THE UNIVERSITY *of* EDINBURGH

A Formalised Approach to the Composition of Processes over Linear Resources

Filip Smola

Doctor of Philosophy

Artificial Intelligence Applications Institute
School of Informatics
University of Edinburgh

2024

Abstract

We present a formal framework for process composition based on actions that are specified by their input and output resources. The correctness of these compositions is verified by translating them into deductions in intuitionistic linear logic. As part of the verification we derive simple conditions on the compositions which ensure well-formedness of the corresponding deduction when satisfied. By connecting the compositions to port graphs, we also verify an alternate characterisation of their correctness in terms of resource dependence between individual actions. We close our theoretical development on an exploration of a probabilistic refinement of non-determinism in our framework.

We mechanise the whole framework, including a deep embedding of ILL and a theory of port graphs, in the proof assistant Isabelle/HOL. Beyond the increased confidence in our proofs, this allows us to automatically generate executable code for our verified definitions.

Lay Summary

Processes are common in our daily lives and work. The way we cook our food is a process, travelling is a process, making a train is a process. And all these processes can be seen as collections of moment-to-moment actions that depend on each other: chopping an onion before adding it to the pan, going through security before boarding the flight, attaching the engine before placing the hood over it.

As these processes get more complex, they get more difficult to carry out. That is made even more difficult when we are cooperating in a team. And for some processes, such as those in healthcare or heavy manufacturing, it is vital to always carry out the process in a specific way.

In our work, we make it easier for computers to understand processes like this so that we can then better instruct them on how to assist us. They can, for example, check that all steps have been carried out in the right order. Or that the proposed process never gets to a point where what is needed for the next step is not present.

We do this in a highly rigorous way, mathematically proving properties of the framework we are creating so that we can trust it is doing what we expect. In this we use a proof assistant called Isabelle/HOL, a computer program purpose-built to help us make proofs and to check that those proofs are correct.

We arrive at a system in which we can define complex processes, check for issues in their structure and visualise them. Thanks to the proof assistant, our framework can be readily used with a number of programming languages to produce tools on a trustworthy foundation.

Acknowledgements

First and foremost, I would like to thank my primary supervisor Jacques Fleuriot. It is his class that showed me the world of interactive theorem proving and I immediately felt that it fits with how my mind works. Through working with Jacques, both on my honours project and as part of research internships, I found a love for research. It was his encouragement that lead me to start on this PhD journey, and it is his support and guidance that saw me through it. Our discussions always helped me refine my work and filled me with ideas for the future, and I am grateful for all the help he has given me over the years.

In addition to my primary supervisor, I would also like to thank the other members of my supervisory team over the years: Petros Papapanagiotou, Jane Hillston and Ohad Kammar. Even before I started the PhD, they helped spark and grow my interest in process modelling, both through undergraduate classes and internships. My work from those internships eventually grew into the research proposal that started my PhD journey. Throughout these four years, they all provided patient feedback on my research and writing, ensuring I keep growing and improving. Their help was invaluable.

Throughout my PhD I have been a member of the AI Modelling Lab. I am deeply grateful to all of them, both for their insightful comments on my work but also the years of friendship. They took what could have been a cold office and made it into a family with our group lunches, trips to Firbush and celebrations of everyone's achievements.

Finally, I would also like to express my thanks to everyone in my life outside of the School of Informatics. To my friends, who helped me through moments of self-doubt and were always happy to listen to me talk excitedly about my work. To my flatmate, who was always there to listen and offer encouragement. And to my parents, who always supported me and made this all possible.

Thank you all.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Filip Smola*)

Contents

Chapter 1

Introduction

In this thesis we present a formal framework for process composition based on actions specified by their input and output. Our framework describes how actions depend on each other's outputs and ensures that they do not compete for resources.

We take inspiration from the proofs-as-processes paradigm [1], which relates processes to linear logic [33]. In our case, we prove that process compositions deemed valid in our framework correspond to well-formed linear deductions, demonstrating that such valid compositions obey our expectations of processes. To make these expectations more concrete, we then formally link compositions to a diagrammatic representation and specify the allowed connections between actions, proving that valid compositions obey them.

We develop our framework in the proof assistant Isabelle/HOL [65] to ensure that its logical underpinnings and the proofs of its properties are fully rigorous. Beyond this, the mechanisation of our framework enables automated generation of verified executable code for our definitions, meaning we can readily use the verified concepts outside of the proof assistant.

To support our work, we mechanise two self-contained theories that can be reused in other projects. One is a deep embedding of intuitionistic linear logic, which we use to translate resources into propositions and process compositions into deductions. The other is a formalisation of port graphs, which we use to graphically represent and reason about process compositions.

Resources. Resources in our framework specify inputs and outputs of individual actions and whole processes, and are discussed in Chapter 2. They can represent physical as well as digital objects, and as such must be manipulated in a linear manner:

preventing the free duplication or discarding of those objects not explicitly marked as allowing it.

This linearity is ensured by our process compositions, a fact that we demonstrate in two ways. First, in Chapter 4, we produce well-formed linear logic deductions for each valid composition, meaning the composition manipulates resources in a way that obeys the rules of linear logic. Second, in Chapter 5, we use port graphs (a refinement of ordinary graphs with ports mediating edge connection), to show that the actions in valid process compositions are connected in a linear way.

Our resources form an algebraic structure [7], allowing for combinations that can represent, for instance, multiple simultaneous objects and non-deterministic outcomes. The atoms of this algebra are not constrained in any way beyond having a notion of equality, which makes them (and the resources induced by them) able to carry extra information such as location or internal state depending on the needs of the domain being modelled. We partition the atoms into two sets: linear and copyable ones. This helps the Isabelle type system control which resources can be duplicated and discarded (e.g. data) and which do not allow that (e.g. physical objects).

Processes. Processes are collections of actions that transform resources, such as manufacturing processes where physical objects are transformed with the use of tools and machines. They are discussed in Chapter 3.

We focus our view on the actions' inputs and outputs, as described by resources, with compositions describing how resources move between individual actions to form a larger process. This view is reminiscent of algorithmic planning [77], but instead of preconditions and postconditions we focus on the objects and data that actions pass to each other.

We formulate a simple condition on how these compositions of processes are formed that ensures they handle the resources in the correct way. This correctness is grounded in linear logic (see Chapter 4) and made more concrete with port graphs (see Chapter 5).

Moreover, in the presence of complex information in resource atoms, this condition can have implications such as ensuring the process is free of bottlenecks or that it obeys a given graph of locations. We illustrate this with a case study in Section 7.4.

Our view is focused on processes as collections of actions connected through their inputs and outputs, rather than focusing on the agents executing those actions. This can be contrasted with many process calculi, whose presentations often focus on agents

performing sequences of actions while communicating with each other or sometimes cooperating on the actions. Our approach is more specific, making correctness conditions simpler to check. We note the possibility of formally connecting to these other formalisms as part of future work in Section 8.1.

Contributions. The contributions of this thesis revolve around a formal language for describing the structure of processes composed from actions. We list the main points:

- Mechanisation of the language for composing processes, supporting automated generation of executable code.
- Verified operations for converting processes between modelling domains while preserving their correctness.
- General mechanisation of intuitionistic linear logic.
- Translation of all valid processes into well-formed deductions of linear logic.
- General mechanisation of port graphs.
- Formalised graphical representation of processes, useful for both visualisation and proof.
- Preliminary exploration of a probabilistic extension to our language.

Thesis Structure. In the rest of Chapter 1 we introduce the main strands of background work that feed into our framework. This includes intuitionistic linear logic (in Section 1.1), to which we appeal when arguing the correctness of our process compositions, and the proofs-as-processes paradigm (in Section 1.2), which inspires this connection between processes and linear logic. We also give a brief overview of interactive theorem proving (in Section 1.3) focusing on Isabelle/HOL, the proof assistant in which we mechanise our work throughout the rest of this thesis.

In Chapter 2 we introduce resources, which we use to specify inputs and outputs of actions. This includes their formalisation as a universal algebra as well as a decision procedure for equality of resources based on rewriting.

In Chapter 3 we build on resources by introducing process compositions. We formally define compositions of processes with resources as inputs and outputs, and give a brief overview of how we visualise them with process diagrams. We then touch on

systematic transformations of processes, which allow us to reuse the structure of a composition in a different domain while preserving its correctness.

In Chapter 4 we demonstrate that the notion of validity for process compositions ensures *linearity*. Following common practice in literature around the proofs-as-processes paradigm, we do so by relating to linear logic. Specifically, we map resources to propositions and processes to deductions, with valid compositions mapping to well-formed deductions.

In Chapter 5 we take an alternative approach to demonstrating linearity compared to Chapter 4: instead of appealing to linear logic for the notion of linearity, we express it directly in terms of resource-bearing connections between individual actions. To formalise these connections, we map process compositions to port graphs where they figure as edges.

In Chapter 6 we explore an extension of our framework to include probabilistic information by furnishing non-deterministic resources with concrete distributions between their options. While this expands the information that process compositions can express, it also significantly complicates their theory. We mitigate some of these complications, we discuss limitations of our approach and suggest future directions.

In Chapter 7 we detail a number of concrete domains and process models. These case studies illustrate the main features of our framework, from representing non-determinism and action refinement to using composition validity to ensure that a process is free of bottlenecks.

In Chapter 8 we summarise the thesis, highlight the major threads of future work and offer some concluding remarks.

Published Work. We have published part of the work described in this thesis in the Journal of Automated Reasoning as "Linear Resources in Isabelle/HOL" [83]. Section 3 of the paper discusses resources, which are discussed in Chapter 2 of this thesis. Section 4 of the paper discusses process compositions, which are discussed in Chapter 3 of this thesis. Section 5 of the paper discusses the translation of resources and process compositions into linear logic, which are discussed in Chapter 4 of this thesis. Section 7 of the paper discusses the modelling of manufacturing in the simulation game Factorio, which is discussed in Section 7.4 of this thesis.

As the paper is based on an earlier version of our work, there are changes both in the formalisation and in its discussion. This thesis refines and expands upon what is described in the paper.

Code. The code supporting this thesis is available online in the Archive of Formal Proofs (AFP)[1] and on GitHub. The archive entries contain the stable parts of our framework, reviewed by editors to ensure they build without errors in the latest version of Isabelle/HOL. The GitHub repositories contain a development version of the mechanisation, with examples and in-progress formalisation, as well as non-proof code.

Our mechanisation of intuitionistic linear logic, corresponding to Section 4.2 and Section 4.5, is available in the AFP as the entry *ILL* [81]. So is the core of our framework and its connection with ILL, corresponding to Chapter 2, Chapter 3 and the rest of Chapter 4, as the entry *ProcessComposition* [82].

These AFP entries have development counterparts in two GitHub repositories, one for the *ILL* entry[2] and one for the *ProcessComposition* entry[3]. The latter repository contains extra theories with respect to the archive, as well as the case studies from Section 7.1, Section 7.3 and Section 7.4. It also includes our mechanisation of process port graphs from Section 5.4, resting on the general mechanisation of port graphs from Section 5.3 which has its own repository[4].

Code generated from our formalisation is tracked in its own repository[5]. This allows us to control the version of the mechanisation from which it is generated and turn it into a Haskell library that can be easily reused. It is used by our final repository[6], which contains the implementation of process diagrams from Section 3.3.

1.1 Intuitionistic Linear Logic

Logical systems usually contain what are called structural rules, ones governing the structure of the argument rather than its logical content. For instance, a core such rule of linear logic is the *Cut* rule, which states that two deductions can be connected to have one satisfy an assumption of the other. When discussing linear logic we are mainly interested in the structural rules *Weakening* and *Contraction*. *Weakening* states that whatever we can deduce from a collection of propositions Γ, we can also deduce from Γ with the addition of any further proposition. *Contraction*, for its part, states that whatever we can deduce from a collection of propositions Γ containing two instances

[1] `https://www.isa-afp.org`
[2] `https://github.com/pilif0/isa-ILL`
[3] `https://github.com/pilif0/isa-ProcessComposition`
[4] `https://github.com/pilif0/isa-PortGraph`
[5] `https://github.com/pilif0/codegen-ProcessComposition`
[6] `https://github.com/pilif0/process-diagram`

of some proposition A, we can also deduce a variant of Γ containing only a single instance of A. That is, in essence, they allow us to disregard and reuse assumptions respectively.

Linear logic is a formal system introduced by Girard [33] which constrains the use of *Weakening* and *Contraction*. As a result, it accounts for the number of propositions being used and not just their presence, making it well suited for representing resources and processes. In this section we discuss linear logic and in particular its intuitionistic fragment. In Chapter 4 we use intuitionistic linear logic to argue correctness of process compositions formed in our framework. As such, the rules of linear logic influence the theory presented in Chapter 2 and Chapter 3 in order for the correctness argument to be valid.

Substructural logics are those that constrain or outright remove one of the structural rules. For instance, relevant logic [27] disallows *Weakening*. Linear logic does not purely disallow *Weakening* and *Contraction*, it instead restricts them with the modal operator !. As a result, deduction of ordinary logic remains accessible under this modality while more controlled *linear* deduction takes place outside of it. It is this linearity, the control of *Weakening* and *Contraction*, that makes linear logic useful when accounting precisely for the resources consumed and produced by processes. This is demonstrated by the frequent use of linear logic in the formal process modelling literature, which we note in Section 1.2.

In the present work we use *intuitionistic* linear logic (ILL), and specifically its sequent calculus formulation. In contrast to classical logic, it only allows deductions with a single conclusion. Specifically with linear logic, this means that ILL has fewer operators and rules than its classical counterpart (CLL). However, the intuitionistic constraint is well suited to our representation of processes as deductions from one input proposition to one output proposition (see Chapter 4).

Given a set of propositional variables A, the propositions of ILL are generated as follows:

$$P, Q = a \mid \mathbf{1} \mid P \otimes Q \mid \mathbf{0} \mid P \oplus Q \mid \top \mid P \,\&\, Q \mid P \multimap Q \mid !P \qquad \text{for } a \in A \tag{1.1}$$

where $\otimes$ is the operator *times*, $\oplus$ is the operator *plus*, & is the operator *with*, $\multimap$ is *linear implication* and ! is *exponential*.

Sequents are of the form $\Gamma \vdash C$ where Γ is a list of propositions, the *antecedents*, and C is a single proposition, the *consequent*. Valid sequents are generated by the sequent calculus rules shown in Figure 1.1, which we take from Bierman's work [12].

$!\Gamma$ denotes the result of exponentiating (i.e. applying ! to) each proposition in the list Γ.

In the rest of this section we introduce the operators of ILL in more detail, with reference to the rules in Figure 1.1, and link them to our use. We pay particular attention to the ! operator.

The operators $\otimes$ and & are the two linear forms of conjunction, with units $\mathbf{1}$ and $\top$ respectively. Following Girard's terminology, $\otimes$ is considered *multiplicative* conjunction while & is considered *additive* conjunction. This is because in the premises of their rules $\otimes_R$ and $\&_R$, $\otimes$ requires distinct formula lists Γ and Δ as antecedents while & requires the same list Γ. Intuitively, $\otimes$ represents simultaneous availability of two formulas while & represents the availability of a choice of one of them. Thus $\otimes_R$ combines into the antecedents those of both premises, while $\&_R$ only propagates one list of antecedents into its conclusion. In our work we only make use of $\otimes$, which forms the counterpart to parallel resources.

The operator $\oplus$ is the only linear form of disjunction in ILL and has $\mathbf{0}$ as its unit. Note that in the premises of its rule $\oplus_L$ it requires the same formula list Γ in the antecedents, making it *additive* disjunction. Intuitively, $\oplus$ represents the non-deterministic availability of one of the two formulas. Thus, just as with $\&_R$, the rule $\oplus_L$ only propagates one list of antecedents into its conclusion. In our work this operator forms the counterpart to non-deterministic resources.

The operator $\multimap$ is the linear form of implication. Note that in the premises of its rule $\multimap_L$, it requires distinct formula lists Γ and Δ in the antecedents, making it *multiplicative* implication. Intuitively, $\multimap$ represents the availability of a transformation from its left-hand formula to its right-hand formula. This can be seen in how it is derived using $\multimap_R$. In our work this operator forms the counterpart to executable resources.

Finally, the ! operator controls the use of weakening and contraction. This is a distinguishing feature of linear logic, limiting the use of weakening and contraction but not outright rejecting them. Notably, reasoning in intuitionistic linear logic with exponentiated formulas recovers ordinary intuitionistic logic. In our work this operator forms the counterpart to copyable resources, restricting copying (contraction) and erasing (weakening) to them.

There are four rules in ILL concerning the ! operator. The first two, Weakening and Contraction, reintroduce these two concepts into the logic but constrain them only to exponentiated formulas. Thus, once we have an exponentiated formula we can get

any number of copies of it or fully discard it. The third rule, Dereliction, says that if something can be derived from a list of formulae then it can be derived with any of them exponentiated. Intuitively this is because having one copy is a special case of being able to get any number of copies. The fourth rule, Promotion, says that if something can be derived from a list of only exponentiated formulas then the result can be exponentiated. Intuitively this is because to get any number of the consequent we need only copy all of the antecedents that many times.

Structural

$$\frac{}{A \vdash A}\ \text{Identity} \qquad \frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, B, A, \Delta \vdash C}\ \text{Exchange} \qquad \frac{\Gamma \vdash B \qquad B, \Delta \vdash C}{\Gamma, \Delta \vdash C}\ \text{Cut}$$

With

$$\frac{\Gamma, A \vdash C}{\Gamma, A \ \& \ B \vdash C}\ \&_{L-1} \qquad \frac{\Gamma, B \vdash C}{\Gamma, A \ \& \ B \vdash C}\ \&_{L-2} \qquad \frac{}{\Gamma \vdash \top}\ \top_R$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \ \& \ B}\ \&_R$$

Plus

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \oplus B}\ \oplus_{R-1} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \oplus B}\ \oplus_{R-2} \qquad \frac{}{\Gamma, \mathbf{0} \vdash C}\ \mathbf{0}_L$$

$$\frac{\Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma, A \oplus B \vdash C}\ \oplus_L$$

Times

$$\frac{\Gamma \vdash A}{\Gamma, \mathbf{1} \vdash A}\ \mathbf{1}_L \qquad \frac{}{\vdash \mathbf{1}}\ \mathbf{1}_R \qquad \frac{\Gamma, A, B \vdash C}{\Gamma, A \otimes B \vdash C}\ \otimes_L \qquad \frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B}\ \otimes_R$$

Linear Implication

$$\frac{\Gamma \vdash A \qquad \Delta, B \vdash C}{\Gamma, \Delta, A \multimap B \vdash C}\ \multimap_L \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B}\ \multimap_R$$

Exponential

$$\frac{\Gamma \vdash B}{\Gamma, !A \vdash B}\ \text{Weakening} \qquad \frac{\Gamma, !A, !A \vdash B}{\Gamma, !A \vdash B}\ \text{Contraction}$$

$$\frac{\Gamma, A \vdash B}{\Gamma, !A \vdash B}\ \text{Dereliction} \qquad \frac{!\Gamma \vdash A}{!\Gamma \vdash !A}\ \text{Promotion}$$

Figure 1.1: ILL Inference Rules

1.2 Proofs-as-Processes

The *proofs-as-processes* paradigm was introduced by Abramsky [1] and examined in more depth by Bellin and Scott [9]. It concerns the connection of linear logic to processes akin to the famous *propositions-as-types* [93] paradigm. In particular, Bellin and Scott examine how a deduction in classical linear logic can be used to synthesise a π-calculus [61] agent. That agent's behaviour mirrors the manipulation of propositional variables described by the deduction, with a correspondence between execution of the agent and cut elimination of the logic.

There are further strands of work within the proofs-as-process paradigm that use linear logic to argue for process correctness. Caires and Pfenning [21] describe a type system for π-calculus corresponding to the proof system of dual intuitionistic linear logic [8]. The types in this system can be viewed as session types, a wider concept introduced by Honda [44] which offers a type discipline for symmetric dyadic communication, such as between a server and a client in a distributed system. The connection between session types and linear logic is reinforced by the work of Wadler [92] who, instead of using π-calculus, introduces the new calculus CP. While session types focus more on communication protocols of virtual agents in distributed systems, they give us confidence that well-formed linear logic deduction is good evidence for correctness of a process.

As such, we take adherence to the rules of linear logic as a formalisation of process correctness. However, instead of synthesising processes from deduction, we compose processes independently in a way that all valid compositions reflect well-formed deductions in intuitionistic linear logic. We verify this in Chapter 4 by mechanising deductions of ILL in Isabelle/HOL and then defining exactly how compositions map to them. Note that we do not formalise a notion of execution for process compositions and thus, unlike Bellin and Scott, we do not relate execution to cut elimination. Nevertheless, our processes take some inspiration from deductions of ILL as well as from the proofs-as-processes literature.

1.2.1 WorkflowFM

One application of the proofs-as-processes paradigm, and a significant source of inspiration for our framework, is WorkflowFM [69], a formal framework for modelling and deploying workflows, which we previously worked on. At its core is a reasoner based on HOL Light [36] which accepts instructions from a graphical user interface

and performs automated deduction in a deeply-embedded classical linear logic (CLL). Following the proofs-as-processes paradigm, specifically the work of Bellin and Scott, the reasoner produces a π-calculus process as a side-effect of the deduction. The generated π-calculus term can be deployed into a framework implemented in Scala for the purposes of simulation and monitoring. WorkflowFM has been applied in the medical and manufacturing domains [56, 68, 70].

Our current framework takes a similar perspective on processes, viewing them as actions connected by their inputs and outputs, and similarly targets physical processes such as manufacturing. However, our connection to linear logic is looser than that of WorkflowFM: its processes are side-effects of deduction while ours are separate objects that can be used to construct a deduction on demand. This decoupling has three notable effects.

First it means that modelling with our framework is not slowed down by the proof assistant. With WorkflowFM, interaction with the prover is a constant part of the modelling process as each operation triggers proof search. We instead verify the operations of our framework ahead of time so that modelling can proceed independently using just the generated verified code. This change speeds up the modelling action-response loop and makes programs that create compositions easier to implement.

Second, we are not limited to *classical* linear logic. Our switch to *intuitionistic* linear logic allows us to express higher-order processes using its linear implication. In CLL linear implication is syntactic sugar, with $A \multimap B = A^{\perp} \parr B$. But the same can also mean an ordinary process from A to B, making higher-order processes indistinguishable from ordinary processes. As a result, WorkflowFM cannot express higher-order processes distinctively as can be done in our framework.

Third, our notion of correctness is not limited to linear logic. In WorkflowFM, correct processes are side-effects of deduction and so changing the notion of correctness would mean changing the logic. In our framework, process composition correctness is expressed by a predicate on the composition itself (see Section 3.2). Correctness with respect to linear logic is expressed by our translation into well-formed deductions of ILL. But, if we were to add more correctness conditions, then that translation would still work and we would only have to further verify those additional conditions.

As a final note, the WorkflowFM framework includes a graphical user interface for composing processes, where processes are visualised using diagrams. Those diagrams are implemented entirely outside of the formal environment and, as a result, their faithfulness to the process formalism cannot be verified. In contrast, our framework

includes a formal connection of process compositions to port graphs, as described in Chapter 5, which can serve as a basis for visualisation.

1.3 Interactive Theorem Proving

Automated and interactive theorem proving both involve the use of computers to check proofs in some formal logical system. In the case of automated theorem proving, the computer is also tasked with finding the proof itself, while in the interactive setting the user guides the proof construction. The interactivity lets us tackle more complex problems, where automated procedures might not find a proof, but it comes at the cost of needing manual intervention. In practice, modern interactive theorem provers often integrate with automated theorem provers and offload simpler goals to these, allowing the user to focus on more complex, higher-level reasoning.

Since our aim is to produce a framework that has been formally verified rather than to fully automatically produce proofs as part of its day-to-day use, we choose to use an interactive theorem prover. This also allows us to annotate the proofs as we build them, shaping them to be more readable than ones found automatically, which aids the trustworthiness of our formalisation.

There are two popular approaches to interactive theorem provers: the Logic of Computable Functions (LCF) [79] approach and the calculus of constructions (CoC) [23]. Since we use Isabelle/HOL, which follows the LCF approach, we focus our discussion on the former, but we also briefly recount the CoC approach.

In the LCF approach, first mechanised by Milner [60], we use the type system of a host language to reduce soundness of proofs to a small kernel that can be manually analysed and trusted. This can be done by using abstract types, so that the only way to construct an object of type "theorem" outside of this kernel is by using sound operations. The rest of the formal environment is then built up from these basic operations using the full range of programming constructs available in the language. This approach is used, for instance, by HOL [34], HOL Light [36] and Isabelle [71].

The CoC approach is itself a dependent type theory, often extended to include (co)inductive types. This means that, in contrast to LCF's use of the host language type system, here the theory *is* the type system. Moreover, systems based on the CoC use dependent type theory, while systems based on LCF use simple type theory. The calculus of constructions is used, for instance, by Rocq [88] and Lean [63].

We use Isabelle/HOL, which is an axiomatisation of higher-order logic in the

generic proof assistant Isabelle [71]. Beyond our familiarity with the system, its advantages lie in its mature proof language resembling written mathematics, a wide range of proof automation tools and its ability to automatically generate executable code from our definitions. In the rest of this section we give more detail on Isabelle/HOL and of its use in our work.

There is nothing in our experience that leads us to believe that an analogous mechanisation could not be achieved using other proof assistants. While we have not found ourselves limited by the simple type theory, a system based on a more involved type theory may allow us to express certain concepts more concisely. For instance, what we now achieve with a combination of datatype and relation — such as valid compositions (see Section 3.2) and well-formed port graphs (see Section 5.3.4) — other systems may be able to express with just the type. Additionally, the automation available for specification, proving and code generation differs from tool to tool. We have found Isabelle/HOL to be comfortably up to the present task.

1.3.1 Isabelle/HOL

We mechanise our work in the proof assistant Isabelle [71] using higher-order logic (HOL). As we noted previously, Isabelle follows the LCF approach with the host language being ML [62], forming a general basis in which many logical systems can be axiomatised. We use the axiomatisation of higher-order logic, known as Isabelle/HOL.

When working with an LCF-style proof assistant, we find it helpful to follow the HOL methodology: making extensions to the theory in a conservative manner, using definitions instead of axiomatisations. A bad definition is at worst not useful, while a bad set of axioms may hide an inconsistency and allow us to "prove" false statements without the kernel needing to be compromised. As such, our work consists of definitions — of datatypes, relations, functions and logical locales — and no axioms.

Isabelle provides the proof language Isar [95] which is close to the language of mathematical proofs. This aids in readability and maintainability of the statements and their proofs, and seamlessly integrates with all the automation available in Isabelle.

It also includes Sledgehammer [13], which allows the user to invoke a number of automated theorem provers on a specified goal with the aim of finding its proof. If they succeed, then their output is used as a guide by Isabelle to reconstruct the proof using its own internal methods. This is especially useful for finding proofs of simpler subgoals, allowing us to concentrate on higher-level reasoning. Even when

unsuccessful, the output may contain information about where the proof lies or that the statement is false.

We found the combination of Isar and Sledgehammer invaluable when mechanising our theory. The robustness of Isar means that, in many cases, when we make a change deep in a formalisation many of the proofs following it will still work. Moreover, with Sledgehammer even those proofs that require different justification can often be rapidly adapted. Thus, what is left for us to focus on are the core implications of the change. This removes much of the cost of exploration and allows us to rapidly iterate through ideas.

1.3.2 Isabele/HOL Syntax

In this thesis we use Isabelle/HOL code blocks to introduce definitions and theorem statements, with terms rendered in italics. These definitions include inductive datatypes (**datatype**), recursive functions (**function** and **fun**) including primitively recursive ones (**primrec**), inductive relations (**inductive**), and general definitions (**definition**). Proven theorems start with the keyword **lemma**, followed by a name and then the statement being proven. In most cases we omit the proofs of our lemmas for the sake of space. Type variables are preceded by $'$ as for instance in $'a$. We also use Isabelle/HOL syntax inline when talking about formal entities, such as the empty list constructor *Nil*.

In our formal statements we use the following Isabelle/HOL notation:

- Meta-level implication: $[\![P; Q]\!] \Longrightarrow P \wedge Q$ expresses the conjunction introduction rule from the assumptions P and Q;
- List construction: $[x, y] = x \# [y]$;
- List append: $[x, y] = [x] \mathbin{@} [y]$;
- List to set conversion: $set\ [1, 2, 7, 4, 2] = \{1, 2, 4, 7\}$
- List quantifiers: $list\text{-}all\ P\ xs = (\forall x \in set\ xs.\ P\ x)$ and $list\text{-}all2\ P\ xs\ ys = (\forall (x, y) \in set\ (zip\ xs\ ys).\ P\ x\ y)$
- Function image: $f\ `\ \{x, y, z\} = \{f\,x, f\,y, f\,z\}$
- String literal: $STR\ ''Hello\ World''$.

1.3.3 Extraction of Code from Isabelle/HOL

To make practical use of our framework, we take advantage of the code generator [35] included in Isabelle/HOL. Given a set of formal objects to export, this system takes the definitions we make and the theorems we prove, and transforms them into executable code in one of several target languages. By default, Isabelle is distributed with support for SML [62], OCaml [52], Haskell [72] and Scala [66], and support for Go [86] is available on the Archive of Formal Proofs [87]. We will focus our attention on Haskell, but in principle any of these languages could be used in its place.

The code generator first collects code equations for all the requested formal objects, which can come directly from their definitions or from proven theorems. It then recursively collects all other formal objects needed for those code equations until it has all that it needs, and preprocesses them using a configurable range of proven theorems. This preprocessing step can for instance be used to replace certain functions with more performant variants, as long as we can prove them to be equal on all inputs.

The final code equations are translated into an intermediate functional language, called Thingol, which is then serialised into source code of the selected target language. The code equations, and any preprocessing done on them, are part of the logical environment of Isabelle, and thus verified. Only the translations into Thingol and into the target language fall outside the logic.

In practice, for most of our formalisation the code generation is set up by the specification tools themselves: inductive datatypes, primitively recursive functions and plain definitions. However, there is a case where we need to manually bridge the gap from our specification to executable code: the resource term equivalence in Section 2.2, whose inductive relation definition does not automatically yield a code equation. It is only once we prove that the equivalence can be decided by a rewriting normalisation procedure that we can register a code equation for it (see Section 2.3.6).

The ability to extract executable code from our definitions enables us to work with the formal objects outside of the proof assistant. This means that we can use them in informal contexts, such as drawing diagrams like those we discuss in Section 3.3. It also means that, if we wish to only construct and consume the process compositions and not make proofs about them, we need not be slowed down by a fully formal environment (see for instance the case study in Section 7.2). Throughout this we have an assurance that the code we are working with accurately reflects the verified theory.

1.4 Conclusion

In the next chapter, we introduce our notion of resources. They are the foundation on which we build process compositions, representing their inputs and outputs, and feature throughout the rest of our work.

Chapter 2

Resources

The role of resources in our framework is to describe the inputs and outputs of processes. Their basis is *atoms*, which are drawn from two unconstrained type parameters: one for linear resources and another for copyable resources.

Starting from these atoms, resources can be combined in several ways to express more complex situations. Parallel combination represents simultaneous presence of several resources. Non-deterministic combination represents exactly one of two resources, without specifying which one. An executable resource represents a single potential execution of a process, combining its input and output resource, while a *repeatably* executable resource does not limit the number of uses. In Chapter 3 we use these combinations to capture the inputs and outputs of processes and their compositions.

We formalise resources as a universal algebra [7], through its terms and equations. The terms, which we describe in Section 2.1, serve to express any combination of atoms. The equations serve to connect different terms for what we consider the same resource, and in Section 2.2 we describe their mechanisation as an equivalence relation on terms. This yields resources as a quotient of the terms by that equivalence relation.

While the equivalence relation specifies what resources are equal, it in itself is not suitable for showing what resources are distinct. In Section 2.3 we discuss a decision procedure for resource term equivalence based on term rewriting [6]. By reducing terms into their normal forms and comparing those, this procedure lets us decide whether or not two resources are equal in a computable way, allowing for generation of executable code involving resources. Beyond deciding the equality of resources, in Section 2.4 we also note how the rewriting relation involved in this procedure allows us to take advantage of more automation available in Isabelle/HOL.

In Chapter 4 we make an argument for the linearity of our process composition by relating them to deductions of intuitionistic linear logic (ILL). As part of that, we relate resource terms to propositions of ILL and resource term equivalence to proofs in ILL (i.e. deductions with no premises).

Running example. To illustrate some of the concepts in this chapter, we will be using a simple vending machine as a running example. This machine can accept cash, dispense one kind of drink at a fixed cost and return change. See Chapter 7 for more involved examples of resources.

2.1 Resource Terms

When describing interesting processes we rarely talk about singular objects. An action may for example require or produce multiple resources, or its result may be non-deterministic (e.g. it can fail). Thus our resources combine in various ways to formally express such situations, building from the individual objects of the domain and two special objects.

Our first step in mechanising resources in Isabelle/HOL is to express the possible combinations as terms. Given two types of resource atoms, $'a$ for linear atoms and $'b$ for copyable atoms, the resource term type $('a, 'b)$ *res-term* has four leaf resources and four resource combinations:

Isabelle Definition 2.1.1 (Datatype of resource terms)

```
datatype ('a, 'b) res-term =
  Res ('a)
| Copyable ('b)
| Empty
| Anything
| Parallel (('a, 'b) res-term list)
| NonD (('a, 'b) res-term) (('a, 'b) res-term)
| Executable (('a, 'b) res-term) (('a, 'b) res-term)
| Repeatable (('a, 'b) res-term) (('a, 'b) res-term)
```

We now describe each of the eight resource term constructors and illustrate the kinds of information the type parameters $'a$ and $'b$ can represent. See Section 3.1 for how resources are used in process compositions, giving a further perspective on the meaning of each resource and combination.

The first two kinds of leaves, constructed by *Res* and *Copyable*, essentially inject the resource atoms into the type (linear and copyable atoms respectively). This means that every resource atom is itself a resource, which can then be combined into more complicated resources. The key difference between the two kinds of atoms is that copyable atoms allow for copying and deleting actions (see Section 3.1.3).

The third leaf, *Empty*, represents as resource the absence of any object. It is useful when we want to describe an action with no input or no output. And, with the equations introduced in Section 2.2, it acts as a unit for parallel combination of resources.

The fourth and final leaf, *Anything*, represents a resource about which we have no information. Any resource, and thus also their combinations, can be turned into this one by "forgetting" all information about it (see Section 3.1.3). In certain cases this allows us to more concisely express a resource, but at the cost of no longer being able to act on the forgotten resources. See Section 7.2 for an example of how this resource can be used to more concisely express the output of a process.

The first combination, *Parallel*, represents the simultaneous presence of a whole list of resources as one. For instance, *Parallel* [*Res Nail, Res Hammer, Res Picture*] could represent what is required to hang a picture on a wall: a nail, a hammer and the picture itself.

The second combination, *NonD*, represents exactly one of two resources without indicating which one. The primary use of this combination is for actions with non-deterministic outcomes. For instance, *NonD* (*Res Success*) (*Res Error*) for a process that can succeed or fail with an error, or *NonD* (*Res Heads*) (*Res Tails*) for the result of a coin toss.

The third combination, *Executable*, represents a single potential execution of a process and is specified by the process input and output resources. This allows us to represent higher-order processes, which are processes that take as input or produce as output other processes. For instance, *Executable* (*Res SteelSheet*) (*Parallel* [*Res SteelTiles, Res Waste*]) represents the ability to cut a steel sheet into tiles while producing some waste.

The fourth and final combination, *Repeatable*, is the repeatable variant of *Executable*. It still represents a potential execution of a process specified by its input and output, but one that can be used repeatedly. This allows us to represent higher-order processes when we do not know how many times they will use this resource. For instance, the number of its executions might be decided by another resource, such as a manufacturing order containing a list of instructions to perform.

Note that the type parameters $'a$ and $'b$ from which resource atoms are drawn are not constrained in any way. This means they can be any type we can define in Isabelle/HOL, only requiring the elements to have a notion of equality. As a result we can build resources from atoms such as the following:

- Objects with internal state: *Glass* c v where $c \in \{\textit{Water, Milk, Juice}\}$ and $\mathrm{v} \in \mathbb{R}$ is the contained volume;
- Objects at graph-like locations: $(\textit{Bowl, Kitchen Counter})$, $(\textit{Coat, Hall Rack})$ where *Kitchen Counter* and *Hall Rack* are vertices of some graph;
- Stacks of objects: $(\textit{Plate, } n)$ where $\mathrm{n} \in \mathbb{N}$ is the count.

This allows us to easily add information to the resources. As compositions of processes use resources, requiring their equality wherever a connection is made, this information has an effect on what compositions are valid (see Section 3.2). For instance, with resource atoms located in a graph we can ensure that any movement of resources is done between adjacent locations.

Example. In the case of our running example, the vending machine, there are three kinds of objects we care about: cash, drinks and the machine itself. So, in that domain, these form our type of resource atoms with constructors *Cash*, *Drink* and *Machine*.

The constructor *Drink* is sufficient to represent a single drink and does not need a parameter. But *Cash* does need a parameter: we represent the amount n of some currency as *Cash* n. Similarly for the vending machine itself, we parameterise the *Machine* constructor with a natural number to represent the amount that the machine currently holds in credit.

Note that for simplicity we assume any amount of currency is one object rather than deal with denominations used in the real world. However, this model could be extended in a straightforward way to account for this detail.

The resource construction most relevant in this domain is parallel combination. For instance, if we have a vending machine with m already in credit and want to buy a drink that costs $m + c$, then we may start in the situation described by the following resource:

$$\textit{Parallel}\ [\textit{Res}\ (\textit{Cash}\ c),\ \textit{Res}\ (\textit{Machine}\ m)]$$

2.2 Resources as Quotient of Terms

Resource terms give many ways of expressing the same resource. For instance, Figure 2.1 shows syntax trees for five resource terms which express the same resource: the single (linear) atom A.

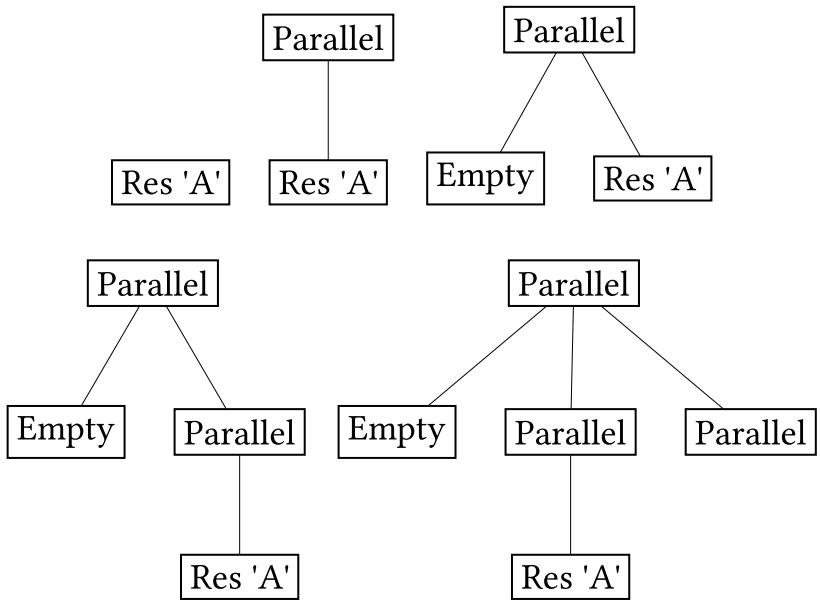


Figure 2.1: Five resource term trees expressing the same resource, the single atom A.

At present, we pay special attention to the variety of terms that can be produced by the parallel combination of resources. This is because parallel combinations arise frequently in our process combinations, and when they do it is often with more than two resources. Rendering the alternatives equal, as we set up in this section, significantly simplifies building compositions at the price of a manageable increase to complexity of their mechanisation.

Other resource combinations can also produce a variety of terms, for example *NonD* x x could be considered the same as x. In Chapter 6 we demonstrate how such further resource equalities can be added to our framework, a task made easier thanks to the automation available in Isabelle/HOL. As such, we consider adding resource equalities to our framework as a recurring part of future work (see Section 2.5).

Note that, although they are not resource equations, further resource transformations can also be achieved through composition operators and resource actions (see Section 3.1.2 and Section 3.1.3).

In our formalisation of resources, we collect all of the possible forms of a resource into one object. We start by defining an equivalence relation on resource terms and

then form their quotient by that relation. This allows us to lift operations on terms to yield operations on resources.

We relate terms that represent the same resource by the relation $\sim$, defined inductively as shown in Definition 2.2.1. The first three introduction rules (lines 3–5) express the core of the equivalence, handling parallel combinations with no children, single child and nested parallel combinations respectively. Then the next eight introduction rules (lines 6–13) close the relation on the resource term structure, meaning the result will be a congruence. The final two rules (lines 14–15) make the relation symmetric and transitive. The fact that it is reflexive, the last condition for it to be an equivalence, can be proven from this definition.

Isabelle Definition 2.2.1 (Equivalence of resource terms)

```
1  inductive res-term-equiv :: ('a, 'b) res-term ⇒ ('a, 'b) res-term ⇒ bool (infix ∼ 100)
2   where
3     Parallel [] ∼ Empty
4   | Parallel [a] ∼ a
5   | Parallel (x @ [Parallel y] @ z) ∼ Parallel (x @ y @ z)
6   | Empty ∼ Empty
7   | Anything ∼ Anything
8   | Res x ∼ Res x
9   | Copyable x ∼ Copyable x
10  | list-all2 (∼) xs ys ⟹ Parallel xs ∼ Parallel ys
11  | ⟦x ∼ y; u ∼ v⟧ ⟹ NonD x u ∼ NonD y v
12  | ⟦x ∼ y; u ∼ v⟧ ⟹ Executable x u ∼ Executable y v
13  | ⟦x ∼ y; u ∼ v⟧ ⟹ Repeatable x u ∼ Repeatable y v
14  | x ∼ y ⟹ y ∼ x
15  | ⟦x ∼ y; y ∼ z⟧ ⟹ x ∼ z
```

Then we can define resources as the quotient of terms by the relation $\sim$, treating each equivalence class as one object. This is easily done in Isabelle/HOL:

Isabelle Definition 2.2.2 (Quotient type of resources)

```
quotient-type ('a, 'b) resource = ('a, 'b) res-term / (∼)
```

For making definitions using the quotient type we use lifting, which takes an expression involving resource terms and *lifts* them to use resources instead. In Isabelle/HOL this is automated by the Lifting package introduced by Huffman and Kunčar [46].

The main tool that Lifting introduces is **lift-definition**. This keyword takes a new name, the desired (lifted) type and original (unlifted) expression. It deduces a formal

connection between the original and new types, producing automatically a definition for a new constant with the desired type. To complete the lifting, we need to prove a respectfulness condition: that applying the original expression to equivalent arguments produces equivalent results, ensuring that the choice of representatives would not matter. With this obligation fulfilled, Isabelle automatically proves a number of facts relating the new constant to the original expression.

For instance, in Definition 2.2.3 we lift the *Parallel* constructor from resource terms to resources. The respectfulness obligation in this case is one of the rules defining the relation $\sim$, and so can be proven in a single step.

Isabelle Definition 2.2.3 (Parallel resource)

```
lift-definition Parallel :: ('a, 'b) resource list ⇒ ('a, 'b) resource is res-term.Parallel
  by (rule res-term-equiv.intros)
```

We use the same approach to lift all the resource term constructors to resources, allowing us to directly construct resources without going through terms every time. In Section 2.3.7 we use lifting to define the normal form representative for every resource. Other functions of resources we define also follow the pattern of first defining them for resource terms and then lifting that to resources, for instance the *parallel-parts* function in Section 3.3 and the *process-refineRes* function in Section 3.4.2. And in Section 2.4 we use more advanced automation to lift additional structure from resource terms to resources.

The main issue that remains is that of characterising the equivalence in a computable way. This is important, because without it we would not be able to generate executable code for anything involving resources. We address this in Section 2.3 with a normalisation procedure based on rewriting.

Before addressing that issue, let us define an infix notation for parallel resources which we call the resource product and denote with $\odot$. This simplifies statements that involve small numbers of concrete resources being in parallel.

Isabelle Definition 2.2.4 (Resource product)

```
definition resource-par :: ('a, 'b) resource ⇒ ('a, 'b) resource ⇒ ('a, 'b) resource
  (infixr ⊙ 120)
  where x ⊙ y = Parallel [x, y]
```

Thanks to resources being a quotient, this operation is associative and has *Empty* as its left and right units (i.e. it forms a monoid).

Isabelle Lemma 2.2.1 (Associativity and unitality of the resource product)

lemma *resource-par-assoc*:
$a \odot (b \odot c) = (a \odot b) \odot c$

lemma *resource-par-unitR*:
$x \odot \mathit{Empty} = x$

lemma *resource-par-unitL*:
$\mathit{Empty} \odot x = x$

Note that, once the quotient is made, it does not matter if we initially defined the parallel resource term combination as having arbitrary arity or as strictly binary. It only affects the phrasing of the equivalence rules, the normalisation procedure we use to decide that equivalence and the representation of resources in generated code. We choose to use lists because they more concisely reflect the monoidal nature of parallel resources and make for a simpler normalisation procedure.

2.3 Resource Term Normalisation by Rewriting

The relation $\sim$ specified which resource terms represent the same resource, but its rules on their own are not enough to show the negative. For instance, even from $\mathit{Empty} \sim \mathit{Anything}$ we do not arrive at contradiction but at the suggestion that if there was some term y such that $\mathit{Empty} \sim y$ and $y \sim \mathit{Anything}$ then the equivalence could hold. It is difficult to prove that no such y could exist.

To remedy this, we give a computable characterisation of the resource term equivalence $\sim$ through a normalisation procedure based on term rewriting. For an introduction to term rewriting, see for instance the book by Baader and Nipkow [6]. In our rewriting system we use the following rules:

$$\mathit{Parallel}\ [] \to \mathit{Empty} \tag{2.1}$$

$$\mathit{Parallel}\ [a] \to a \tag{2.2}$$

$$\mathit{Parallel}\ (x\ @\ [\mathit{Parallel}\ y]\ @\ z) \to \mathit{Parallel}\ (x\ @\ y\ @\ z) \tag{2.3}$$

$$\mathit{Parallel}\ (x\ @\ [\mathit{Empty}]\ @\ y) \to \mathit{Parallel}\ (x\ @\ y) \tag{2.4}$$

The rules (2.1)–(2.3) are obtained directly from the introduction rules of the equivalence $\sim$ by picking a specific direction (see the first three rules in Figure 2.2.1). The rule (2.4) is obtained from a theorem about the equivalence $\sim$, allowing us to drop any *Empty* resource within a *Parallel* one in a single step.

In the rest of this section we detail the normalisation procedure and how it characterises the equivalence relation. We start by defining what it means for a term to be normalised. Next we specify the rewriting relation, using single-direction variants of the equivalence rules, and give an upper bound on the number of rewriting steps that may apply to any given term. Then we define the rewriting step function to implement the rewriting we specified and show that its repeated application to a term eventually reaches the normal form. Finally, we verify that terms having equal normal forms is the same as them being equivalent and define the representative term for a resource to be the normal form in its equivalence class.

2.3.1 Normalised Terms

A resource term is normalised if:

- It is a leaf node (i.e. one of *Empty*, *Anything*, *Res* or *Copyable*), or
- It is a non-parallel internal node (i.e. one of *NonD*, *Executable* or *Repeatable*) and all of its children are normalised, or
- It is a parallel internal node (i.e. *Parallel*) and all of the following hold:
  - All of its children are normalised, and
  - None of its children are empty or parallel resource terms (i.e. one of *Empty* or *Parallel*), and
  - It has at least two children.

We formalise this in Isabelle as the predicate *normalised* through structural recursion on the type of resource terms:

Isabelle Definition 2.3.1 (Normal form predicate for resource terms)

```
primrec normalised :: ('a, 'b) res-term ⇒ bool where
  normalised Empty = True
| normalised Anything = True
| normalised (Res x) = True
| normalised (Copyable x) = True
| normalised (Parallel xs) =
  ( list-all normalised xs ∧
    list-all (λx. ¬ is-Empty x) xs ∧ list-all (λx. ¬ is-Parallel x) xs ∧
    1 < length xs)
| normalised (NonD x y) = (normalised x ∧ normalised y)
| normalised (Executable x y) = (normalised x ∧ normalised y)
| normalised (Repeatable x y) = (normalised x ∧ normalised y)
```

2.3.2 Rewriting Relation

We define the rewriting relation, the congruence closure of the rules (2.1)–(2.4), as the inductive relation *res-term-rewrite*:

Isabelle Definition 2.3.2 (Rewriting relation on resource terms)

```
inductive res-term-rewrite :: ('a, 'b) res-term ⇒ ('a, 'b) res-term ⇒ bool where
  res-term-rewrite Empty Empty
| res-term-rewrite Anything Anything
| res-term-rewrite (Res x) (Res x)
| res-term-rewrite (Copyable x) (Copyable x)
| res-term-rewrite (Parallel []) Empty
| res-term-rewrite (Parallel [a]) a
| res-term-rewrite (Parallel (x @ [Parallel y] @ z)) (Parallel (x @ y @ z))
| res-term-rewrite (Parallel (x @ [Empty] @ z)) (Parallel (x @ z))
| list-all2 res-term-rewrite xs ys ⟹ res-term-rewrite (Parallel xs) (Parallel ys)
| ⟦res-term-rewrite x y; res-term-rewrite u v⟧
  ⟹ res-term-rewrite (NonD x u) (NonD y v)
| ⟦res-term-rewrite x y; res-term-rewrite u v⟧
  ⟹ res-term-rewrite (Executable x u) (Executable y v)
| ⟦res-term-rewrite x y; res-term-rewrite u v⟧
  ⟹ res-term-rewrite (Repeatable x u) (Repeatable y v)
```

Note that this relation is reflexive rather than partial, so its normal forms are fixpoints rather than terminal elements. The rewriting step function (see Section 2.3.4) will have to be total, like all functions defined in Isabelle. By making the rewriting relation reflexive we can have the rewriting step function graph be a subset of this relation, which is useful for reusing proof.

As further assurance, we show that a resource term satisfies the predicate *normalised* if and only if it is a fixpoint of the relation *res-term-rewrite*. So these two definitions agree on what terms are normalised.

Isabelle Lemma 2.3.1 (Normalised terms are fixpoints of rewriting)

```
lemma normalised-is-rewrite-refl:
  normalised x = (∀y. res-term-rewrite x y ⟶ x = y)
```

2.3.3 Rewriting Bound

The rewriting bound expresses the upper limit on how many rewriting steps may be applied to a particular resource term. For this bound we disregard many details of the resource term at hand in order to arrive at a simple definition, which means that even

terms in normal form can have a positive rewriting bound — this is not the *least* upper bound. But there being a finite bound is sufficient to show that normalisation by this rewriting terminates.

We define the bound in Definition 2.3.3 through structural recursion on the type of resource terms with the following major cases:

- If the term is a leaf (i.e. one of *Empty*, *Anything*, *Res* or *Copyable*), then its bound is zero.
- If the term is a non-parallel internal node (i.e. one of *NonD*, *Executable* or *Repeatable*), then its bound is the sum of bounds for its children.
- If the term is a parallel internal node (i.e. *Parallel*), then its bound is the sum of bounds for its children plus its length (for possibly dealing with unwanted children) plus one (for possibly ending up with too few children).

Isabelle Definition 2.3.3 (Upper bound for number of rewrites)

```
primrec res-term-rewrite-bound :: ('a, 'b) res-term ⇒ nat
 where
  res-term-rewrite-bound Empty = 0
| res-term-rewrite-bound Anything = 0
| res-term-rewrite-bound (Res a) = 0
| res-term-rewrite-bound (Copyable x) = 0
| res-term-rewrite-bound (Parallel xs) =
    sum-list (map res-term-rewrite-bound xs) + length xs + 1
| res-term-rewrite-bound (NonD x y) =
    res-term-rewrite-bound x + res-term-rewrite-bound y
| res-term-rewrite-bound (Executable x y) =
    res-term-rewrite-bound x + res-term-rewrite-bound y
| res-term-rewrite-bound (Repeatable x y) =
    res-term-rewrite-bound x + res-term-rewrite-bound y
```

We show two crucial properties of this bound. First, for every resource term not already in normal form this bound is positive.

Isabelle Lemma 2.3.2 (Positive bound for all unnormalised terms)

```
lemma res-term-rewrite-bound-not-normalised:
 ¬ normalised x ⟹ res-term-rewrite-bound x ≠ 0
 by (induct x ; fastforce)
```

Second, the terms related by rewriting to any term have bound at most as high as it does. In other words, rewriting a term does not increase this bound.

Isabelle Lemma 2.3.3 (Rewriting does not increase the bound)

```
lemma res-term-rewrite-non-increase-bound:
  res-term-rewrite x y ⟹ res-term-rewrite-bound y ≤ res-term-rewrite-bound x
  by (induct x y rule: res-term-rewrite.induct)
     (simp-all add: sum-list-mono-list-all2 list-all2-conv-all-nth)
```

2.3.4 Rewriting Step

The rewriting relation in Definition 2.3.2 specifies all possible rewriting paths. When implemented, a specific algorithm must be chosen, which yields a rewriting function. Our rewriting function is given in Definition 2.3.4, followed by a description of the rewriting path it implements.

Isabelle Definition 2.3.4 (Rewriting algorithm for resource terms)

```
primrec step :: 'a res-term ⇒ 'a res-term where
   step Empty = Empty
 | step Anything = Anything
 | step (Res x) = Res x
 | step (Copyable x) = Copyable x
 | step (NonD x y) =
     ( if ¬ normalised x then NonD (step x) y
       else if ¬ normalised y then NonD x (step y)
       else NonD x y)
 | step (Executable x y) =
     ( if ¬ normalised x then Executable (step x) y
       else if ¬ normalised y then Executable x (step y)
       else Executable x y)
 | step (Repeatable x y) =
     ( if ¬ normalised x then Repeatable (step x) y
       else if ¬ normalised y then Repeatable x (step y)
       else Repeatable x y)
 | step (Parallel xs) =
     ( if list-ex (λx. ¬ normalised x) xs then Parallel (map step xs)                (i.)
       else if list-ex is-Parallel xs then Parallel (merge-one-parallel xs)          (ii.)
       else if list-ex is-Empty xs then Parallel (remove-one-empty xs)               (iii.)
       else (case xs of
               [] ⇒ Empty                                                            (iv.)
             | [a] ⇒ a                                                               (v.)
             | - ⇒ Parallel xs))                                                     (vi.)
```

There are two choices when rewriting: the order in which we rewrite children of internal nodes, and the order in which we apply the rewriting rules. For an example of the latter: the term *Parallel* [*Empty*] could be rewritten directly into *Empty* by rule

(2.2), or first into *Parallel* [] by rule (2.4) and only then into *Empty* by rule (2.1). Our algorithm in Definition 2.3.4 approaches these choices as follows:

- For the internal nodes *NonD*, *Executable* and *Repeatable*, always rewrite the first child until it reaches its normal form and only then start rewriting the second child if that one is not also already normalised.
- For the internal node *Parallel* we proceed in phases:
  i. If any child is not normalised, then rewrite all the children (note that rewriting does not change normalised terms by Lemma 2.3.1); otherwise
  ii. If there is some nested *Parallel* node in the children, then merge one up; otherwise
  iii. If there is some *Empty* node in the children, then remove one; otherwise
  iv. If there are no children, then return the term *Empty*; otherwise
  v. If there is exactly one child, then return that term; otherwise
  vi. Do nothing and return the same resource.

We show that the graph of *step* is a sub-relation of the rewriting relation. Thus normalised resource terms are exactly those for which this function acts as identity and the input resource term is always equivalent to the result.

Isabelle Lemma 2.3.4 (Rewriting relation contains rewriting step)

lemma *res-term-rewrite-contains-step*:
res-term-rewrite x (*step* x)

With this more specific formulation of rewriting we can prove that, for any term not already normalised, this *step* function strictly decreases its rewriting bound:

Isabelle Lemma 2.3.5 (Rewriting algorithm decreases bound)

lemma *res-term-rewrite-bound-step-decrease*:
assumes ¬ *normalised* x
shows *res-term-rewrite-bound* (*step* x) < *res-term-rewrite-bound* x

Finally, we show that applying *step* produces a term equivalent to the original. Transitivity will then allow us to repeatedly apply this rewriting and still produce a term equivalent to the original. The proof proceeds rests on noting that the rewriting relation is a one-directional form of the equivalence and *step* is contained by that relation.

Isabelle Lemma 2.3.6 (Rewriting algorithm produces an equivalent term)

lemma *res-term-equiv-step*:
$x \sim$ *step* x
using *res-term-rewrite-contains-step res-term-rewrite-imp-equiv* **by** *blast*

2.3.5 Normalisation

With this rewriting function the normalisation procedure is quite simple: keep applying *step* as long as the resource term is not normalised. We mechanise this as the function *normal-rewr*, and prove that its pattern is complete and consistent, and use the rewriting bound to show that it terminates:

Isabelle Definition 2.3.5 (Normalisation procedure for resource terms)

function *normal-rewr* :: $('a, 'b)$ *res-term* $\Rightarrow$ $('a, 'b)$ *res-term*
where *normal-rewr* $x =$ (*if* *normalised* x *then* x *else* *normal-rewr* (*step* x))
by *pat-completeness auto*

Isabelle Lemma 2.3.7 (Normalisation procedure terminates)

termination *normal-rewr*
using *res-term-rewrite-bound-step-decrease*
by (*relation Wellfounded.measure res-term-rewrite-bound*, *auto*)

We now prove that any resource term is equivalent to the result of its normalisation, proceeding by induction on the normalisation procedure. If the term is already normalised, then the result is the same term and equivalent by reflexivity of $\sim$. Otherwise we know by inductive hypothesis that applying *step* to the term gives us something equivalent to the result of normalisation. And from Lemma 2.3.6 we know that the original term is equivalent to applying *step* to it, so by transitivity of $\sim$ we prove the inductive case as well.

Isabelle Lemma 2.3.8 (Every resource is equivalent to its normalisation)

lemma *res-term-equiv-normal-rewr*:
$x \sim$ *normal-rewr* x

2.3.6 Characterising the Equivalence

In order to characterise the resource term equivalence we need to prove the following statement:

Isabelle Lemma 2.3.9 (Equivalence is equality of normal forms)

lemma *res-term-equiv-is-normal-rewr*:
$x \sim y = (\mathit{normal\text{-}rewr}\ x = \mathit{normal\text{-}rewr}\ y)$

First, the $\Longleftarrow$ direction is simpler. We have already shown that every term is equivalent to the result of applying the rewriting step to it. Because the normalisation is repeated application of the step, by transitivity of the equivalence every term is equivalent to its normalisation. So two terms with equal normal forms can be shown equivalent using transitivity and symmetry of the resource term equivalence.

$$\begin{array}{c} x \sim \mathit{step}\ x \sim \mathit{step}\ (\mathit{step}\ x) \sim \cdots \sim \mathit{normal\text{-}rewr}\ x \\ \| \\ y \sim \mathit{step}\ y \sim \mathit{step}\ (\mathit{step}\ y) \sim \cdots \sim \mathit{normal\text{-}rewr}\ y \end{array}$$

Second, the $\Longrightarrow$ direction is more complex. It relies on showing that equivalent resources are *joinable* by the rewriting function, meaning there is a sequence of rewrite steps from each term to some common (possibly intermediate) form. We formalise this statement by casting the rewriting step in the language of the Abstract Rewriting [85] theory already mechanised for the IsaFoR/CeTA project [89] and available in the Archive of Formal Proofs. This leaves us with the following statement to prove, where *step-irr* is the graph of *step* without its reflexive part[1]:

Isabelle Lemma 2.3.10 (Joinability of equivalent resource term)

lemma *res-term-equiv-joinable*:
$x \sim y \Longrightarrow (x, y) \in \mathit{step\text{-}irr}^{\downarrow}$

We prove this by induction on the resource term equivalence. This means that we have to prove for every rule introducing the equivalence (see Definition 2.2.1) that there exist rewriting paths bringing the two terms to a common form. For the rules that apply to leaf terms the proof is trivial, since the terms are already equal. For rules ensuring congruence on resource combinations we use the path formed by fully normalising both children, at which point the terms are equal because, by inductive hypothesis, the equivalent children are joinable and so their normal forms are equal.

The main complexity of this proof is in the rules that simplify parallel combinations of resource terms. Here we first repeat the pattern of fully normalising the children, since that is what *step* does first (see Definition 2.3.4). Then we consider how the merging of all nested *Parallel* terms, removal of all *Empty* terms and simplification of

[1] The Abstract Rewriting theory defines normal forms as elements that do not rewrite further, instead of our approach where they are elements that rewrite to themselves. The *step-irr* relation bridges this.

the resulting list will proceed in different cases of the rules' parameters. We find it helpful in these cases to first prove a number of lemmas relating the simple approach in *step* of merging or removing one term at a time to a more concise formulation of merging or removing all present occurrences in one pass of the list.

With the joinability proven, we start from Lemma 2.3.8: every term is equivalent to its normal form. So, by transitivity and symmetry, normal forms of equivalent terms are themselves equivalent. Then, by the joinability rule we just showed, we have that these normal forms are joinable. But, because they are normalised terms, we know that each only rewrites to itself. Therefore the form that joins them can only be those normalised terms, and so they must be equal. This concludes the proof of Lemma 2.3.9.

As a result we get a computable characterisation of the resource term equivalence $\sim$. We add Lemma 2.3.9 to the code generator, meaning that Isabelle/HOL is no longer blocked from generating code for anything involving resources. This characterisation is also important to how we translate resources into linear logic in Section 4.3.

2.3.7 Representative Term

Now that the resource term normalisation is verified, we can use it to define a representative term for every resource. While every resource is an equivalence class of terms, there is exactly one normalised term among them. We denote the representative of x as *of-resource* x. Having such a representative is useful, for instance, for visualising the resource.

The representative can be constructed by applying the rewriting normalisation procedure to any term in the class. As with the resource constructors (see Section 2.1), this definition is facilitated by the Lifting package [46].

Thus, we define *of-resource* to be the normalisation procedure but lifted to have resources as its domain. This requires that equivalent terms have equal normal forms, which we proved as part of Lemma 2.3.9 in the previous section.

Isabelle Definition 2.3.6 (Representative term for resources)

lift-definition *of-resource* :: *'a resource* ⇒ *'a res-term* **is** *normal-rewr*
by (*rule res-term-equiv-normal-rewr*)

The resulting function is equivalent to the following, where *Rep-resource* x is the equivalence class representing x and *SOME a. P a* chooses an arbitrary element that satisfies the predicate P:

$$\textit{of-resource}\ x = \textit{normal-rewr}\ (\textit{SOME}\ a.\ a \in \textit{Rep-resource}\ x)$$

2.4 Resource Type is a Bounded Natural Functor

When modelling processes in a certain context, the resources are built from available resource atoms. It is then useful to have a method for systematically changing the content of a resource (the resource atoms it contains) without changing its shape, translating the resource from one context to another. In general, such a method is called a *mapper* [32].

Suppose, for instance, that we were using a specific currency in our vending machine example, say Czech crowns. Then we may want to *map* a process from that domain to one using Polish złoty instead. We would do that by taking every resource in the process and systematically applying the currency conversion to all atoms the resource contains. The mapper generalises this operation for any function on resource atoms. For a further discussion of the resource mapper and its use in transforming processes, see Section 3.4.1.

For resource terms, Isabelle/HOL automatically defines the mapper and proves its properties (e.g. that it respects the identity function and commutes with function composition). This is because every inductive datatype in Isabelle/HOL is a *bounded natural functor* (BNF), a structure for compositional construction of datatypes in HOL presented by Traytel et al. [90] and integrated in Isabelle/HOL by Blanchette et al. [14]. It therefore remains for us to lift this term-level mapper to the quotient and transfer its properties.

Fortunately, lifting the BNF structure from a concrete type to a quotient has been automated by Fürer et al. [32] who reduce this task to two proof obligations concerning the equivalence relation being used and parts of the BNF structure of the concrete type (resource terms in our case). Once these obligations are proven, the constants required for the BNF structure of the quotient and their properties are automatically derived.

The first obligation concerns the relator, an extension of the mapper to relations on the content instead of functions. It ensures that the generated relator will commute with relation composition. While its proof would usually be quite difficult, Fürer et al. describe how this condition can be proven using a confluent rewriting relation whose equivalence closure contains the equivalence relation we used to define the quotient type. In our case, the resource term normalisation procedure described in Section 2.3 gives us exactly such a relation.

The second obligation concerns the setter, which gathers the content into a set while discarding the shape. It ensures that the generated setter will be a natural trans-

formation, that is applying the setter after mapping a function is the same as using the setter first and then applying that function to every element of its result. In our case this condition can be proven using structural induction on resource terms and Isabelle's automated methods.

As a result we get the BNF constants (mapper, relator and setter) and properties, all already integrated with the automated tools within Isabelle.

2.5 Conclusion

In this chapter we described our mechanisation of resources in Isabelle/HOL. We start with a datatype of resource terms and define an equivalence relation on them to express different syntactic forms describing the same resource. We then obtain resources as the quotient of their terms by this relation. To provide a decision procedure for the equivalence, we mechanise a rewriting system for terms and show that equality of normal forms is the same as equivalence of the terms. We use that rewriting system to automatically lift some of the structure from resource terms to resources themselves.

Note that the set of resource term equivalences we use is not exhaustive. For instance, in Section 6.5 we add three more. As such, we see the identification of further practical equivalences and their integration into the resource algebra as a continual part of future work. Fortunately, the automation available in Isabelle removes much of the friction in that process.

We make use of these resources in the next chapter to describe the inputs and outputs of processes. The work done internally by the resource algebra, through the term equivalence relation, makes our mechanisation of processes simpler. For instance, we automatically get that parallel composition of processes with no inputs also has no input itself. And the decision procedure for resource term equivalence, apart from aiding in proofs, allows Isabelle to automatically generate executable code for process compositions, which enables us to use them outside of the proof assistant.

Chapter 3

Process Compositions

The role of process compositions in our framework is to describe how larger processes are built from smaller parts using a handful of simple operations. With our focus on the inputs and outputs of processes, a composition describes how the outputs of some actions are used to fulfil the inputs of other ones. A central concept is that of *valid* compositions, which uses simple rules to ensure the connections between action outputs and inputs are correct. For instance, it ensures that every input resource comes from a known origin and that linear resources are used exactly once.

We start in Section 3.1 with our formalisation of the process compositions themselves. This revolves around a datatype representing them as a range of basic actions combined with four composition operations. We describe each basic action and composition in that datatype, and the resources describing their inputs and outputs.

On this datatype we then formally define validity of process compositions in Section 3.2, and illustrate what malformed compositions it prevents. In Chapter 4 we connect this to the idea of linearity by relating valid compositions to well-formed deductions in linear logic. And in Section 5.6 we characterise the allowed connections between actions and show that valid compositions obey them. In Section 7.4 we formalise a modelling domain whereby validity of compositions in it ensures the processes are free from bottlenecks.

In Section 3.3 we describe our visualisation of process compositions through diagrams. We favour this visualisation over purely algebraic composition expressions wherever possible, because it abstracts a number of bookkeeping aspects and focuses on the connections between actions. This often renders the diagrams much more readable. The process diagrams used in this thesis are drawn by a Haskell program that uses code generated from our mechanisation, ensuring their faithfulness to the theory.

One advantage of our mechanised process compositions is that we can define and formally verify transformations of compositions. In Section 3.4 we describe three such transformations: mapping resource atoms to move to a different modelling domain, refining resource atoms with arbitrarily complex resources and refining processes by substituting whole processes for primitive actions. We demonstrate their use by refining a process model in Section 7.1.

Running example. In this chapter we continue to use our running example from Chapter 2, a simple vending machine. Recall that this machine can accept cash, dispense one kind of drink at a fixed cost and return change. See Chapter 7 for more involved examples of process compositions.

3.1 Process Composition Datatype

We formalise compositions describing how complex processes are built as trees of composition actions with simple actions at their leafs. Simple actions are split into two groups: primitive actions represent actions we assert as part of the modelling domain, while resource actions allow us to manipulate the form of resources. Composition actions take one or more processes to build a larger one, reflecting simple instructions such as "do one then the other" or "do both of these".

We first introduce our mechanisation of process compositions in Isabelle/HOL as a datatype along with the definition of their inputs and outputs. Then, in the rest of this section, we describe in more detail the nodes of the process composition trees. In Section 3.2 we then discuss the validity conditions on compositions.

We mechanise process compositions as the datatype $('a, 'b, 'l, 'm)$ *process*. The type variables $'a$ and $'b$ represent linear and copyable resource atoms respectively, while $'l$ represents labels of primitive actions (e.g. using string literals) and $'m$ represents any other metadata attached to primitive actions (e.g. cost to perform). The full datatype is shown in Definition 3.1.1. For the input and output resources we define primitively recursive functions *input* and *output*, whose full definition is show in Definition 3.1.2.

Note that, for brevity, we may also use the notation $P\colon x \to y$ to say that process P has input resource x and output resource y. Recall also that in Definition 2.2.4 we defined $x \odot y$ as infix syntax for *Parallel* $[x, y]$. We have for instance *Swap* $a\ b\colon a \odot b \to b \odot a$ (see Section 3.1.3).

Isabelle Definition 3.1.1 (Datatype of process compositions)

datatype $('a, 'b, 'l, 'm)$ *process* =

Primitive	$(('a, 'b)$ *resource*$)$ $(('a, 'b)$ *resource*$)$ $('l)$ $('m)$	Primitive Action
| *Seq*	$(('a, 'b, 'l, 'm)$ *process*$)$ $(('a, 'b, 'l, 'm)$ *process*$)$	Composition Operations
| *Par*	$(('a, 'b, 'l, 'm)$ *process*$)$ $(('a, 'b, 'l, 'm)$ *process*$)$	Composition Operations
| *Opt*	$(('a, 'b, 'l, 'm)$ *process*$)$ $(('a, 'b, 'l, 'm)$ *process*$)$	Composition Operations
| *Represent*	$(('a, 'b, 'l, 'm)$ *process*$)$	Composition Operations
| *Identity*	$(('a, 'b)$ *resource*$)$	Resource Actions
| *Swap*	$(('a, 'b)$ *resource*$)$ $(('a, 'b)$ *resource*$)$	Resource Actions
| *InjectL*	$(('a, 'b)$ *resource*$)$ $(('a, 'b)$ *resource*$)$	Resource Actions
| *InjectR*	$(('a, 'b)$ *resource*$)$ $(('a, 'b)$ *resource*$)$	Resource Actions
| *OptDistrIn*	$(('a, 'b)$ *resource*$)$ $(('a, 'b)$ *resource*$)$ $(('a, 'b)$ *resource*$)$	Resource Actions
| *OptDistrOut*	$(('a, 'b)$ *resource*$)$ $(('a, 'b)$ *resource*$)$ $(('a, 'b)$ *resource*$)$	Resource Actions
| *Duplicate*	$('b)$	Resource Actions
| *Erase*	$('b)$	Resource Actions
| *Apply*	$(('a, 'b)$ *resource*$)$ $(('a, 'b)$ *resource*$)$	Resource Actions
| *Repeat*	$(('a, 'b)$ *resource*$)$ $(('a, 'b)$ *resource*$)$	Resource Actions
| *Close*	$(('a, 'b)$ *resource*$)$ $(('a, 'b)$ *resource*$)$	Resource Actions
| *Once*	$(('a, 'b)$ *resource*$)$ $(('a, 'b)$ *resource*$)$	Resource Actions
| *Forget*	$(('a, 'b)$ *resource*$)$	Resource Actions

Isabelle Definition 3.1.2 (Input and output resources for composition trees)

```
primrec input :: ('a, 'b, 'l, 'm) process ⇒ ('a, 'b) resource
where
  input (Primitive ins outs l m) = ins
| input (Seq p q) = input p
| input (Par p q) = input p ⊙ input q
| input (Opt p q) =
    NonD (input p) (input q)
| input (Represent p) =
    Empty
| input (Identity a) = a
| input (Swap a b) = a ⊙ b
| input (InjectL a b) = a
| input (InjectR a b) = b
| input (OptDistrIn a b c) =
    a ⊙ (NonD b c)
| input (OptDistrOut a b c) =
    NonD (a ⊙ b) (a ⊙ c)
| input (Duplicate a) =
    Copyable a
| input (Erase a) = Copyable a
| input (Apply a b) =
    a ⊙ (Executable a b)
| input (Repeat a b) =
    Repeatable a b
| input (Close a b) = Repeatable a b
| input (Once a b) = Repeatable a b
| input (Forget a) = a
```

```
primrec output :: ('a, 'b, 'l, 'm) process ⇒ ('a, 'b) resource
where
  output (Primitive ins outs l m) = outs
| output (Seq p q) = output q
| output (Par p q) = output p ⊙ output q
| output (Opt p q) =
    output p
| output (Represent p) =
    Repeatable (input p) (output p)
| output (Identity a) = a
| output (Swap a b) = b ⊙ a
| output (InjectL a b) = NonD a b
| output (InjectR a b) = NonD a b
| output (OptDistrIn a b c) =
    NonD (a ⊙ b) (a ⊙ c)
| output (OptDistrOut a b c) =
    a ⊙ (NonD b c)
| output (Duplicate a) =
    Copyable a ⊙ Copyable a
| output (Erase a) = Empty
| output (Apply a b) =
    b
| output (Repeat a b) =
    Repeatable a b ⊙ Repeatable a b
| output (Close a b) = Empty
| output (Once a b) = Executable a b
| output (Forget a) = Anything
```

3.1.1 Primitive Actions

We start with *primitive actions* which depend on the domain we are modelling: they represent operations we can perform in the domain. For instance, if the domain has a notion of resources with locations, then there will be some kind of movement action which may move an object freely between locations or be constrained to edges of some graph. We consider these to be assumptions of the model.

In Isabelle/HOL, we represent a primitive action by *Primitive in out lab meta*, where *in* and *out* are its input and output resources respectively, *lab* is its label and *meta* is any other associated metadata.

The label serves to distinguish actions that may have equal inputs and outputs but different meaning, such as two modes of moving an object between locations with different costs and speeds (e.g. walking *vs.* running). The term "label" is used because it is often set to a printable type, such as a string, which we can then use as a label in visualisations such as the process diagrams discussed in Section 3.3.

The metadata can carry any further information we may wish to associate with the primitive actions, such as cost of execution or parameters for implementation. See for instance our model of manufacturing in Section 7.4 where action metadata carries the information needed to implement the process in the simulation environment, such as what needs to be manufactured, at what rate and how the products should be moved.

Both the label and metadata are taken from arbitrary types (the unconstrained type variables $'l$ and $'m$ of $('a, 'b, 'l, 'm)\ process$) so they can carry any kind of complex information.

As these primitive actions correspond to the assumptions about the modelling domain on which the composition relies, we make it convenient to collect the assumptions by defining the function *primitives* with the following signature:

$$\begin{aligned} &primitives :: ('a, 'b, 'l, 'm)\ process \\ &\qquad \Rightarrow (('a, 'b)\ resource \times ('a, 'b)\ resource \times 'l \times 'm)\ list \end{aligned}$$

It gathers the parameters of every *Primitive* node, returns the empty list for every other leaf of the composition and gathers children's results for every internal node using list append.

Example. In our running example, the vending machine, we have three primitive actions: paying into the machine, getting a drink and getting change. As linear resource atoms we use those introduced in Section 2.1: *Cash* for an amount of currency, *Drink* for a single drink and *Machine* for a vending machine with some amount in credit.

We use string literals as labels and assign no metadata to these actions (by using the constant () of type *unit*).

Paying money into a vending machine requires the machine, which can hold an amount of existing credit, and the cash being paid in. It produces a machine with the combined amount in credit. We define the action as a function of the two variables:

Isabelle Definition 3.1.3 (Adding to credit of a vending machine)

```
definition add-to-credit credit cash =
  Primitive (Res (Machine credit) ⊙ Res (Cash cash))
            (Res (Machine (credit + cash)))
            STR "Add to credit" ()
```

Returning funds from a vending machine only requires a machine and it produces a machine with zero credit left and the relevant amount of cash dispensed:

Isabelle Definition 3.1.4 (Returning credit as cash)

```
definition refund credit =
  Primitive (Res (Machine credit))
            (Res (Machine 0) ⊙ Res (Cash credit))
            STR "Get change" ()
```

Getting a drink requires a machine with credit of at least the price of the drink and produces a machine with decreased credit and the drink. We define this action as a partial function of the price and credit:

Isabelle Definition 3.1.5 (Buying a drink)

```
definition get-drink price credit = (if price ≤ credit
  then Primitive (Res (Machine credit))
                 (Res (Machine (credit − price)) ⊙ Res Drink)
                 STR "Get drink" ()
  else undefined)
```

Note that we separate the actions of getting a drink and returning change, while often vending machines return change automatically along with the purchase. Our formulation separates these two concerns, which allows returning change and cancelling to both be handled by the *refund* action.

3.1.2 Composition Operators

Next we have four ways of creating processes from one or two simpler ones, allowing us to inductively describe how to orchestrate the primitive actions:

Sequential composition — *Seq P Q: input P → output Q*

First execute P and then use its output to execute Q.

Parallel composition — *Par P Q: input P ⊙ input Q → output P ⊙ output Q*

Execute P and Q in parallel, combining their inputs into one parallel resource and doing the same for their outputs.

Optional composition — *Opt P Q: NonD (input P) (input Q) → output P*

Given two processes P and Q with the same output, take as input the non-deterministic combination of their inputs. Execute exactly one of them based on the branch of the input that is supplied at runtime, producing their common output no matter which branch is taken. Thus, this operation is used to eliminate non-determinism from resources by reacting to it.

Representation — *Represent P: Empty → Repeatable (input P) (output P)*

Introduce the executable resource representing P, which could be an arbitrarily complex composition in its own right. This requires no input, analogously to a constant being viewed as a nullary function.

Note that the output could have been defined using *Executable*, meaning it would allow only a single use. But based on experience with modelling processes we decided to use *Repeatable* instead, meaning we do not have to know ahead of time how many times we intend to use the representation.

The simplicity of these composition operations may seem restrictive. For instance, what if we wish to use only part of the output of P as input to Q, essentially using Q to further process some objects produced by P while keeping others untouched? In such a case our composition operations require us to state this explicitly so there is no ambiguity: do P and then do Q in parallel with "doing nothing" for the remainder of P's outputs. This is the way our process compositions approach the *frame problem* [57]: the lack of change must be stated. The way in which we express such lack of change is through resource actions, discussed in the next section.

Example. Perhaps the simplest process composition in our running example is paying money into a vending machine and then getting a drink. For example we can pay 10 into an empty machine and then get a drink costing 5 from the machine which then has 5 left as credit:

$$\begin{gathered} \mathit{Seq}\ (\mathit{add\text{-}to\text{-}credit}\ 0\ 10)\ (\mathit{get\text{-}drink}\ 5\ 10): \\ \mathit{Res}\ (\mathit{Machine}\ 0) \odot \mathit{Res}\ (\mathit{Cash}\ 10) \to \mathit{Res}\ (\mathit{Machine}\ 5) \odot \mathit{Res}\ \mathit{Drink} \end{gathered}$$

3.1.3 Resource Actions

Expressing concepts such as "doing nothing" is where *resource actions* come in. These are similar to primitive actions but, instead of expressing an assumption about some particular domain, they express the ways that we can always transform resources.

As an inspiration for this range of actions we use theorems of linear logic: if an action can be argued in linear logic without making any assumption about the domain, then we can consider it as always doable with resources. However, not all theorems of linear logic have counterparts in process compositions because we do not make use of all of its operators — see Section 4.3 for our translation of resources into linear logic propositions. Note also that to argue for some of these actions one needs more than one inference rule, such as with *OptDistrIn*. We aim for convenience of expressing process compositions more than for atomicity with respect to the logic.

The resource actions along with their inputs and outputs are as follows (see also Definitions 3.1.1 and 3.1.2):

Doing nothing — *Identity* a*:* $a \to a$

Take any resource and return it unchanged. For instance, to keep a resource aside while doing something with other resources.

Reordering parallel resources — *Swap* $a\ b$*:* $a \odot b \to b \odot a$

Swap the order of any two parallel resources. This allows us to compose processes when their interface has the same resources but in a different order, while retaining information about *how* they were reordered.

Injections — *InjectL* $a\ b$*:* $a \to$ *NonD* $a\ b$ **and** *InjectR* $a\ b$*:* $b \to$ *NonD* $a\ b$

Take one resource to its non-deterministic combination with another. For instance, if a process Q has input *NonD* (*Res Tin*) (*Res Copper*) and we can deterministically obtain the atom *Tin* from another process P, then *InjectL* (*Res Tin*) (*Res Copper*) is the bridge that connects from P to Q.

Furthermore, injections in combination with *Opt* can also be used to optionally compose[1] any two processes P*:* $x \to a$ and Q*:* $y \to b$ to one of the form *NonD* $x\ y \to$ *NonD* $a\ b$.

Inward distributing — *OptDistrIn* $a\ x\ y$*:* $a \odot$ *NonD* $x\ y \to$ *NonD* $(a \odot x)\ (a \odot y)$

Distribute a parallel resource into both branches of a non-deterministic one. This

[1] Specifically: *Opt* (*Seq* P (*InjectL* (*output* P) (*output* Q))) (*Seq* Q (*InjectR* (*output* P) (*output* Q)))

is useful when a is deterministically available but needed to resolve the branches of *NonD* x y. For instance, if always available tools are needed to repair a machine that may break during its use.

Outward distributing — *OptDistrOut* a b c*: NonD* $(a \odot x)$ $(a \odot y) \to a \odot$ *NonD* x y

Distribute a parallel resource out of both branches of a non-deterministic one. This is useful when processing some non-deterministic resource produces a partially deterministic result. For instance, if using a machine may result in a successful or failed product but always also outputs the machine itself. (Note that this action could be defined as a composition of others[2], but we keep it as a primitive for convenience and symmetry.)

Duplicating copyable — *Duplicate* a*: Copyable* $a \to$ *Copyable* $a \odot$ *Copyable* a

Duplicate a copyable resource into two copies. For instance, to use a single password input for multiple actions requiring it.

Erasing copyable — *Erase* a*: Copyable* $a \to$ *Empty*

Discard a copyable resource, producing nothing. For instance, when an action produces a digital alert to which we do not wish to react.

Higher-order application — *Apply* a b*:* $a \odot$ *Executable* a $b \to b$

Take any resource and an executable resource with matching input, evaluate the process represented by the latter and produce its output. If we use a process to prepare an *Executable* resource, then this allows us to execute it.

Duplicating repeatable — *Repeat* a b*: Repeatable* a $b \to$ *Repeatable* a $b \odot$ *Repeatable* a b

Duplicate a repeatably executable resource into two copies, allowing for multiple parallel executions.

Erasing repeatable — *Close* a b*: Repeatable* a $b \to$ *Empty*

Discard a repeatably executable resource, producing nothing. This captures that being executable any number of times includes zero times.

Dropping repeatability — *Once* a b*: Repeatable* a $b \to$ *Executable* a b

Turn a repeatably executable resource into a matching single-use executable resource, which can then be executed with *Apply*.

[2] Specifically: *Opt* (*Par* (*Identity* a) (*InjectL* x y)) (*Par* (*Identity* a) (*InjectR* x y))

Forgetting resource details — *Forget a: a → Anything*

Forget all information about some resource, producing *Anything*. For instance: if we have a drawer with socks of two colours, then we can specify the output of a process that finds two matching socks more concisely as: *NonD* (*Black* ⊙ *Black* ⊙ *Anything*) (*White* ⊙ *White* ⊙ *Anything*) (for more on this sock example see Section 7.2 and Dixon et al. [25]).

Note that what is forgotten are details about the resource and not its presence, only copyable resources can be erased.

Note how none of these actions interact with the internal state of resource atoms, only changing how they are arranged. Indeed, interacting with the internal state of atoms is impossible for these actions, since they are defined regardless of the resource atoms we choose to use. This includes the case where atoms have no internal state to interact with.

Example. With resource actions we can extend our previous vending machine process to also get change after buying the drink. We do this by following the previous composition with refunding in parallel with identity on the drink, defined formally as follows and visualised using process diagrams (see Section 3.3) in Figure 3.1:

Seq (*Seq* (*add-to-credit 0 10*) (*drink 5 10*)) (*Par* (*refund 5*) (*Identity* (*Res Drink*)))

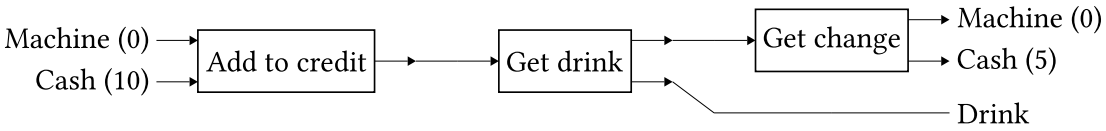


Figure 3.1: Process diagram of paying into a vending machine, getting a drink and the remaining change

3.2 Valid Compositions

When building compositions of processes, we wish to ensure that we are doing so sensibly, in particular that we are manipulating resources correctly. We call such process compositions *valid*.

One approach to ensuring correctness would be to perform each step as a deduction in linear logic, which is generally accepted to produce correct processes (see discussion

of proofs-as-processes in Section 1.2). However, this would limit the concepts we can express in compositions and their validity to only what we can express within linear logic.

Instead, we set up rules purely in the language of resources about what compositions "make sense". In Chapter 4 we then show that the mechanical translation of any composition into a linear logic deduction will be well-formed if the composition satisfies these rules. On the process composition side, this approach yields conditions simpler than the full rules of linear logic. And the decoupling allows process compositions and their validity to involve concepts outside of linear logic, for example probabilities as we discuss in Chapter 6.

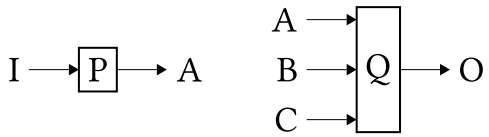
In our mechanisation of composition validity, shown in Definition 3.2.1, we consider primitive actions and all resource actions to be valid. For all composition operations we require at least that their child processes be valid, and in the sequential and optional composition cases there are further conditions:

- In sequential composition *Seq P Q*, the output of the first process *P* must be equal to the input of the second process *Q*.
- In optional composition *Opt P Q* the outputs of both processes must be equal, so that no matter the branch actually taken we know what the output will be.

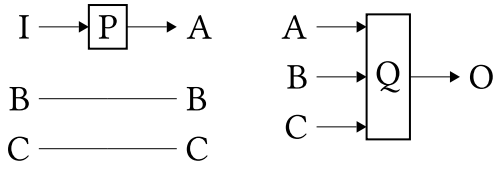
Isabelle Definition 3.2.1 (Validity of process compositions)

```
primrec valid :: ('a, 'b, 'l, 'm) process ⇒ bool where
  valid (Primitive ins outs l m) = True
| valid (Seq p q) = (valid p ∧ valid q ∧ output p = input q)
| valid (Par p q) = (valid p ∧ valid q)
| valid (Opt p q) = (valid p ∧ valid q ∧ output p = output q)
| valid (Identity a) = True
| valid (Swap a b) = True
| valid (InjectL a b) = True
| valid (InjectR a b) = True
| valid (OptDistrIn a b c) = True
| valid (OptDistrOut a b c) = True
| valid (Duplicate a) = True
| valid (Erase a) = True
| valid (Represent p) = valid p
| valid (Apply a b) = True
| valid (Repeat a b) = True
| valid (Close a b) = True
| valid (Once a b) = True
| valid (Forget a) = True
```

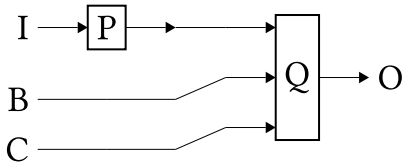
As an example consider processes P: $Res\ I \to Res\ A$ and Q: $Res\ A \odot Res\ B \odot Res\ C \to Res\ O$. The sequential composition of P then Q is not valid, because the output of P is not the input of Q. However, by composing P in parallel with identity processes on $Res\ B$ and $Res\ C$ we can "fill out" its output to make the sequential composition valid. This situation is visualised using process diagrams (see Section 3.3) in Figure 3.2.



(a) Invalid sequential composition



(b) Valid sequential composition



(c) Resulting process

Figure 3.2: Example of how sequential composition can fail to be valid, visualised through process diagrams (and treating P and Q as primitive actions)

Validity ensures that resources are only changed as explicitly stated by the composition. Furthermore, composition operations and resource actions at most manipulate the structure of resource combinations and not the contents of atoms, to which they do not have access. Thus, the only way to change the contents of resource atoms is via primitive actions, which are all assumptions about the domain.

As a result, the information within resource atoms can interact with this notion of composition validity in interesting ways. For instance, if the atoms are located at nodes of a graph and we only use movement actions corresponding to edges of the graph, then validity of compositions in such a domain ensures all resources move along valid paths of the graph. As a more involved example, see our model of manufacturing in Section 7.4 where validity ensures the whole manufacturing process is balanced (i.e. contains no bottlenecks).

3.3 Process Diagrams

We find it helpful to visualise process compositions using diagrams, such as those in Figure 3.2. These can more concisely communicate processes compared to their algebraic definition, whose readability quickly declines with process size. In this section we give an overview of the process diagrams used in this thesis.

In process diagrams we visualise individual actions as shapes (e.g. primitive actions as labelled boxes), with compositions laying those shapes out and connecting them. These connections represent the flow of resources between actions, the core information in process compositions.

The advantage of diagrams comes from how they more effectively express aspects of the compositions. For instance, recall that parallel combination of resources represents their simultaneous presence; in a diagram we express this by visualising parallel resources as parallel connections. To aid in this, we define the function *parallel-parts* to split any resource into its simplest parallel constituents. Following the pattern from Section 2.2, we define it first for resource terms and then lift it to resources:

Isabelle Definition 3.3.1 (Parallel parts of a resource term)

```
primrec parallel-parts :: ('a, 'b) res-term ⇒ ('a, 'b) res-term list
 where
  parallel-parts Empty = []
| parallel-parts Anything = [Anything]
| parallel-parts (Res a) = [Res a]
| parallel-parts (Copyable a) = [Copyable a]
| parallel-parts (Parallel xs) = concat (map parallel-parts xs)
| parallel-parts (NonD a b) = [NonD a b]
| parallel-parts (Executable a b) = [Executable a b]
| parallel-parts (Repeatable a b) = [Repeatable a b]
```

Isabelle Definition 3.3.2 (Parallel parts of a resource)

```
lift-definition parallel-parts :: ('a, 'b) resource ⇒ ('a, 'b) resource list
  is ResTerm.parallel-parts
 by (simp add: equiv-parallel-parts)
```

Our process diagrams are made to correspond mechanically to the process composition syntax: individual actions are represented by concrete diagrams and composition operations are represented by operations on the diagrams of their children. For instance, sequential composition is represented by horizontally juxtaposing diagrams of the two children and then connecting the output points of the first diagram to the input points of the second diagram.

Because it is not possible to draw images directly in Isabelle/HOL, we implement a program to draw these diagrams on top of code automatically generated from our mechanisation. (See Section 1.3.3 for a brief introduction to code generation facilities in Isabelle/HOL.) We do so in Haskell, both because it is one of the languages Isabelle can generate and because it has the Diagrams library [97], which suits our purpose.

Note that this means this diagram construction cannot be used as part of logical statements or proofs, nor can we verify properties of the construction itself. In contrast, in Chapter 5 we discuss a fully formal mapping from process compositions to port graphs, which we then use in proofs about processes.

The crucial point of our implementation rests in uniquely identifying resource occurrences in a process composition, because this allows us to name points in the diagram and then easily look them up and connect them. We discuss our approach to this in Section 3.3.1, followed by a discussion of the actual diagrams in Section 3.3.2. In Chapter 5 we make further use of these unique identifiers (see Section 5.3.1 and Section 5.4.1).

3.3.1 Identifying Resource Occurrences

Connections in the process diagrams are shown as *wires* going from action outputs into action inputs. To best draw these wires, we can assign names to individual points in the diagrams and later use those names when placing the connecting line. This means that we require unique names for every occurrence of a resource in the process composition.

Intuitively, an occurrence of a resource in a process composition is always as a parallel part of an input or output of some process. As such, we can uniquely identify a resource occurrence by identifying: the process, whether it is in the input or output and which parallel part it is. And, because process compositions are trees, every subprocess is uniquely identified by the path through that tree. These are exactly the ingredients we use in the following mechanisation.

We first define the components of a process composition tree path, a token for each direction we could go at every internal node:

Isabelle Definition 3.3.3 (Components for composition tree paths)
datatype *process-inner* = *SeqL* | *SeqR* | *ParL* | *ParR* | *OptL* | *OptR* | *Rep*

A sequence of these components (*process-inner list* in Isabelle/HOL) uniquely identifies a subtree of a process composition. Note that not every sequence makes

sense for every composition: for instance, a sequence longer than a composition tree is deep cannot make sense for that tree. However, whether the sequence makes sense can be easily checked by attempting to traverse the tree while following that sequence, and if it does then the corresponding subprocess can be extracted.

We call the occurrence of a resource in the input or output of a process a *port*, and when combined with a process identifier we call it a *qualified port*. (This terminology alludes to *port graphs*, which serve as an inspiration for our process diagrams and are more extensively discussed in Chapter 5.) Each port is on a specific side (input or output) of a process, has an index within that side and is labelled with the resource it carries.

For the sake of generality, we use type parameters for the range of sides on which the ports can appear, their labels and the atoms qualifying them, as shown in Definition 3.3.4. In Section 5.3.1 we make further use of these ports in our general mechanisation of port graphs. For process diagrams we use two sides, input and output, label the ports with resources and qualify them with composition tree paths. We use these again in Section 5.4.1 when specialising port graphs to represent process compositions.

Isabelle Definition 3.3.4 (Datatypes for sides, ports and qualified ports)

datatype *process-side* = *In* | *Out*
datatype ('*s*, '*a*) *port* = *Port* (*side*: '*s*) (*index*: *nat*) (*label*: '*a*)
datatype ('*s*, '*a*, '*p*) *qualified-port* = *QPort* (*port*: ('*s*, '*a*) *port*) (*name*: '*p list*)

We define convenience functions to construct a list of ports from the parallel parts of a single resource, given a starting index and a common side:

Isabelle Definition 3.3.5 (Ports from lists and parallel parts)

fun *listPorts* :: *nat* ⇒ '*s* ⇒ '*a list* ⇒ ('*s*, '*a*) *port list*
 where *listPorts n s as* = *map* (λ(*x*, *y*). *Port s x y*) (*zip* [*n*..< *n* + *length as*] *as*)
fun *parallelPorts* :: *nat* ⇒ '*s* ⇒ ('*a*, '*b*) *resource* ⇒ ('*s*, ('*a*, '*b*) *resource*) *port list*
 where *parallelPorts n s a* = *listPorts n s* (*parallel-parts a*)

For instance, consider a primitive action with the input resource *Res A* ⊙ *Res B*. The ports corresponding to this resource are generated as follows:

$$\begin{aligned} &\mathit{parallelPorts}\ 0\ \mathit{In}\ (\mathit{Res}\ A \odot \mathit{Res}\ B) \\ &= \mathit{listPorts}\ 0\ \mathit{In}\ ([\mathit{Res}\ A, \mathit{Res}\ B]) \\ &= [\mathit{Port}\ 0\ \mathit{In}\ (\mathit{Res}\ A), \mathit{Port}\ 1\ \mathit{In}\ (\mathit{Res}\ B)] \end{aligned}$$

and, if it is the first child of sequential composition, they are uniquely identified as:

$$[\mathit{QPort}\ [\mathit{SeqL}]\ (\mathit{Port}\ 0\ \mathit{In}\ (\mathit{Res}\ A)), \mathit{QPort}\ [\mathit{SeqL}]\ (\mathit{Port}\ 1\ \mathit{In}\ (\mathit{Res}\ B))]$$

3.3.2 Drawing the Diagrams

With a way to uniquely identify resource occurrences, the rest of the diagram construction is a matter of drawing the relevant shapes and connecting points using their names. This part is done fully outside the proof assistant using the generated Haskell code.

We define the diagrams for process compositions by pattern matching on the top constructor. For each primitive action or resource action we construct a concrete diagram using its parameters (e.g. the two resources *a* and *b* being swapped in *Swap a b*). For each composition operation we first recursively construct the diagrams for all of its children and then combine those in a way specific to that operation.

Note that in our diagrams we arrange sequential composition from left to right and parallel composition from top to bottom. This is merely a presentation choice and the implementation could easily be changed to alter these directions.

Note also that, to make the diagrams more compact, we simplify the printing of resources with respect to the actual formal terms. When constructing the diagram we require that both the linear and copyable resource atoms be strings so they can be printed. This can be conveniently satisfied using the resource mapping in Section 3.4.1. To indicate the different kinds of resources we use a variety of symbols inspired by both our Isabelle syntax (specifically $\odot$) as well as the syntax of their ILL translation (see Section 4.3). However, the font used by the diagram drawing library is limited and does not contain the $\odot$ or $\multimap$ symbols. As such, we use the following notation, with examples shown in figures throughout the rest of this section:

- $A \cdot B$ to indicate parallel resources,
- $A \oplus B$ to indicate non-deterministic resources,
- $A \to B$ to indicate executable resources,
- $!A$ to indicate copyable atoms,
- $![A \to B]$ to indicate repeatably executable resources,
- 1 to indicate the *Empty* resource, and
- $\top$ to indicate the *Anything* resource.

In the remainder of this section, we show illustrative examples of diagrams for resource actions and for composition operations, and describe how they relate to the formal objects.

A primitive action, which represents an assumption about what can be done in the domain, is visualised by a box with inputs on its left, outputs on its right and the action label in the centre. This is intended to indicate that a resource goes in, another comes out and we have no further detail about what happens inside to make that transformation. An example with two parallel resource atoms as input and one as output is shown in Figure 3.3.

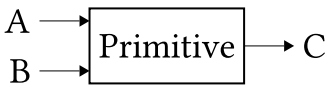
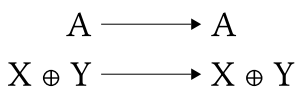


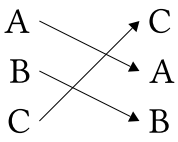
Figure 3.3: *Primitive* (*Res A* ⊙ *Res B*) (*Res C*) *STR "Primitive"* ()

The identity and swap resource actions only interact with the parallel combinations of resources, so we only need wires to visualise them. For the identity action this means simply a wire going from left to right for each parallel part. For the swap action these wires go diagonally, crossing over in two groups to swap the order of the wires. These are shown on examples in Figure 3.4.

Note that the wires representing the identity action can be shortened to save space in larger diagrams, all the way down to points. This loses no information, since the action does nothing with those resources. In Section 5.4.3 we come back to this idea, using port graphs to formalise how the identity process is the unit of sequential composition (Lemma 5.4.4).



(a) *Identity* (*Res A* ⊙ *NonD* (*Res X*) (*Res Y*))

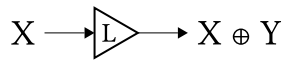


(b) *Swap* (*Res A* ⊙ *Res B*) (*Res C*)

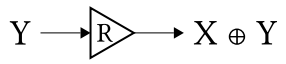
Figure 3.4: Illustrative diagrams of simple resource actions

The injection resource actions interact with non-deterministic resource combinations, so cannot be shown simply with wires. As such, we visualise them using a triangle, since that can be adapted to any number of input wires coming into its base and there will always be one output wire coming from its point. Inside the triangle we place either "L" or "R" to denote whether the action is injecting into the left or right side of the non-deterministic resource. Example instances are shown in Figure 3.5, each injecting into one side of the non-deterministic combination of atoms X and Y.

The idea is similar for the distribution actions, except that they have three resource parameters so the shape we choose is a diamond: three of its points accommodate



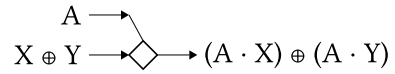
(a) *InjectL* (*Res X*) (*Res Y*)



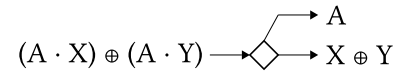
(b) *InjectR* (*Res X*) (*Res Y*)

Figure 3.5: Illustrative diagrams of injection actions into non-deterministic resources

one non-deterministic input, one non-deterministic output and the arbitrary resource(s) being distributed in or out. Example instances are shown in Figure 3.6, distributing the atom *A* into and out of the same non-deterministic combination.



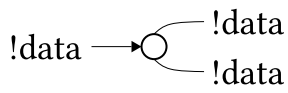
(a) *OptDistrIn* (*Res A*) (*Res X*) (*Res Y*)



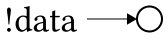
(b) *OptDistrOut* (*Res A*) (*Res X*) (*Res Y*)

Figure 3.6: Illustrative diagrams of distribution actions over non-deterministic resources

For the duplication and erasing actions on copyable atoms, it would be sufficient to visualise them using a fork in a wire and an end to a wire respectively. The fact that the visualisation is constructed from a process composition would ensure that this is only ever done with copyable resources, since no other resources can figure in these actions. We however choose to emphasise these actions by adding a small circle into their visualisation, making it clear that this is intentional and not just an error in the program building the diagram. Example instances duplicating and erasing a copyable atom *data* are shown in Figure 3.7.



(a) *Duplicate data*

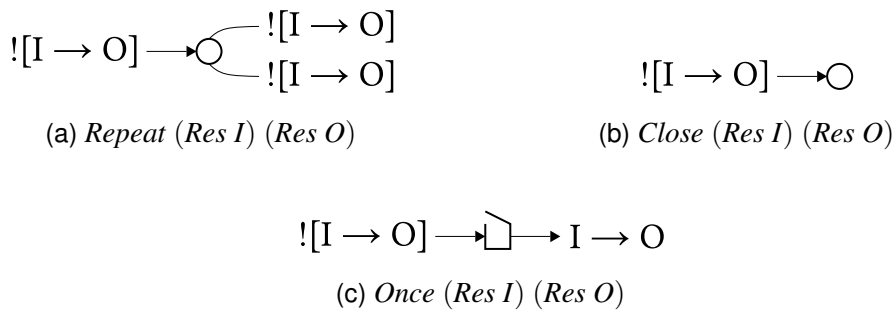


(b) *Erase data*

Figure 3.7: Illustrative diagrams of resource actions duplicating and erasing a copyable resource atom

We use the same visualisations for the repetition and closing of repeatably executable resources, because these represent the copyable aspect of these resources. For the third action related to this type of resources, turning them into plain executable resources, we use an open box shape to suggest the “unboxing”. Example instances of these actions are shown in Figure 3.8, using a repeatably executable resource with input atom *I* and output atom *O*.

For the application of an executable resource to an input we use use a box with two groups of incoming wires — one group for the input resource and one wire for the



(a) *Repeat* (*Res I*) (*Res O*)

(b) *Close* (*Res I*) (*Res O*)

(c) *Once* (*Res I*) (*Res O*)

Figure 3.8: Illustrative diagrams of characteristic resource actions for repeatably executable resources

executable resource — and output wires for the corresponding output resource. To distinguish this box from the visualisation of primitive actions, we connect the wire carrying the executable resource to the bottom of the box, separating it from the ordinary input and output, and include two bent wires inside the box. These wires suggest that the input resource is brought to the executable resource, and that the output resource is brought from it. This visualisation is meant to match that of *Represent* (shown in Figure 3.14), which can be used to introduce (repeatably) executable resources. An example visualisation for an instance of the *Apply* action is shown in Figure 3.9.

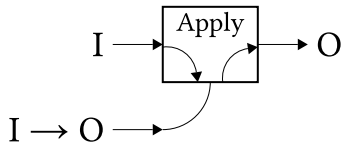


Figure 3.9: *Apply* (*Res I*) (*Res O*)

Final among the resource actions, forgetting is visualised with a cross to whose centre all input wires are brought, with a single wire coming out carrying the *Anything* resource. This is intended to indicate all information about the input resource being discarded, including any parallel combination structure it might have had. An example instance is shown in Figure 3.10, in this case forgetting the two parallel atoms *A* and *B*.

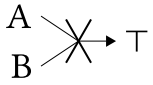


Figure 3.10: *Forget* (*Res A* ⊙ *Res B*)

To represent sequential composition, we take the diagrams visualising the processes being composed and lay them out horizontally with a set amount of spacing

between them. We then connect the ends of output wires in the first diagram to the starts of input wires in the second diagram. Assuming that the composition we are visualising is valid, we know that the output and input resources match (see Section 3.2) and as a result so will the wires. An example instance is shown in Figure 3.11.

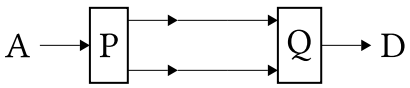


Figure 3.11: *Seq* (*Primitive* (*Res A*) (*Res B* $\odot$ *Res C*) *STR* *"P"* ())
(*Primitive* (*Res B* $\odot$ *Res C*) (*Res D*) *STR* *"Q"* ())

The case of parallel composition is simpler, since we do not make any new connections. We simply lay the two diagrams out vertically, with a set amount of space between them. Additionally, to improve the layout of the diagrams, we stretch the smaller diagram of the two to the same width as the other by extending its input and output wires. An example instance is shown in Figure 3.12.

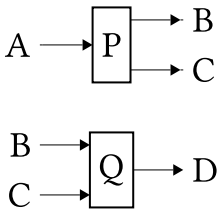


Figure 3.12: *Par* (*Primitive* (*Res A*) (*Res B* $\odot$ *Res C*) *STR* *"P"* ())
(*Primitive* (*Res B* $\odot$ *Res C*) (*Res D*) *STR* *"Q"* ())

The visualisation of optional composition is complicated by the fact that we have already used the two axes available to us in the plane for sequential and parallel composition. As such, we cannot show it with just juxtaposition. As the basis of our visualisation we lay out the diagrams of the children vertically, and then we introduce two special shapes: an input "splitter" shown as the $\oplus$ symbol and an output "combiner" shown as a triangle pointing left. These are meant to depict the input non-deterministic resource being split into its branches, each routed to the corresponding child of the composition, and the children's outputs being merged together to get the overall output. We use the $\oplus$ shape to suggest the involvement of non-deterministic resources, and the triangle because its angled sides can face the child diagrams and its vertical side can accommodate any number of parallel outputs. An example instance is shown in Figure 3.13.

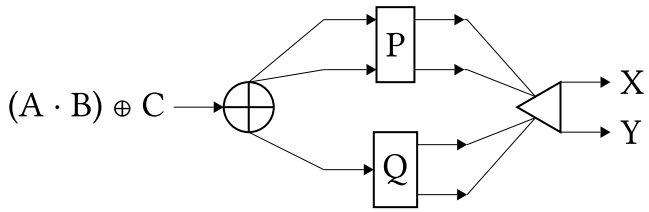


Figure 3.13: *Opt* (*Primitive* (*Res A* ⊙ *Res B*) (*Res X* ⊙ *Res Y*) *STR* "*P*" ())
(*Primitive* (*Res C*) (*Res X* ⊙ *Res Y*) *STR* "*Q*" ())

Finally, the visualisation of process representation as a repeatably executable resource only needs to include one child diagram within it. We surround it with a box to clarify that this is the nested part. Then we add a wire going from the input side of the nested diagram and wrapping around to the right, matching the fact that the representation of a process takes no input and instead both the input and output of the child process play a role in the representation's output. Since the repeatably executable resource is not the same as a parallel combination of resources, we enclose the diagram in another box to avoid suggesting as much and give it the one output wire for the output that this process actually has. An example instance is shown in Figure 3.14.

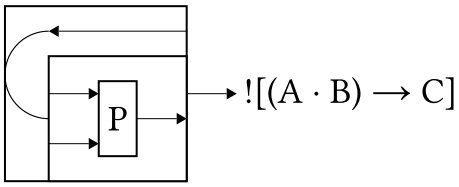


Figure 3.14: *Represent* (*Primitive* (*Res A* ⊙ *Res B*) (*Res C*) *STR* "*P*" ())

3.4 Process Transformations

When modelling processes, we often find ourselves wanting to refine the modelling domain or the process itself. We may for example want to take a formalised process talking about one set of objects, and formalise the same process over a different set of objects. Or we may want to make an abstract process more specific by replacing one of its actions with a composition of several other actions.

The simplest ways to make such refinements is to formalise the new domain and repeat the same composition steps in it, possibly going into more detail at some point. But, with our mechanised framework we can make such refinements more rigorous as an operation on process compositions. This allows us to automate process transfor-

mations, make explicit the connection between the original process and the result, and formally verify properties of the transformations.

In this section we discuss three process transformations: mapping resource atoms to a new domain, refining resource atoms into arbitrarily complex resources and refining primitive actions. All three are mechanised as part of our framework, allow for code generation and we verify under what conditions they preserve composition validity.

In Section 7.1 we demonstrate these transformations by taking an abstract model of coursework marking and refining it with concrete information only available later in the run of a course. The transformations allow us to derive a model that is more faithful to the practice of that process while retaining its status as a refinement of the initial outline. As a result, any change to the abstract model is automatically projected to the refined model.

3.4.1 Resource Mapping

Resource mapping refers to transforming all atoms present in a resource, either within one domain or to another domain entirely. When we define the resource terms in Definition 2.1.1, Isabelle/HOL automatically defines the function *map-res-term* f g, which transforms a resource term by applying the function f to any linear atom within it and the function g to any copyable atom. This function is lifted to resources as *map-resource* when we prove they inherit the BNF structure of the terms (see Section 2.4). Finally, when we define process compositions in Definition 3.1.1, Isabelle/HOL uses the lifted function to define *map-process*, which takes as arguments a function for each of: linear atoms, copyable atoms, labels and metadata.

For instance, consider a domain whose linear atoms are amounts of some currency (e.g. as a pair of natural number and currency tag). Then, given the conversion ratios, we can map each of those atoms to the corresponding amount in some unified currency (e.g. that of the account we are paying from). Mapping this over all resources in a process can be considered as transforming that process within one domain because the result still has amounts of currency as atoms, just with different magnitudes.

Or, consider a domain whose linear atoms are objects placed at some locations. Then we can map each of those atoms to just the relevant object, forgetting its location. Applying this map to a process then represents moving to a simplified domain where we do not account for locations. For example, if the located atoms are represented by

object-location pairs, then we can use the projection *fst* to retain only the objects. A mug in the office, another mug in the bedroom and a book in the office then become just a mug, a mug and a book:

$$\textit{map-resource fst fst}\ (\textit{Res}\ (\textit{Mug, Office}) \odot \textit{Res}\ (\textit{Mug, Bedroom}) \odot \textit{Res}\ (\textit{Book, Office})) \\ = \textit{Res Mug} \odot \textit{Res Mug} \odot \textit{Res Book}$$

The fact that this transformation changes resource atoms involved in a process while preserving its structure gives it strong properties. First, the input and output of the transformed process are the input and output of the original process transformed accordingly. Second, any process that was valid before the transformation will remain valid after it.

Note that the second point does not require anything from the functions. Any valid process composition transformed in any way will always remain valid. But the converse is not the case: it is possible to map distinct atoms to the same one, erasing what made a composition invalid.

3.4.2 Resource Refinement

Resource refinement generalises the idea of resource mapping: instead of replacing atoms with other atoms, we allow the replacement to be any resource. This leads to opaque atoms being replaced with resources that have known structure, refining the description.

Note, however, that we can only allow such a refinement for linear atoms, because otherwise we could use it to refine a copyable resource into a linear resource. This would make it very easy for this transformation to render a process composition invalid, for instance if it involves the *Duplicate* action. As such, we restrict the replacements for copyable atoms to other copyable atoms.

We define this operation on resource terms by recursion through all its constructors, using functions passed as arguments in the linear and copyable atom cases to form the replacement subterm. This we then lift to resources, and define the function *process-refineRes* to apply it across all resources within a process.

For instance, consider our model of coursework marking (see also Section 3.4 and Section 7.1). In the abstract model we use a single resource atom to represent all of the students. Once the number of students is known, we can use resource refinement to replace every occurrence of this monolithic resource with a parallel combination of the right number of atoms to represent individual students. This refinement not only

makes the process reflect the new information, it also paves the way towards breaking up monolithic actions: once we consider the students as individuals, we can conceive of marking their coursework submissions as separate actions. Consider the following simplified example (see Section 7.1.2 for the full version):

$$\textit{refine-resource}\ (\lambda x.\ \textit{Parallel}\ (\textit{replicate}\ 3\ (\textit{Res Student})))\ (\lambda x.\ x)\ (\textit{Res Students}) \\ = \textit{Res Student} \odot \textit{Res Student} \odot \textit{Res Student}$$

Once again, this operation has strong properties. The input and output of the refined process are the input and output of the original process, refined accordingly. And any process that was valid before the refinement will remain valid after it.

3.4.3 Primitive Action Substitution

To refine a process composition, we substitute an arbitrarily complex process in place of a primitive action. The simplest approach would be to target a single action (perhaps identified by its path in the composition tree, see Section 3.3.1) and replace it with a given process. But our definition is more complex in order for the operation to be more generally useful.

We define the substitution as a function with two parameters, *process-subst* $P\ f$. The first parameter acts as a predicate on the four primitive action parameters (input, output, label and metadata) and determines targets for the substitution. The second parameter takes those four parameters of every target and generates its replacement process. As with resource refinement, the function recursively passes over a composition and for every primitive action it checks whether it satisfies P and, if so, replaces it with the relevant output of f. This allows us to define substitutions of multiple actions at once that react to what they are replacing.

Referring back to our initial model of coursework marking from Section 3.4.2, once we have used the resource refinement to split the monolithic *Students* resource into multiple individuals, we can use process substitution to break up a monolithic marking action into several parallel actions marking individual submissions. The predicate can identify the target action by its input (all students). The replacement can then be a parallel composition of individual marking actions, with their number determined by the number of students. See Section 7.1.2 for concrete examples of the target predicate and replacement function. Applying this substitution expresses that, once we model students and their submissions individually, we can mark each submission independently of the others.

The increased complexity of this transformation is reflected in the assumptions of its properties. If the replacement function does not change the target action's input, then the input of the composition does not change. If it does not change the input and output, then the output of the composition also does not change. If, furthermore, every replacement it generates is valid, then any composition that was valid before the substitution will remain valid after it.

3.5 Conclusion

In this chapter we described our mechanisation of process compositions as trees of individual actions and composition operations. We use resources to describe the inputs and outputs of processes, with the compositions describing how these inputs and outputs combine. For instance, sequential composition describes that the output of one process is used as input to another. We then formalised a validity condition for process compositions, expressing when the composition operations are used correctly.

We also gave a brief overview of how we generate process diagrams to visualise our compositions, using Haskell code generated automatically by Isabelle from our mechanisation. And we closed on a discussion of the kinds of systematic process transformations we can define and verify in our framework.

As with the equivalences that build the resource algebra, the set of resource actions we use is not exhaustive. For instance, in Section 6.6.1 we suggest two possible additions. As such, we see the identification and addition of further resource actions as a continual part of future work.

We use process compositions in the remainder of this thesis. In the next chapter, we connect them to deductions of linear logic in such a way that valid compositions always yield well-formed deductions. In Chapter 5 we make the ideas underlying our process diagrams more formal and use them to verify properties of process compositions.

Chapter 4

Linearity of Process Compositions

In this chapter we describe our argument for the linearity of resources in process compositions, connecting the compositions to deductions in intuitionistic linear logic (see Section 1.1).

The argument is, in short, that every composition corresponds to a deduction in ILL with the following properties:

Well-formedness

For every valid composition the corresponding deduction is well-formed: it follows the rules of ILL.

Input-Output Correspondence

The conclusion of the deduction is a sequent $I \vdash O$ where I and O are propositions corresponding to the composition's input and output respectively.

Primitive Correspondence

Primitive actions of the composition correspond exactly to the premises of the corresponding deduction.

Structural Correspondence

The structure of the composition matches that of the corresponding deduction.

To express these properties, we must embed ILL in Isabelle/HOL. We first mechanise a shallow embedding, allowing us to formally talk about what sequents are valid in the logic. But this is not enough to fully demonstrate the properties, so we also mechanise a deep embedding which allows us to formally talk about the deductions themselves and their structure. We prove that the deep embedding is sound and complete with respect to the shallow embedding. We refer the reader to the work of Dawson

and Goré [24], for instance, for a further discussion of shallow and deep embeddings in Isabelle/HOL.

When connecting resources to ILL propositions we encounter a difficulty in the many concrete terms that may represent one resource. To resolve this, we use its normal form and mirror the normalisation procedure described in Section 2.3 with ILL deductions to show that any equivalent resource terms correspond to logically equivalent propositions.

Note that the connection we make is from process compositions to linear logic deductions, but not necessarily the other way. There exist well-formed ILL deductions which do not reflect any of our process compositions, for instance because we do not make any connection to the ILL & operator. Similarly, the resources and process compositions may be extended to carry information that cannot be expressed in ILL (see for example Chapter 6).

4.1 Related Work

Our use of ILL for this argument is informed by previous work on the mechanical construction of plans. Dixon et al. [25] describe how proof search in ILL corresponds to planning, with search algorithms corresponding to planning strategies. They implement this relation in Isabelle/HOL, producing plans as terms witnessing the truth of an existentially quantified conjecture about the preconditions, postconditions and other constraints on the plan. Küngas and Matskin [51] for their part formalise cooperative planning of multiple agents — where those agents exchange capabilities to reach goals none could reach on their own — in terms of partial deduction in ILL. This work supports ILL as a suitable formalism for expressing processes composed from individual actions.

There already exists a shallow embedding of ILL in Isabelle due to Kalvala and de Paiva [49]. However, this is as part of the object logic Isabelle/Sequents, which is distinct from Isabelle/HOL. While this development is not compatible with our Isabelle/HOL one, we adapt some of their recommendations into our mechanisation of ILL. For instance, we adjust antecedents in the inference rules of ILL to remove implicit assumptions. As far as we are aware, theirs is the only published mechanisation of ILL in Isabelle.

Outside of Isabelle there are the multiple mechanisations of linear logic in Rocq:

- Power and Webster [73] developed a shallow embedding of ILL.

- Laurent created YALLA[1], a deep embedding of multiple fragments of propositional linear logic.
- Xavier et al. [96] mechanised propositional and first-order linear logic with focusing.

While these Rocq developments are noteworthy, they have not influenced the work described in this chapter.

Linear logic is also used outside the theorem proving world, especially in logic programming and type systems. Within logic programming there are for instance the languages Lolli by Hodas and Miller [39] and its extension, Forum, by Miller [59]. Since linear logic refines the control over copying and deleting propositions, logic programming languages based on it inherits this increased control. This means that rules express not only which expressions are needed to derive their goal, but also how many of them. As in our use, this is particularly useful when they represent resources that are being consumed.

Similarly, type systems based on linear logic offer this increased control to programs. With linear types the compiler checks that a (linear) value is used exactly once. This can again be used to improve handling of resources, such as memory locations, file handles or even quantum states. Bernardy et al. [11] describe how linear types can be integrated into existing languages, demonstrating this for Haskell. Radanne et al. [75] extend ML with linear and affine (one or zero uses) types.

Going beyond single use, quantitative type theory [4] includes multiplicity in the type information drawn from some given semiring. This can be used to express more complex use patterns beyond the unrestricted use of ordinary types and exactly one use of linear types.

Finally, recall the session types we mention in Section 1.2. These too are based on linear logic and can be integrated into programming languages to check that communication complies with specified protocols. For instance, session types have been integrated into the web programming language Links [54], Haskell [53, 67, 47, 74], Scala [78], Rust [48] and OCaml [58].

[1] `https://github.com/olaure01/yalla`

4.2 Shallow Embedding of ILL Deductions

A shallow embedding of ILL deductions is simpler to define than the deep one and it allows us to reuse much of the automation available in Isabelle. However, it is limited to formalising what sequents are valid *within* the logic rather than directly talking about the structure of its deductions.

There is already a shallow embedding of ILL by Kalvala and de Paiva [49] distributed with Isabelle, but this is part of the Isabelle/Sequents system and is not compatible with our HOL development. Nevertheless, their approach provides inspiration for some aspects of our mechanisation.

First we mechanise the propositions of ILL as the datatype *'a ill-prop*, mirroring the specification (1.1) from Section 1.1, along with relevant notation. The type variable *'a* represents the type from which we draw the propositional atoms.

Isabelle Definition 4.2.1 (ILL propositions)

```
datatype 'a ill-prop =
  Prop  ('a)
| Times ('a ill-prop) ('a ill-prop) (infixr ⊗) | One (1)
| With  ('a ill-prop) ('a ill-prop) (infixr &) | Top (⊤)
| Plus  ('a ill-prop) ('a ill-prop) (infixr ⊕) | Zero (0)
| LImp  ('a ill-prop) ('a ill-prop) (infixr ⊸)
| Exp   ('a ill-prop) (!)
```

Then we represent the valid sequents of ILL as an inductive relation between a list of propositions (*antecedents*) and a single proposition (*consequent*). We denote it infix by $\vdash$ and the full definition is shown in Definition 4.2.2.

Every rule in this definition represents one of the inference rules of ILL shown in Figure 1.1. However, we adjust their precise statement following the work of Kalvala and de Paiva [49] to remove implicit assumptions which make them less useful for pattern matching. For instance we adjust the $\otimes_L$ rule by adding Δ so that we no longer assume that A and B are the last antecedents:

$$\frac{\Gamma, A, B \vdash C}{\Gamma, A \otimes B \vdash C}\otimes_L \quad \text{becomes} \quad \frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, A \otimes B, \Delta \vdash C}\otimes_L$$

Note that, compared to our statement in Section 1.1, ILL is often stated with multisets for antecedents instead of lists as in our formalisation. In such contexts the order of antecedents does not matter and the structural inference rule Exchange is made implicit.

Isabelle Definition 4.2.2 (Shallow embedding of valid ILL sequents)

```
inductive sequent :: 'a ill-prop list ⇒ 'a ill-prop ⇒ bool (infix ⊢ 60) where
  identity:   [a] ⊢ a
| exchange:   G @ [a] @ [b] @ D ⊢ c ⟹ G @ [b] @ [a] @ D ⊢ c
| cut:        ⟦G ⊢ b; D @ [b] @ E ⊢ c⟧ ⟹ D @ G @ E ⊢ c
| timesL:     G @ [a] @ [b] @ D ⊢ c ⟹ G @ [a ⊗ b] @ D ⊢ c
| timesR:     ⟦G ⊢ a; D ⊢ b⟧ ⟹ G @ D ⊢ a ⊗ b
| oneL:       G @ D ⊢ c ⟹ G @ [1] @ D ⊢ c
| oneR:       [] ⊢ 1
| limpL:      ⟦G ⊢ a; D @ [b] @ E ⊢ c⟧ ⟹ G @ D @ [a ⊸ b] @ E ⊢ c
| limpR:      G @ [a] @ D ⊢ b ⟹ G @ D ⊢ a ⊸ b
| withL1:     G @ [a] @ D ⊢ c ⟹ G @ [a & b] @ D ⊢ c
| withL2:     G @ [b] @ D ⊢ c ⟹ G @ [a & b] @ D ⊢ c
| withR:      ⟦G ⊢ a; G ⊢ b⟧ ⟹ G ⊢ a & b
| topR:       G ⊢ ⊤
| plusL:      ⟦G @ [a] @ D ⊢ c; G @ [b] @ D ⊢ c⟧ ⟹ G @ [a ⊕ b] @ D ⊢ c
| plusR1:     G ⊢ a ⟹ G ⊢ a ⊕ b
| plusR2:     G ⊢ b ⟹ G ⊢ a ⊕ b
| zeroL:      G @ [0] @ D ⊢ c
| weaken:     G @ D ⊢ b ⟹ G @ [!a] @ D ⊢ b
| contract:   G @ [!a] @ [!a] @ D ⊢ b ⟹ G @ [!a] @ D ⊢ b
| derelict:   G @ [a] @ D ⊢ b ⟹ G @ [!a] @ D ⊢ b
| promote:    map Exp G ⊢ a ⟹ map Exp G ⊢ !a
```

With our shallow embedding of ILL (using explicit Exchange, see *exchange* in Definition 4.2.2) we can prove that this move is admissible in the logic. We do so by proving that any two sequents whose antecedents form equal multisets are equally valid, stated more generally in Isabelle as follows:

Isabelle Lemma 4.2.1 (Multiset exchange)

```
lemma exchange-mset:
  assumes mset A = mset B
    shows G @ A @ D ⊢ c = G @ B @ D ⊢ c
```

This fact relies on the theories of multisets and of combinatorics already formalised in Isabelle/HOL. We first note that any two lists forming the same multiset are related by a permutation, which is a sequence of element transpositions. By induction on this sequence of transpositions we show that we can derive each sequent from the other.

4.3 Resources as Linear Propositions

Before relating process compositions to ILL deductions we first need to relate the resources within the former to ILL propositions. Then we can translate the input and

output of processes into ILL.

Because resources are defined as a quotient, we first define the translation for resource terms:

Isabelle Definition 4.3.1 (ILL translation of resource terms)

```
primrec res-term-to-ill :: 'a res-term ⇒ 'a ill-prop
 where
  res-term-to-ill Empty = 1
| res-term-to-ill Anything = ⊤
| res-term-to-ill (Res x) = Prop x
| res-term-to-ill (Copyable r) = !(res-term-to-ill r)
| res-term-to-ill (Parallel rs) = compact (map res-term-to-ill rs)
| res-term-to-ill (NonD r t) = (res-term-to-ill r) ⊕ (res-term-to-ill t)
| res-term-to-ill (Executable a b) = (res-term-to-ill a) ⊸ (res-term-to-ill b)
| res-term-to-ill (Repeatable a b) = !((res-term-to-ill a) ⊸ (res-term-to-ill b))
```

Note how all but the *Parallel* case map a resource term constructor directly to a constructor of ILL propositions. The *Parallel* case instead uses the helper function *compact* to combine the list of translations of the children using the binary $\otimes$ operator of ILL. This function is defined as follows:

Isabelle Definition 4.3.2 (ILL proposition compacting)

```
function compact :: 'a ill-prop list ⇒ 'a ill-prop where
  xs ≠ [] ⟹ compact (x # xs) = x ⊗ compact xs
| xs = [] ⟹ compact (x # xs) = x
| compact [] = 1
```

Then we extend this to resources by translating the normal form representative term obtained via *of-resource* (see Section 2.3.7):

Isabelle Definition 4.3.3 (ILL translation of resources)

```
fun resource-to-ill :: 'a resource ⇒ 'a ill-prop
 where resource-to-ill x = res-term-to-ill (of-resource x)
```

The crucial property of this translation is that for equivalent resource terms the translation of one can be derived within ILL from the translation of the other. The derivation in the opposite direction then also follows by symmetry of resource term equivalence. In other words, the logic agrees with the relation we defined.

Isabelle Lemma 4.3.1 (Equivalent terms yield equivalent propositions)

lemma *res-term-ill-equiv*:
assumes $a \sim b$
shows $[\textit{res-term-to-ill}\ a] \vdash \textit{res-term-to-ill}\ b$
and $[\textit{res-term-to-ill}\ b] \vdash \textit{res-term-to-ill}\ a$

In the translation of process compositions to deductions we may construct translations of non-normal terms, so this property is vital to the resulting deductions being well-formed. Consider a situation that may arise from parallel composition of a process with two inputs, say A and B, with a process with one input, say C:

- The combined input is $\textit{Parallel}\ [\textit{Res}\ A, \textit{Res}\ B, \textit{Res}\ C]$ which translates to $\textit{Prop}\ A \otimes (\textit{Prop}\ B \otimes \textit{Prop}\ C)$.
- But the first input is $\textit{Parallel}\ [\textit{Res}\ A, \textit{Res}\ B]$, which translates to $\textit{Prop}\ A \otimes \textit{Prop}\ B$, and the second input is $\textit{Res}\ C$, which translates to $\textit{Prop}\ C$.
- The product of those translations is $(\textit{Prop}\ A \otimes \textit{Prop}\ B) \otimes \textit{Prop}\ C$ which is not the same proposition.

We prove the property by induction on the resource term equivalence relation. In each case we use Isabelle's automated methods to find the deduction pattern needed to transform one translation into the other. These methods make use of facts about ILL as well as facts about the proposition compacting operation, such as the following:

Isabelle Lemma 4.3.2 (Compacting appended lists is equivalent to the $\otimes$ operator)

lemma *times-equivalent-append*:
shows $[\textit{compact}\ a \otimes \textit{compact}\ b] \vdash \textit{compact}\ (a\ @\ b)$
and $[\textit{compact}\ (a\ @\ b)] \vdash \textit{compact}\ a \otimes \textit{compact}\ b$

4.4 Shallow Embedding is Not Enough

With the shallow embedding of ILL we can start formalising our argument for linearity of process compositions. In this section we describe how this embedding allows us to demonstrate the *Well-formedness* and *Input-Output Correspondence* properties and how it is insufficient for the *Primitive Correspondence* and *Structural Correspondence* properties. This insufficiency motivates our use of a deep embedding in the following sections.

For every process composition p, we have the following ILL sequent formed from translating its input and output (call it the *input-output sequent*):

$$[\textit{resource-to-ill}\ (\textit{input}\ p)] \vdash \textit{resource-to-ill}\ (\textit{output}\ p)$$

We can show that, for every valid process composition, its input-output sequent is valid in ILL given the validity of input-output sequents of primitive actions occurring in the composition:

Isabelle Lemma 4.4.1 (Shallow linearity theorem)

lemma *shallow-linearity*:
assumes *valid p*
and $\forall$ *ins outs l m*.
$(\textit{ins}, \textit{outs}, l, m) \in \textit{set}\ (\textit{primitives}\ p)$
$\longrightarrow [\textit{resource-to-ill}\ \textit{ins}] \vdash \textit{resource-to-ill}\ \textit{outs}$
shows $[\textit{resource-to-ill}\ (\textit{input}\ p)] \vdash \textit{resource-to-ill}\ (\textit{output}\ p)$

We prove this statement by structural induction on the process. In each case we make use of Isabelle's automated methods to find an ILL sequent derivation from the translation of the input to the translation of the output. However, as the following discussion details, this proof is insufficient to ensure all of the properties we want. We address this in Section 4.7 by using a deep embedding of deductions to construct concrete instances of this general proof for any given process.

The *Well-formedness* property is demonstrated by the proof being checked by Isabelle, while *Input-Output Correspondence* is demonstrated by the conclusion being the input-output sequent of the composition.

Primitive Correspondence is not sufficiently demonstrated by this theorem. Its assumption only says that input-output sequents of the primitive actions are sufficient for the proof. This does not necessarily mean that they are necessary nor that they are used as many times as the primitive actions occur in the composition. Thus we cannot conclude that the primitive actions correspond exactly to the premises of this deduction.

Structural Correspondence is also not demonstrated by the theorem. The structure of its proof does not necessarily follow the structure of the composition. We know that there exists *some* proof of the input-output sequent, but that proof may not have any further relation to the composition itself and so this is not a satisfying argument for its linearity.

Consider for example process compositions whose input is equal to their output. We can prove the input-output sequent for any such composition to be valid in ILL directly by the Identity inference rule:

$$\frac{}{A \vdash A}\ \text{Identity}$$

In this way, the sequential composition of primitive actions P: $A \to B$ and then Q: $B \to A$ can have its input-output sequent shown to be valid in ILL without using any premise at all.

Moreover, consider the sequential composition of identities on resources A, then on B and then again on A (where A and B are distinct). Its input-output sequent can again be shown to be valid in ILL, despite this composition being invalid because it creates and discards the resource B.

In the following sections we develop a deep embedding of ILL deductions which allows us to demonstrate the *Primitive Correspondence* and *Structural Correspondence* properties. With the deep embedding we can say that ILL accepts not just the input and output of a process composition but every step within it.

4.5 Deep Embedding of ILL Deductions

A deep embedding of ILL deductions represents them as objects we can directly construct. This gives up much of the automation Isabelle offers during proof, but it allows us to build deductions whose structure we know matches the process composition.

For this, we mechanise deductions as the datatype $('a, 'l)$ *ill-deduct*, whose elements are trees with nodes exactly mirroring the defining rules of the sequent relation in Definition 4.2.2. We also include a node for explicitly representing premises (the meta-level assumption of a particular sequent) to let us express contingent deductions.

The type of deductions is parameterised by two type variables: $'a$ and $'l$. The type $'a$ represents the type from which we draw the propositional variables, just as it does in the type $'a$ *ill-prop* of propositions. The type $'l$ represents the type of labels we attach to the premise nodes. We use these labels to distinguish premises that may assume the same sequent but have different intended meaning. For instance, once we get a drink from a vending machine we can drink it or we can pour it out on the ground. Both these actions take the drink as input and result in an empty can, so their representation as premises will assert the same sequents, but they have very different meaning.

In total this datatype has 22 constructors, each with up to eight parameters. The deduction tree's semantics are defined via two functions: *ill-conclusion* expresses the deduction's conclusion sequent while the predicate *ill-deduct-wf* checks whether the deduction is well-formed. Full definitions are shown in Appendix A.1, but as an example we consider the Cut rule which is stated in the shallow embedding as follows:

$$[\![G \vdash b; D\ @\ [b]\ @\ E \vdash c]\!] \Longrightarrow D\ @\ G\ @\ E \vdash c$$

Its deep embedding, the term $Cut\ G\ b\ D\ E\ c\ P\ Q$, represents the deduction tree shown in Figure 4.1. Note that P and Q correspond to deep embeddings of the two assumptions in the shallow rule: $G \vdash b$ and $D\ @\ [b]\ @\ E \vdash c$ respectively.

$$\dfrac{\begin{array}{cc} P & Q \\ \vdots & \vdots \\ G \vdash b & D\ @\ [b]\ @\ E \vdash c \end{array}}{D\ @\ G\ @\ E \vdash c}\ \text{Cut}$$

Figure 4.1: Deduction tree represented by the term $Cut\ G\ b\ D\ E\ c\ P\ Q$

The semantic functions take the following values for this rule[2]:

$$\begin{array}{l} ill\text{-}conclusion\ (Cut\ G\ b\ D\ E\ c\ P\ Q) = D\ @\ G\ @\ E \vdash c \\ ill\text{-}deduct\text{-}wf\ (Cut\ G\ b\ D\ E\ c\ P\ Q) = \\ \quad (ill\text{-}deduct\text{-}wf\ P \wedge ill\text{-}conclusion\ P = G \vdash b \wedge \\ \quad\ ill\text{-}deduct\text{-}wf\ Q \wedge ill\text{-}conclusion\ Q = D\ @\ [b]\ @\ E \vdash c) \end{array}$$

Additionally, a function called *ill-deduct-premises* recursively gathers the list of all the premise leaves in a deduction (represented by antecedents-consequent-label triples). Continuing the above example, the premises of a cut node are those of its two child deductions:

$$\begin{array}{l} ill\text{-}deduct\text{-}premises\ (Cut\ G\ b\ D\ E\ c\ P\ Q) = \\ \quad (ill\text{-}deduct\text{-}premises\ P\ @\ ill\text{-}deduct\text{-}premises\ Q) \end{array}$$

We verify this deep embedding by proving it is sound and complete with respect to the shallow one. Soundness requires that the conclusions of well-formed deductions are valid sequents given the validity of the premises, stated in Isabelle as:

Isabelle Lemma 4.5.1 (Soundness of deeply embedded deductions)

lemma *ill-deduct-sound*,
assumes *ill-deduct-wf P*
and $\forall (a, c, l) \in ill\text{-}deduct\text{-}premises\ P.\ ill\text{-}sequent\text{-}valid\ (Sequent\ a\ c)$
shows *ill-sequent-valid* (*ill-conclusion P*)

Completeness requires that for every valid sequent there exist a well-formed deduction with it as conclusion and with no premises:

[2]Formally, the *ill-conclusion* value is pair of proposition list and single proposition with custom notation. This is to emphasise the connection with the shallowly embedded relation for valid sequents. The predicate *ill-sequent-valid* ties these values back to the relation. See Appendix A.1 for the definitions.

Isabelle Lemma 4.5.2 (Completeness of deeply embedded deductions)

lemma *ill-deduct-complete*:
assumes $G \vdash c$
obtains P
where *ill-conclusion* $P = \mathit{Sequent}\ G\ c$
and *ill-deduct-wf* P
and *ill-deduct-premises* $P = []$

Because the deduction tree nodes mirror the sequent relation, we can prove these statements rather simply by induction either on the deduction structure or on the sequent relation. Note that for completeness we require the deduction to have no premises because otherwise it would be trivial: we could just assume the sequent.

4.6 Deeply Embedded Equivalence of Resource Translations

Recall that Lemma 4.3.1 shows that translations of equivalent resource terms can be derived from one another:

$$a \sim b \Longrightarrow [\mathit{res\text{-}term\text{-}to\text{-}ill}\ a] \vdash \mathit{res\text{-}term\text{-}to\text{-}ill}\ b$$

We use this fact to fill gaps between linear logic translations of different but equivalent resource terms, which will be vital when we construct deductions from process compositions in the next section. To make use of this fact in the deep embedding, we first need to describe how a witness deduction is constructed for every case. This mirrors the earlier problem of deciding the resource term equivalence, so our solution is similar to the normalisation procedure described in Section 2.3.

We build a deduction from one resource term to its normal form and another deduction into the other term from its normal form (note the opposite direction). These two deductions can then be connected because the two terms have equal normal forms. Thus the core of our solution are two functions which construct, for any term a, deductions with the respective conclusions:

$$[\mathit{res\text{-}term\text{-}to\text{-}ill}\ a] \vdash \mathit{res\text{-}term\text{-}to\text{-}ill}\ (\mathit{normal\text{-}rewr}\ a)$$
$$[\mathit{res\text{-}term\text{-}to\text{-}ill}\ (\mathit{normal\text{-}rewr}\ a)] \vdash \mathit{res\text{-}term\text{-}to\text{-}ill}\ a$$

In the definitions of these functions we again mirror the normalisation procedure. However, instead of a rewriting step that transforms the resource term, we build a deduction (in the desired direction) proving the transformation is allowed in ILL. These are then chained with the Cut rule until the resource term is normalised.

We prove that these functions in all cases produce well-formed deductions with the above conclusions. Furthermore, we prove that they have no premises and are thus theorems of ILL.

We name the two functions *ill-deduct-res-term-from-normal-rewr* and *ill-deduct-res-term-to-normal-rewr*: ILL deductions that connect a resource term from or to its normal form. They are used in the next section when constructing deductions from process compositions to fill gaps between linear logic translations of different but equivalent resource terms.

4.7 Process Compositions as Linear Deductions

With the deep embedding of ILL deductions we can construct a proof of Lemma 4.4.1 for any specific process composition, showing that its input-output sequent is valid in ILL. This way we can ensure that every such proof satisfies the *Primitive Correspondence* and *Structural Correspondence* properties which we identified at the start of this chapter, namely that: premises of the deduction correspond exactly to primitive actions of the composition and the deduction matches the composition in structure.

We do this by recursively constructing an ILL deduction for every process composition, mechanised as the following function

$$\textit{to-deduct} :: ('a, 'l, 'm)\ \textit{process} \Rightarrow ('a, 'l \times 'm)\ \textit{ill-deduct}$$

By associating every constructor of process compositions with a pattern of ILL inferences, this function ensures that the resulting deduction reflects every step of the composition.

We then prove that the resulting deductions demonstrate the *Well-formedness*, *Input-Output Correspondence* and *Primitive Correspondence* properties. That is, we show that for any valid process composition the deduction is well-formed:

Isabelle Lemma 4.7.1 (Valid compositions yield well-formed deductions)

lemma *valid-to-deduct-wf*:
$\textit{valid}\ P = \textit{ill-deduct-wf}\ (\textit{to-deduct}\ P)$

and that the conclusion is always the input-output sequent:

Isabelle Lemma 4.7.2 (Conclusion is the input-output sequent)

lemma *to-deduct-conclusion*:
$\textit{ill-conclusion}\ (\textit{to-deduct}\ P) = [\textit{resource-to-ill}\ (\textit{input}\ P)] \vdash \textit{resource-to-ill}\ (\textit{output}\ P)$

and that the premises correspond to the primitive actions that occur in the composition (including in number and order):

Isabelle Lemma 4.7.3 (Premises correspond to primitive actions)

lemma *primitives-give-premises*:
map (λ(*a*, *b*, *l*, *m*). ([*resource-to-ill a*], (*resource-to-ill b*), (*l*, *m*))) (*primitives P*)
= *ill-deduct-premises* (*to-deduct P*)

By the soundness of the deep embedding (Lemma 4.5.1) each thus constructed deduction is then a proof of Lemma 4.4.1 for a specific process composition, with added guarantees about the proof's structure.

We next outline precisely how *to-deduct* constructs the deductions. In some cases the translation is direct, for instance *Primitive* and *Identity* are translated into premises and identity rules respectively:

to-deduct (*Primitive a b l m*) = *Premise* [*resource-to-ill a*] (*resource-to-ill b*) (*l*, *m*)
to-deduct (*process.Identity a*) = *ill-deduct.Identity* (*resource-to-ill a*)

In other cases the deduction being constructed may be more complex, especially where different forms of one resource are involved. For instance with parallel composition we need to:

1. Separate the proposition translation of a parallel resource into translations of the two inputs,
2. Use the children's deductions to connect translations of their inputs with translations of their outputs.
3. Merge the translations of the two outputs back into one proposition for the combined resource.

In Isabelle/HOL we define this case as shown in Figure 4.2a and visualised as a proof tree in Figure 4.2b. The Isabelle/HOL definition uses helper functions *ill-deduct-simple-cut* for a frequent instantiation of the Cut rule and *ill-deduct-tensor* to juxtapose two deductions using the $\otimes_R$ and $\otimes_L$ rules, and the connections between a resource term and its normal form.

Thus we have fully mechanised our goal with this translation. We show that all valid process compositions are linear on the grounds that they obey rules of linear logic by mechanically producing the specific ILL deduction witnessing this fact. Further, because the way we construct this deduction involves no proof search, we we include it in the code we export from Isabelle/HOL, allowing us to construct the witness even outside of the proof assistant.

```
to-deduct (Par p q) =
  ill-deduct-simple-cut
    (ill-deduct-res-term-from-normal-rewr
      (res-term.Parallel [of-resource (input p), of-resource (input q)]))
    (ill-deduct-simple-cut
      (ill-deduct-tensor (to-deduct p) (to-deduct q))
      (ill-deduct-res-term-to-normal-rewr
        (res-term.Parallel [of-resource (output p), of-resource (output q)])))
```

(a) Embedded deduction for parallel composition

$$\dfrac{\text{From normal form}\ \vdots\ [(\!|a \odot x|\!)] \vdash (\!|a|\!) \otimes (\!|x|\!) \qquad \dfrac{\dfrac{\dfrac{\overset{\textit{to-deduct } p}{\vdots}\ [(\!|a|\!)] \vdash (\!|b|\!) \qquad \overset{\textit{to-deduct } q}{\vdots}\ [(\!|x|\!)] \vdash (\!|y|\!)}{[(\!|a|\!)], (\!|x|\!)] \vdash (\!|b|\!) \otimes (\!|y|\!)}\otimes_R}{[(\!|a|\!) \otimes (\!|x|\!)] \vdash (\!|b|\!) \otimes (\!|y|\!)}\otimes_L \qquad \dfrac{\text{To normal form}\ \vdots\ [(\!|b|\!) \otimes (\!|y|\!)] \vdash (\!|b \odot y|\!)}{[(\!|a|\!) \otimes (\!|x|\!)] \vdash (\!|b \odot y|\!)}\textit{Cut}}{[(\!|a \odot x|\!)] \vdash (\!|b \odot y|\!)}\textit{Cut}$$

(b) Deduction for parallel composition of processes p: $a \to b$ and q: $x \to y$. For the sake of space we use $(\!|r|\!)$ to denote *resource-to-ill* r.

Figure 4.2: Witness of linearity for parallel composition

4.8 Conclusion

In this chapter we provided support for our definition of process composition validity by demonstrating that valid compositions map to well-formed deductions of linear logic. Our use of linear logic here ties into formalisations of process correctness often found in the proofs-as-processes literature. As part of the verification, we show that primitive actions of the composition correspond to premises of the deduction, expressing the view that primitive actions are assumptions about the domain being modelled. To mechanise this connection and its properties in Isabelle/HOL, we produce a deep embedding of intuitionistic linear logic.

Our demonstration in this chapter establishes correctness of valid compositions by linking to the rules of linear logic. In the next chapter we approach the correctness more directly, from the perspective of the resource connections between individual actions. We use a graphical approach to represent actions as nodes and resource connections as edges, and prove that the kinds of connections present are highly constrained by the resources they carry. For instance, linear resources must have a clear origin and a single destination.

Chapter 5

Process Compositions as Port Graphs

In Chapter 3 we introduced process compositions that formally described how a process is formed from smaller ones by means of simple operations. The core information that we can extract from process compositions is how individual actions in the process connect through the resources they produce and consume. In Section 3.3 we introduced process diagrams, visualising process compositions from this point of view. They represent actions as boxes, which are connected by wires to represent resource dependencies. While this visualisation is useful for concisely communicating the composition, it is done outside of the proof assistant. As a result, it is not fully rigorous: we make no formal connection between the composition tree and the diagram beyond the Haskell function drawing it. More specifically, we cannot formally verify any claims about the visualisation or use it in proofs.

In this chapter we make more rigorous the connection between composition trees, which can be seen as the "algebraic" representation, and process diagrams, which can be considered to be the "graphical" representation. For this, we mechanise in Isabelle/HOL *port graphs*, the notion that underpins our process diagrams. We then define a mapping from process compositions into port graphs and formally verify a number of its properties. See Figure 5.1 for an illustrative port graph constructed by our framework to represent a manufacturing process composition.

Our verification culminates in a graphical characterisation of composition linearity and its proof for every valid composition. It serves as a counterpart to Chapter 4, where linearity is demonstrated by appealing to linear logic. This characterisation of linearity specifies more directly which connections between actions are allowed.

At present, our approach is limited to sequential and parallel process compositions, because we find the base form of port graphs insufficient for expressing non-

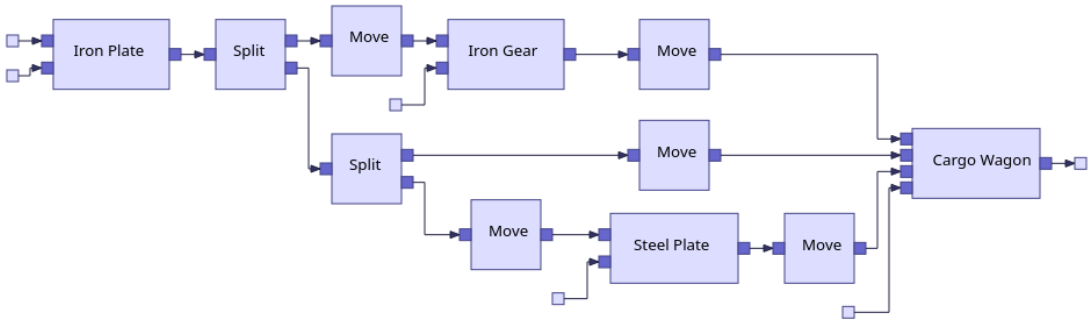


Figure 5.1: Visualisation of a port graph constructed by our framework to represent a composition from a manufacturing domain (see Section 7.4 for a discussion of that domain). The composition has 11 primitive actions, represented here by nodes. Composition operations and resource actions yield the connections between ports of those nodes, representing resources.

deterministic and higher-order features. As a result, we do not provide a port graph mapping for optional composition, process representation or their associated resource actions (such as *InjectL* and *Apply*). The most direct strand of future work, outlined in Section 5.8, aims to extend port graphs and their mechanisation to cover such features and allow for a fully formal graphical representation of *all* process compositions.

The remainder of this chapter proceeds as follows. In Section 5.1 we highlight some related work on port graphs and then in Section 5.2 we introduce the notion of port graphs with a focus on the aspects relevant to our mechanisation. In Section 5.3 we describe our mechanisation of port graphs in general, which forms a self-contained theory that can be reused for other applications. Then in Section 5.4 we give our mapping of process compositions to a specific kind of port graphs and discuss some of its basic properties. In Section 5.5 we use port graphs to prove when sequential and parallel process composition operations distribute over each other. In Section 5.6 we use port graphs to characterise process linearity in terms of allowed connections between actions, and prove it to hold for valid process compositions. In Section 5.7 we define a transition system for port graphs, showing that equivalent port graphs have equivalent transitions. This gives a behavioural dimension to equivalences between port graphs that we prove in prior sections. Finally, in Section 5.8 we give concluding remarks and outline future work.

5.1 Related Work

Port graphs and port graph rewriting systems have been used to model various complex systems. For instance, Andrei et al. [3] use them to model biochemical processes and Vallet et al. [91] use them to model social networks. In comparison to ordinary graphs, the advantage of port graphs is that they allow us to model the specific points where edges connect to nodes. This is useful to represent, for instance, protein sites in biochemistry or communication ports in computer networks. In our case, they are crucial to representing specific input and output resources of individual actions.

Port graphs are also connected to graphical representations arising in applications of category theory. In their book on applied category theory, Fong and Spivak [31] use port graphs in their formalisation of signal flow graphs, a graphical language used for instance in signal processing. As shown for instance by Bonchi et al. [16], signal flow graphs can be viewed as string diagrams, which are widely used in the study of monoidal categories [55]. Through monoidal categories, string diagrams connect to a wide range of graphical representations in applied category theory, such as the ZX calculus in the study of quantum circuits [22, 50]. As such, there is evidence for the fruitful use of port graphs to formalise graphical representations.

Note that, in the literature, port graphs are often formalised with a set of nodes, a set of ports and a function from ports to nodes expressing to which node each port is attached. In our mechanisation we diverge from this representation and instead formalise the ports attached to each node as part of that node. This means that, in mechanising operations on port graphs, we are modifying collections instead of updating functions. We find this approach to be easier to verify. Nevertheless, we can recover the attachment function from our formalisation and, as such, we do not lose anything by using our approach.

Another use of port graphs is the Incredible Proof Machine [18], a tool for visual construction of proofs. It uses port graphs to represent proofs, with nodes representing inference rules and ports their premises and conclusions. Notably, the meta theory of this tool is verified in Isabelle/HOL [19]. As far as we are aware, this is the only published mechanisation of port graphs. However, it is specific to their use as part of the tool being verified. For instance, they do not support open ports, which we find vital to defining operations connecting multiple port graphs (see Section 5.3.9). Instead of adapting this mechanisation, we choose to develop our own.

5.2 Port Graphs

Port graphs [2] refine the notion of directed graphs. Recall that the latter consist of a collection of nodes and a collection of edges going from one node to another. (Undirected graphs simply discard the distinction between edge origin and destination.)

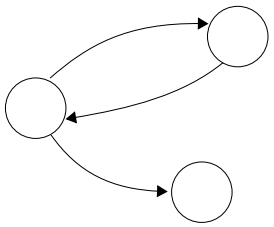


Figure 5.2: Example directed graph with three nodes and three edges

As suggested by their name, port graphs modify directed graphs with *ports* to mediate the connection between nodes and edges. They are still formed by collections of nodes and edges, but every node has a collection of ports attached to it and edges go between ports instead of the nodes themselves. Note that this does not require all ports to have an adjacent edge. The ports allow us to express where and how nodes are connected, instead of simply saying that they are connected or not.

In practice, these ports have been used to express sites on proteins when representing biological processes [3], or input and output signals in signal flow graphs [31].

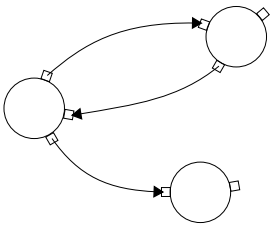


Figure 5.3: Example port graph with three nodes, three edges and eight ports, two of which are left unconnected

Note that it is possible to turn any port graph into a directed graph, essentially forgetting the connection details that the ports express. We can do this by taking the same range of nodes as the port graph and then, for every edge of the port graph we ensure there is an edge in the directed graph that goes from the originating port's node to the destination port's node. If the port graph edges express dependencies, as they do for our process port graphs, then the directed graph constructed in this way can be used to check for dependency loops. We make use of this when discussing a

transition system for port graphs in Section 5.7 to show when it terminates. This graph is sometimes called the *internal flow graph* [31].

Labelled port graphs [30] extend this idea further by labelling each node, port and edge with arbitrary data. This is of great practical use when the port graph is used to represent additional information. For instance, a representation of a distributed system may label nodes with IP addresses and ports with port numbers.

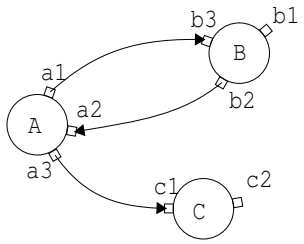


Figure 5.4: Example port graph with simple text labels on nodes and ports

In our case we use node and port labels for multiple purposes. We name nodes based on the corresponding primitive action's position in the composition to uniquely identify them and we annotate them with its label and metadata. We label ports with a combination of side (e.g. input and output for processes) and index to uniquely identify them in their collections and we annotate them with the resources they carry.

Open port graphs address connecting port graphs to form larger ones. They add a number of ports not attached to any node, which we may call *open*, that are available for connection from some outside environment. Then two port graphs may be combined by describing how their open ports are to be connected to form new edges. Note that having open ports not attached to any node allows us to have meaningful port graphs with no nodes. These are useful when combining port graphs, for instance to reorder how existing edges connect to open ports or to disconnect some of them.

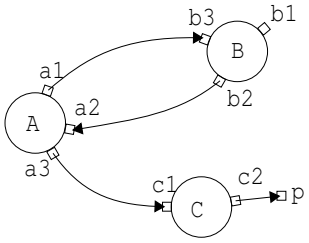


Figure 5.5: Example labelled port graph with port `c2` connected to the open port `p`

This variant is useful when reasoning about port graphs in category theory, where it is vital to have a notion of interface (which here would be the collection of open ports) and composition [31].

In our case, we use open ports for the input and output resources of the process being represented. Sequential process composition then uses those open ports to form the relevant new connections.

Hierarchical port graphs [29] allow for nodes to contain whole other port graphs within them. As the name suggests, these are useful for going beyond connection and representing hierarchical structure as well. This allows them to represent multi-layer systems as well as representing multiple layers of abstraction.

At present, we do not mechanise hierarchical port graphs or make use of them to represent process compositions. However, their existence suggests a future extension to our mechanisation of port graphs that may improve the range of features they can express. See Section 5.8 for our discussion of future work.

In the next section we describe our mechanisation of open port graphs with port and node labels. We then use this theory in succeeding sections to graphically represent process compositions and verify properties about connections between individual actions.

5.3 Mechanisation of Port Graphs

We mechanise port graphs in two parts: their data and the constraints on that data. We then define operations on the data and verify that they preserve those constraints (among other properties).

The data itself we represent using a datatype, which gathers the nodes, edges and open ports. To make implementing operations on port graphs simpler, we use lists for all three collections to represent finite sets.

The constraints on the data we represent as locales, which are Isabelle's construct for collecting together assumptions and the facts stemming from them. This means that if the data satisfies the locale assumptions, then the facts of the locale are true of the data.

The operations we define are ultimately in service of graphically representing process compositions. However, we keep this part of the theory more general to make it reusable in other contexts.

5.3.1 Ports

Before we state exactly how the data is represented, recall from Section 3.3.1 our definition of (qualified) ports (reproduced in Definition 5.3.1). We will now be using their general definitions to mechanise port graphs that allow for any port sides, labels and qualifying atoms. In Section 5.4.1 we then specialise these type variables in the same way we did for process diagrams in Section 3.3.1 to yield process port graphs. That is, we will use input and output as port sides, resources as port labels and composition tree paths as atoms qualifying the ports.

Isabelle Definition 5.3.1 (Datatypes of ports and qualified ports)

datatype $('s, 'a)$ *port* = *Port* (*side*: $'s$) (*index*: *nat*) (*label*: $'a$)
datatype $('s, 'a, 'p)$ *qualified-port* = *QPort* (*port*: $('s, 'a)$ *port*) (*name*: $'p$ *list*)

Note that names are sequences of atoms rather than simply an arbitrary type. This allows us to further qualify an already qualified port with an additional atom. When done to two collections of ports with two distinct atoms, this allows us to easily make those two collections disjoint. For qualified ports, this operation is defined as follows:

Isabelle Definition 5.3.2 (Further qualifying a qualified port)

primrec *qualifyQPort* :: $'p \Rightarrow ('s, 'a, 'p)$ *qualified-port* $\Rightarrow ('s, 'a, 'p)$ *qualified-port*
where *qualifyQPort* x (*QPort* *port* *path*) = *QPort* *port* (x # *path*)

5.3.2 Nodes, Edges and Places

Nodes, edges and places are the different parts of port graphs. While the first two represent exactly those basic graph components, places represent the union of both ports attached to nodes (which we call *ground* ports) and open ports, which are not attached to a node. In other words, places are what edges can connect.

Nodes are formed from a name (again a sequence of atoms drawn from type $'p$), a label drawn from type $'l$ and the ports attached to the node (themselves drawing sides from type $'s$ and labels from type $'a$). Mechanised in Isabele/HOL:

Isabelle Definition 5.3.3 (Datatype of nodes)

datatype $('s, 'a, 'p, 'l)$ *node* =
Node (*node-name*: $'p$ *list*) (*node-label*: $'l$) (*node-ports*: $('s, 'a)$ *port list*)

Note that in Section 5.4.1, when specifying port graphs for process compositions, we instantiate the node labels to carry the labels and metadata of primitive actions.

Places can be either ground ports, qualified by a node name, or open ports. We mechanise them in Isabelle/HOL as follows (with type variables 's, 'a and 'p again just as with qualified ports):

Isabelle Definition 5.3.4 (Datatype of places)

```
datatype ('s, 'a, 'p) place =
  place-ground: GroundPort (('s, 'a, 'p) qualified-port)
| place-open: OpenPort (('s, 'a) port)
```

The automatically generated predicates *place-ground* and *place-open* allow us to easily distinguish the two kinds of places, which is a frequent point in proofs.

With both nodes and places defined, we can for instance define how all the ground places induced by a node can be collected:

Isabelle Definition 5.3.5 (Collecting ground places of a node)

```
fun nodePlaces :: ('s, 'a, 'p, 'l) node ⇒ ('s, 'a, 'p) place list
 where nodePlaces n =
  map (λp. GroundPort (QPort p (node-name n))) (node-ports n)
```

Edges are made up of two places, the origin and destination, meaning they are simple and directed edges. Mechanised in Isabelle/HOL (with type variables 's, 'a and 'p again just as with qualified ports):

Isabelle Definition 5.3.6 (Datatype of edges)

```
datatype ('s, 'a, 'p) edge =
 Edge (edge-from: ('s, 'a, 'p) place) (edge-to: ('s, 'a, 'p) place)
```

We now illustrate these constructs before using them in port graphs. Consider the *add-to-credit* action from Definition 3.1.3, say with existing credit of zero and adding ten in cash. This action has two input atoms (*Machine 0* and *Cash 10*) and one output atom (*Machine 10*). Let this action be found as the first child under two sequential compositions, such as in the example in Section 3.1.3. While our full process port graph construction uses more complex labels (described in Section 5.4), we could represent this action with the node:

```
Node [SeqL, SeqL]
     STR "Add to credit"
     [Port In 0 (Res (Machine 0)), Port In 1 (Res (Cash 10)),
      Port Out 0 (Res (Machine 10))]
```

and one of its places would then be:

$$GroundPort\ (QPort\ (Port\ In\ 0\ (Res\ (Machine\ 0)))\ [SeqL,\ SeqL])$$

Then, if this node was within a larger port graph that has the following open input port:

$$Port\ In\ 3\ (Res\ (Machine\ 0))$$

we could connect this open port to the node's input with the following edge:

$$Edge\ (OpenPort\ (Port\ In\ 3\ (Res\ (Machine\ 0))))\\ (GroundPort\ (QPort\ (Port\ In\ 0\ (Res\ (Machine\ 0)))\ [SeqL,\ SeqL]))$$

5.3.3 Port Graph Data

With all the constituent parts defined, we can now mechanise the port graph data itself, with sides drawn from type $'s$, port labels drawn from type $'a$, name atoms drawn from type $'p$ and node labels drawn from type $'l$:

Isabelle Definition 5.3.7 (Datatype of port graphs)

datatype $('s, 'a, 'p, 'l)$ *port-graph* =
PGraph (*pg-nodes*: $('s, 'a, 'p, 'l)$ *node list*)
(*pg-edges*: $('s, 'a, 'p)$ *edge list*)
(*pg-ports*: $('s, 'a)$ *port list*)

On this data alone we make several useful definitions: a function to qualify the whole port graph, a function to collect all places occurring in it and a relation for port graphs with no names in common.

The function *qualifyPortGraph* takes a name atom and prefixes it to names of all nodes in the port graph as well as the names qualifying ground places in edges. We define it using a series of functions with the same effect on different targets, called *qualifyNode*, *qualifyEdge* and *qualifyPlace*, culminating in *qualifyPortGraph*. All of these either qualify their target's constituent parts or, in the case of *qualifyNode*, prefix its name with the atom just as *qualifyQPort* in Section 5.3.1 does.

The function *pgraphPlaces* collects all the places occurring in the port graph by combining those attached to its nodes with its open ports. This function figures frequently in proofs about port graphs and is defined as follows:

Isabelle Definition 5.3.8 (Collecting open and ground places of a port graph)

```
fun pgraphPlaces :: ('s, 'a, 'p, 'l) port-graph ⇒ ('s, 'a, 'p) place list
 where pgraphPlaces x =
  concat (map nodePlaces (pg-nodes x)) @
  map OpenPort (pg-ports x)
```

Being able to tell that two port graphs do not have a name in common is useful because such a shared name would interfere when combining them. While this makes a frequent assumption in our verification, it is easy to satisfy by using *qualifyPortGraph* to qualify all names within each port graph with one of two distinct name atoms. We call port graphs satisfying this relation *disjoint* and define it in Isabelle/HOL as follows:

Isabelle Definition 5.3.9 (Disjoint port graphs)

```
definition pg-disjoint :: ('s, 'a, 'p, 'l) port-graph ⇒ ('s, 'a, 'p, 'l) port-graph ⇒ bool
 where pg-disjoint x y =
  (∀m n. m ∈ set (pg-nodes x) ∧ n ∈ set (pg-nodes y)
        ⟶ node-name m ≠ node-name n)
```

5.3.4 Well-Formed Port Graphs

While we can now express the data of a port graph, nothing guarantees that it is sensible. We could for instance include an edge between places that are not in the port graph, or two nodes with the same name.

To address this, we make a number of requirements of the port graph data in order to call it an actual port graph, which we collect into the locale *port-graph* shown in Definition 5.3.10. A locale acts as a predicate, but with more convenient automation in proofs. We list the requirements next, with a brief motivation in each case:

edge-from-pg **and** *edge-to-pg*

> Origin and destination of every edge in the port graph must be places of that port graph.

node-unique-name

> No two nodes of the port graph can have the same name, so we can use the name to identify a node.

ports-index-bound

> Index of every open port of the port graph is less than the total number of open ports on the same side.

open-ports-label-eq

Any two open ports of the port graph that have the same side and index must also have the same label, and thus be the same port.

node-ports-label-eq

Any two ports of a node of the port graph that have the same side and index must also have the same label, and thus be the same port.

nodes-distinct, *edges-distinct* **and** *ports-distinct*

Node, edge and open port lists do not contain duplicates, so that they represent finite sets.

Isabelle Definition 5.3.10 (Well-formed port graphs)

```
locale port-graph =
  fixes G :: ('s, 'a, 'p, 'l) port-graph
  assumes edge-from-pg:
    ⋀e. e ∈ set (pg-edges G) ⟹ edge-from e ∈ set (pgraphPlaces G)
      and edge-to-pg:
    ⋀e. e ∈ set (pg-edges G) ⟹ edge-to e ∈ set (pgraphPlaces G)
      and node-unique-name:
    ⋀m n. ⟦m ∈ set (pg-nodes G); n ∈ set (pg-nodes G); node-name m = node-name n⟧
          ⟹ m = n
      and ports-index-bound:
    ⋀p. p ∈ set (pg-ports G)
        ⟹ port.index p < length (filter (λx. port.side x = port.side p) (pg-ports G))
      and open-ports-label-eq:
    ⋀p q. ⟦p ∈ set (pg-ports G); q ∈ set (pg-ports G);
           port.side p = port.side q; port.index p = port.index q⟧
          ⟹ port.label p = port.label q
      and node-ports-label-eq:
    ⋀n p q. ⟦n ∈ set (pg-nodes G); p ∈ set (node-ports n); q ∈ set (node-ports n);
             port.side p = port.side q; port.index p = port.index q⟧
            ⟹ port.label p = port.label q
      and nodes-distinct: distinct (pg-nodes G)
      and edges-distinct: distinct (pg-edges G)
      and ports-distinct: distinct (pg-ports G)
```

One implication of this locale is that we can define a function that given a ground port retrieves the node in the port graph it is attached to. The good behaviour of this function relies on assumptions of the locale for both the existence and uniqueness of the node. As noted in Section 5.1, some formulations of port graphs in the literature start with such a function instead of making ports part of the node data as we do. We can recover this function from data satisfying our assumptions.

5.3.5 Port Graph With Flow

Beyond the basic requirements, the port graphs used to represent process compositions are even more specific. They have two special sides, input and output, and the direction of edges represents the flow of resources between actions. As such, they should not contain, for instance, edges going into an output port of an action, because that is not a place that can receive a resource. We characterise this through the following extra requirements on well-formed port graphs:

- The possible sides include two specially designated ones, input and output, and
- All edges that touch some input or output place have both:
  - Origin in an open input port or ground output port, and
  - Destination in an open output port or ground input port.

When mechanising these assumptions in Isabelle/HOL we first define a simple type class to generically capture the two designated sides. Type classes [94] are a variant of locales constrained to a single type variable, letting us define constants and assumptions about them, that act as constraints on types. They allow us to sort types and safely overload definitions. In order to use the class constants with a type, we need to show that this type is an instance of the class by proving that it satisfies all assumptions of the class.

Here we use a type class to define convenient syntax for the input and output sides (*In* and *Out*) and capture the implicit assumption that they are distinct. This lets us later instantiate those elements as we please, see for instance Section 5.4.1.

Isabelle Definition 5.3.11 (Typeclass capturing distinct input and output sides)

class *side-in-out* =
  fixes *In* :: $'a$ **and** *Out* :: $'a$
  assumes *in-out-distinct*: *In* $\neq$ *Out*

We then abbreviate the precondition on edges, with *edge-in-flow e* meaning that *e* has either origin or destination with side *In* or *Out*:

Isabelle Definition 5.3.12 (Flow precondition on edges)

definition *edge-in-flow* :: $('s ::$ *side-in-out*, $'a$, $'p)$ *edge* $\Rightarrow$ *bool*
  where *edge-in-flow e* = (*place-side* (*edge-from e*) $\in$ {*In*, *Out*} $\vee$
    place-side (*edge-to e*) $\in$ {*In*, *Out*})

and with it we mechanise the full assumptions as the locale *port-graph-flow*:

Isabelle Definition 5.3.13 (Port graphs with flow)

```
locale port-graph-flow =
  port-graph G for G :: ('s :: side-in-out, 'a, 'p, 'l) port-graph +
  assumes edge-from-open:
    ⟦e ∈ set (pg-edges G); place-open (edge-from e); edge-in-flow e⟧
     ⟹ place-side (edge-from e) = In
       and edge-to-open:
    ⟦e ∈ set (pg-edges G); place-open (edge-to e); edge-in-flow e⟧
     ⟹ place-side (edge-to e) = Out
       and edge-from-ground:
    ⟦e ∈ set (pg-edges G); place-ground (edge-from e); edge-in-flow e⟧
     ⟹ place-side (edge-from e) = Out
       and edge-to-ground:
    ⟦e ∈ set (pg-edges G); place-ground (edge-to e); edge-in-flow e⟧
     ⟹ place-side (edge-to e) = In
```

5.3.6 Equivalence of Port Graphs

Our formalisation of port graphs uses names to identify nodes, and it represents the collections of nodes, edges and ports with lists. This means that two port graphs could differ in nothing but names assigned to one node, or in the order of elements in the lists, and be considered entirely distinct. But we would like to draw a connection between such almost-identical port graphs.

We formalise this connection as an equivalence relation between port graphs, which we denote as $\approx$. For two port graphs to be equivalent we require that either both are ill-formed (i.e. do not satisfy the *port-graph* locale) or they are both well-formed, have the same set of open ports and it is possible to systematically rename nodes of each to give the nodes of the other in an invertible way.

Isabelle Definition 5.3.14 (Witnesses of port graph equivalence)

```
definition pgEquiv-witness :: ('p list ⇒ 'p list) ⇒ ('p list ⇒ 'p list)
                              ⇒ ('s, 'a, 'p, 'l) port-graph ⇒ ('s, 'a, 'p, 'l) port-graph
                              ⇒ bool
  where pgEquiv-witness f g x y ≡
    renameNode f ‘ (set (pg-nodes x)) = set (pg-nodes y) ∧
    set (pg-nodes x) = renameNode g ‘ (set (pg-nodes y)) ∧
    renameEdge f ‘ (set (pg-edges x)) = set (pg-edges y) ∧
    set (pg-edges x) = renameEdge g ‘ (set (pg-edges y)) ∧
    (∀l. l ∈ node-name ‘ set (pg-nodes x) ⟶ g (f l) = l) ∧
    (∀l. l ∈ node-name ‘ set (pg-nodes y) ⟶ f (g l) = l)
```

Isabelle Definition 5.3.15 (Equivalent port graphs)

```
definition pgEquiv :: ('s, 'a, 'p, 'l) port-graph ⇒ ('s, 'a, 'p, 'l) port-graph
                      ⇒ bool (infix ≈ 50)
 where pgEquiv x y ≡
  (¬ port-graph x ∧ ¬ port-graph y) ∨
  ( port-graph x ∧ port-graph y ∧ set (pg-ports x) = set (pg-ports y) ∧
   (∃f g :: 'p list ⇒ 'p list. pgEquiv-witness f g x y))
```

In our Isabelle/HOL mechanisation we use *renameNode* and *renameEdge* to represent the systematic renaming, with each taking a function and using it to update the name contained in a node or both places of an edge. To aid in proof automation, we separate out the conditions on the functions witnessing the equivalence.

Note that in our definition we use variants of $set\ xs = set\ ys$ when comparing the node, edge and port collections. This compares the finite sets the lists represent, thus ignoring element order and any duplicates. Because in the *port-graph* locale we require those collections to have no duplicates, it is only the order that could matter.

This port graph equivalence relates port graphs whose only substantial difference is in the way they name nodes, not in the shape of the graph itself. Node names are only important to uniquely identify them in connections, a fact also used in graph rendering algorithms to identify constituent parts of the visualisation. But the specific choice of those names is not important.

As simple cases, we prove two equivalences. First, qualifying a port graph with a name atom produces a port graph equivalent to the original:

Isabelle Lemma 5.3.1 (Equivalence of port graph qualification)

```
lemma pgEquiv-qualifyPortGraph:
 assumes port-graph x
   shows qualifyPortGraph a x ≈ x
```

Second, a port graph is equivalent to any other port graph that has the same nodes, edges and ports but possibly in different order:

Isabelle Lemma 5.3.2 (Equivalence of permuted port graphs)

```
lemma pgEquiv-permute:
 assumes port-graph x
     and set (pg-nodes x) = set (pg-nodes y)
     and set (pg-edges x) = set (pg-edges y)
     and set (pg-ports x) = set (pg-ports y)
     and distinct (pg-nodes y)
     and distinct (pg-edges y)
     and distinct (pg-ports y)
   shows x ≈ y
```

5.3.7 Simple Example Port Graph

We now give a simple example port graph as an illustration before we move on to operations we define over port graphs in general. This port graph has a single node with incoming and outgoing edges. The general definition in Isabelle/HOL is shown in Definition 5.3.16, with an instance visualised in Figure 5.6. (See Section 5.3.10 for a discussion of how such visualisations can be obtained.)

Isabelle Definition 5.3.16 (Single-node port graph)

```
1 fun nodePortGraph :: 'p list ⇒ 'l ⇒ 'a list ⇒ 'a list
2                      ⇒ ('s :: side-in-out, 'a, 'p, 'l) port-graph
3  where nodePortGraph n l ins outs = PGraph
4   [Node n l (listPorts 0 In ins @ listPorts 0 Out outs)]
5   (map2 Edge (map OpenPort (listPorts 0 In ins))
6              (map (λp. GroundPort (QPort p n)) (listPorts 0 In ins)) @
7    map2 Edge (map (λp. GroundPort (QPort p n)) (listPorts 0 Out outs))
8              (map OpenPort (listPorts 0 Out outs)))
9   (listPorts 0 In ins @ listPorts 0 Out outs)
```

Let us describe this definition in more detail. Line 3 takes in the node name *n*, label *l*, input port data list *ins* and output port data list *outs*, and starts constructing the port graph. Line 4 forms the single node from its name, label and ports formed from the input and output data on the corresponding sides with indices starting at zero. The function *listPorts* uses an initial index and common side to construct consecutive ports from a list of data for them. Lines 5–8 build the edges: line 5 and line 6 form edges from open input ports to node inputs, while line 7 and line 8 form edges from node outputs to open output ports. Finally, line 9 builds the open ports from input and output data. Note that in this simple case open ports match those of the node, which is not the case in general.

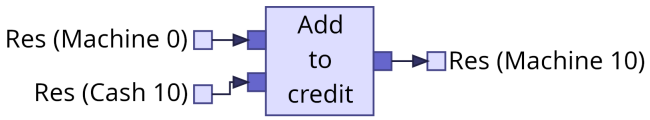


Figure 5.6: Port graph *nodePortGraph* [] *STR ''Add to credit''* [*Res* (*Machine 0*), *Res* (*Cash 10*)] [*Res* (*Machine 10*)], representing an instance of the *add-to-credit* from Definition 3.1.3.

5.3.8 Juxtaposition

Our first complex operation on port graphs is *juxtaposition*, which we will use to represent parallel composition of processes (see Section 5.4.2). Informally, it is very simple: take the nodes, edges and open ports of two port graphs and put them into one. The formal definition, however, needs to be careful about potential clashes when unifying that data.

The first possible clash, and the easiest to solve, is over nodes of the two input port graphs. We simply assume that the port graphs are disjoint, as characterised by the *pg-disjoint* relation defined in Section 5.3.3. To ensure the latter is satisfied before we perform this operation, we can use distinct name atoms to qualify the two port graphs apart (we do so in Section 5.4.2).

The second possible clash involves the open ports. Because these ports are identified by their side and the index within, we can avoid the clash by shifting the indices of open ports from one of the input port graphs. We choose to shift those of the second input, increasing the index of every open port coming from it by the number of open ports on that side coming from the first input. Since, in a well-formed port graph (see Section 5.3.4), the number of ports on a side is the upper bound for their index, we know shifting up by that amount will avoid any clash. Note that this change needs to be done not just in the open ports themselves, but also in any edges that may be adjacent to them.

The last possible clash is with the edges. However, because edges are pairs of places, which in turn are either ports of nodes or open ports, any potential clashes in them will be taken care of by the adjustments we make to nodes and open ports.

With these considerations in mind, our definition of juxtaposition of port graphs in Isabelle/HOL is as follows:

Isabelle Definition 5.3.17 (Juxtaposing port graph)

```
fun juxtapose :: ('s :: side-in-out, 'a, 'p, 'l) port-graph ⇒ ('s, 'a, 'p, 'l) port-graph
                 ⇒ ('s, 'a, 'p, 'l) port-graph
 where juxtapose x y = PGraph
  ( pg-nodes x @ pg-nodes y)
  ( pg-edges x @
   map (shiftOpenInEdge (λs. length (filter (λp. port.side p = s) (pg-ports x)))
                        (λs. length (filter (λp. port.side p = s) (pg-ports x))))
       (pg-edges y))
  ( pg-ports x @
   map (shiftPort (λs. length (filter (λp. port.side p = s) (pg-ports x))))
       (pg-ports y))
```

where *shiftPort* and *shiftOpenInEdge* shift indices of ports, either directly or applying to any open ports contained in an edge. The amount each port is shifted by is a function of that port's side, allowing for more concise statements.

Figure 5.7 illustrates juxtaposition of two simple port graphs representing primitive actions (see the example introduced in Figure 5.6). The resulting port graph now has three open input ports and three open output ports. Consider, for instance, the open output port labelled with *Res D*. It starts with index *0* in its original port graph but has index *1* after juxtaposition due to the open output port in the other port graph. In Section 5.4.2, where we use juxtaposition to represent parallel composition of processes, we also ensure the constituent port graphs are disjoint by qualifying them with distinct name atoms.

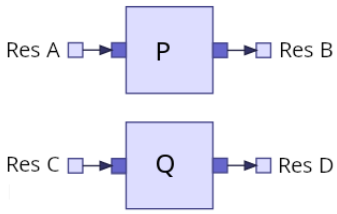


Figure 5.7: Juxtaposition of two port graphs, which we use to represent parallel composition (see Section 5.4.2).

The first property we verify is that this operation preserves the port graph locales we defined. We state this as the following two theorems, each proven by proving the relevant locale's assumptions:

Isabelle Lemma 5.3.3 (Juxtaposition preserves well-formedness)

```
lemma port-graph-juxtapose:
  fixes x y ::
    ('s, 'a, 'p, 'l) port-graph
  assumes port-graph x
      and port-graph y
      and pg-disjoint x y
    shows port-graph (juxtapose x y)
```

```
lemma port-graph-flow-juxtapose:
  fixes x y ::
    ('s :: side-in-out, 'a, 'p, 'l) port-graph
  assumes port-graph-flow x
      and port-graph-flow y
      and pg-disjoint x y
    shows port-graph-flow (juxtapose x y)
```

Next we verify that the operation is associative up to the port graph equivalence. We prove this according to the definition of port graph equivalence (Definition 5.3.15): both sides are well-formed port graphs with the same set of open ports, and with nodes and edges related by renaming. By assuming that the corresponding component port graphs are equivalent, we know that there are renaming functions between them. We use these to form the combined renaming function and show that it relates the nodes

and edges correctly. By assuming that the component port graphs are disjoint, we can determine the source of every node in the two new port graphs and, by extension, the appropriate renaming to apply. We state the theorem as follows:

Isabelle Lemma 5.3.4 (Juxtaposition is associative up to equivalence)

lemma *juxtapose-assoc-pgEquiv*:
fixes $x\ y\ z$ :: $('s, 'a, 'p, 'l)$ *port-graph*
assumes *port-graph* x **and** *port-graph* y **and** *port-graph* z
and *pg-disjoint* $x\ y$ **and** *pg-disjoint* $y\ z$ **and** *pg-disjoint* $x\ z$
and *port-graph* x' **and** *port-graph* y' **and** *port-graph* z'
and *pg-disjoint* $x'\ y'$ **and** *pg-disjoint* $y'\ z'$ **and** *pg-disjoint* $x'\ z'$
and $x \approx x'$ **and** $y \approx y'$ **and** $z \approx z'$
shows *juxtapose* (*juxtapose* $x\ y$) $z \approx$ *juxtapose* x' (*juxtapose* $y'\ z'$)

Finally, we verify that juxtapositions of equivalent port graphs are themselves equivalent. The proof takes the same general form as the associativity proof, being again a proof of equivalence using the same operation, but is rendered simpler by considering fewer port graphs (four instead of six). We state the theorem as follows:

Isabelle Lemma 5.3.5 (Juxtapositions of equivalent port graphs are equivalent)

lemma *juxtapose-resp*:
fixes $x\ y$ :: $('s, 'a, 'p, 'l)$ *port-graph*
assumes *port-graph* x **and** *port-graph* y **and** *pg-disjoint* $x\ y$
and *port-graph* x' **and** *port-graph* y' **and** *pg-disjoint* $x'\ y'$
and $x \approx x'$ **and** $y \approx y'$
shows *juxtapose* $x\ y \approx$ *juxtapose* $x'\ y'$

5.3.9 Sequencing

As our second complex operation on port graphs we define their *sequencing*, which makes use of port graph inputs, outputs and the flow between them. We use this operation to represent sequential composition of processes (see Section 5.4.2). Sequencing goes further than juxtaposition, not just putting the port graphs together but also making new connections and removing other superseded elements. Informally, on top of combining the port graphs into one, this operation: takes the open output ports of one port graph, matches them with open input ports of the other and "stitches" together the edges incident on them. While doing this, it also applies the same methods for avoiding clashes as juxtaposition (i.e. assuming disjoint node names and open port shifting).

The core complexity of this operation is forming the new edges to represent the stitched connection in the result. For this we define the helper function *seqInterfaceEdges* on the two input port graphs x and y which:

1. Forms a mapping from open output ports of x to the edges in x going into them.
2. Forms a mapping from open input ports of y to the edges in y coming from them.
3. For each open output port of x, uses the mappings to retrieve the edge lists to be stitched: those of x going into that port and those of y coming from the matching open input port of y.
4. For each such pair of edge lists, form an edge for each combination going from the origin in x to the destination in y.

The definition of *seqInterfaceEdges* uses the mechanisation of mappings available in Isabelle/HOL, which represents them as functions from a key type to an optional value type. For the purposes of code generation, this representation can be automatically mapped to a list of key-value pairs. The full definition is given in Appendix A.2.

In our proofs we mainly interact with this function through two rules: one introducing the fact that an edge is one of its results (Lemma 5.3.6) and the other decomposing this fact back to its necessary premises (Lemma 5.3.7). Both of these tie an edge resulting from *seqInterfaceEdges* to the existence of two edges, one in each input port graph, that "meet" at matching open ports and which, when combined, yield that edge. Note that, since properties we wish to prove (such as the locales in Definition 5.3.10 and Definition 5.3.13) disregard order of the edges, so do these two rules. This significantly simplifies them in comparison to the function's implementation with mappings described above.

Isabelle Lemma 5.3.6 (Introduction rule for elements of *seqInterfaceEdges*)

```
lemma seqInterfaceEdges-setI:
  fixes x y :: ('s :: side-in-out, 'a, 'p, 'l) port-graph
  assumes port-graph-flow x
    and port-graph-flow y
    and edge-from e = edge-from a
    and a ∈ set (pg-edges x)
    and edge-to e = edge-to b
    and b ∈ set (pg-edges y)
    and edge-to a = OpenPort (portSetSide Out p)
    and edge-from b = OpenPort (portSetSide In p)
    and port.side p = Out
  shows e ∈ set (seqInterfaceEdges x y)
```

Isabelle Lemma 5.3.7 (Destruction rule for elements of *seqInterfaceEdges*)

```
lemma seqInterfaceEdges-setD:
    fixes x y :: ('s :: side-in-out, 'a, 'p, 'l) port-graph
  assumes e ∈ set (seqInterfaceEdges x y)
   obtains a b and p :: ('s, 'a) port
    where edge-from e = edge-from a
      and a ∈ set (pg-edges x)
      and edge-to e = edge-to b
      and b ∈ set (pg-edges y)
      and edge-to a = OpenPort (portSetSide Out p)
      and edge-from b = OpenPort (portSetSide In p)
      and port.side p = Out
```

With the edges that stitch together the output-input connection, we no longer need the original edges or the open ports used. To remove the open ports we simply filter them out. To remove the edges we use the helper function *disconnectFromPlaces ps es* which filters the edges *es* to remove any that are incident on a place in *ps*.

Isabelle Definition 5.3.18 (Removing edges adjacent on given places)

```
definition disconnectFromPlaces :: ('s, 'a, 'p) place list ⇒ ('s, 'a, 'p) edge list
                                   ⇒ ('s, 'a, 'p) edge list
  where disconnectFromPlaces ps es =
   filter (λe. edge-from e ∉ set ps ∧ edge-to e ∉ set ps) es
```

Note that these removals need to be taken into account when shifting open ports (and their occurrences in edges) to avoid clashes: one port graph contributes no output and the other contributes no inputs, so only ports on other sides need to be shifted up.

Our definition of port graph sequencing in Isabelle/HOL is shown in Definition 5.3.19. To form the nodes of the result, it simply appends the two lists (line 4). The edges of the result are split into three groups: first are the new edges forming the connection between the two port graphs (line 5), then the edges of the first port graph except for those that were going to its open outputs (lines 6–8), and finally the edges of the second port graph except for those coming from its open inputs and with the appropriate shifting of indices applied (lines 9–16). This shifting of indices applies to open ports that are neither inputs nor outputs, and ensures that there can be no clash in open ports coming from the two input port graphs. Finally, the ports of the result are also split into three groups: all non-output ports of the first port graph (line 17), all output ports of the second port graph (line 18), and all other ports of the second port graph with their indices shifted in the same way (lines 19–20).

Isabelle Definition 5.3.19 (Sequencing of port graphs)

```
fun seqPortGraphs :: ('s :: side-in-out, 'a, 'p, 'l) port-graph ⇒ ('s, 'a, 'p, 'l) port-graph
                     ⇒ ('s, 'a, 'p, 'l) port-graph
 where seqPortGraphs x y = PGraph
  ( pg-nodes x @ pg-nodes y)
  ( seqInterfaceEdges x y @
   disconnectFromPlaces
     (map OpenPort (filter (λx. port.side x = Out) (pg-ports x)))
     (pg-edges x) @
   map (shiftOpenInEdge
           (λs. if s = In ∨ s = Out then 0
                else length (filter (λx. port.side x = s) (pg-ports x)))
           (λs. if s = In ∨ s = Out then 0
                else length (filter (λx. port.side x = s) (pg-ports x))))
       (disconnectFromPlaces
          (map OpenPort (filter (λx. port.side x = In) (pg-ports y)))
          (pg-edges y)))
  ( filter (λx. port.side x ≠ Out) (pg-ports x) @
   filter (λx. port.side x = Out) (pg-ports y) @
   map (shiftPort (λs. length (filter (λp. port.side p = s) (pg-ports x))))
       (filter (λx. port.side x ≠ In ∧ port.side x ≠ Out) (pg-ports y)))
```

Figure 5.8 illustrates sequencing of the same simple port graphs as our example of juxtaposition in Figure 5.7. The resulting port graph now only has one open input port and one open output port. The open ports labelled *Res B* and *Res C* in the original port graphs are stitched together by *seqInterfaceEdges* to form two edges between ground ports instead. As with juxtaposition, in Section 5.4.2 we qualify the constituent port graphs to ensure they are disjoint.

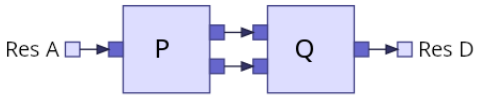


Figure 5.8: Sequencing of two port graphs, which we use to represent sequential composition (see Section 5.4.2).

Once again, the first property we verify is that this operation preserves the port graph locales. In this case, however, even proving that the base locale remain satisfied requires facts from the flow locale. For instance, if an edge is part of the stitched connection, then its origin is only guaranteed to be in the port graph because we know it could not have been one of the open output ports that were removed. More complex situations requiring facts about the flow arise as part of proving that this process does

not introduce any duplicate edges. As such, we state this as a single theorem, albeit in almost the same form as the one for juxtaposition:

Isabelle Lemma 5.3.8 (Sequencing port graphs preserves well-formedness)

```
lemma port-graph-flow-seqPortGraphs:
    fixes x y :: ('s :: side-in-out, 'a, 'p, 'l) port-graph
  assumes port-graph-flow x
      and port-graph-flow y
      and pg-disjoint x y
    shows port-graph-flow (seqPortGraphs x y)
```

The other two properties, namely that the operation is associative and respects port graph equivalence, are stated in the same form as those for juxtaposition except for the port graphs needing to satisfy the stronger *port-graph-flow* locale. It is the proofs of these properties that are made more complex due to the operation being more complex. For instance, when proving that edges of the two resulting port graphs are related by renaming, we need to consider more cases: an edge could come from any of the constituent port graphs or be the result of one of the stitched interfaces. In each case we need to find the corresponding edge in the other resulting port graph and show that the combined renaming function indeed relates them.

Isabelle Lemma 5.3.9 (Sequencing is associative up to equivalence)

```
lemma seqPortGraphs-assoc-pgEquiv:
    fixes x y z :: ('s :: side-in-out, 'a, 'p, 'l) port-graph
  assumes port-graph-flow x and port-graph-flow y and port-graph-flow z
      and pg-disjoint x y and pg-disjoint y z and pg-disjoint x z
      and port-graph-flow x' and port-graph-flow y' and port-graph-flow z'
      and pg-disjoint x' y' and pg-disjoint y' z' and pg-disjoint x' z'
      and x ≈ x' and y ≈ y' and z ≈ z'
    shows seqPortGraphs (seqPortGraphs x y) z ≈
          seqPortGraphs x' (seqPortGraphs y' z')
```

Isabelle Lemma 5.3.10 (Sequencings of equivalent port graphs are equivalent)

```
lemma seqPortGraphs-resp:
    fixes x y :: ('s :: side-in-out, 'a, 'p, 'l) port-graph
  assumes port-graph-flow x and port-graph-flow y and pg-disjoint x y
      and port-graph-flow x' and port-graph-flow y' and pg-disjoint x' y'
      and x ≈ x' and y ≈ y'
    shows seqPortGraphs x y ≈ seqPortGraphs x' y'
```

5.3.10 Port Graph Export

Our mechanisation allows us to specify concrete port graphs, manipulate them and prove their properties. However, being firmly in the algebraic environment of Isabelle, we cannot directly visualise them. That means we miss out on a significant advantage offered by their graphical nature to human comprehension.

Outside the proof assistant there exist tools which, given the data describing a port graph, lay the individual elements out on a plane and turn it into an image. We take advantage of this by mechanising a function from port graphs to the data such a tool needs.

We target the Eclipse Layout Kernel (ELK) [26], which can represent (hierarchical) port graphs and implements a range of graph layout algorithms, and Eclipse Sprotty [1], which is a web-based diagramming framework compatible with ELK. Their integration is shown by the online ELK Demonstrators [2].

Within Isabelle/HOL, we use the existing mechanisation of JSON due to Brucker [20] to define a function from port graphs into an ELK-compatible JSON object. While most of the implementation (given in full in Appendix A.3) consists of directly turning each aspect of our port graphs into the ELK JSON counterpart, we also need to convert the type variables port graphs use into a form comprehensible to ELK. As such, we parameterise the conversion function with the following domain-dependent functions:

- Mapping of port graph sides to the five port sides that ELK supports: `UNDEFINED`, `NORTH`, `EAST`, `SOUTH`, `WEST`. For instance, with process port graphs we map *In* to `WEST` and *Out* to `EAST`.
- Mapping of node names to string literals, so they can be used as components in unique identifiers. For instance, with process port graphs we simply print the path to the relevant action in the composition tree to uniquely identify it.
- Mapping of node labels to string literals, so they can be used as node labels in the visualisation. For instance, with process port graphs we use the action's label.

Once instantiated for a specific type of port graphs, such as process port graphs, the conversion function produces formal JSON objects. These we then turn into text and visualise using the online ELK JSON Demonstrator [3]. We use this method to visualise

[1] `https://projects.eclipse.org/projects/ecd.sprotty`
[2] `https://rtsys.informatik.uni-kiel.de/elklive/`
[3] `https://rtsys.informatik.uni-kiel.de/elklive/json.html`

the port graphs in the present chapter, but in Section 5.8 we note the potential for a more closely tailored tool.

5.3.11 Summary

We have mechanised a self-contained theory of port graphs, which we will use in the following sections to graphically reason about process compositions. It includes the data representing port graphs, constraints on that data and two significant port graph compositions.

While certainly influenced by our intention, we believe this theory is sufficiently general to be of use for formal verification in other contexts. As noted in Section 5.1, port graphs have been used to model biochemical processes and social networks, and are connected to graphical languages used, for instance, to reason about quantum circuits.

In the next section we specialise this general theory to our case of graphical representation of process compositions and use the language of port graphs to prove their various properties. This perspective is particularly useful when talking about resource connections between actions.

5.4 Process Port Graphs

Turning process compositions into port graphs, we use nodes to represent primitive actions and edges to represent the resource connections between those actions. These resource connections are the result of composition operations and resource actions, which we represent with port graph operations and node-less port graphs respectively.

Note that with process port graphs we can use the tools described in Section 5.3.10 to visualise process compositions. This differs in several ways from the process diagrams of Section 3.3. First, as discussed in more detail in Section 5.4.2, our construction only applies to a subset of process compositions while the process diagrams apply to all valid compositions. However, second, the process port graphs tie compositions *formally* to the graphical representation, allowing us to prove their properties, while process diagrams are informal. Third, where both visualisations exist they will often differ in their layout, since the layout of port graphs is determined by an algorithm based on the present connections while the layout of process diagrams is determined naïvely from the composition structure. But the connections between actions that both

visualisations represent will be the same.

We start this section by discussion of how we specialise the general type of port graphs to suit our purposes. Then we describe the translation itself and continue on to its verified properties. In the following section we will discuss our main theorem stemming from this translation, a graphical proof of process linearity.

5.4.1 Preliminaries

Before we can even state how process compositions relate to port graphs, we need to instantiate the type variables of port graphs to specify the sides, name atoms, node labels and port labels. We give these instantiations relative to the process composition type variables: $'a$ and $'b$ for linear and copyable resource atoms respectively, and $'l$ and $'m$ for primitive process label and metadata respectively. Note that, with regards to ports, the instantiation matches that of Section 3.3.1 for process diagrams.

To represent process compositions we only need two sides: input and output. We reuse our formalisation of these sides from Section 3.3.1 and prove it to be an instance of the *side-in-out* type class from Section 5.3.5.

Isabelle Definition 5.4.1 (Datatype of process sides, an instance of *side-in-out*)

```
datatype process-side = In | Out
instantiation process-side :: side-in-out
begin
  definition In = process-side.In
  definition Out = process-side.Out
  instance by standard (simp add: In-process-side-def Out-process-side-def)
end
```

Nodes in this case represent primitive actions, which can be uniquely identified in a process composition by the path to that leaf. See Section 3.3.1 for our earlier discussion of these paths and their formalisation, which we reuse here (reproduced in Definition 5.4.2. As such, the name atoms we use are the type *process-inner*.

Isabelle Definition 5.4.2 (Components for composition tree paths)

```
datatype process-inner = SeqL | SeqR | ParL | ParR | OptL | OptR | Rep
```

Furthermore, because the only nodes are primitive actions, we annotate them with the extra data of those actions. For this we define a new datatype, $('l, 'm)$ *node-content*, to collect together an action's label and metadata:

Isabelle Definition 5.4.3 (Datatype holding node content)

datatype ('l, 'm) *node-content* = *NodePrimitive* ('l) ('m)

Ports represent parallel parts of the input and output resources of an action, so we annotate them with the relevant resources. See our discussion of ports in the context of process diagrams in Section 3.3.1. As such, the port labels we use are the type ('a, 'b) *resource*.

All together, the specific type of port graphs we use is the following:

(*process-side*, ('a, 'b) *resource*, *process-inner*, ('l, 'm) *node-content*) *port-graph*

which we abbreviate as:

('a, 'b, 'l, 'm) *process-port-graph*

5.4.2 Process Port Graph Construction

We construct port graphs from processes by associating each action with a port graph template and each composition operation with an operation on port graphs, as shown in Definition 5.4.4. We follow it with more detailed discussion of its cases.

Isabelle Definition 5.4.4 (Constructing a port graph from a process composition)

```
primrec pgConstruct :: ('a, 'b, 'l, 'm) process ⇒ ('a, 'b, 'l, 'm) process-port-graph
 where
  pgConstruct (Primitive ins outs l m) =
  nodePortGraph [] (NodePrimitive l m) (parallel-parts ins) (parallel-parts outs)
| pgConstruct (Seq p q) =
  seqPortGraphs (qualifyPortGraph SeqL (pgConstruct p))
                (qualifyPortGraph SeqR (pgConstruct q))
| pgConstruct (Par p q) =
  juxtapose (qualifyPortGraph ParL (pgConstruct p))
            (qualifyPortGraph ParR (pgConstruct q))
| pgConstruct (Identity a) = idPortGraph (parallel-parts a)
| pgConstruct (Swap a b) = swapPortGraph (parallel-parts a) (parallel-parts b)
| pgConstruct (Duplicate a) = forkPortGraph (Copyable a)
| pgConstruct (Erase a) = endPortGraph [Copyable a]
| pgConstruct (Repeat a b) = forkPortGraph (Repeatable a b)
| pgConstruct (Close a b) = endPortGraph [Repeatable a b]
| pgConstruct (Once a b) = oncePortGraph a b
| pgConstruct (Forget a) = forgetPortGraph a
```

Note that we provide no definitions for the non-deterministic and higher-order cases, as those cannot be captured faithfully by our present theory of port graphs.

While we use port indices to represent parallel resources, a more complicated approach would be needed to also represent non-deterministic combinations of resources in a way that works well with optional composition and the relevant resource actions. For instance, we may want the graphical representation of an injection action followed by optional composition to result in just the corresponding branch of the composition being reachable. In the case of representing a composition as a repeatable executable resource, we would need at least hierarchical port graphs to allow nodes to contain whole port graphs. As such, expanding the construction to all compositions is part of future work, as described in Section 5.8).

In the primitive action case, we make use of the simple single-node port graph previously discussed in Section 5.3.7 and shown again in Figure 5.9. Because this action by itself is the root of its own composition tree, we use the empty name [] for the node. The label we give it is the action's label and metadata, while the input and output port data lists are the parallel parts of the input and output resources.

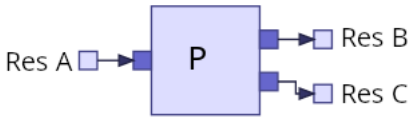
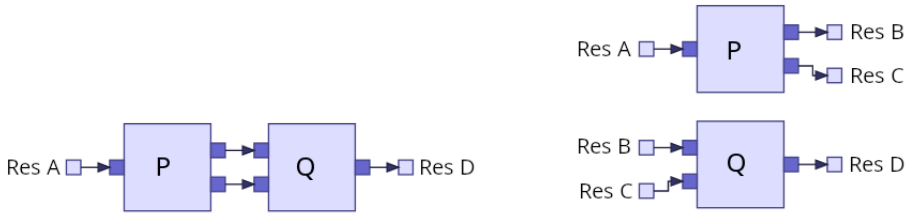


Figure 5.9: Port graph *Primitive* (*Res A*) (*Res B* ⊙ *Res C*) *STR* ″*P*″ ()

For the sequential and parallel composition cases, we make use of the port graph sequencing (see Section 5.3.9) and juxtaposition (see Section 5.3.8) respectively. We first recursively construct the port graphs of the child processes and qualify them with distinct name atoms to prevent an overlap in the names they contain. Then we apply the relevant operation to the resulting port graphs, which takes care of all other aspects given disjoint port graphs. Instances for two primitive actions composed in sequence and in parallel are shown in Figure 5.10.

For *Identity* we build a port graph consisting of no node and a set of edges, one for each parallel part of the relevant resource going between two open ports, one input and one output. We abstract this pattern in general port graphs for any list of data (in our case the parallel parts of a resource) and call it *idPortGraph*. Its instance for the three-atom resource *Res A* ⊙ *Res B* ⊙ *Res C* is shown in Figure 5.11.

For *Swap* we build again a port graph consisting of no node and a set of edges, just like for *Identity*, using parallel parts of the two resources being swapped. But, in this case, we change the indices of the open output ports to reflect the swapped order



(a) Sequential composition (b) Parallel composition

Figure 5.10: Process port graphs for sequential and parallel composition of primitive actions P: $Res\ A \to Res\ B \odot Res\ C$ and Q: $Res\ B \odot Res\ C \to Res\ D$

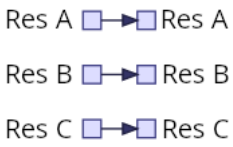


Figure 5.11: Process port graph for *Identity* $(Res\ A \odot Res\ B \odot Res\ C)$

of the resources. We call the resulting pattern in general port graphs *swapPortGraph*, parameterised by two lists of data for the ports.

Note that, during the layout step of visualising this port graph on its own, the different order stops being visually apparent due to the layout algorithm's goal of minimising edge crossings. However, as shown in Figure 5.12, the reordering becomes apparent when we fix the port positions using primitive action nodes.

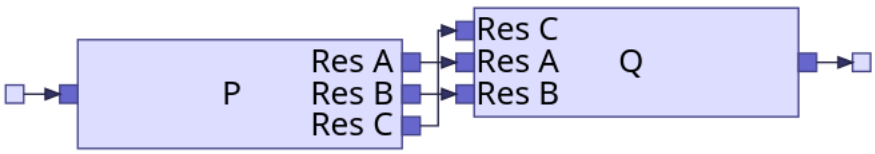


Figure 5.12: Process port graph involving *Swap* $(Res\ A \odot Res\ B)\ (Res\ C)$

For *Duplicate* and *Repeat* we use the same pattern: we build a port graph consisting of no node and one input port with two edges going from it to two output ports. We call this pattern *forkPortGraph*, parameterised by the single piece of data for all its open ports. Its instance for duplicating a copyable atom *data* is shown in Figure 5.13.

For *Erase* and *Close* we use a very simple pattern: a single input port with no edges or output ports. This represents closing off whatever output port this single input is composed with. Its instance for erasing a copyable atom *data* is shown in Figure 5.14.

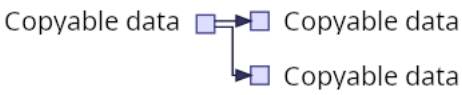


Figure 5.13: Process port graph for *Duplicate data*

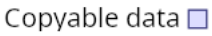


Figure 5.14: Process port graph for *Erase data*

For *Once* we use a pattern similar to *Identity* but, because of how its input and output interact with *parallel-parts*, there is only one edge. Moreover, the origin and destination of that edge carry different resources as data: the origin carries a *Repeatable* resource while the destination carries the corresponding *Executable* resource. Note that this is the first port graph pattern we use in which the labels on origin and destination of an edge differ. Its instance for a repeatable process from x to y is shown in Figure 5.14.

Figure 5.15: Process port graph for *Once* x y

For *Forget* we build a port graph consisting of no node, a number of input ports and a single output port, and edges from every input to the sole output. We use parallel parts of the input resource to label the input ports and the output *Anything* resource to label the output port. This represents any complex resource merging into a single *Anything* resource, forgetting any details about it including that it may have consisted of multiple parts. Note that in this port graph we also may have edges with origin and destination labelled differently. Its instance for the resources *Res A*, *Res B* and *Res C* is shown in Figure 5.16.

This concludes the patterns we use to generate port graphs from process compositions. Figure 5.17 illustrates the port graph construction on a more complex process composition, the manufacturing of four iron gears per second defined in Section 7.4.6. We next turn to the properties that our process port graphs satisfy.

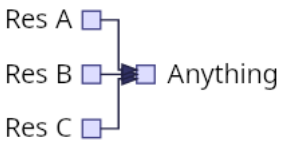


Figure 5.16: Process port graph for *Forget Res A* ⊙ *Res B* ⊙ *Res C*

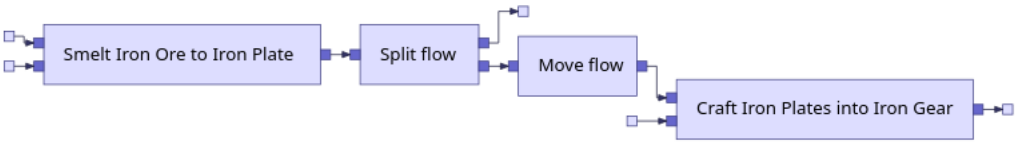


Figure 5.17: Process port graph for manufacturing four iron gears per second (see Section 7.4.6 for details of this process). We omit open port labels due to their size.

5.4.3 Properties of Process Port Graphs

Before we discuss specific properties, recall that our process port graph construction is not defined for processes that make use of non-deterministic or higher-order features. To exclude such processes from consideration, we define the predicate *pgDefined* (see Appendix A.4 for its full definition). We then assume that any process used in *pgConstruct* satisfies this condition.

We start our verification by proving that our construction, given a composition it is fully defined on, results in a well-formed port graph with flow. Proving that each of the port graph patterns we use is well-formed and satisfies the flow requirements is quite simple, because having concrete port graphs simplifies much of the quantification in the locale assumptions. The cases for parallel and sequential composition are simple, because we prove that both port graph juxtaposition and sequencing satisfy the *port-graph-flow* locale just after defining them (Lemma 5.3.3 and Lemma 5.3.8). Therefore we now only need to prove that qualifying the child port graphs with distinct atoms (*SeqL* and *SeqR*, or *ParL* and *ParR*) indeed makes them disjoint, which is easily done. As a result, we have the following theorem in Isabelle/HOL:

Isabelle Lemma 5.4.1 (Process port graphs are well-formed)

```
lemma port-graph-flow-pgConstruct:
  assumes pgDefined x
    shows port-graph-flow (pgConstruct x)
```

Then we show that the open ports of the constructed port graph correspond to

parallel parts (given in Definition 3.3.2) of the process composition's input and output resources:

Isabelle Lemma 5.4.2 (Open ports correspond to input and output)

```
lemma pgConstruct-ports:
  assumes pgDefined x
    shows set (pg-ports (pgConstruct x)) =
          set (parallelPorts 0 In (input x) @ parallelPorts 0 Out (output x))
```

Moreover, the nodes of the port graph also correspond to the primitive actions present in the process composition. To prove this, we define the function *namedPrimitives* that operates exactly as the function *primitives* (see Section 3.1.1) but also collects the path to each primitive action. We show that the path is the node name, the action's label and metadata are the node label, and the action's input and output form the node ports:

Isabelle Lemma 5.4.3 (Nodes correspond to primitive actions)

```
lemma pgConstruct-nodes:
  assumes pgDefined x
    shows pg-nodes (pgConstruct x) =
          map (λ(n, ins, outs, l, m).
                 Node n (NodePrimitive l m) (parallelPorts 0 In ins @
                                             parallelPorts 0 Out outs))
              (namedPrimitives x)
```

The remainder of the properties we highlight in this section concern situations where different process compositions yield equivalent port graphs. These capture a deeper aspect of process compositions than the surface-level syntax: they say that two compositions, although syntactically different, represent the same structure of connections between actions. In Section 5.8 we discuss future work taking further advantage of this aspect of the equivalence.

To start with, we can show that, up to equivalence of constructed port graphs, the identity process behaves as unit for both sequential and parallel composition:

Isabelle Lemma 5.4.4 (Identity process is sequential unit)

```
lemma pgConstruct-seq-unit-pgEquiv:
  assumes pgDefined x
    shows pgConstruct (Seq (Identity (input x)) x) ≈ pgConstruct x
      and pgConstruct (Seq x (Identity (output x))) ≈ pgConstruct x
```

Isabelle Lemma 5.4.5 (Identity process is parallel unit)

```
lemma pgConstruct-par-unit-pgEquiv:
 assumes pgDefined x
   shows pgConstruct (Par (Identity Empty) x) ≈ pgConstruct x
     and pgConstruct (Par x (Identity Empty)) ≈ pgConstruct x
```

We can also take the theorems about when port graph sequencing and juxtaposition are associative and show that the process port graphs satisfy their assumptions. This is because qualifying port graphs with distinct name atoms makes them disjoint while being equivalent to the original. As a result, we get the following two theorems:

Isabelle Lemma 5.4.6 (Parallel and sequential composition are associative)

```
lemma
 assumes pgDefined x and pgDefined y and pgDefined z
   shows pgConstruct-Par-assoc:
         pgConstruct (Par (Par x y) z) ≈ pgConstruct (Par x (Par y z))
     and pgConstruct-Seq-assoc:
         pgConstruct (Seq (Seq x y) z) ≈ pgConstruct (Seq x (Seq y z))
```

And we can show that the *Duplicate* action forms a monoid in this way with the *Erase* action as unit: erasing either result of duplication is as if we did nothing, and nesting duplications is associative. Because the corresponding port graphs have no nodes that would require renaming, the relations can be shown as *equalities* of port graphs instead of just equivalences. All three statements are as follows:

Isabelle Lemma 5.4.7 (*Duplicate* and *Erase* form a monoid)

```
lemma
 shows pgConstruct-duplicate-eraseL:
  pgConstruct (Seq (Duplicate x) (Par (Erase x) (Identity (Copyable x)))) =
  pgConstruct (Identity (Copyable x))
   and pgConstruct-duplicate-eraseR:
  pgConstruct (Seq (Duplicate x) (Par (Identity (Copyable x)) (Erase x))) =
  pgConstruct (Identity (Copyable x))
   and pgConstruct-duplicate-assoc:
  pgConstruct (Seq (Duplicate x) (Par (Identity (Copyable x)) (Duplicate x))) =
  pgConstruct (Seq (Duplicate x) (Par (Duplicate x) (Identity (Copyable x))))
```

Increasing the statement complexity, we can prove that any process that composes like an identity action produces a port graph equivalent to it. More precisely, by composing like an identity action we mean that (i) sequential composition of the candidate process with itself is valid, and (ii) for any other process whose sequential composition with the candidate is valid the result produces a port graph equivalent to that other

process. This yields two theorems, Lemma 5.4.8 and Lemma 5.4.9, which differ in the ordering of the sequential composition and thus in the identity action using the input or output resource.

Isabelle Lemma 5.4.8 (Left unit of *Seq* is equivalent to an identity on input)

```
lemma pgConstruct-ide-input:
  assumes pgDefined x
      and valid (Seq x x)
      and ⋀f. ⟦pgDefined f; valid (Seq f x)⟧
              ⟹ pgConstruct (Seq f x) ≈ pgConstruct f
    shows pgConstruct x ≈ pgConstruct (Identity (input x))
```

Isabelle Lemma 5.4.9 (Right unit of *Seq* is equivalent to an identity on output)

```
lemma pgConstruct-ide-output:
  assumes pgDefined x
      and valid (Seq x x)
      and ⋀f. ⟦pgDefined f; valid (Seq x f)⟧
              ⟹ pgConstruct (Seq x f) ≈ pgConstruct f
    shows pgConstruct x ≈ pgConstruct (Identity (output x))
```

In the following two sections we turn to two statements about processes that this new graphical perspective allows us to prove. One describes when parallel and sequential composition of processes can pass through each other without affecting the connections between actions. The other characterises the range of connections that can occur in valid process compositions, serving as a graphical counterpart to our demonstration of linearity in Chapter 4.

5.5 Process Interchange

In general, parallel and sequential composition of processes do not distribute over each other. (For a more thorough discussion of behaviour one expects from concurrent programs and process calculi, see Hoare and van Staden [38].) In our notation, this means we would expect *Seq* (*Par R S*) (*Par T U*) to have more constrained behaviour than *Par* (*Seq R T*) (*Seq S U*) (when they are both valid compositions). This is because, in a direct reading, the latter form allows for more interleaving of the processes: the former stipulates that we do R and S before T and U, while the latter allows us to do T before S as long as it follows R. In practice, this may for instance be the case if U depends on R in order to execute.

However, for both forms to be valid in our framework we must have *output* R = *input* T and *output* S = *input* U. Note that this means there is no resource passing from R to U and from S to T. And, because resources in our framework model dependency between actions, we would expect to be able to do T without first doing S. Because through resources in our framework we have this extra information about the dependency between actions, we would like to equate the two forms when both are valid.

We cannot equate the two forms in themselves, because they are different compositions of processes. But, because our claim about their close relation is based in connections, we can relate them through the port graphs they generate. A simple indication of this is that, when both compositions are valid, our implementation of process diagrams (see Section 3.3) draws the same diagram for both, shown in Figure 5.18.

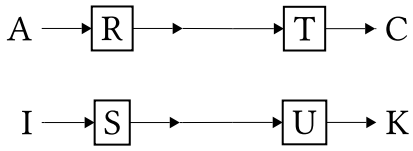


Figure 5.18: Common process diagram for the compositions *Seq* (*Par R S*) (*Par T U*) and *Par* (*Seq R T*) (*Seq S U*)

With our process port graphs, we can make this relation fully formal: for any four processes, if both the composition forms are valid then the port graphs resulting from them are equivalent. Note that validity of one subsumes validity of the other, so we need only assume one is valid. That is, in Isabelle/HOL:

Isabelle Lemma 5.5.1 (Interchange of process port graphs)

lemma *pgConstruct-interchange*:
assumes *pgDefined r* **and** *pgDefined t* **and** *pgDefined s* **and** *pgDefined u*
and *valid* (*Par* (*Seq r t*) (*Seq s u*))
shows *pgConstruct* (*Seq* (*Par r s*) (*Par t u*)) ≈
pgConstruct (*Par* (*Seq r t*) (*Seq s u*))

We prove this fact first as a general theorem about port graphs and then show that process port graphs satisfy its assumptions. In that, we again use the fact that port graph qualification done as part of the construction makes the child port graphs disjoint while keeping them equivalent to the originals, and that process port graphs only use the input and output sides for ports. Beyond all the parameters being well-formed port graphs with flow that are disjoint from each other, there are two interesting assumptions we need to satisfy. They require that the number of output ports of *pgConstruct r* must

be equal to the number of input ports of *pgConstruct* t, and the same for s and u respectively. We prove these for process port graphs from the validity assumption and the fact that open ports of process port graphs correspond to the inputs and outputs of the relevant process.

As a result, while the two compositions are syntactically distinct and in the general context of concurrent process modelling we would not expect them to be the same, we can use the extra information expressed by resources in our framework and our connection to port graphs to prove when they result in the same connections between actions. Figure 5.19 visualises the port graph that results from both forms, which, as expected, matches the unformalised process diagram in Figure 5.18.

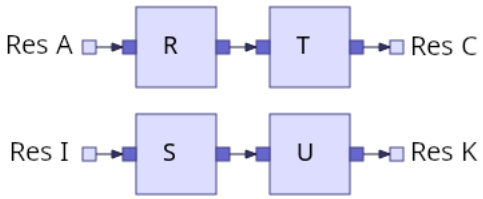


Figure 5.19: Port graph for both forms of nested parallel and sequential composition

5.6 Graphical Linearity

Our work in Chapter 4 demonstrates process correctness, more specifically their linearity, by appealing to linear logic. We transform all process compositions into deductions in linear logic while preserving their structure and then prove that for valid compositions the resulting deduction follows the rules of linear logic.

However, that demonstration does not say what linearity means directly in terms of the resources and processes, just that adherence to the rules of linear logic demonstrates it. In this section we use our port graph construction to split all occurrences of resources in the process (represented by places of the constructed port graph) into five categories based on how they connect to other occurrences. This characterises all the ways resources are manipulated in the process and allows us to then argue that they are being manipulated correctly by addressing each category. We first present the formal statement, followed by discussion of the cases it identifies.

Isabelle Lemma 5.6.1 (Graphical linearity of process port graphs)

```
lemma pgConstruct-linearity:
 assumes pgDefined x and valid x
     and p ∈ set (pgraphPlaces (pgConstruct x))
  obtains
  (Linear)
          ¬(∃a. port.label (place-port p) = Copyable a)
     and ¬(∃a b. port.label (place-port p) = Repeatable a b)
     and port.label (place-port p) ≠ Anything
     and ∃!e ∈ set (pg-edges (pgConstruct x)). edge-from e = p ∨ edge-to e = p
| (NonLinear-Origin)
          (∃a. port.label (place-port p) = Copyable a) ∨
          (∃a b. port.label (place-port p) = Repeatable a b)
     and ¬(∃e ∈ set (pg-edges (pgConstruct x)). edge-to e = p)
| (NonLinear-Destin)
          (∃a. port.label (place-port p) = Copyable a) ∨
          (∃a b. port.label (place-port p) = Repeatable a b)
     and ¬(∃e ∈ set (pg-edges (pgConstruct x)). edge-from e = p)
     and ∃!e ∈ set (pg-edges (pgConstruct x)). edge-to e = p
| (Anything-Origin)
          port.label (place-port p) = Anything
     and ∃!e ∈ set (pg-edges (pgConstruct x)). edge-from e = p
     and ¬(∃e ∈ set (pg-edges (pgConstruct x)). edge-to e = p)
| (Anything-Destin)
          port.label (place-port p) = Anything
     and ¬(∃e ∈ set (pg-edges (pgConstruct x)). edge-from e = p)
```

As with other properties of process port graphs up to this point, we start by assuming that the process x in question has no non-deterministic or higher-order features. Additionally, we assume that it is valid. While that is not always needed for the resulting port graph to have certain properties (such as Lemma 5.4.2), it is vital in this case. Then we fix a place p in the port graph constructed from x and prove that each such place falls into one of five cases:

- It is labelled with a linear resource and there is a unique edge incident on it, or
- It carries a copyable or repeatable resource (*Copyable* or *Repeatable*) and:
  - there is no edge coming into it but an arbitrary number of edges coming from it, or
  - there is no edge coming from it but there is a unique edge coming into it; or

- It carries the *Anything* resource and:
  - there is a unique edge coming from it but no edge coming into it, or
  - there is a no edge coming from it but an arbitrary number of edges coming into it.

The proof of this theorem consists of six major sub-lemmas totalling around 2500 lines of Isabelle script. Each proceeds by induction on the process composition structure, and case analysis covering the possible resource labels and incident edges. In each case we simplify the port graph construction and arrive either at a pattern fitting one of the above five cases or a contradiction, meaning that incompatible pattern cannot arise by constructing a port graph from a valid process composition.

One crucial fact used in this proof is that all edges in the port graph construction have the same resource label on their origin and their destination, except for those turning a *Repeatable* resource into an *Executable* resource or any resource into the *Anything* resource. This allows us to prove that our goal property is preserved through sequencing of port graphs.

Let us now discuss in more detail the five cases into which this theorem splits connections in process port graphs, along with what they mean for the resources involved.

The first case is the most important: every linear resource (i.e. not copyable, repeatable or *Anything*) is moved from exactly one origin to exactly one destination. As a result, no linear resource is left where it is produced or used to satisfy multiple requirements, and every action requiring a linear resource has a unique source for it.

The remaining cases characterise the allowed exceptions to the first case. The second and third cases say that copyable and repeatable resources can link to any number of destinations but must still have a unique origin. That is, they can be copied and erased, but not merged.

The fourth and fifth cases say that the *Anything* resource can be made by merging any number of other resources but after formation must be treated linearly. This represents the idea of *Anything* as grouping together arbitrary resources and treating them as one homogeneous object, which is then still a resource and must be treated as such.

With reference to our use of linear logic for similar purposes in Chapter 4, the first case corresponds to the base way how linear logic manipulates propositions, while the second and third cases correspond to the extra allowances for the ! modality and the fourth and fifth cases correspond to the extra allowances of the $\bot$ proposition.

However, here we are explicitly using the language of connections between nodes which represent primitive actions.

Note that, because our present port graph construction does not cover all process compositions, this does not supersede the demonstration through linear logic. Nevertheless, it offers a valuable perspective on the same high-level issue of process composition correctness and shows that our graphical approach detailed in this chapter can be fruitfully applied to verifying process models.

5.7 Port Graph Transition System

So far this chapter, we have been using port graphs to represent the structure of a process composition in a way inspired by the process diagrams of Section 3.3. We can, however, form an additional relationship between port graphs and processes: behaviour, expressed as a transition system. In this section we describe a transition system on port graphs, which in turn induces a transition system on process compositions through the port graphs constructed from them. That in turn allows us to better formalise the meaning of different process compositions.

The intuition behind the port graph transition system relies on viewing nodes as actions and edges as dependencies between them, just as we do with process port graphs. To make a transition, we find a node that has no incoming edge from another node. This represents its lack of dependency on any other node. Then we remove that node and all its adjacent edges from the port graph, which represents performing that action and distributing its results. In the absence of dependency loops, this will either result in another transition candidate or a port graph with no more nodes.

In our mechanisation of this idea, we start by defining the *node flow*. This relation orders nodes based on the edges going between them, placing one node before another node if there is an edge from a port of the former to a port of the latter. In Isabelle/HOL we define this as an inductive relation:

Isabelle Definition 5.7.1 (Node flow)

```
inductive node-flow :: ('s, 'a, 'p, 'l) port-graph
                       ⇒ ('s, 'a, 'p, 'l) node ⇒ ('s, 'a, 'p, 'l) node ⇒ bool
 where
  ⟦x ∈ set (pg-nodes G); y ∈ set (pg-nodes G); e ∈ set (pg-edges G);
   edge-from e ∈ set (nodePlaces x); edge-to e ∈ set (nodePlaces y)⟧
  ⟹ node-flow G x y
```

We can then define the concept of an *enabled* node of a port graph, which is a node with no incoming edge that originates from another node:

Isabelle Definition 5.7.2 (Enabled node)

```
definition nodeEnabled :: ('s, 'a, 'p, 'l) port-graph ⇒ ('s, 'a, 'p, 'l) node ⇒ bool
 where nodeEnabled G n ≡
  n ∈ set (pg-nodes G) ∧
  (∀e. e ∈ set (pg-edges G) ∧ place-ground (edge-from e)
       ⟶ edge-to e ∉ set (nodePlaces n))
```

The node flow allows us to then prove the important fact that, in a port graph that has some nodes and whose node flow is acyclic, there always exists some node that is enabled:

Isabelle Lemma 5.7.1 (Every non-empty acyclic port graphs has an enabled node)

```
lemma node-enabled-obtain:
 assumes port-graph G
      and pg-nodes G ≠ []
      and acyclicP (node-flow G)
  obtains n where nodeEnabled G n
```

What remains to be defined before the transition itself is the function to remove a node and all its adjacent edges. We do this by filtering the relevant parts of the port graph data:

Isabelle Definition 5.7.3 (Removing a node and its adjacent edges)

```
definition removeNode :: ('s, 'a, 'p, 'l) node ⇒ ('s, 'a, 'p, 'l) port-graph
                       ⇒ ('s, 'a, 'p, 'l) port-graph
 where removeNode n G =
 PGraph
  (filter ((≠) n) (pg-nodes G))
  (disconnectFromPlaces (nodePlaces n) (pg-edges G))
  (pg-ports G)
```

Then we define the transition relation between port graphs labelled with the node being used as follows[4]:

Isabelle Definition 5.7.4 (Transition relation on port graphs)

```
inductive pgTrans :: ('s, 'a, 'p, 'l) port-graph ⇒ ('s, 'a, 'p, 'l) node
                   ⇒ ('s, 'a, 'p, 'l) port-graph ⇒ bool
 where ⟦nodeEnabled G n; G' = removeNode n G⟧ ⟹ pgTrans G n G'
```

We can then prove, for instance, that it is impossible for a port graph with no nodes to have a transition. Because each transition removes a node, this means that a chain of transitions will eventually terminate once all nodes have been consumed.

Isabelle Lemma 5.7.2 (Port graphs with no nodes do not transition)

lemma *pgTrans-no-nodes*:
assumes *pg-nodes* $G = []$
shows $\neg$ *pgTrans* $G\ n\ G'$
using *assms* **by** (*clarsimp elim*!: *pgTrans.cases simp add*: *nodeEnabled-def*)

However, the most important property relates to port graph equivalence. Consider two equivalent port graphs that are both well-formed and a transition on one to some result. Then there exists a node renaming such that the other port graph will transition to a port graph equivalent to the result of the assumed transition, and it will do so along a node that is equal to the label of the assumed transition up to the renaming. (In our proof we use the renaming function that witnesses the equivalence of the initial port graphs.) Or, in Isabelle/HOL:

Isabelle Lemma 5.7.3 (Equivalent port graphs transition to equivalent results)

lemma *pgTrans-pgEquiv*:
assumes $X \approx Y$
and *port-graph* X
and *pgTrans* $X\ n\ X'$
obtains f **where** *pgTrans* Y (*renameNode* $f\ n$) (*removeNode* (*renameNode* $f\ n$) Y)
and $X' \approx$ (*removeNode* (*renameNode* $f\ n$) Y)

This means that equivalent port graphs have equivalent transitions. As such, all of the process port graph equivalence facts we have proven over the previous sections imply equivalence of behaviour under this view.

5.8 Conclusion

In this chapter we formally connected our process compositions to the graphical structure of port graphs. With it we bring to surface the information contained in compositions about how individual actions are connected by the resources they require and produce. We first mechanised a self-contained theory of port graphs in Isabelle/HOL

[4]Note that we use the **inductive** keyword more to get automatically generated theorems for use in proofs rather than defining a truly inductive relation.

and then defined how specific port graphs can be constructed from our process compositions. With these port graphs we were able to state and verify two interesting properties of process compositions: when sequential and parallel composition distribute over each other and what process linearity means in terms of connections between actions. We closed by giving a transition system for port graphs, suggesting a behavioural interpretation to them.

Next we highlight some threads of future work stemming from our mechanisation and use of port graphs. Then, in the next chapter, we explore an approach for adding probabilistic information into our framework.

Behaviour through port graphs. In future work we plan to further pursue the behavioural angle suggested by the port graph transition system. Recall that equivalent port graphs have equivalent transitions and that many of our theorems about process port graphs have as conclusion their equivalence. This suggest that we could define an equivalence relation on process compositions using equivalence of port graphs constructed from them (in addition to requiring they have equal inputs, outputs and validity). Then we could quotient the type of process compositions with this relation, just like we quotient resources with their equivalence in Section 2.2. On the resulting type, the process port graph equivalences would become equalities while respecting the transition system. This may yield a type closer to our intuition of processes.

Connection to category theory. Such a quotient of process compositions would be useful for formally verifying that our framework forms a monoidal category of resources and processes. In Section 5.2 we briefly mentioned the connection of port graphs to string diagrams of monoidal categories. Monoidal categories themselves form a suitable theory of parallel processes with their formalisation of sequential and parallel composition. See for instance the work of Breiner et al. [17], who find monoidal categories useful as a model of process plans: the algebraic side is suitable for digital uses while string diagrams keep them readily interpretable for humans.

In this chapter we have formed a connection between our framework for process compositions and port graphs, which in turn are related to string diagrams. Along with our framework capturing parallel processes, this suggests that our framework should satisfy the axioms of a monoidal category. Our preliminary work in this direction suggests that by leveraging the equivalence of constructed port graphs to relate process compositions expressing the same intuition, we can formally verify that they form

a monoidal category. In this, we use the mechanisation of monoidal categories in Isabelle/HOL due to Stark [84].

This suggests that a fruitful thread of future work would be exploring the implications this has for our framework as well as what connections category theory can facilitate. There may be theorems about monoidal categories that have a meaningful interpretation in terms of process compositions. For instance, the composition pattern we discuss in Section 5.5 can be seen from the categorical perspective as talking about parallel composition being a *binary endofunctor*.

Expanded construction coverage. Recall that our current port graph construction does not cover process compositions with non-deterministic or higher-order features. In order to address this in future work, we will explore extensions to our mechanised theory of port graphs that could accommodate these features.

For instance, hierarchical port graphs would allow us to nest port graphs within nodes of other port graphs. With these, we may be able to express *Represent* P as a node containing the port graph for the child process P. However, we expect that the inclusion of higher-order features would mean nodes of the port graph no longer represent just primitive actions, because the evaluation of an arbitrary *Executable* resource may require a node to properly express.

While hierarchical port graphs may allow us to express the optional composition *Opt* P Q as a node holding port graphs for its two child processes P and Q, they are unlikely to elegantly resolve, for instance, its interaction with a preceding injection process. But, what may be more effective, is generalising the notion of port graphs from the plane into space. Noting that in our current construction sequential composition corresponds to the left-right axis and parallel composition to the up-down axis, the addition of a dimension may allow us to express optional composition (and other non-deterministic effects) in the front-back axis without interfering. However, such a generalisation would be a significant undertaking, because it includes both generalising the idea of port graphs and mechanising the resulting formalism before we can take advantage of it with process compositions.

Tailored visualisation. Recall from Section 5.3.10 that we can convert our port graphs into a format compatible with external visualisation tools, namely the Eclipse Layout Kernel (ELK). Given that our conversion is mechanised in Isabelle/HOL, it is also possible to automatically generate executable Scala code for it. Because both

ELK and Sprotty have Java interfaces, this could serve as the basis for a tool visualising formally verified port graphs in general and process port graphs in particular.

As noted in Section 5.4, the visualisation of process port graphs is connected to our process diagrams from Section 3.3. The fact that process port graphs are fully formal would contribute to the trustworthiness of a visualisation based on them. Furthermore, the integration with ELK would make it possible to use the various graph layout algorithms that it implements to improve the appearance and readability of the visualisation.

In contrast with our current use of the online ELK Demonstrators environment, a custom environment would allow us to tailor all aspects of how the port graphs are rendered. Such an environment could in turn form the basis for a graphical process composer that retains the formally verified core.

Chapter 6

Probabilistic Resources

Our non-deterministic resources (introduced in Chapter 2) represent possible outcomes of a process, with optional composition (introduced in Section 3.1.2) building non-deterministic actions reacting to those outcomes. When we aim to distil quantitative information from a process compositions, such as the number of steps taken or the total amount of some resource (e.g. electricity) consumed, we may need to weigh the contribution of different execution paths if any part of the process reacts to such non-deterministic outcomes. Without more information, we can at best use unrepresentative values such as assuming all outcomes are equally likely.

In this chapter we explore how such probabilistic information could be added to our non-deterministic resources. With this, we can weigh the contribution of different execution paths by their relative probability and arrive at an expected value for the numeric information we seek.

Note, however, that sometimes it may not be appropriate to give a non-deterministic outcome of an action any explicit probability. For instance, compare using a machine with a known failure rate to the probability of the password being wrong in a log-in process. The latter is nearly impossible to accurately quantify without access to large amounts of data and is likely to vary significantly between different users. To ensure that framework can express both situations, we make the probability information optional when extending our resources. Uses of that information are then conditional on its presence, a fact enforced by the type system of both Isabelle/HOL and any target languages such as Haskell or OCaml.

We proceed with this exploration by progressively extending the framework described in the preceding chapters and addressing the resulting issues. We set some general context by looking at related work in the next section. Then in Section 6.3,

we describe how we extend our mechanisation of resources to carry the probabilistic information. This is done with minimal use of the information itself, mainly to observe the effects of its presence. In Section 6.4 we discuss complications to process compositions arising from that information being present, and in Section 6.5 we discuss our approach to alleviating those complications. This includes three new equations for our resource algebra (one described in Section 6.5.1 and two in Section 6.5.2), each working to uncover implicit forms of determinism in the resources and thus simplify them. We close with concluding remarks and future work in Section 6.7.

Isabelle/HOL makes the process of extending our theory significantly smoother than it might be if we were not using a proof assistant, because every change we make in the theory instantly propagates to parts depending on it. Every definition and proof following the change is re-run and either still works after the change or it is highlighted by the system. At such breakages, we are given the state just before and a description of the issue. This makes fundamental changes to the theory easier to enact, because we can focus on newly arisen issues while trusting that proofs not highlighted by Isabelle/HOL are still fully correct.

6.1 Related Work

Many process calculi include the information we are concerned with in a stochastic manner: the probability of certain action outcomes is not described directly as a probability distribution but is the result of different actions racing to complete with delays generated according to a random distribution. Most often this distribution is the negative exponential distribution, which describes the time between events of a Poisson process, because it has mathematical properties aiding in efficient model parameterisation and simulation (e.g. it is memory-less). This approach is for instance taken by generalised stochastic Petri nets [64], the stochastic π-calculus [61] and PEPA [37].

This is in contrast to our approach in this chapter, where we assign the probability distribution explicitly within the resource describing the action output. We find that having two actions racing to completion, with one being the winner and outcome as if the other did not happen, would clash with the notion of linearity in our processes.

Recall that our framework for process composition is inspired by linear logic (see Chapter 4 for the formal connection to ILL). While linear logic does not concern itself with probabilities, Horne [45] introduces the notion of sub-additives to linear logic and links it to probabilistic choice in processes. This refinement of additive operators of

linear logic is along the same lines as our extensions of non-deterministic resources in this chapter. Horne's work thus opens the possibility of verifying probabilistic aspects of our framework with (this extension of) linear logic, not just their linearity. While we do not do so at present, we note this as part of future work in Section 6.7.

Recall also WorkflowFM, which is the most significant inspiration for our approach (see Section 1.2.1). The process models of WorkflowFM do not themselves include probabilistic information. However, part of the larger framework is an agent-based simulator for deployed models with which we can carry out stochastic simulations for an implementation of the process [70]. This is unlike the present work, where we include the probabilistic information in the model itself.

Finally, we should note that the concept of non-determinism can relate to multiple aspects of processes. In this chapter we focus on the non-determinism introduced by an action having multiple possible outcomes. However, another kind of non-determinism is introduced by the possible interleavings of parallel actions. See for instance the overview by Segala [80] for a discussion of both kinds of non-determinism and how different probabilistic process models refine them. We do not consider a probabilistic refinement of this second form of non-determinism at present.

6.2 Probability Theory in Isabelle/HOL

We now give a brief introduction to the way probability theory is mechanised in Isabelle/HOL. We focus on probability mass functions and their operations, as those are what we use in this chapter.

We use the mechanisation of probability theory distributed with Isabelle as the session (collection of theories) *HOL-Probability*. This mechanisation is based on measure theory, and its core is described in Hölzl's thesis [40] and further discussed in papers by Hölzl and Heller [42], Avigad, Hölzl and Serafin [5], and Hölzl, Lochbihler and Traytel [43]. The resulting probability theory has been used, for instance, to mechanise Markov Chains and Markov Decision Processes [41] and a compiler for probability density functions described by probabilistic programs [28].

Intuitively, a measure is a collection of sets along with a function that assigns non-negative values to those sets such that the value it assigns to the union of two sets is the sum of the values it assigns to those sets. A probability distribution is a measure that assigns the value 1 to the whole space — in that context, we consider the whole space as the events and the measure as the probability of some subset of events. A

probability mass function (PMF) is a distribution whose whole space is countable. As such, we can also view a PMF as a function assigning a real value to elements of the space that sums up to 1.

In Isabelle/HOL, PMFs over the type *'a* are represented by the type *'a pmf*. For instance, the Bernoulli distribution assigning probability *p* to *True* is *bernoulli-pmf p* :: *bool pmf*. We use several functions to manipulate PMFs in our work:

- *map-pmf f p* uses the function *f* to transform the domain of the PMF *p*. Note that if two elements are mapped to the same one then this adds their probability.
- *pmf p x* is the probability that the PMF *p* assigns to *x*.
- *return-pmf x* constructs the trivial PMF that assigns probability 1 to *x*.
- *bind-pmf p f* uses *f* to take every element in the space of the PMF *p* to some PMF, and then combines those PMFs according to the probability that *p* assigns to the relevant elements of its space. We use its infix syntax: $p \gg\!= f$.

To illustrate the function *bind-pmf*, consider the joint PMF defined as follows [1]:

Isabelle Definition 6.2.1 (Product PMF)

definition *pair-pmf A B* = $A \gg\!= (\lambda x.\ B \gg\!= (\lambda y.\ (x, y)))$

which assigns the expected probabilities:

Isabelle Lemma 6.2.1 (Probability of a pair in product PMF)

lemma *pmf-pair*:
pmf (*pair-pmf M N*) (*a*, *b*) = *pmf M a* ∗ *pmf N b*

6.3 Adding Probabilistic Information

The probabilistic information we add to our non-deterministic resources represents the probability distribution over the two children of that resource combination. We represent this with a PMF, because it is a probability distribution over a discrete domain: the first child and the second child. See Section 6.2 for a brief overview of how PMFs and probability theory in general are mechanised in Isabelle/HOL.

[1] https://isabelle.in.tum.de/dist/library/HOL/HOL-Probability/Probability_Mass_Function.html#Probability_Mass_Function.pair_pmf|const

Another aspect of the information we are adding is its optional nature, because in some situations it is not possible to assign reasonable probabilities to the outcomes. For this we use the option monad, which in Isabelle/HOL is mechanised as the type *'a option* for contents represented by type *'a*. It has two constructors, *Some 'x* and *None*, that respectively wrap an existing value and represent absence.

The last aspect is the actual domain we will use for the PMF. Because it only needs to have two elements, we could use the type *bool*, for instance representing an answer to "do you mean the first child?" But, to make our code more readable, in Definition 6.3.1 we define a new datatype called *bchoice* (as in, "binary choice") with two elements: *First* and *Second*. For convenience, we use the existing Bernoulli distribution in Definition 6.3.2 to define *first-pmf p*, which assigns probability *p* to *First* and *1 − p* to *Second*.

Isabelle Definition 6.3.1 (Datatype annotating binary choice)

```
datatype bchoice = First | Second
```

Isabelle Definition 6.3.2 (Binary choice PMF from the first option's probability)

```
definition first-pmf :: real ⇒ bchoice pmf
  where first-pmf p =
    map-pmf (λx. case x of True ⇒ First | False ⇒ Second) (bernoulli-pmf p)
```

6.3.1 Extending Resources

Bringing all of these together in one parameter, we expand the non-deterministic resource term constructor and rename it to *NonDP*:

Isabelle Definition 6.3.3 (Updated datatype of resource terms)

```
datatype ('a, 'b) res-term =
   Res ('a)
 | Copyable ('b)
 | Empty
 | Anything
 | Parallel (('a, 'b) res-term list)
 | NonDP (bchoice pmf option) (('a, 'b) res-term) (('a, 'b) res-term)
 | Executable (('a, 'b) res-term) (('a, 'b) res-term)
 | Repeatable (('a, 'b) res-term) (('a, 'b) res-term)
```

After this change, we also update all of the resource theory to accommodate the new parameter. At this stage we avoid making extensive use of the new information,

such as in the resource term equivalence, which will be covered in the next sections. Thus, for two $NonD_P$ terms to be equivalent they have to be assigned the same probabilistic information. Updating all of the resource theory is mechanical, showing that expanding the information that resources carry is unproblematic for that part of our framework.

As an example, at this point we can express the outcome of using a laser cutting machine with 30% failure rate as the following resource:

$$NonD_P\ (Some\ (first\text{-}pmf\ 0.7))\ (Res\ Cutter \odot Res\ CutShape)\ (Res\ CutterBlocked)$$

6.3.2 Extending Compositions

When it comes to the process theory, we need to match the expanded $NonD_P$ resources by adding that same probabilistic information parameter (*bchoice pmf option*) to the relevant constructors. These are: the optional composition *Opt*, and the resource actions *OptDistrIn* and *OptDistrOut*. Note that we do not need such a parameter in *InjectL* and *InjectR*, even though their outputs involve non-deterministic resources, because the probabilistic information they use is constant: full probability of 1 towards the first and second child respectively.

With this information we can update the *input* and *output* functions (originally given in Definition 3.1.2) to provide the relevant information to $NonD_P$ resources. We use either a new parameter of the process (here named d in all cases) or a constant (via *first-pmf*), yielding the following new defining equations:

Isabelle Definition 6.3.4 (Probabilistic input and output definitions)

$$
\begin{array}{l|l}
| \ input\ (Opt\ d\ p\ q) = & | \ output\ (Opt\ d\ p\ q) = \\
\quad NonD_P\ d\ (input\ p)\ (input\ q) & \quad output\ p \\
| \ input\ (InjectL\ a\ b) = & | \ output\ (InjectL\ a\ b) = \\
\quad a & \quad NonD_P\ (Some\ (first\text{-}pmf\ 1))\ a\ b \\
| \ input\ (InjectR\ a\ b) = & | \ output\ (InjectR\ a\ b) = \\
\quad b & \quad NonD_P\ (Some\ (first\text{-}pmf\ 0))\ a\ b \\
| \ input\ (OptDistrIn\ d\ a\ b\ c) = & | \ output\ (OptDistrIn\ d\ a\ b\ c) = \\
\quad a \odot (NonD_P\ d\ b\ c) & \quad NonD_P\ d\ (a \odot b)\ (a \odot c) \\
| \ input\ (OptDistrOut\ d\ a\ b\ c) = & | \ output\ (OptDistrOut\ d\ a\ b\ c) = \\
\quad NonD_P\ d\ (a \odot b)\ (a \odot c) & \quad a \odot (NonD_P\ d\ b\ c)
\end{array}
$$

The rest of the adjustments just mechanically account for the new parameters. Note that, at this stage, we do not yet make any changes to process composition validity, meaning optional composition still requires that the outputs of the two child processes

be equal. As a result, we lose our original way of transforming branches of a non-deterministic resource without merging them, because it relies on *InjectL* and *InjectR* actions whose outputs now differ (see Section 3.1.3). We address this point in the remainder of this chapter after describing how we adjust linearity demonstration from Chapter 4.

6.3.3 Linearity Demonstration Through ILL

The adjustment needed for the ILL translation demonstrating our compositions' linearity (see Chapter 4) is almost as simple and mechanical as the adjustment to resources. Its crucial point is that ILL cannot express probabilistic information, so in our translation of resources into ILL propositions we are forced to drop that part of $NonD_P$ resources. While this renders the translation no longer injective, because resources only differentiated by their probabilistic information go to the same proposition, all of the properties mentioned in Chapter 4 still hold.

As noted in Section 3.2, if our framework was fully reliant on linear logic, then we could not include in our processes and resources information that the logic cannot express. The advantage of our loosened relation is that we can use linear logic to demonstrate linearity while remaining able to include further concepts in our process compositions. This makes our framework easy to extend, as seen in this chapter.

6.3.4 Summary

In the previous sections, we discussed how to extend resources with optional probabilistic information for non-deterministic resources, without yet making significant use of it. We observed that the mere addition of this information preserves the vast majority of the proven statements in our theory. The exception are its effects on what compositions are considered valid, particularly in the presence of injection actions whose outputs are made distinct by the probabilistic information. In the next sections we make use of the probabilistic information and adjust optional composition in its presence.

In all of the above changes to our mechanisation, the automation available in Isabelle is invaluable. In most cases it assures us that our changes to the constructors do not break proofs without costly manual re-checking. Where proofs of true statements do break, our use of the structured Isar language and the integration of automated provers through Sledgehammer aids in their quick patching. And in statements that are

no longer true, the identification of the precise point of proof failure helps us understand why the statement is no longer true.

6.4 Complications to Composition Validity

While adding probabilistic information allows us to express the relative likelihood of actions outcomes, it comes at a cost. This new information, by virtue of distinguishing some resources that would have previously been considered equal, complicates process composition validity (defined in Section 3.2).

For an example of this issue, recall the approach for progressing both branches of a non-deterministic resource described in Section 3.1.3. Given two processes, P: $x \to a$ and Q: $y \to b$, we follow each up with the relevant injection action into $NonD\ a\ b$ (that is, *InjectL* and *InjectR* respectively) and then compose the results using *Opt*. The final composition was valid because both branches had the same output, resulting in:

$$(Opt\ (Seq\ P\ (InjectL\ a\ b)\ (Seq\ Q\ (InjectR\ a\ b)))) : NonD\ x\ y \to NonD\ a\ b$$

However, with the addition of probabilistic information, those outputs are now different because they assign different probabilities: *first-pmf 1* and *first-pmf 0* respectively. While this now violates the validity condition, we would intuitively expect this composition to still be valid and for it to combine the two outputs in a way that weighs the relative probabilities of taking each branch, resulting in:

$$(Opt\ d\ P\ Q) : NonD_P\ d\ x\ y \to NonD_P\ d\ a\ b$$

It is thus clear that the validity condition on *Opt* and the way its output is formed must change in the presence of probabilistic information. Our approach to this change is detailed in the following sections.

6.5 Simple Optional Composition

Recall that, in its original form, optional composition applies two processes to the non-deterministic combination of their inputs with the aim of producing a common output and thus merging the branches. In the presence of probabilistic information this is now too restrictive. We find that the simplest way of changing optional composition is for it to no longer aim to merge the outputs and instead produce their non-deterministic combination, thus allowing the processes to have any outputs.

More formally, recall the original definition of the optional composition validity (Definition 3.2.1) and output (Definition 3.1.2):

$$valid\ (Opt\ x\ y) \equiv valid\ x \wedge valid\ y \wedge output\ x = output\ y$$
$$output\ (Opt\ x\ y) = output\ x$$

We change these two defining equations to the following:

$$valid\ (Opt\ d\ x\ y) \equiv valid\ x \wedge valid\ y$$
$$output\ (Opt\ d\ x\ y) = NonD_P\ d\ (output\ x)\ (output\ y)$$

Using this condition and output for *Opt* allows us to once again optionally compose processes whose outputs were rendered different by the probabilistic information. But, because optional composition no longer merges the two processes' outputs, we lose the only way of eliminating non-determinism in resources. Even with equal outputs, say r, we get the output resource $NonD_P\ d\ r\ r$. As such, resource expressions can quickly grow unwieldy.

However, we can control some of that non-deterministic resource complexity with resource equations. We explore two examples in the next sections: (i) merging the two branches when they are the same, just like optional composition did before our change, and (ii) taking advantage of the probabilistic information to internalise the actions *InjectL* and *InjectR* in the resource algebra. Both of these express a kind of "hidden determinism" in the resource, and use it to simplify the resource expression.

Before we add those equations, let us briefly turn to the translation of process compositions into ILL deductions, which needs to be fixed after our change to validity and output of optional composition. Previously, the linearity of this composition action was demonstrated using the $\oplus_L$ rule of ILL. That is, denoting *resource-to-ill* r with $(\!|r|\!)$, for processes $P\colon x \to a$ and $Q\colon y \to b$ we used the following deduction:

$$\frac{\begin{array}{cc} to\text{-}deduct\ P & to\text{-}deduct\ Q \\ \vdots & \vdots \\ [(\!|x|\!)] \vdash (\!|a|\!) & [(\!|y|\!)] \vdash (\!|b|\!) \end{array}}{[(\!|x|\!) \oplus (\!|y|\!)] \vdash (\!|a|\!)}\ \oplus_L\ (a = b)$$

Because we removed the assumption on which this deduction relies ($a = b$) and changed the composition output, we now also need to use the $\oplus_{R-1}$ and $\oplus_{R-2}$ rules to make the deduction well-formed again. For the same processes we now use the following deduction:

$$\dfrac{\dfrac{\begin{array}{c}\textit{to-deduct } P\\ \vdots\\ [(\!|x|\!)] \vdash (\!|a|\!)\end{array}}{[(\!|x|\!)] \vdash (\!|a|\!) \oplus (\!|b|\!)}\oplus_{R-1} \quad \dfrac{\begin{array}{c}\textit{to-deduct } Q\\ \vdots\\ [(\!|y|\!)] \vdash (\!|b|\!)\end{array}}{[(\!|y|\!)] \vdash (\!|a|\!) \oplus (\!|b|\!)}\oplus_{R-2}}{[(\!|x|\!) \oplus (\!|y|\!)] \vdash (\!|a|\!) \oplus (\!|b|\!)}\oplus_{L}$$

6.5.1 When Probability Does Not Matter

Our first step towards controlling the complexity of non-deterministic resources is to add a resource equation that recovers what optional composition was doing originally. That is, if we have a non-deterministic combination of resources that are equal, then we equate it with that child resource — no matter which branch it ends up being, the outcome is the same. In Isabelle/HOL notation:

$$\mathit{NonD}_P\ d\ x\ x = x$$

To add this equation into the resource algebra, we need to add the relevant rule to resource term equivalence (Definition 2.2.1) and update the normalisation procedure (see Section 2.3). The implications of this change will then propagate through the rest of the resource and process theory. Isabelle's automation is invaluable in making this process smooth.

Note that this resource equation does not make use of the probabilistic information. We present it here, because in the presence of probabilistic information and the resulting issue this equation becomes vital to usability of our framework. However, changes analogous to those we describe here could be carried out in the non-probabilistic setting to achieve the same effects.

6.5.1.1 Resource Terms

The rule that we add to the resource term equivalence is almost identical to our target equation, just with $=$ replaced with $\sim$ and the two sides being resource *terms*:

Isabelle Definition 6.5.1 (Additional equivalence when children are equal)

$$\mathit{NonD}_P\ d\ x\ x \sim x$$

To update the normalisation procedure we first need to decide on the new normal form and then make the procedure pursue it. In this we aim towards the right-hand side of the new equivalence rule, because it is the simpler, shallower form. For the normal form represented by the predicate *normalised* (Definition 2.3.1), we no longer consider any non-deterministic resource with equal children to be normalised:

Isabelle Definition 6.5.2 (Normal form that checks for equality of children)

$$normalised\ (NonD_P\ d\ x\ y) \equiv normalised\ x \wedge normalised\ y \wedge x \neq y$$

To make the rewriting-based normalisation procedure pursue this new normal form, we make additions to both the rewriting relation specification and the rewriting step implementation. For the specifying relation (Definition 2.3.2), we add a rule corresponding to the left-to-right direction of the new equation:

Isabelle Definition 6.5.3 (Additional term rewriting rule when children are equal)

$$res\text{-}term\text{-}rewrite\ (NonD_P\ d\ x\ x)\ x$$

For the step implementation (Definition 2.3.4), we add a new case split on whether the normalised children are equal into the equation for $NonD_P$ terms:

Isabelle Definition 6.5.4 (Rewriting step that checks for equality of children)

$$\begin{array}{l} step\ (NonD_P\ d\ x\ y) = \\ (\quad\ \ if\ \neg\ normalised\ x\ then\ NonD_P\ d\ (step\ x)\ y \\ \ \ else\ if\ \neg\ normalised\ y\ then\ NonD_P\ d\ x\ (step\ y) \\ \ \ else\ if\ x = y\ then\ x \\ \ \ else\ NonD_P\ d\ x\ y) \end{array}$$

For the rewriting bound (Definition 2.3.3), which is used to prove that the normalisation procedure terminates, we add *1* to the original bound for non-deterministic resources to represent possibly merging the two children if they end up equal.

In order for the full normalisation procedure *normal-rewr* to decide the resource term equivalence (see Section 2.3.6), we require (among other simpler facts) that for every rule introducing the relation its two sides be joinable by a series of rewriting steps. We fulfil this obligation for the new rule by joining them according to the following diagram (where arrows represent rewriting steps and stars their reflexive and transitive closure):

$$NonD_P\ d\ x\ x \rightarrow^*\ NonD_P\ d\ (normal\text{-}rewr\ x)\ (normal\text{-}rewr\ x) \rightarrow normal\text{-}rewr\ x\ {}^*\!\leftarrow x$$

As for the translation of resource terms to ILL propositions, a change to the resource term equivalence requires a change to the deduction demonstrating it in ILL (see Section 4.6). Just as the original definition of that deduction follows the original definition of the normalisation procedure, so do our changes to the deduction follow those we make to the normalisation procedure

This concludes our changes to resource terms, which insert the new resource term equivalence and ensure the normalisation procedure decides that equivalence. We now turn to the rest of resource and process theory, where the added equivalence becomes an equation of resources and affects the rest of our framework.

6.5.1.2 Resources and Compositions

Recall from Section 2.2 that resources are obtained as a quotient of resource terms by the equivalence relation $\sim$. In Section 2.4 we described those resources inherit the Bounded Natural Functor structure from resource terms. Our proof of that fact (following Theorem 4 of Fürer et al. [32]) relies on the resource term mapper *map-res-term* commuting with the equivalence relation $\sim$.

But this is no longer the case. Consider distinct resource atoms A and B. The resource term $\mathit{NonD}_P\ d\ (\mathit{Res}\ A)\ (\mathit{Res}\ B)$ is not equivalent to the term $\mathit{Res}\ A$. But we can use *map-res-term* to map both A and B to A, resulting in $\mathit{NonD}_P\ d\ (\mathit{Res}\ A)\ (\mathit{Res}\ A)$ which is equivalent $\mathit{Res}\ A$ thanks to the newly added rule. As such, the resource term mapper does not commute with the equivalence relation. From a wider perspective, the issue lies in equivalence of resource terms now considering the *content* of the terms, not jut their *shape*.

As a result, we no longer get a convenient automatic definition and set of theorems for the resource mapper and relator. But the resource mapper is vital for transforming processes between domains, allowing us to systematically translate the resources they use (see Section 3.4.1). We therefore lift the mapper definition from the term-level through the quotient, still calling it *map-resource*, and manually verify the properties it does retain: that mapping of identity is identity and that composition of mappers is a mapper of composition.

Note that this more manual definition of the resource mapper is now not used in the automatically-generated mapper for process compositions. This means we need to manually define and verify an alternative process mapper to make use of it, which we call *map-process′*.

While these difficulties with the mappers are inconvenient, decreasing the level of automation in our mechanisation, they do not stand in the way of any important properties of resources and process compositions.[2]

As for the translation of process compositions into ILL deductions, it only needs to be made more robust. Previously, because normalisation passed through the *NonD* term constructor without changing it, we did not have to worry about it before and after the core deduction we described earlier in Section 6.5. But, with the new equivalence

[2] An additional minor change is that the function *parallel-parts* (originally defined in Definition 3.3.2) needs to check whether the two child terms in its NonD_P equation are equivalent. This makes it respect resource term equivalence and function properly when lifted to resources. However, we make no further use of this function in the present chapter and, as such, gloss over the details.

rule, normalisation may change $NonD_P$ terms to simpler ones and we need to account for that around that core deduction. For instance, if both processes have the same input resource x, then for the deduction translation to be well-formed we first need to turn *resource-to-ill* x into *resource-to-ill* $x \oplus$ *resource-to-ill* x. We do this using the deductions demonstrating resource term equivalence, which are originally defined in Section 4.6, by preceding and following the core deduction with deductions out of and into the relevant normal forms respectively.

6.5.1.3 Results

With this resource equation successfully added and process theory now updated, we regain the elimination feature of optional composition before our changes:

$$[\![P\colon a \to x;\ Q\colon b \to x]\!] \Longrightarrow (Opt\ d\ P\ Q)\colon NonD_P\ d\ a\ b \to x$$

Additionally, on top of recovering that behaviour, it now applies everywhere there is a non-deterministic resource. For instance, we can now express taking one of two actions on the same resource with any relative probability:

$$[\![eatSoup\colon Res\ Hungry \to Res\ Bowl;\ eatPasta\colon Res\ Hungry \to Res\ Plate]\!] \Longrightarrow \\ (Opt\ d\ eatSoup\ eatPasta)\colon Res\ Hungry \to NonD_P\ d\ (Res\ Bowl)\ (Res\ plate)$$

Further, with this change it is now possible to illustrate how an expected value can be extracted from a non-deterministic process composition. Consider again a laser cutting machine with 30% failure rate. Let using that machine to cut out a given shape be a primitive action that occupies the machine and consumes a metal plate, and may produce either the cut out shape and free the machine or block the machine. Then let fixing the machine be a primitive action that turns the blocked machine back into an available one, along with producing metal scrap for disposal. Furthermore, let the cost of operating the machine be £20 and the cost of fixing it be £30. We can formalise these two actions as follows:

Isabelle Definition 6.5.5 (Operating a faulty machine)

```
definition operate =
  Primitive (Res Cutter ⊙ Res Plate)
            (NonDP (Some (first-pmf 0.7)) (Res Cutter ⊙ Res CutShape)
                                          (Res CutterBlocked))
            STR "Operate"
            20
```

Isabelle Definition 6.5.6 (Fixing a blocked machine)

```
definition fix =
 Primitive (Res CutterBlocked)
           (Res Cutter ⊙ Res Scrap)
           STR ''Fix''
           30
```

Then, operating the machine and fixing any failure that occurs can be represented by the following composition (visualised in Figure 6.1):

Isabelle Definition 6.5.7 (Operating a faulty machine, possibly unblocking it)

```
definition operate-and-fix =
 Seq operate
     (Seq (Opt (Some (first-pmf 0.7))
               (Identity (Res Cutter ⊙ Res CutShape))
               fix)
          (OptDistrOut (Some (first-pmf 0.7))
                       (Res Cutter) (Res CutShape) (Res Scrap)))
```

resulting in the laser cutting machine being preserved while the metal plate is turned either into the cut out shape or scrap:

$$\textit{operate-and-fix}\text{: } \textit{Res Cutter} \odot \textit{Res Plate} \rightarrow \textit{Res Cutter} \odot \textit{NonD}_P\ (\textit{Some}\ (\textit{first-pmf}\ 0.7))\ (\textit{Res CutShape})\ (\textit{Res Scrap})$$

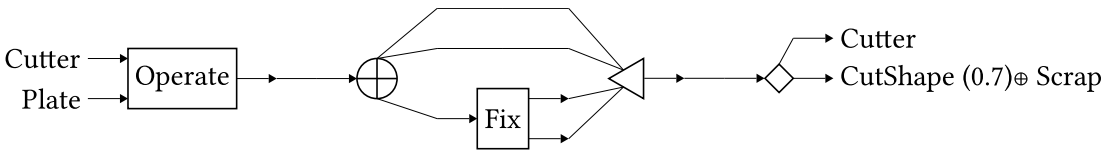


Figure 6.1: Process diagram for operating a faulty machine, with possibly fixing it

We can then define a function *cost-pmf*, given in Definition 6.5.8, to calculate the distribution of cost from a process composition. It does so by associating individual actions with cost distributions, and composition actions with ways of combining PMFs. More specifically, for primitive actions we use a deterministic cost extracted from their metadata and for resource actions we use a deterministic zero cost (i.e. we consider those free). For sequential and parallel composition we form the joint distribution from the child PMFs and add each pair of values, adding up the costs of the two child processes. For optional composition we combine the child PMFs by weighing them using the overall PMF assigned to that composition action, or equally if the distribution in unspecified.

Isabelle Definition 6.5.8 (Cost PMF calculation)

```
primrec cost-pmf :: ('a, 'b, 'l, rat) process ⇒ rat pmf
 where
  cost-pmf (Primitive ins outs l m) = return-pmf m
| cost-pmf (Identity a) = return-pmf 0
| cost-pmf (Swap a b) = return-pmf 0
| cost-pmf (Seq p q) =
    cost-pmf p ⤜ (λx. cost-pmf q ⤜ (λy. return-pmf (x + y)))
| cost-pmf (Par p q) =
    cost-pmf p ⤜ (λx. cost-pmf q ⤜ (λy. return-pmf (x + y)))
| cost-pmf (Opt dopt p q) = ( case dopt of
     Some d ⇒ d ⤜ case-bchoice (cost-pmf p) (cost-pmf q)
   | None ⇒ cost-pmf p ⤜ (λx. cost-pmf q ⤜ (λy. return-pmf (x + y / 2))))
| cost-pmf (OptDistrIn d a b c) = return-pmf 0
| cost-pmf (OptDistrOut d a b c) = return-pmf 0
| cost-pmf (Duplicate a) = return-pmf 0
| cost-pmf (Erase a) = return-pmf 0
| cost-pmf (Repeat a b) = return-pmf 0
| cost-pmf (Close a b) = return-pmf 0
| cost-pmf (Once a b) = return-pmf 0
| cost-pmf (Forget a) = return-pmf 0
```

Note that it is not currently clear to us how to assign a cost to higher-order features, namely the *Represent* and *Apply* cases. In the *Apply* case, it is unclear how much executing an arbitrary executable resource should cost. In the *Represent* case, it is not clear whether representing a composition should be free or not. We choose to omit these two cases, meaning our *cost-pmf* is only applicable to compositions with no higher-order features. This is sufficient for our discussion in this section.

Then, by applying *cost-pmf* to our composition from Definition 6.5.7, we get that the expected cost of operating the laser cutter in this way is £29:

Isabelle Lemma 6.5.1

```
lemma operate-and-fix-cost:
  measure-pmf.expectation (cost-pmf operate-and-fix-cost) real-of-rat = 29
```

6.5.2 Determined Non-Determinism

Our second step towards controlling the complexity of non-deterministic resources makes use of the new probabilistic information. In some cases, such as the outputs of resource actions *InjectL* and *InjectR*, the probability distribution tells us that, out of what is outwardly two outcomes, only one is actually possible. This is the case for

$NonD_P\ (Some\ d)\ x\ y$ where d assigns either *First* or *Second* probability of *1*. We formalise this by adding the following two conditional equations to the resource algebra (where $pmf\ d\ x$ is the probability that d assigns to x):

$$pmf\ d\ First\ \ = 1 \Longrightarrow NonD_P\ (Some\ d)\ x\ y = x$$
$$pmf\ d\ Second = 1 \Longrightarrow NonD_P\ (Some\ d)\ x\ y = y$$

6.5.2.1 Resource Terms

The overall approach is the same as in Section 6.5.1, so we briefly list the specific changes expressing those equations:

- We add the corresponding resource term equivalence rules:

  Isabelle Definition 6.5.9 (Additional equivalences when a child is impossible)

  $$pmf\ d\ First\ \ = 1 \Longrightarrow NonD_P\ (Some\ d)\ x\ y \sim x$$
  $$pmf\ d\ Second = 1 \Longrightarrow NonD_P\ (Some\ d)\ x\ y \sim y$$

- We refine the normal form specification, requiring neither *First* nor *Second* to have probability *1* or the distribution to be unspecified:

  Isabelle Definition 6.5.10 (Normal form that checks for impossible children)

  $$normalised\ (NonD_P\ d\ x\ y) \equiv normalised\ x \wedge normalised\ y \wedge x \neq y \wedge$$
  $$(case\ d\ of\ Some\ p \Rightarrow pmf\ p\ First \neq 1 \wedge pmf\ p\ Second \neq 1 \mid None \Rightarrow True)$$

- We add to the rewrite specification rules corresponding to the left-to-right direction of the equations:

  Isabelle Definition 6.5.11 (Term rewriting rules when either child is impossible)

  $$pmf\ d\ First\ \ = 1 \Longrightarrow res\text{-}term\text{-}rewrite\ (NonD_P\ (Some\ d)\ x\ y)\ x$$
  $$pmf\ d\ Second = 1 \Longrightarrow res\text{-}term\text{-}rewrite\ (NonD_P\ (Some\ d)\ x\ y)\ y$$

- We expand the rewriting step implementation with two new cases:

  Isabelle Definition 6.5.12 (Rewriting step that checks for impossible children)

```
step (NonDP d x y) =
 (      if (case d of Some p ⇒ pmf p First = 1 | None ⇒ False) then x
  else if (case d of Some p ⇒ pmf p Second = 1 | None ⇒ False) then y
  else if ¬ normalised x then NonDP d (step x) y
  else if ¬ normalised y then NonDP d x (step y)
  else if x = y then x
  else NonDP d x y)
```

The proof that the full normalisation procedure *normal-rewr* decides the resource term equivalence once again requires us to show that the two sides of the new equations are joinable. In this case, this is made trivial by our implementation of the rewriting step: the first step in normalisation of the left-hand sides results immediately in the right-hand side. We only need to prove that $pmf\ d\ Second = 1$ implies $pmf\ d\ First \neq 1$, which is a simple consequence of d representing a probability distribution.

What in this case requires more notable changes compared to the previous equation is the translation of resource into ILL propositions. Recall, as we noted when adding the probabilistic information in Section 6.3, that ILL cannot express the probabilistic information and as such we drop it when translating resources into its propositions. Recall also that in our linearity demonstration through ILL we use a deduction mimicking the resource term normalisation procedure to connect translations of equivalent resources with ILL proofs (see Section 4.6).

As a result of the newest two equivalences, the probabilistic information plays a role in the normalisation procedure but cannot be followed by an ILL deduction. For instance, with $pmf\ d\ First = 1$, the term $NonD_P\ (Some\ d)\ x\ y$ is equivalent to x but the following is not a valid sequent of ILL:

$$[res\text{-}term\text{-}to\text{-}ill\ x \oplus res\text{-}term\text{-}to\text{-}ill\ y] \nvdash res\text{-}term\text{-}to\text{-}ill\ x$$

We choose to resolve this as part of *res-term-to-ill* itself, translating $NonD_P\ d\ x\ y$ into *res-term-to-ill* x or *res-term-to-ill* y if d is a PMF fully biased to one of the two choices (otherwise proceeding as before). The deductions following those steps of the normalisation (i.e. the first two branches of the expanded $NonD_P$ case of *step* shown above) then become identity deductions and all the relevant properties are preserved.

This concludes our changes to resource atoms, resulting in the desired equation at the resource level with no new issues in that part of our framework. We next turn to the process theory, where the new equations render the *InjectL* and *InjectR* actions redundant.

6.5.2.2 Compositions

The new equations essentially internalise the *InjectL* and *InjectR* actions in the resource algebra itself. Because their inputs are now equal to their outputs, they can be represented by the identity process and we can prove statements such as the following:

Isabelle Lemma 6.5.2

lemma *output-NonDP-first*:
output (*Identity* (*NonD*$_P$ (*Some* (*return-pmf First*)) *x y*)) = *x*
by *simp*

As such, we remove the two actions from the type of process compositions and all relevant definitions.

Similarly to the translation of resources into ILL, we also need to expand the translation of process compositions (specifically, the equations for *Opt*, *OptDistrIn* and *OptDistrOut*) to treat the new special cases. In the case of *Opt*, if given a distribution biased fully to one of the two choices, we use just the deduction of that child process instead of always combining both. In the case of *OptDistrIn* and *OptDistrOut*, we use an identity deduction for such situations because there is nothing to distribute into or out of.

With these updates, while verifying the properties of the translation of process compositions into deductions of ILL requires more steps in the affected cases, most of the properties are preserved. That is, the resulting deduction has as conclusion the input-output sequent and is well-formed (see Section 4.7). But, because *Opt* may discard one of its branches in the translation, we no longer have that *all* primitives occurring in the process correspond to premises of the deduction. This is now only true for compositions where *Opt* is never fully determined, with a relaxed subset relation holding in general.

As a result of the new equations, the resource algebra will simplify any expression where the probabilistic information allows us to rule out one of the two outcomes. Beyond the immediate effect of making explicit *InjectL* and *InjectR* actions redundant, this simplification can also allow other resource equations to apply and simplify the resource even further. For instance, consider the following chain of equalities which is newly possible:

$$\begin{aligned} &\mathit{NonD}_P\ d\ (\mathit{NonD}_P\ (\mathit{Some}\ (\mathit{return\text{-}pmf}\ \mathit{First}))\ x\ y) \\ &\qquad (\mathit{NonD}_P\ (\mathit{Some}\ (\mathit{return\text{-}pmf}\ \mathit{Second}))\ y\ x) \\ &\qquad = \quad \mathit{NonD}_P\ d\ x\ x \\ &\qquad = \quad x \end{aligned}$$

6.6 Limitations

With the changes made so far, we can express probabilistic processes like the simple faulty machine example in Section 6.5.1.3. We can also use the probabilistic informa-

tion in the resource algebra to remove the need for explicit injection actions. However, there are situations that our extended framework does not capture well. We give some examples in Section 6.6.1 and Section 6.6.2.

As a result, we find that our approach in this chapter does not produce a fully effective framework for reasoning about probabilistic resources. While we are able to integrate probabilistic information into resources and make use of it in processes, its presence induces resource expressions of impractical complexity. That complexity interferes with the ability of our framework to be a helpful tool in practice.

While we find some success in curbing the complexity with the addition of resource equations, and we expect further steps could be made with more resource equations and resource actions, there are deeper issues that they cannot address, such as parallel resources requiring independence of their children. We suspect that, instead of relieving individual sources of the complexity, a more fruitful future research direction would be investigating approaches that more deeply integrate the probabilistic information into processes and resources. See Section 6.7 for our discussion of future work.

6.6.1 Need for More Simplifications

For instance, consider the faulty machine example with a simple change: there is a cheap action with a chance (say 20%) of unblocking the machine and getting a successful product. We can then attempt to unblock first and only move to the (expensive) repair if that fails. This would allow us to explore different expected costs of the process with varying action costs and failure probabilities — for example, in some situations it might be cheaper to skip the unblock attempt altogether. However, that unblocking action would in our present framework result in an overly complex resource.

We can define the unblocking action and the overall process as follows (visualised in Figure 6.2):

Isabelle Definition 6.6.1 (Attempting to unblock the machine)

```
definition unblock =
 Primitive (Res CutterBlocked)
           (NonDP (Some (first-pmf 0.2)) (Res Cutter ⊙ Res CutShape)
                                         (Res CutterBlocked))
           STR ''Unblock''
           2
```

Isabelle Definition 6.6.2 (Operating, attempting to unblock and fixing the machine)

```
definition operate-unblock-fix =
 Seq operate
     (Opt (Some (first-pmf 0.7))
          (Identity (Res Cutter ⊙ Res CutShape))
          (Seq unblock
               (Opt (Some (first-pmf 0.2))
                    (Identity (Res Cutter ⊙ Res CutShape))
                    fix)))
```

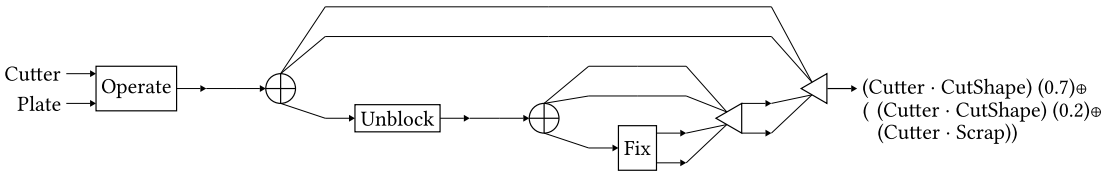


Figure 6.2: Process diagram for operating a faulty machine, with possibly unblocking and fixing it

From this composition we get the output:

$$\begin{aligned} &NonD_P\ (Some\ (first\text{-}pmf\ 0.7)) \\ &\quad (Res\ Cutter \odot Res\ CutShape) \\ &\quad (NonD_P\ (Some\ (first\text{-}pmf\ 0.2)) \\ &\quad\quad (Res\ Cutter \odot Res\ CutShape) \\ &\quad\quad (Res\ Cutter \odot Res\ Scrap)) \end{aligned}$$

Note that this output contains the resource $Res\ Cutter \odot Res\ CutShape$ twice. If they were direct children of one $NonD_P$ term then our resource equations would be able to simplify it. But, because of the separation in nested $NonD_P$ terms, this is currently not possible.

This situation is an instance of a more general pattern where the way nested $NonD_P$ terms are associated stands in the way of simplifying the overall term. The two forms of this pattern are:

$$NonD_P\ (Some\ p)\ x\ (NonD_P\ (Some\ q)\ x\ y)$$
$$NonD_P\ (Some\ p)\ (NonD_P\ (Some\ q)\ x\ y)\ y$$

One possible remedy to such patterns would be the addition of two resource actions to swap the way nested $NonD_P$ resources are associated, appropriately recalculating the associated probabilities. However, we believe that the relatively ad-hoc nature of such fixes does not address the full issue.

6.6.2 Dependence in Parallel Resources

Consider the following three parallel resources:

$$\mathit{Res}\ A \odot \mathit{Res}\ B \odot \mathit{Res}\ C$$
$$\mathit{Res}\ A \odot \mathit{Res}\ B \odot \mathit{Res}\ D$$
$$\mathit{Res}\ X \odot \mathit{Res}\ Y \odot \mathit{Res}\ C$$

We can non-deterministically combine them, assigning probability p to the first, $(1-p)q$ to the second and $(1-p)(1-q)$ to the third. This is expressed by the following resource:

$$\begin{aligned}&\mathit{NonD}_P\ (\mathit{Some}\ (\mathit{first\text{-}pmf}\ p))\\ &\quad(\mathit{Res}\ A \odot \mathit{Res}\ B \odot \mathit{Res}\ C)\\ &\quad(\mathit{NonD}_P\ (\mathit{Some}\ (\mathit{first\text{-}pmf}\ q))\\ &\qquad(\mathit{Res}\ A \odot \mathit{Res}\ B \odot \mathit{Res}\ D)\\ &\qquad(\mathit{Res}\ X \odot \mathit{Res}\ Y \odot \mathit{Res}\ C))\end{aligned}$$

Taking a different view of this situation, we can see that the first atom is either A or X, second is either B or Y and third is either C or D. As such, we might try to express it as the following resource:

$$\begin{aligned}&\mathit{NonD}_P\ (\mathit{Some}\ (\mathit{first\text{-}pmf}\ p_A))\ (\mathit{Res}\ A)\ (\mathit{Res}\ X) \odot\\ &\mathit{NonD}_P\ (\mathit{Some}\ (\mathit{first\text{-}pmf}\ p_B))\ (\mathit{Res}\ B)\ (\mathit{Res}\ Y) \odot\\ &\mathit{NonD}_P\ (\mathit{Some}\ (\mathit{first\text{-}pmf}\ p_C))\ (\mathit{Res}\ C)\ (\mathit{Res}\ D)\end{aligned}$$

But there is no way to assign the probabilities p_A, p_B and p_C in a way that accurately captures the situation. This is because of dependency between the atoms: if the first atom is A then we know the second must be B. We have no way of expressing this dependency and thus can only accurately represent the situation with the earlier resource.

More generally, parallel combination of resources assumes that any non-determinism in each of its children is independent of the other children, because it has no way of expressing any dependency that might be present. As such, in any situation that involves dependency in the probabilities, only the form with non-deterministic combination above parallel combination is accurate.

This in turn can lead to convoluted process compositions. If we, for instance, want to react to the presence of C or D we cannot do so with a simple optional composition but must use multiple optional compositions, parallel composition and identity actions to carefully account for the other atoms and the involved probabilistic information.

6.7 Conclusion

In this chapter we explored how our base framework of resources and process compositions can be expanded to include probabilistic information in its non-deterministic elements. We found that, while adding the probabilistic information itself to resources is unproblematic, the way in which it differentiates previously equal resources has significant implications for which process compositions are valid. This was chiefly a result of optional composition of processes eliminating non-determinism in its output.

After dropping that elimination of non-determinism and, as a result, relaxing the constraints on validity of optional composition, we find that, while the difficulties with composition validity are solved, this leads to increasingly complex and redundant resource expressions. The main source of this is the lack of *any* elimination of non-determinism and the resulting inability to reduce the size of non-deterministic resources even if informally they represent a deterministic situation.

We add three equations to our resource algebra to help address this one-way expansion, aiming to take advantage of "hidden determinism" in the resources. The first regains the elimination of non-determinism, this time within the resource algebra itself rather than a composition operation, by simplifying any non-deterministic resource whose both branches are equal. The other two take advantage of the new probabilistic information to simplify cases where the distribution is biased fully in favour of one branch. These have a positive effect on resource complexity, giving it ways to decrease.

Throughout mechanising the content of this chapter we found the automation available in Isabelle invaluable. It made our exploration of this wide-reaching change to our framework tractable, giving us the confidence that no errors were introduced by our many local changes.

We now highlight some threads of future work based on the work in this chapter.

Deeper solution. As we note in Section 6.6, more resource equations could be added in the future following the same blueprint as our work in this chapter. Adding new resource actions is also possible where the target connection is not suitable for inclusion in the resource algebra, for instance due to not having a clear normal form. Each of these has the potential to improve how efficiently certain situations can be phrased with resources.

However, we choose not to further pursue the addition of even more resource equations and resource actions at present. While they help the situation, the inefficient

resource expressions hint at a deeper problem with adding this kind of probabilistic information to our framework. We expect that this will require a more systematic solution to be satisfyingly resolved, rather than a series of local patches.

One interesting possibility to investigate would be using distributions of deterministic resources as their inputs and outputs of processes. This follows to the extreme the idea that, to express dependent resources, we should place the probabilistic information "above" those resources in the term tree, by moving that information up and out of the resource.

Extended linear logic. When discussing the changes relevant to our demonstration of linearity through linear logic, we noted that it cannot express the probabilistic information. As such, we drop it as part of our translation and proceed with linear logic only demonstrating the linearity aspect of valid process composition.

However, as previously noted, Horne [45] introduces the idea of sub-additives into linear logic, which can express probabilistic choice. As such, in the future we could extend our mechanisation of ILL with these sub-additives and use them to demonstrate probabilistic aspects of our compositions. Note that Horne's work is based on classical linear logic, so adaptations would have to be made to suit our intuitionistic setting.

Chapter 7

Case Studies

In this chapter we describe a number of case studies that demonstrate the main features of our framework in a variety of domains.

In Section 7.1 we demonstrate the process transformations described in Section 3.4 by modelling the process of marking a coursework assignment. We start with an abstract model of the process, which we then refine to include knowledge about the students taking part in the coursework. This refinement allows us to better capture the practical reality of the process. The transformation functions automate the refinement, while their formal verification ensures they preserve validity of the abstract composition.

In Section 7.2 we recreate an example used by Dixon et al. [25] and demonstrate the use of non-deterministic resource combinations and our *Anything* resource. This domain concerns picking socks from a drawer until we have two of the same colour, for which we formalise two plans: one conformant and one contingent.

In Section 7.3 we model assembly workflows in the context of manufacturing metal components. To effectively model the whole range of possible workflows, we define a recursive function that builds tailored process compositions from data available when the component is ordered. We then prove that, for any possible order, the resulting process composition is valid.

In Section 7.4 we model manufacturing in the logistics simulation game Factorio[1]. We formalise the domain in such a way that valid process compositions represent manufacturing processes free from bottlenecks and with correct logistics connections. We implement one such process in the game by following instructions derived from its formal description, and validate that the production rates and power usage match those

[1] `https://factorio.com/`

computed in Isabelle/HOL.

Note that, while we implement most of the models in this chapter in Isabelle/HOL, one could also do so with code generated from our framework into Haskell (or, indeed, any other target language supported by the code generator). The definitions would be nearly identical, because the necessary range of functions is extracted with the same names. This is demonstrated by our formalisation of picking matching socks in Section 7.2 which is done fully in Haskell. What we lose by working outside the proof assistant is our ability to prove properties of these definitions, such as validity of the result for all possible parameters (as we do for instance for the assembly workflows in Lemma 7.3.1). In exchange we get improved performance and interoperability, since we no longer require that everything be formally verified.

7.1 Model of Coursework Marking

Recall our process transformations from Section 3.4. They allow us to systematically transform and refine resources and processes, preserving a formal connection between the original and the result. We now use the example domain of marking a coursework assignment to demonstrate how they can refine a process outline with additional information.

This domain concerns the lifetime of the coursework, from its release to the students, through their submission to eventually releasing marks back to the students. Initially, we only know that the coursework will take place and the outline of the marking process. Later, we obtain the specific students who will take part and can thus refine the outline. Our process transformations allow us to take advantage of the new information without having to remake the process from scratch.

In our initial model, the resources moving between the relevant actions are quite abstract: for example, we represent all students as one resource atom. As a result, that initial model is overly constrained compared to the process in practice: for example, it requires that all student submissions be marked before any mark can be submitted. Once refined with information about the students taking part, this unwanted synchronisation can be removed.

We next introduce the initial model and then the concrete refinements that integrate information about the students taking part. Note that our model could be made more complex to cover further aspects of coursework marking in practice, such as late submissions, non-submissions or internal dependencies within the coursework struc-

ture. We use a simpler model to better demonstrate the transformations, but the same approach would apply to those extensions.

7.1.1 Initial Model

We start our initial model with the resource atoms it involves. The linear atoms cover: the students taking part, their submissions, their marks, the fact that the marks have been submitted and the fact that all marks have been released.

Isabelle Definition 7.1.1 (Initial linear resource atoms)

datatype *lres* = *Students* | *Submissions* | *Marks* | *MarksSubmitted* | *MarksReleased*

The single copyable atom we introduce represents the coursework instructions, such as a list of problems to solve. Note that our initial model does not require it to be copyable, but in the refined model we will want to copy it for each student taking part.

Isabelle Definition 7.1.2 (Copyable resource atoms)

datatype *cres* = *Instructions*

Next, we define the primitive actions that the model will use. These are the basic steps of the process: students making their submissions, marking the submissions, submitting the marks into the system and releasing those marks back to the students once they are all submitted. We mechanise all of these as primitive actions with a simple string label and no metadata:

Isabelle Definition 7.1.3 (Collecting submissions)

definition *collectSubs* =
Primitive (*Copyable Instructions* ⊙ *Res Students*) (*Res Submissions*)
STR ''*Collect Submissions*'' ()

definition *markAll* =
Primitive (*Res Submissions*) (*Res Marks*)
STR ''*Mark Submissions*'' ()

definition *submitMarks* =
Primitive (*Res Marks*) (*Res MarksSubmitted*)
STR ''*Submit Marks*'' ()

definition *releaseMarks* =
Primitive (*Res MarksSubmitted*) (*Res MarksReleased*)
STR ''*Release Marks*'' ()

The full process composition is simple in this initial model, all four actions are composed in sequence. We visualise it with the process diagram in Figure 7.1a, and prove that it is valid and has the desired input and output:

Isabelle Lemma 7.1.1 (Input, output and validity of the initial marking model)

```
lemma marking:
  shows (marking): Copyable Instructions ⊙ Res Students ⟶ Res MarksReleased
    and valid marking
  by (simp-all add: markingProcess-def action-defs)
```

Note how the monolithic resource atom and primitive actions impose constraints on the process that we may not want in practice. For instance, we may want to mark submissions and submit the resulting marks in batches. But for that we need to integrate more knowledge into the model.

7.1.2 Refined Model

The knowledge we integrate is that our initial abstraction of "students" actually consists of a number of individuals, producing individual submissions and requiring individual marks. This then allows us to model that the individual marking actions do not depend on each other, which in turn allows their execution to interleave. That better reflects the practical reality.

To model the new knowledge, we enclose the refined model in a locale that fixes a list of students (which we call *students*) drawn from the type variable *'s*. We name this locale *refined* and enclose the initial model in a trivial locale called *abstract* to avoid any name clashes. In the *refined* locale we also fix a function *name* that assigns a string literal to every member of that type, representing the student's name for purposes of visualisation.

Isabelle Definition 7.1.4 (Refinement locale with named students)

```
locale refined =
  fixes students :: 's list and name :: 's ⇒ String.literal
```

We make this refinement in two main steps: refinement of resources and refinement of actions. As a target for the refinement of resources, we define a new datatype of linear resources to represent the more individual view (we keep the same copyable atom type *cres*):

Isabelle Definition 7.1.5 (Elaborated linear resource atoms)

```
datatype 's lres = Student ('s) | Submission ('s) | Mark ('s)
                 | MarkSubmitted ('s) | MarksReleased
```

Then we can define the function that refines the linear atoms of the initial model into resources of the refined model. For instance, the original atom *Students* is mapped to a parallel combination of *Student s* resources for *s* from *students*.

Isabelle Definition 7.1.6 (Refining initial linear atoms into elaborated resources)

```
primrec refinement :: abstract.lres ⇒ ('s refined.lres, abstract.cres) resource
 where
  refinement abstract.Students = Parallel (map (Res ∘ refined.Student) students)
| refinement abstract.Submissions =
    Parallel (map (Res ∘ refined.Submission) students)
| refinement abstract.Marks = Parallel (map (Res ∘ refined.Mark) students)
| refinement abstract.MarksSubmitted =
    Parallel (map (Res ∘ refined.MarkSubmitted) students)
| refinement abstract.MarksReleased = Res refined.MarksReleased
```

We can then apply this refinement function to resources throughout the original process composition with the following term:

$$\textit{process-refineRes refinement } (\lambda x.\ x)\ \textit{abstract.marking}$$

Instantiating the locale with four illustrative students, the resulting composition is visualised in Figure 7.1b. Using to the properties of *process-refineRes*, we prove that the composition remains valid and its input and output are suitably refined.

Isabelle Lemma 7.1.2 (Input, output and validity after refining resources)

```
lemma marking-refined-resources:
 defines x ≡ process-refineRes refinement id abstract.marking
  shows (x): Copyable Instructions ⊙ Parallel (map (Res ∘ Student) students)
           ⟶ Res MarksReleased
    and valid x
 by (simp-all add: abstract.marking-def abstract.action-defs refine-resource-par)
```

With resources refined, we can use primitive action substitution (see Section 3.4.3) to break up the monolithic collection of submissions, their marking and the submission of marks. For each of those, the substitution requires two definitions: a predicate defining the actions to be replaced and a function generating their replacements. In our case, the three predicates use the input and output of the primitive actions to single them out, while the three functions compose the relevant number of individualised

actions in parallel as a replacement. The collection replacement also needs to make sufficient copies of the *Instructions* resource and interleave them with the individual student resources. We illustrate the substitution on the most complex example, the collection of submissions.

The first part is the target action predicate, which we call *collectionSplit-cond*. In this replacement we are looking for refined *collectSubs* actions, so the predicate is defined as follows:

Isabelle Definition 7.1.7 (Predicate targeting collection of submissions)

```
definition collectionSplit-cond :: ('s refined.lres, abstract.cres) resource
                                 ⇒ ('s refined.lres, abstract.cres) resource
                                 ⇒ String.literal ⇒ unit ⇒ bool
 where collectionSplit-cond in out l m = (
  in = Copyable abstract.Instructions ⊙
       Parallel (map (Res ∘ refined.Student) students) ∧
  out = Parallel (map (Res ∘ refined.Submission) students))
```

The second part is the individualised action that will form the replacement, taking in a copy of the coursework instructions and the individual student, and producing the individual submission:

Isabelle Definition 7.1.8 (Collecting individual submission)

```
definition collectStudent :: 's
                           ⇒ ('s refined.lres, abstract.cres, String.literal, unit) process
 where collectStudent s =
  Primitive (Copyable abstract.Instructions ⊙ Res (Student s))
            (Res (Submission s))
            (STR ''Collect submission of '' + name s) ()
```

Before we can construct the actual replacement compositions, we need to copy our one *Instructions* atom enough times to have one for each student. The input of the *collectStudent* action also requires that we then interleave these copies with the individual student actions. In order to achieve both of these, we define the following two functions to construct the relevant compositions of resource actions.

One is *duplicateToN n x*, which uses the *Duplicate* action to produce *n* copies of the copyable atom *x*. We define it by recursion on the natural number *n*:

Isabelle Definition 7.1.9 (Duplicating an atom into *n* copies)

```
fun duplicateToN :: nat ⇒ 'b ⇒ ('a, 'b, 'l, 'm) process
 where
  duplicateToN (Suc 0) x = Identity (Copyable x)
| duplicateToN (Suc n) x = Seq (Duplicate x)
                               (Par (Identity (Copyable x)) (duplicateToN n r))
| duplicateToN 0 x = Erase x
```

The other is *swapInterleave xs ys*, which uses the *Swap* action to take two lists of parallel resources and interleave them. It does so by recursively swapping the first element of *ys* with all but the first element of *xs* until either runs out:

Isabelle Definition 7.1.10 (Interleaving two lists of resources)

```
fun swapInterleave :: ('a, 'b) resource list ⇒ ('a, 'b) resource list
                      ⇒ ('a, 'b, 'l, 'm) process
 where
  swapInterleave (x # xs) (y # ys) =
   Seq (Par (Par (Identity x) (Swap (Parallel xs) y)) (Identity (Parallel ys)))
       (Par (Identity (x ⊙ y)) (swapInterleave xs ys))
| swapInterleave [] ys = Identity (Parallel ys)
| swapInterleave xs [] = Identity (Parallel xs)
```

With these two functions, we can construct the replacement process composition for collecting submissions. This composition in sequence: duplicates the *Instructions* atom for each student, then interleaves the copies with the student atoms and finally performs *collectStudent* in parallel for each student[2]:

Isabelle Definition 7.1.11 (Replacement for abstract collection of submissions)

```
definition collectionSplit-func :: ('s refined.lres, abstract.cres) resource
                                   ⇒ ('s refined.lres, abstract.cres) resource
                                   ⇒ String.literal ⇒ unit
                                   ⇒ ('s refined.lres, abstract.cres, String.literal, unit) process
 where collectionSplit-func in out l m =
 Seq (Seq (Par (duplicateToN (length students) Instructions)
               (Identity (Parallel (map (Res ∘ refined.Student) students))))
          (swapInterleave (replicate (length students) (Copyable Instructions))
                          (map (Res ∘ refined.Student) students)))
     (par-process-list (map collectStudent students))
```

Using a simpler variant of this pattern we construct target predicates and replacement functions to refine the marking and mark submission actions. We can apply all three action refinements after the resource refinement with the following term:

[2]Full definition of *par-process-list* is given in Appendix A.5.

```
process-subst submitSplit-cond submitSplit-func
( process-subst markSplit-cond markSplit-func
  ( process-subst collectionSplit-cond collectionSplit-func
    ( process-refineRes refinement (λx. x) abstract.marking)))
```

Instantiating the locale with the same four illustrative students, the fully refined composition is visualised in Figure 7.1c. Using to the properties of *process-subst*, we prove that the composition still remains valid and its input and output are unchanged from the resource refinement for any arbitrary list of students.

Isabelle Lemma 7.1.3 (Input, output and validity after refining actions)

```
lemma marking-refined-actions:
 shows (refined.marking):
         Copyable Instructions ⊙ Parallel (map (Res ∘ Student) students) →
         Res MarksReleased
   and valid refined.marking
 using marking-refined-resources by (simp-all add: refined.marking-def)
```

As a result, we have a model that better reflects the marking process without unnecessary synchronisation. We construct it via refinements of an initial outline, representing the integration of more knowledge. Most notably, if we were to change the initial model then our refinements would automatically project that change into the refined model. We see that as an invaluable feature for the maintenance of complex process models.

7.1.3 Concluding Remarks

We chose marking to demonstrate process refinement, because it is a simple case where some information about the process is only available at a later point in time but contributes significantly to the process model.

Moreover, marking is often done by multiple people who need to coordinate their work. While we do not implement this at present, we envision a graphical checklist built on top of this model to support that coordination (see future work in Section 8.1). In this tool, both the course organiser and the individual markers would be able to see the status of all submissions, which would help them more effectively mark all of them.

As the complexity of the process increases, the coordination effort saved by such a tool would also increase. This could, for instance, be the case for coursework assignments that are split into multiple parts, with different markers for different parts. Note that such a split into parts could also be performed in our framework as another refinement of the process in Figure 7.1c.

(a) Process diagram of the initial marking model

(b) Process diagram of the marking process after resource refinement

(c) Process diagram of the marking process after all resource and action refinements

7.2 Three Socks Problem

The three socks problem is an example by Dixon et al. [25], which we use to demonstrate our non-deterministic and *Anything* resources (see Section 2.1) and the *Forget* resource action (see Section 3.1.3). While their work focuses on using deduction in ILL to construct a plan (or, a process), our framework focuses on representing the structure of process compositions. As such, we do not replicate their work in automatically finding the composition but rather represent the two plans that they discuss.

We also take this opportunity to demonstrate how compositions can be built outside the proof assistant using the code generated from our mechanisation (see Section 1.3.3). As such, we use Haskell to implement the resources and process compositions in this section. Note how, apart from the keywords of the language itself and capitalisation changes, the approach remains largely the same as within Isabelle. The main difference is that we cannot prove properties of the processes.

The setting is a simple case of non-determinism in daily life. To quote the authors:

> *The problem is to get a pair of socks from the back of a chest. Because of the location of the socks, their colour cannot be seen until they are taken. The two-colour version of this problem is when there are only black and white socks.*

The domain of this problem has three kinds of socks as linear resource atoms: black, white and hidden. We represent these with a simple datatype with three constructors, one for each kind of sock.

Listing 7.1: Linear atoms for picking socks (with abbreviations)

```
data LRes = Hidden | Black | White
  deriving (Eq, Read, Show)

hidden = res Hidden
black = res Black
white = res White
```

We have one action available to us in this domain: pick a hidden sock from the drawer. The output is a sock that is no longer hidden but either black or white.

Listing 7.2: Primitive action for picking a sock

```
pick :: Process LRes () String ()
pick = Primitive hidden (nonD black white) "Pick sock" ()
```

For convenience, before building compositions, we also define # as infix syntax for the resource product as we did $\odot$ in Definition 2.2.4.

Listing 7.3: Infix syntax for resource product

```
(#) :: Resource a b -> Resource a b -> Resource a b
(#) = resource_par
```

In the case with three socks, our goal is to compose a process P with the following input and output:

$$P\text{: } \mathit{Res\ Hidden} \odot \mathit{Res\ Hidden} \odot \mathit{Res\ Hidden}$$
$$\rightarrow$$
$$\mathit{NonD}\ (\mathit{Res\ Black} \odot \mathit{Res\ Black} \odot \mathit{Anything})\ (\mathit{Res\ White} \odot \mathit{Res\ White} \odot \mathit{Anything})$$

Dixon et al. use the ILL term $\top$ (`top` in their notation) in the goal sequent for their plan to "allow solutions containing more socks than needed." This is because in ILL it is valid to derive $\top$ from any proposition (via the $\top_R$ rule, see Figure 1.1), in this case a proposition representing any number of arbitrarily coloured socks. While nothing else can be done in ILL with this $\top$ proposition, it is perfectly suited to standing in for any configuration of extra socks.

Our resource counterpart to ILL's $\top$ is the *Anything* resource, which represents the presence of some resources without any details of them. The resource action *Forget* is our counterpart to the $\top_R$ rule, accepting any resource as input and producing the *Anything* resource.

A conformant plan for this problem, such as the one shown by Dixon et al. (see Listing 7.4), represents an agent that does not react to the colour of the picked sock. This means that it always uses the *pick* action to resolve all three socks (in any order), even though in some cases it already has a matching pair after resolving only two. Each possible outcome is then massaged into the same form as the goal output.

Listing 7.4: Conformant plan shown by Dixon et al.

```
case_or (pick h1)
(λb1. case_or (pick h2)
  (λb2. case_or (pick h3) (λb3. inl b1⊗b2⊗b3) (λw3. inl b1⊗b2⊗w3))
  (λw2. case_or (pick h3) (λb3. inl b1⊗b3⊗w2) (λw3. inr w2⊗w3⊗b1)))
(λw1. case_or (pick h2)
  (λb2. case_or (pick h3) (λb3. inl b2⊗b3⊗w1) (λw3. inr w1⊗w3⊗b2))
  (λw2. case_or (pick h3) (λb3. inr w1⊗w2⊗b3) (λw3. inr w1⊗w2⊗w3)))
```

Our composition representing the same plan, shown in Listing 7.5, is significantly longer because it is fully explicit about how each case is massaged into the goal output.

Note that this composition also starts with three pick actions in parallel. This is followed by a composition of only resource actions to merge the three non-deterministic outputs. Figure 7.2 shows the process diagram generated by this composition.

Listing 7.5: Conformant plan using our framework

```
conformantPlan :: Process LRes () String ()
conformantPlan = seq_process_list
  [ par_process_list [pick, pick, pick]
  , OptDistrIn (nonD black white # nonD black white) black white
  , Opt
      (seq_process_list
        [ Swap (nonD black white # nonD black white) black
        , OptDistrIn (black # nonD black white) black white
        , Opt
            (seq_process_list
              [ Par (Identity black) (Swap (nonD black white) black)
              , OptDistrIn (black # black) black white
              , Opt (innerNoSwap True black) (innerNoSwap True white)])
            (seq_process_list
              [ Par (Identity black) (Swap (nonD black white) white)
              , OptDistrIn (black # white) black white
              , Opt (innerSmallSwap True) (innerBigSwap False)])])
      (seq_process_list
        [ Swap (nonD black white # nonD black white) white
        , OptDistrIn (white # nonD black white) black white
        , Opt
            (seq_process_list
              [ Par (Identity white) (Swap (nonD black white) black)
              , OptDistrIn (white # black) black white
              , Opt (innerBigSwap True) (innerSmallSwap False)])
            (seq_process_list
              [ Par (Identity white) (Swap (nonD black white) white)
              , OptDistrIn (white # white) black white
              , Opt (innerNoSwap False black) (innerNoSwap False white)])])]
  where fInj isLeft = if isLeft then InjectL else InjectR
        picked isLeft = if isLeft then black else white
        unpicked isLeft = if isLeft then white else black
        innerNoSwap isLeft r =
          let s = picked isLeft; f = fInj isLeft
          in Seq
              (Par (Identity (s # s)) (Forget r))
              (f (black # black # anything) (white # white # anything))
```

```
innerSmallSwap isLeft =
  let s = picked isLeft; r = unpicked isLeft
  in Seq (Par (Identity s) (Swap r s)) (innerNoSwap isLeft r)
innerBigSwap isLeft =
  let s = picked isLeft; r = unpicked isLeft
  in Seq (Swap r (s # s)) (innerNoSwap isLeft r)
```

In contrast, a contingent plan (not shown explicitly by Dixon et al.) represents an agent capable of reacting to the colour of the picked sock. For instance, if the plan is being executed by a human, they could be looking at the socks as they pick them and stop once they have two of the same colour. Same as with the conformant plan, the outcome in each case is massaged into the same form as the goal output, but in two cases the extra sock remains hidden. The composition representing this plan is given in Listing A.1 in the Appendix and visualised in Figure 7.3.

Compared to the term that Dixon et al. show for their conformant plan (Listing 7.4), our process composition is more explicit with respect to *how* the outcomes are massaged to fit the goal output. One part of this is that our composition framework is explicit about the order of parallel resources, which helps us avoid ambiguity. This means that when we encounter *Res Black* $\odot$ *Res White* $\odot$ *Res Black* we must use a swap action to match the first case of the goal output. In the work of Dixon et al., this step does not explicitly appear in the resulting term.

Another point where our representation is more explicit is that we use the *Forget* action to turn extra socks into the *Anything* resource, while no equivalent step appears in the term shown by Dixon et al. The connection between any remaining socks and the `top` term is made by the deduction that generates their plan.

This highlights a subtle difference in our two approaches: we are verifying that a process representing the plan has a certain output, while Dixon et al. are using proof search to generate a plan that satisfies their specification. The conformant plan they present considers the expected eight outcomes resulting from three binary variables and it is in showing that it satisfies the specification that those outcomes are collapsed into two options. In our case, we are deriving the precise output of the process and, as such, take explicit actions as part of that process to bring the outcomes into alignment. This increased focus on the precise output stems from our interest in composition of processes, where their inputs and outputs feature prominently.

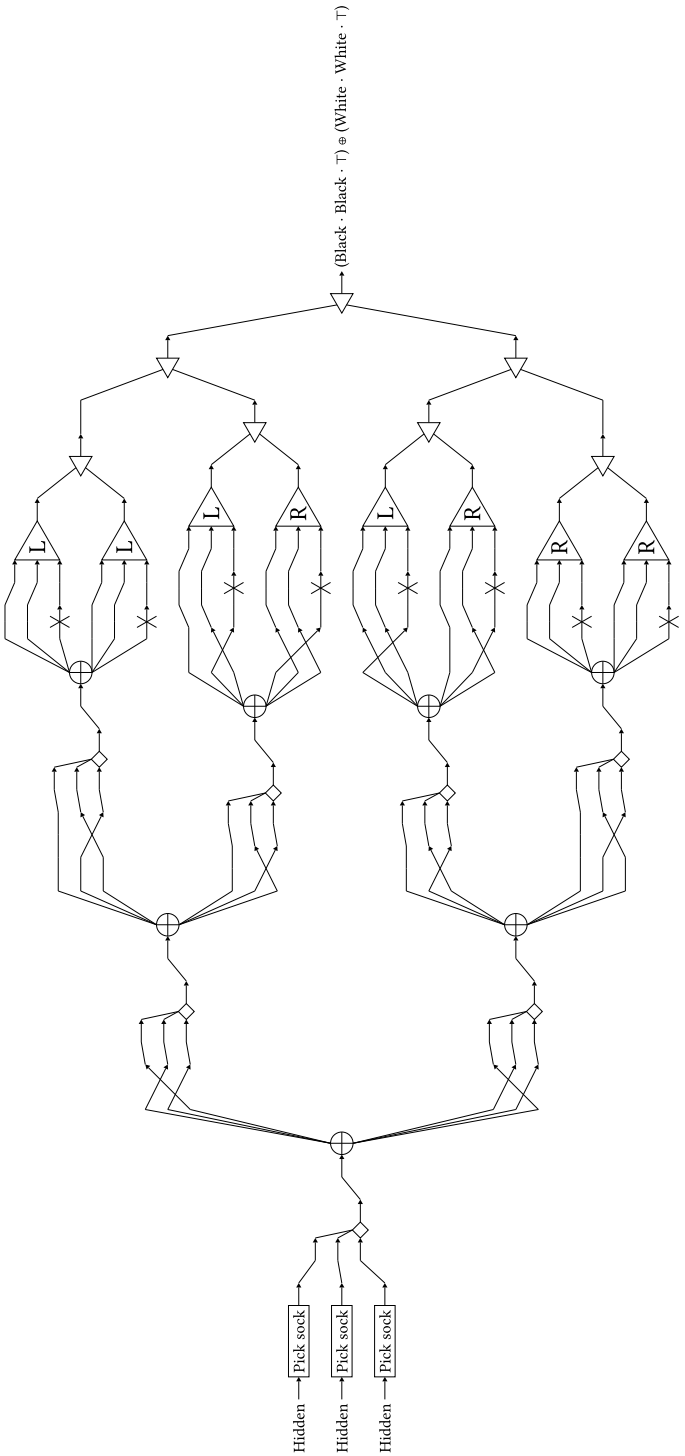


Figure 7.2: Process diagram of the conformant plan for picking two socks of the same colour

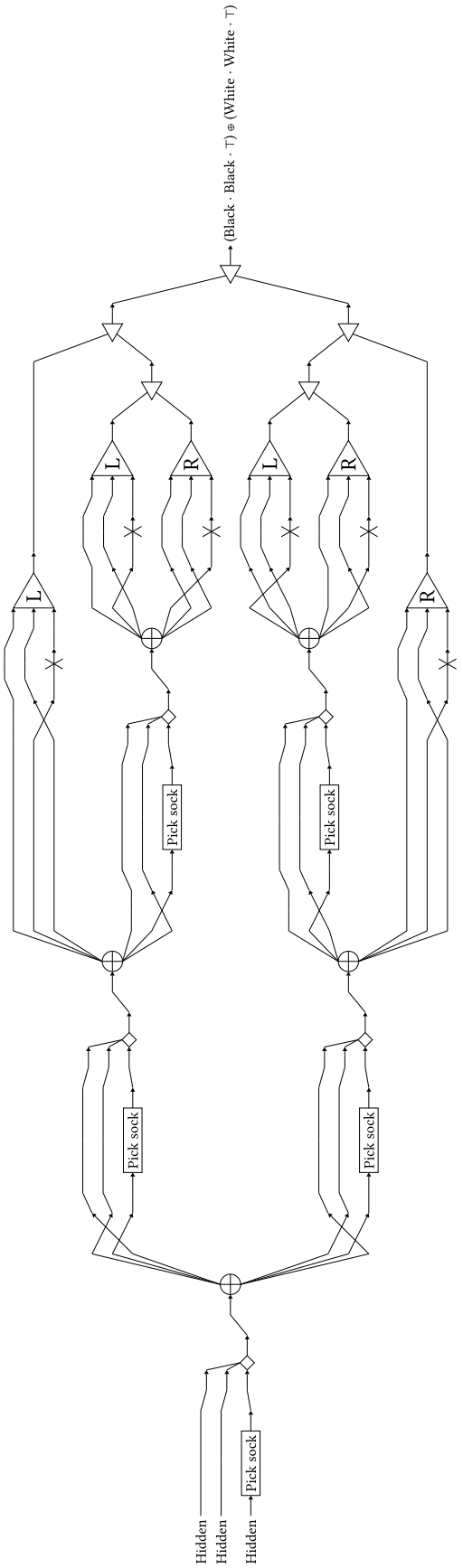


Figure 7.3: Process diagram of the contingent plan for picking two socks of the same colour

7.3 Assembly Workflow Model

As part of our previous work on WorkflowFM (see Section 1.2.1), we presented two case studies of formal manufacturing process models and their use to improve accuracy of scheduling production flows [70]. Those case studies focused on two kinds of process models: standard operating workflows and assembly workflows. In this section we revisit the latter using our framework.

The advantage that our framework offers for assembly workflow is the ability to define compositions through recursive functions. This allows us to generate compositions customised to every individual assembly, something that is not possible to model with compositions in WorkflowFM. Crucially, we prove in general that every instance for any assembly order will be a valid composition.

7.3.1 Assemblies

The context for this case study is the manufacturing of metal components, with each job having a custom production flow. As such, we organise the jobs into *assemblies* that collect the required materials and perform a sequence of operations on them. Crucially, each of the objects brought together can be another assembly, and, as a result, they form trees. We call the data describing the assembly to be produced the *assembly order*.

We wish to model the process of fulfilling an assembly order as the following steps:

1. Fulfil all subassemblies,
2. Gather all subassemblies and materials into one production lot,
3. Perform each operation in sequence on the combined production lot.

The high variability of assembly orders is what makes modelling this process difficult. For instance, there is no limit on the breadth or depth of the assembly tree and so every model may have an arbitrary number of “assemble” actions and operation sequences. Similarly, there is no limit to the number of operations in the sequence and so every model may have an arbitrary number of “perform operation” actions. The precise values for these parameters are only known with a concrete assembly order, not at modelling time.

In the WorkflowFM model [70], this is handled outside of the reasoner as part of the simulation environment. Using our present framework, we define a function from assembly orders to process compositions. One could view this function as a kind of

generic description of the process and the result of running the function for a particular order as the actual model. This allows us to prove that these compositions are valid for all assembly orders.

7.3.2 Assembly Order Formalisation

We start by formalising the assembly orders, which consist of other orders and operations. In our model we give each order a string identifier and we label each operation with a string describing it. In a practical setting these types could contain additional information, such as machine descriptions and configuration, possibly drawn from an enterprise resource planning (ERP) system.

Isabelle Definition 7.3.1 (Datatypes for operations and assembly orders)

datatype *operation* = *Op* (*op-label*: *String.literal*)
datatype *order* =
Order (*orderID*: *String.literal*) (*suborders*: *order list*) (*operations*: *operation list*)

7.3.3 Resource Atoms

The first part formalising this domain is the resource atoms. Here all our atoms are linear: production lots (abstracting what in practice would be an identifier from an ERP system) and machines.

Isabelle Definition 7.3.2 (Linear resource atoms)

datatype *lres* = *Lot* | *Machine*

We use string literals as labels for actions and attach no metadata to them. Thus the process type for this domain is the following:

(*lres, unit, String.literal, unit*) *process*

7.3.4 Primitive Actions

The second part of formalising this domain is the primitive actions:

- Reserve and free a machine. These represent interaction with the larger (unmodelled) environment of the factory and are a crucial point when simulating a large number of process instances from this domain with some shared pool of machines. In simulation, the implementation of reserving a machine would block

until a machine was provided to it by the external environment. Note that by introducing this primitive we are making an assumption that it is always possible to eventually provide the machine.

Isabelle Definition 7.3.3 (Reserving and freeing a machine)

definition *reserve* = *Primitive Empty* (*Res Machine*) *STR ''Reserve machine''* ()
definition *free* = *Primitive* (*Res Machine*) *Empty STR ''Free machine''* ()

- Perform some operation using a machine on a production lot.

  Isabelle Definition 7.3.4 (Performing an operation)

  definition *perform op* =
  Primitive (*Res Machine* ⊙ *Res Lot*) (*Res Machine* ⊙ *Res Lot*)
  (*STR ''Do ''* + *op-label op*) ()

- Bring a number of production lots together, each resulting from a subassembly. This also represents retrieving any required raw resources from storage.[3]

  Isabelle Definition 7.3.5 (Assembling lots for an order into one)

  definition *assemble o* =
  Primitive (*Parallel* (*replicate* (*length* (*suborders o*)) (*Res Lot*))) (*Res Lot*)
  (*STR ''Assemble ''* + *orderID o*) ()

Note that interaction with the environment through the *Reserve* action could lead to deadlock of multiple interacting process instances. For instance, if there are two machines in the shared pool and two process instances each hold one and require reserving a second machine before they free the first one. Modelling individual processes does not prevent this situation, we would have to model all of the interacting processes to guarantee deadlock freedom.

7.3.5 Assembly Composition

First, recall that our goal process needs to perform a sequence of operations. We define two functions, one that handles performing the operation while reserving and freeing a machine, and another to do this for a sequence of operations:

[3] We could also model this by expanding the assembly order type and adding another primitive action. This would complicate the model without enhancing our point, so we choose to keep our model simple for this discussion.

Isabelle Definition 7.3.6 (Performing an operation with a machine[4])

```
fun op :: operation ⇒ (lres, String.literal, meta) process
 where op x = seq-process-list [
  Par reserve (Identity (Res Lot))
 , perform x
 , Par free (Identity (Res Lot))]
```

Isabelle Definition 7.3.7 (Performing a sequence of operations)

```
fun ops :: operation list ⇒ (lres, String.literal, meta) process
 where
  ops [] = Identity (Res Lot)
| ops [x] = op x
| ops (x#xs) = Seq (op x) (ops xs)
```

Next, the assembly process composition for any order is given in Definition 7.3.8. While defined using general recursion, it terminates because every suborder is necessarily smaller than the order that contains it.

Isabelle Definition 7.3.8 (Assembling according to given order[5])

```
function assm :: order ⇒ (lres, String.literal, meta) process
 where assm x = seq-process-list [
  par-process-list (map assm (suborders x))
 , assemble (length (suborders x))
 , ops (operations x)]
```

This function has three sequential steps. First, we recursively apply it in parallel to all subassemblies. Second, we assemble their results into one production lot. Third, we perform the desired sequence of operations on them.

As was our goal, *assm* builds tailored compositions for assembly orders with any depth, breadth and number of operations. It does so fully within our framework, allowing us to prove that it is always valid and results in a production lot:

Isabelle Lemma 7.3.1 (Assembling is correct for any order)

```
lemma assm:
 (assm order): Empty → Res Lot
 valid (assm order)
```

[4] Full definition of *seq-process-list* is given in Appendix A.5.
[5] Full definition of *par-process-list* is given in Appendix A.5.

7.3.6 Concluding Remarks

In this case study we represented workflows for a whole range of assembly orders by taking advantage of the fact that we can compose processes through arbitrary recursive functions. This demonstrates our framework's ability to verify compositions that can be tailored with complex parameters.

Note that our inspiration for this case study also involves simulation of these processes in order to find the optimal schedule [70]. At present, our framework does not integrate with a simulator and, as such, we cannot replicate this aspect. However, once such an integration is available, we believe our models will be easier to simulate, because much of their complexity is contained in the construction of the model and not the process itself.

7.4 Balanced Manufacturing in Factorio

In this section we demonstrate how we can use our framework to formalise solid manufacturing in the logistics simulation game Factorio[6]. Its notion of manufacturing fits well with our notion of processes with inputs and outputs, and its simulation engine offers a way to implement a process and validate its properties in an idealised situation.

This game has previously been used to simulate logistics and process performance. Reid et al. [76] use it to formulate the logistic transport belt problem, which concerns the optimal placement of logistic elements to transport items between locations in the presence of obstacles. Boardman and Krejci [15] use Factorio as a simulation and visualisation aid in lessons about production and inventory control.

We start our discussion with the motivation for the formalisation choices for this domain. Then we discuss our formalisation of item flows and their logistics actions. We then extend this domain with machine blocks, allowing us to express manufacturing. With all of these defined, we can start forming process compositions, which we exemplify with a specific manufacturing process. Then we discuss a Haskell program, based on code generated from these definitions, that constructs instructions for implementing these compositions in the game.

[6] `https://factorio.com/`

7.4.1 Problem Setup

We represent manufacturing in Factorio as process compositions, taking a *steady state* perspective. This means representing the average performance over an infinite period of observation. For instance, if a machine takes five minutes to finish an operation and produces 600 units, then we would treat it as having an output rate of 2 units per second. This is natural in Factorio: we build factories that continually transform inputs into outputs without player intervention. The same steady state perspective is also taken by various user-developed assistants, such as Helmod[7] and Factory Planner[8].

We want the validity of compositions in this domain to ensure that the process is perfectly balanced. This means that all connected input and output rates match throughout the process. As such, one kind of object is so-called "item flows" which represent the arrival of some amount of a certain item type per second.

Beyond the rate of production and consumption, we want the compositions to contain information about the required logistics connections, i.e. how items move between locations. Thus we include location as part of item flows to represent *where* the items are arriving. Note that representing locations by coordinates would make our models overspecified, so we instead use abstract named locations. The precise layout of machines and routing of logistics is left to the agent (human or artificial) implementing the process. Figure 7.4 depicts a section of a transport belt navigating between obstacles.

Finally, the compositions should also account for the machines taken up by the manufacturing. Because in this domain we often use large numbers of machines, we use "machine blocks" to represent groups of machines performing the same recipe in parallel using common input and output flows. These machine blocks integrate with logistics through these input and output flow locations. Figure 7.5 depicts a block of four assembling machines in-game, all manufacturing iron gears from iron plates.

We first define a domain with just item flows as resource atoms (see Section 7.4.2) and primitive actions that represent logistics: splitting, merging and moving the flows (see Section 7.4.3). Then we extend this domain with machine blocks as further resource atoms (see Section 7.4.4), which allows us to represent manufacturing with a primitive action in the combined domain (see Section 7.4.5).

As an application of the process models from this domain, we define how every valid composition can be turned into a sequence of instructions for its implementation in the game (see Section 7.4.7). By mechanically following these instructions a

[7] `https://mods.factorio.com/mod/helmod`
[8] `https://mods.factorio.com/mod/factoryplanner`

Figure 7.4: Complex transport belt layout, avoiding a rock and a tree.

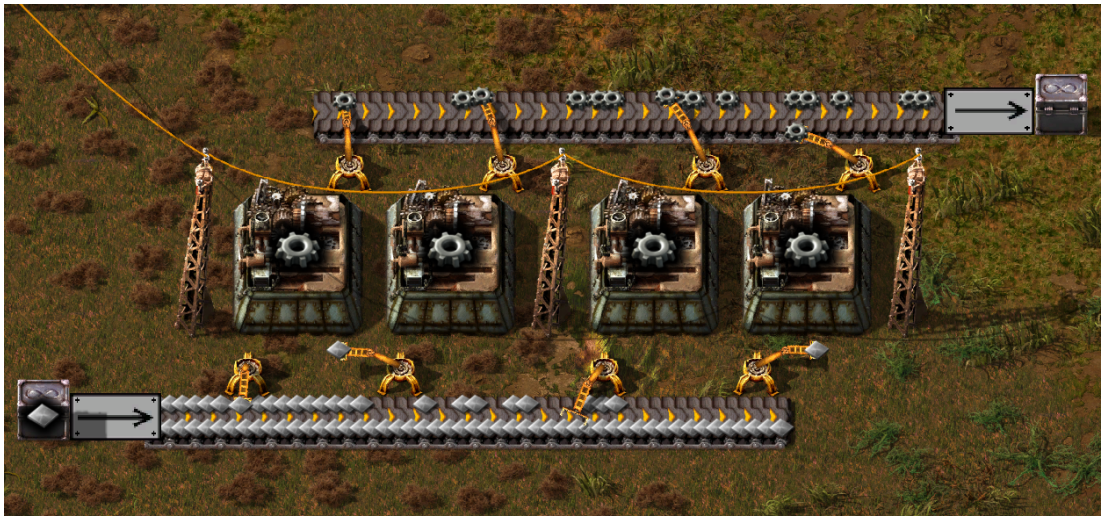


Figure 7.5: Block of four assembling machines, all manufacturing iron gears from iron plates, with an input location on the left edge and output location on the right edge.

player can implement the process. Given a solution to the machine and logistics layout problems, an automated agent could also be made to mechanically follow these instructions.

This last part builds a text output (see Listing 7.6 in Section 7.4.7) based on the given composition, so there is little we could gain by formalising it in Isabelle/HOL. We instead implement it in Haskell on top of the formally verified code generated from the formalisation of this domain.

7.4.2 Item Flows

We start our formalisation of this domain by defining located flows of items. For this we have to formalise locations, item types and rates of flow.

We only need locations to be uniquely identified, so that we know when two things are at the same location. We choose to use string literals, rather than numeric identifiers, because they are more convenient when for example printed as part of the instructions. The same reasoning applies when using string literals for action, item and machine labels in the rest of this section.

Isabelle Definition 7.4.1 (Type synonym designating locations as string literals)
type-synonym *loc* = *String.literal*

We define item types simply by their label, such as an iron plate:

Isabelle Definition 7.4.2 (Datatype for item types, with an example)
datatype *item* = *Item* (*itemLabel*: *String.literal*)
definition *iron-plate* = *Item STR ''Iron Plate''*

For rates of flow we use rational numbers (*rat* in Isabelle/HOL). They are sufficient to express rates and, compared to for instance the reals, their formalisation has a simple translation into executable code in the form of fractions.

Item flows simply combine the item type, rate of flow and location in one datatype:

Isabelle Definition 7.4.3 (Datatype for flows of items)
datatype *flow* = *Flow* (*flowItem*: *item*) (*flowRate*: *rat*) (*flowLoc*: *loc*)

Because we will often be expressing item flows as individual resources, we define more convenient Isabelle notation for them:

item[*rate*] *at location*

stands for:

Res (*Flow item rate location*)

7.4.3 Logistics Actions

We have five primitive actions representing logistics tools of the game for item flows (with first three shown in Figure 7.7):

Moving a flow

Take a flow of given item type and rate from one location to another.

Splitting a flow

Split a flow into two flows of given rates. The rate the original flow must have is the sum of the given rates.

Merging flows

Take two flows of the same item at the same location and merge them into one.

Creating and empty flow

Create from nothing a flow of any item type at any location with zero rate.

Discarding an empty flow

Discard a flow of any item type at any location that has zero rate.

Before defining the actions themselves we specify the metadata they will carry. In general, we use this data to support applications that inspect or consume compositions. In this case, the application is the construction of a sequence of instructions for implementing the process. Thus our metadata carries the information the user needs to implement each action. We formalise the metadata in the same order as the inductive datatype *flow-meta* in Isabelle/HOL:

Isabelle Definition 7.4.4 (Datatype for metadata of logistics actions)

datatype *flow-meta* =
  Move (*item*) (*rat*) (*loc*) (*loc*)
| *Split* (*item*) (*loc*) (*rat*) (*rat*)
| *Merge* (*item*) (*loc*) (*rat*) (*rat*)
| *Unit* (*item*) (*loc*)
| *Counit* (*item*) (*loc*)

The processes in this domain then use item flows as linear atoms, no copyable atoms, string literals as labels and *flow-meta* as metadata. We define the moving, merging and splitting of item flows as follows:

Isabelle Definition 7.4.5 (Moving, splitting and merging flows of items)

definition *move* :: *item* ⇒ *rat* ⇒ *loc* ⇒ *loc*
⇒ (*flow*, 'c, *String.literal*, *flow-meta*) *process*
where *move i r l k* = *Primitive* (*i*[*r*] *at l*) (*i*[*r*] *at k*)
STR ''*Move flow*'' (*Move i r l k*)

definition *split* :: *item* ⇒ *loc* ⇒ *rat* ⇒ *rat*
⇒ (*flow*, 'c, *String.literal*, *flow-meta*) *process*
where *split i l r s* = *Primitive* (*i*[*r*+*s*] *at l*) (*i*[*r*] *at l* ⊙ *i*[*s*] *at l*)
STR ''*Split flow*'' (*Split i l r s*)

definition *merge* :: *item* ⇒ *loc* ⇒ *rat* ⇒ *rat*
⇒ (*flow*, 'c, *String.literal*, *flow-meta*) *process*
where *merge i l r s* = *Primitive* (*i*[*r*] *at l* ⊙ *i*[*s*] *at l*) (*i*[*r*+*s*] *at l*)
STR ''*Merge flows*'' (*Merge i l r s*)

where, for instance, a *merge i l r s* action takes two flows of the same item *i* at the same location *l* but with different rates *r* and *s*, and produces one with item and location unchanged but with summed rate. The *split* action does the opposite and the *move* action changes only the location. See Figure 7.6 for example diagrams of these actions and Figure 7.7 the in-game transport belt segments.

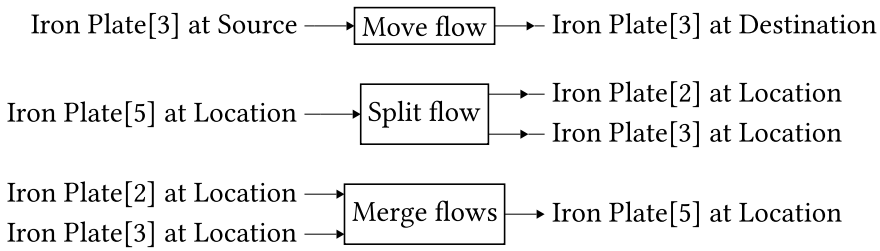


Figure 7.6: Process diagrams for example instances of *move*, *split* and *merge*

The remaining two actions represent our assumption that we can create and discard zero-rate flows for any item at any location. We name these actions *unit* and *counit* and define them as follows, visualised with process diagrams in Figure 7.8:

Isabelle Definition 7.4.6 (Creating and discarding zero-rate flows of items)

definition *unit* :: *item* ⇒ *loc* ⇒ (*flow*, 'c, *String.literal*, *flow-meta*) *process*
where *unit i l* = *Primitive Empty* (*i*[*0*] *at l*) *STR* ''*Create zero flow*'' (*Unit i l*)

definition *counit* :: *item* ⇒ *loc* ⇒ (*flow*, 'c, *String.literal*, *flow-meta*) *process*
where *counit i l* = *Primitive* (*i*[*0*] *at l*) *Empty STR* ''*Discard zero flow*'' (*Counit i l*)

7.4.4 Machine Blocks

Next we formalise machines and machine blocks, so that we can add them to the resource atoms and use them in manufacturing actions. Machines are defined by their

Figure 7.7: Three transport belt segments representing the logistic actions *move*, *split* and *merge* respectively.

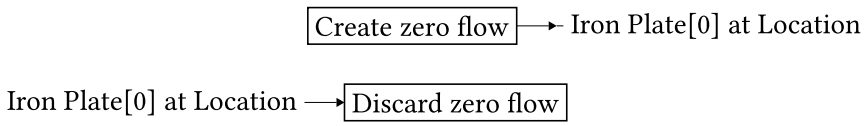


Figure 7.8: Process diagrams for example instances of *unit* and *counit* actions

name, processing speed, base energy demand and running energy demand. For instance, the electric furnace processes items at double of base speed, consumes 6 *kW* of power while idle and an additional 180 *kW* while running.

Isabelle Definition 7.4.7 (Datatype for machines, with an example)

datatype *machine* =
Machine (*machineLabel*: *String.literal*) (*machineSpeed*: *rat*)
(*machineDrain*: *nat*) (*machineConsu*: *nat*)

definition *electric-furnace* = *Machine STR "Electric Furnace" 2 6000 180000*

Note that the machine processing speed is a rational number to allow for machines taking more than base time (i.e. speed less than one). The two energy demands, one when the machine is idle and one when it is running, are both natural numbers representing watts.

Machine blocks consist of the machine description, number of machines, input location and output location:

Isabelle Definition 7.4.8 (Datatype for blocks of machines working together)

datatype *mach-block* =
MachBlock (*mblockMach*: *machine*) (*mblockCount*: *nat*)
(*mblockIn*: *loc*) (*mblockOut*: *loc*)

As with item flows, we define more convenient Isabelle notation for machine blocks as individual resources:

$$mach\langle count\rangle \ at\ in\text{-}loc \rightarrow out\text{-}loc$$

stands for:

$$Res\ (MachBlock\ mach\ count\ in\text{-}loc\ out\text{-}loc)$$

7.4.5 Manufacturing Action

Manufacturing in our formalisation of this domain means using a block of machines to perform a recipe, converting one set of item flows into another. Recipes are defined by two lists of item–count pairs (one for the inputs and one for the outputs), the base time required, the machine used and a name. For instance, there is a recipe for crafting an iron gear from two iron plates, taking half a second and using a tier-one assembling machine.

Isabelle Definition 7.4.9 (Datatype for manufacturing recipes, with an example)

datatype *recipe* =
Recipe (*recIn*: (*item* × *nat*) *list*) (*recOut*: (*item* × *nat*) *list*)
(*recTime*: *rat*) (*recMach*: *machine*) (*recLab*: *String.literal*)

definition *craftGear* =
Recipe [(*iron-plate*, 2)] [(*iron-gear*, 1)] *0.5 assembling-machine-1 STR ''Craft Gear''*

Performing a recipe with some number of machines in parallel requires flows of its inputs at one location and produces flows of its outputs at another location. We define a function to help us construct these flows:

Isabelle Definition 7.4.10 (Constructing a flow from recipe and machine information)

fun *stacksPerTimeAtSpeed* :: *rat* ⇒ *loc* ⇒ *nat* ⇒ *rat* ⇒ *item* × *nat* ⇒ *flow*
where *stacksPerTimeAtSpeed t l n s* (*i*,*c*) = *Flow i* (*s* ∗ (*n*∗*c*) / *t*) *l*

which takes the recipe's time requirement t, machine block location l, number n of parallel machines running the recipe, their processing speed s, the item i and its count c, and constructs the corresponding item flow. The last two arguments being taken as

a pair is important to make this function suitable for mapping over the item–count lists with which recipes are defined.

The final piece of preparation is the metadata, which in this case collects the recipe, number of parallel instances, input location and output location:

Isabelle Definition 7.4.11 (Datatype for metadata of the manufacturing action)

datatype *manu-meta* = *Perform* (*recipe*) (*nat*) (*loc*) (*loc*)

To express manufacturing actions, we move into a domain that combines both item flows and machine blocks. This means that our processes now use as linear resource atoms the sum of those two types: *flow* + *mach-block*. Similarly, as metadata they use the sum of item flow and manufacturing metadata: *flow-meta* + *manu-meta*. The two constructors of Isabelle's sum type, which inject a value from either summand into the sum type, are *Inl* :: $'a \Rightarrow 'a + 'b$ and *Inr* :: $'b \Rightarrow 'a + 'b$.

We use a recipe, number of machines to use and two locations to define the manufacturing primitive action *perform* as follows:

Isabelle Definition 7.4.12 (Performing a recipe with a block of machines)

```
fun perform :: recipe ⇒ nat ⇒ loc ⇒ loc
                ⇒ (flow + mach-block, 'c,
                    String.literal, flow-meta + manu-meta) process
 where perform (Recipe ins outs time m name) n l1 l2 =
 Primitive
  ( Parallel (map (Res ∘ Inl ∘ stacksPerTimeAtSpeed time l1 n (machineSpeed m))
                  ins) ⊙
    Res (Inr (MachBlock m n l1 l2)))
  ( Parallel (map (Res ∘ Inl ∘ stacksPerTimeAtSpeed time l2 n (machineSpeed m))
                  outs))
   name
  ( Inr (Perform (Recipe ins outs time m name) n l1 l2))
```

where the input resource is a parallel combination of item flows (one for each input item of the recipe) along with a suitable block of machines. The output resource is a parallel combination of only the item flows for the recipe's outputs. The function *machineSpeed* merely projects the second argument from the *Machine* constructor, its processing speed.

Figure 7.9 shows an example diagram of the *perform* action, using a recipe for making train cargo wagons. It takes as input three item flows of the recipe ingredients (iron gears, iron plates, steel plates) arriving at the location named "Source" and the two assembling machines used to perform the recipe. Those machines take inputs

from "Source" and deposit their products at another location named "Destination". See Figure 7.10 for further instances of this action.

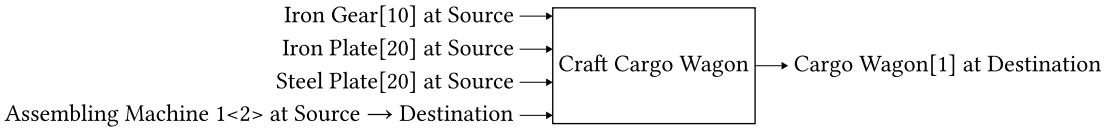


Figure 7.9: Process diagram for crafting cargo wagons on two machines in parallel.

The machine block is consumed by this action to represent occupying those machines with continuous production. This prevents us from using the same group of machines for two tasks at the same time. The block's locations are set to match those where the input and output flows are generated, and its size is set to match the multiplier used to compute the flow rates.

With all the primitive actions set up we can proceed with constructing compositions. In the following two sections we describe one such composition and how instructions are mechanically generated from such compositions.

From this point on we work in the combined domain, so we redefine our custom notation to use the appropriate injections. The item flow notation *item*[*rate*] *at location* and the machine block notation *mach*⟨*count*⟩ *at in-loc* → *out-loc* now apply *Inl* and *Inr* respectively before constructing the resource atom.

Note that all processes from the item flow domain can be easily translated into the combined domain. We just apply the left injection constructor *Inl* to all linear resource atoms and metadata (leaving copyable atoms and labels unchanged) through the process map: *map-process Inl id id Inl*. This map is defined automatically by Isabelle thanks to the BNF structure of resources (see Section 2.4). See Section 3.4 for a discussion of such systematic transformations of processes.

7.4.6 Example Composition: Four Iron Gears

As an example composition we consider the process of turning iron ore into four iron gears per second. This has two manufacturing steps: the iron ore must be smelted into plates and those then crafted into gears. More precisely, considering the flow rates and locations, the process has four steps:

1. Smelt iron ore into plates with 13 furnaces in parallel.

2. Split the resulting iron plates into two flows with rates 0.125 and 8.

3. Move 8 iron plates per second to the next location.

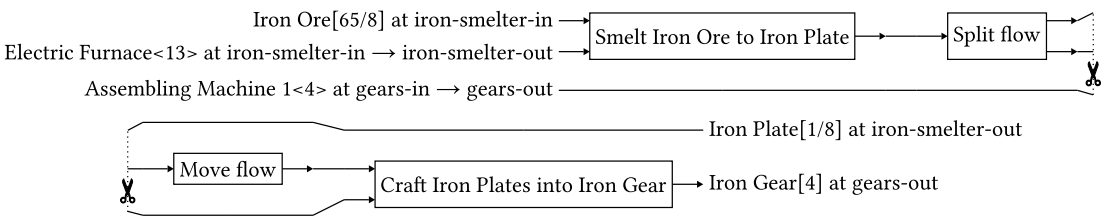
4. Craft iron plates into gears with 4 assembly machines in parallel.

Our choice of targeting 100% machine utilisation means that the furnaces produce more iron plates than we need, which could be made available to another process later on. Alternatively, one could relax the utilisation requirement to require for n machines utilisation of at least $\frac{n-1}{n}$, meaning at most one machine from the group is not fully utilised. Then the action of performing a recipe could be given a fractional number of parallel instances to meet any rational output flow rate. We go with the 100% requirement that sometimes yields a surplus, but our composition rules ensure that it is correctly accounted for and never lost (see Section 3.2).

The earlier four steps are reflected in the actual process composition, which is shown in Figure 7.10: Note that this composition is parameterised by the four involved locations, as they only have to be kept consistent between the individual actions. In the second and third steps we translate logistics action, splitting and moving a flow respectively, into the combined domain using the process mapper.

```
definition fourGears lIPlateIn lIPlateOut lGearIn lGearOut =
  ( perform smeltIronOre 13 lIPlateIn lIPlateOut ||
    Identity (assembling-machine-1<4> at lGearIn → lGearOut)) ;;
  ( map-process Inl id id Inl (split iron-plate lIPlateOut 0.125 8) ||
    Identity (assembling-machine-1<4> at lGearIn → lGearOut)) ;;
  ( Identity (iron-plate[0.125] at lIPlateOut) ||
    map-process Inl id id Inl (move iron-plate 8 lIPlateOut lGearIn) ||
    Identity (assembling-machine-1<4> at lGearIn → lGearOut)) ;;
  ( Identity (iron-plate[0.125] at lIPlateOut) ||
    perform craftGear 4 lGearIn lGearOut)
```

(a) Definition of the process. We use ;; and || as infix notation in Isabelle for sequential and parallel composition respectively.



(b) Process diagram.

Figure 7.10: Process definition and diagram for crafting iron gears from ore

On top of the four core steps, the above composition also involves identity processes that ensure resources that are not being used in a given step are carried safely to the next. One can find a pattern to these accommodations and define helper functions to more easily insert the identity processes, but we omit this to keep the example more direct.

Now we briefly discuss what implications there are from the validity of this composition, and more generally other compositions in this domain. As discussed in Section 3.2, process composition validity in general requires that resources match on both sides of a connection.

Because two item flows are only equal if their rates are equal, we know that there is no bottleneck introduced from the connections made by the composition. Then, because all of the manufacturing actions are set to run their machines at full capacity, we know the whole process will be running at 100% utilisation.

Furthermore, item flows are also only equal if their locations are equal, which means that any change in item locations must come from primitive actions — in our case either a movement or a manufacturing action. Therefore these primitive actions describe all the required logistics connections.

Finally, the input and output of the composition tell us:

- What items the process needs to keep working;
- What items it will produce;
- What kind of production capacity (i.e. machines) it requires to do so.

This information, along with the primitive actions contained in the composition, is all we need to generate instructions for implementing the composition as a factory in the game. We discuss these instructions next, focusing on the process composition we introduced above.

7.4.7 Generating User Instructions

We generate a sequence of instructions that can be mechanically carried out to build the factory. In these we instruct the user on the placement and connection of the various machines.

Implementing our process compositions as factories in the game requires laying out each individual machine in the world and precisely routing the logistics connections. In the absence of a fully automated solution for this problem, we rely on a human player

to carry out the instructions. The instructions may state that two locations should be connected, but they do not try to specify how exactly that connection should look in terms of coordinates. However, if this layout is automated in the future, then our instructions could be adapted for an automated agent.

The instructions (see Listing 7.6 for an example) are split into four parts:

- Requirements, which are generated from the process input and state what item flows and machines need to be provided.
- Actions, which are generated from the primitive actions in the composition and describe how those should be implemented. For manufacturing this means describing the recipe and the machine block to use. For movement this means describing the item type, minimum required throughput, origin and destination.
- Effects, which are generated from the process output and state what item flows (and potentially unused machines) we get out of the process.
- Energy demand, which is calculated from the energy drain of machines consumed by the process. The fact that all used machines are fully utilised makes this calculation a simple sum.

Listing 7.6: Instructions for the *fourGears* composition

```
Process composition "fourGears":
Requirements:
 - Have Iron Ore arriving to iron-smelter-in at the rate of 65/8 per
    second
 - Have 13 of Electric Furnace taking inputs from iron-smelter-in and
    passing outputs to iron-smelter-out
 - Have 4 of Assembling Machine 1 taking inputs from gears-in and passing
     outputs to gears-out
Actions:
 - Set 13 of Electric Furnace to perform recipe "Smelt Iron Ore to Iron
    Plate" on inputs from iron-smelter-in and pushing outputs to iron-
    smelter-out
 - Split flow of Iron Plate located at iron-smelter-out from rate 65/8
    per second into 1/8 per second and 8 per second
 - Move 8 per second of Iron Plate from iron-smelter-out to gears-in
```

```
 - Set 4 of Assembling Machine 1 to perform recipe "Craft Iron Plates
     into Iron Gear" on inputs from gears-in and pushing outputs to gears-
     out
Effects:
 - Have Iron Plate arriving to iron-smelter-out at the rate of 1/8 per
     second
 - Have Iron Gear arriving to gears-out at the rate of 4 per second
Energy demand: 2728000 W
```

The way we generate these instructions is the only part of our work on this domain that is outside of the proof assistant. We use Haskell code automatically generated by Isabelle/HOL for all of the above definitions, including the *fourGears* composition. On top of this code we implement a number of functions for pretty-printing the data involved, such as printing the fully formal natural numbers as plain integers, and construct the actual instructions. We construct individual instructions by taking the relevant part of the composition (e.g. its input or primitive actions) and expressing it in natural language.

We validated the above instructions for the *fourGears* composition by carrying them out in-game, with the result shown in Figure 7.11. The resulting factory exhibits no bottlenecks between the machine blocks, just as validity of the composition predicts. Furthermore, the production rates recorded by the simulation engine match those in the composition, and the recorded energy demand matches the one calculated from the consumed machines. The only difficulty we encountered in following the instructions is implementing the precise item flow split ratio (in this case $1 : 64$) with the simple logistics tools made available (a $1 : 1$ splitter).

7.4.8 Possible Extensions

While our formalisation of this domain is already non-trivial, there are still many aspects of Factorio that we do not model. However, most of those could be modelled with our framework. We note some here as possible future work extending this case study.

For instance, we currently only formalise solid manufacturing, while Factorio also involves fluids. These have their own logistics system through the use of pipes, pumps and tanks. They could be included in a way analogous to the item flows, while extending recipes and manufacturing actions to include them.

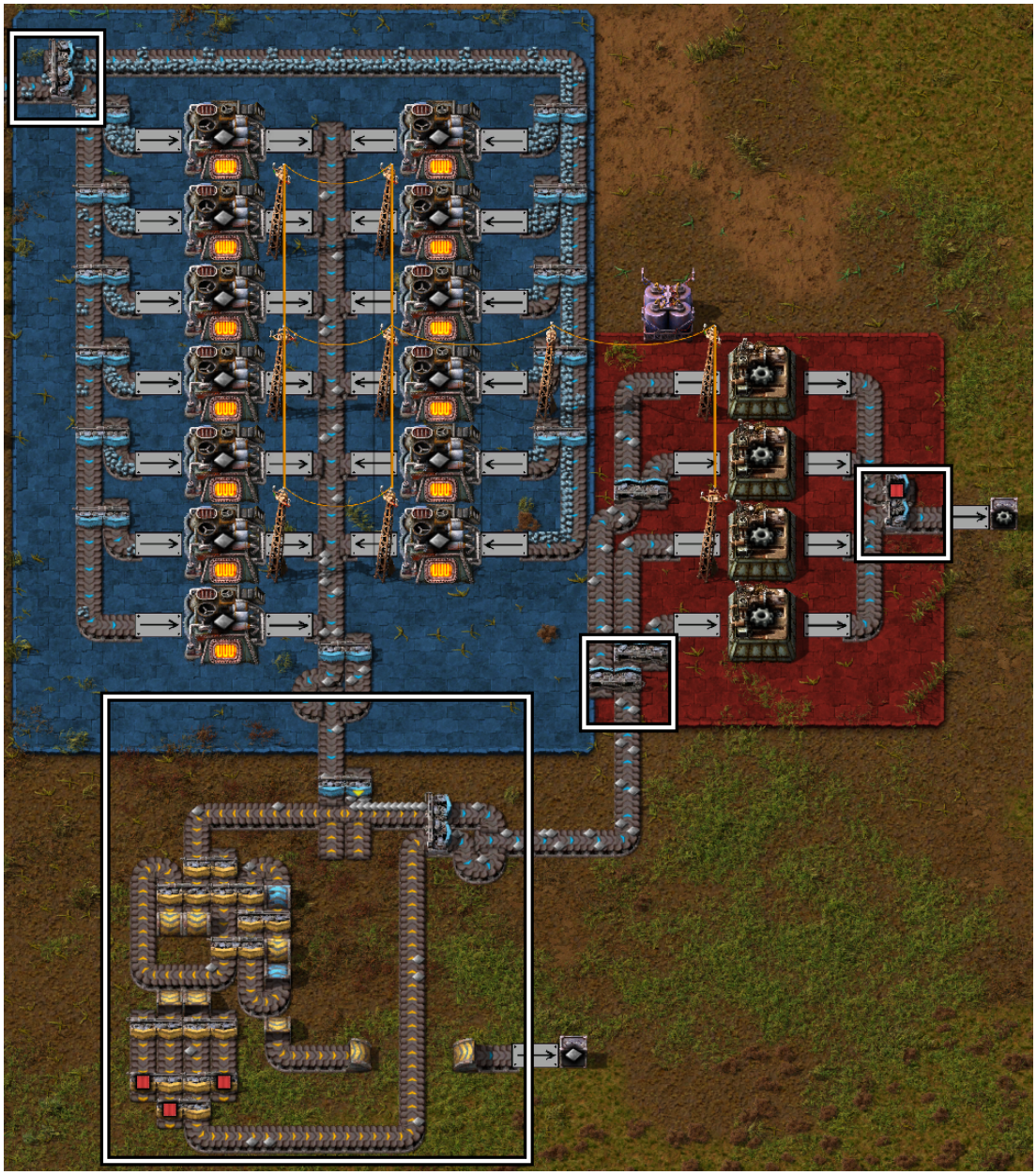


Figure 7.11: In-game implementation of the *fourGears* composition with the electric furnace block highlighted in blue and assembling machine block highlighted in red. The four named locations are highlighted with black-white squares. Bottom segment achieves the $1 : 64$ split of iron plates.

Another aspect we do not currently model in its entirety is fuel consumption. We constrain ourselves only to machines that consume electricity, while the game also includes ones that consume solid or fluid fuels. This could be included as a further input to performing a recipe, using the time taken, energy drain and energy value of a given fuel to compute how much is needed.

Beyond aspects of the game that we do not model, we could also improve the integration of our formalisation with the game. At present, we manually encode the item types, machines and recipes that we use. However, Factorio allows us to export the whole range of item types, machines and recipes from a running copy of the game as a JSON file. We could load this data into Isabelle to make it all available to our formalisation with values guaranteed to be accurate.

7.5 Conclusion

In this chapter we gave a number of case studies to demonstrate the main features of our framework. We touch on a variety of domains: education, daily life and manufacturing. The case studies we presented suggest some of the possible uses of our framework and the tools that could be based on it.

In the next chapter we conclude the thesis, tying together the thread running through it and giving a brief overview of potential future work.

Chapter 8

Conclusion

In the preceding chapters we developed a framework for verified composition of processes based on linear resources. We did so in the proof assistant Isabelle/HOL to ensure a high degree of confidence in the framework's properties and to allow for automated generation of executable code.

We connected our process compositions to two other concepts: linear logic deductions and port graphs. The first focuses on the correspondence of deduction in linear logic to the structure of correct processes, while the second focuses on viewing large processes as collections of actions connected by their inputs and outputs. We use both to demonstrate that valid process compositions manipulate resources correctly.

We also explored an extension of our framework to include probabilistic information in its description of non-determinism. While the present outcome of this exploration is open to definite improvements, it did demonstrate how invaluable a proof assistant is in exploring changes to the foundations of a formal framework. With its assistance, making such changes is significantly easier and, as a result, exploratory studies of formalised systems become more viable.

We demonstrated the application of our framework with a number of case studies in different domains. While these do not cover the full breadth of possibility, they illustrate the framework's main features and suggest more complex uses.

8.1 Future Work

In this section we highlight some of the main threads of future work arising from this thesis.

Process behaviour. A formalisation of behaviour for our process compositions would allow us to prove which of the latter, although syntactically distinct, represent the same behaviour. This in turn would allow us to verify, for instance, behaviour-preserving simplifications of process compositions such as removing identity actions from sequential composition.

We see process port graphs from Section 5.4 and the transition system for port graphs from Section 5.7 as providing a promising path towards formalising a notion of behaviour for our process compositions. A future task for this thread lies in expanding our formalisation of port graphs so that they can capture non-determinism and higher-order processes.

Connection to other formalisms. A complementary approach to process composition behaviour would be to formally connect them to other process formalisms, such as π-calculus or Petri nets. We could then use their established theory and tools in our framework. That includes making use of the their transition systems to characterise the behaviour of our process compositions.

In order to mechanise the connection, we would require mechanisations of those process formalisms in Isabelle/HOL. There exists such a mechanisation for the π-calculus due to Bengtson [10]. As far as we are aware, there is no published mechanisation of Petri nets in Isabelle/HOL.

Both of these formalisms make use of connections between individual actions: as channels between agents in π-calculus and as places between transitions in Petri nets. As such, we believe a significant part of formally connecting process compositions to them would be handled by process port graphs. Then, beyond finding a suitable mechanisation of the target formalism, the major task would be connecting to it from port graphs.

Probabilistic information. As noted, the outcome of our exploration of probabilistic resources in Chapter 6 is relatively preliminary. While the extended framework can express the probabilistic information and make use of it in computing expected values from process compositions, it produces impractically complex resource expressions. As such, one thread of future work would be about revising how probabilistic information can be better integrated into resources and process compositions in light of lessons learned through our exploration.

Concrete tools. Mechanising our framework in Isabelle/HOL allows us to automatically generate executable code from it. We make use of it to generate process diagrams (see Section 3.3) and in our model of manufacturing in Factorio to generate instructions for implementing a process as a factory (see Section 7.4.7).

There are more tools we could implement by utilising the verified code generated by Isabelle/HOL, boosting the practicality of our framework. For instance, recall our definition of process port graphs (see Section 5.4) and our conversion of port graphs into a format that can be visualised by the Eclipse Layout Kernel and Sprotty (see Section 5.3.10). We could make use of the verified code for these to implement a visualisation tool tailored to process compositions. Unlike our process diagrams, such a tool could be made interactive, allowing us to better inspect the process in question or even construct processes in a graphical environment.

We could additionally make use of the port graph transition system (see Section 5.7) to identify enabled actions. Then we could distinguish the relevant nodes in the visualisation, forming a kind of graphical checklist for the process. It could be made interactive, allowing users to mark actions as completed to update the highlighted set of enabled actions. We believe this would aid coordination in settings where multiple people cooperate to execute a large process.

Our ability to generate verified code in Scala, Haskell, OCaml and SML offers the opportunity to integrate with a wide variety of external frameworks while retaining the rigour of our formally verified approach.

8.2 Concluding Remarks

Our mechanised framework gives us a solid foundation for tools whose implementation of processes can be checked, so that the assistance they provide can be relied upon. The language of process compositions is simple and avoids ambiguity, so that our expectations are clear both when modelling processes and when using those models. Moreover, the verified code generated by Isabelle ensures that what the tools use is what we formalised.

Looking toward the future, we will use this foundation to build such tools and expand the range of domains where we have applied our framework. Reaching more domains and modelling increasingly complex processes will allow us to truly push the framework to its limits, and then, with the assistance of Isabelle, expand those limits in a rigorous way.

With the formalisations presented in this thesis, we can implement, for instance, a graphical checklist to help large teams coordinate their work on complex processes in potentially safety-critical situations. The formal basis, given an accurate specification, will ensure that no team member is left without what they need or not knowing what they have to do, and that the graphical representation is faithful to the underlying process model. We believe this would help support people in a variety of domains, from manufacturing through administration all the way to healthcare.

It is our aim to leverage the best abilities of machines to help people perform complex tasks. We have helped machines understand how processes are structured, and now we can make them better at assisting us.

Appendix A

Appendix

A.1 ILL Deduction Embedding Functions

The conclusions of deeply embedded ILL deductions (see Section 4.5) are represented by values from the *ill-sequent* datatype, capturing the antecedents and consequent:

Isabelle Definition A.1.1 (Datatype for arbitrary ILL sequents)

datatype *'a ill-sequent* = *Sequent* (*'a ill-prop list*) (*'a ill-prop*)

We tie those values to the shallowly embedded relation of valid sequents (see Section 4.2) through the *ill-sequent-valid* function:

Isabelle Definition A.1.2 (Validity predicate on ILL sequents)

primrec *ill-sequent-valid* :: *'a ill-sequent* ⇒ *bool*
where *ill-sequent-valid* (*Sequent a c*) = *a* ⊢ *c*

We set up a declaration bundle *deep-sequent* which we can use to switch the turnstile notation from the relation to these values:

Isabelle Definition A.1.3 (Declaration bundle switching turnstile notation)

bundle *deep-sequent* **begin**

no-notation *sequent* (**infix** ⊢ *60*)
notation *Sequent* (**infix** ⊢ *60*)

end

The full definition for conclusions of deeply embedded ILL deductions is then as follows:

Isabelle Definition A.1.4 (Conclusions of deeply embedded ILL deductions)

```
primrec ill-conclusion :: ('a, 'l) ill-deduct ⇒ 'a ill-sequent where
  ill-conclusion (Premise G c l) = G ⊢ c
| ill-conclusion (Identity a) = [a] ⊢ a
| ill-conclusion (Exchange G a b D c P) = G @ [b] @ [a] @ D ⊢ c
| ill-conclusion (Cut G b D E c P Q) = D @ G @ E ⊢ c
| ill-conclusion (TimesL G a b D c P) = G @ [a ⊗ b] @ D ⊢ c
| ill-conclusion (TimesR G a D b P Q) = G @ D ⊢ a ⊗ b
| ill-conclusion (OneL G D c P) = G @ [1] @ D ⊢ c
| ill-conclusion (OneR) = [] ⊢ 1
| ill-conclusion (LimpL G a D b E c P Q) = G @ D @ [a ▷ b] @ E ⊢ c
| ill-conclusion (LimpR G a D b P) = G @ D ⊢ a ▷ b
| ill-conclusion (WithL1 G a b D c P) = G @ [a & b] @ D ⊢ c
| ill-conclusion (WithL2 G a b D c P) = G @ [a & b] @ D ⊢ c
| ill-conclusion (WithR G a b P Q) = G ⊢ a & b
| ill-conclusion (TopR G) = G ⊢ ⊤
| ill-conclusion (PlusL G a b D c P Q) = G @ [a ⊕ b] @ D ⊢ c
| ill-conclusion (PlusR1 G a b P) = G ⊢ a ⊕ b
| ill-conclusion (PlusR2 G a b P) = G ⊢ a ⊕ b
| ill-conclusion (ZeroL G D c) = G @ [0] @ D ⊢ c
| ill-conclusion (Weaken G D b a P) = G @ [!a] @ D ⊢ b
| ill-conclusion (Contract G a D b P) = G @ [!a] @ D ⊢ b
| ill-conclusion (Derelict G a D b P) = G @ [!a] @ D ⊢ b
| ill-conclusion (Promote G a P) = map Exp G ⊢ !a
```

Their well-formedness is defined using these conclusions as follows:

Isabelle Definition A.1.5 (Well-formedness of deeply embedded ILL deductions)

```
primrec ill-deduct-wf :: ('a, 'l) ill-deduct ⇒ bool where
  ill-deduct-wf (Premise G c l) = True
| ill-deduct-wf (Identity a) = True
| ill-deduct-wf (Exchange G a b D c P) =
    (ill-deduct-wf P ∧ ill-conclusion P = G @ [a] @ [b] @ D ⊢ c)
| ill-deduct-wf (Cut G b D E c P Q) =
    ( ill-deduct-wf P ∧ ill-conclusion P = G ⊢ b ∧
      ill-deduct-wf Q ∧ ill-conclusion Q = D @ [b] @ E ⊢ c)
| ill-deduct-wf (TimesL G a b D c P) =
    (ill-deduct-wf P ∧ ill-conclusion P = G @ [a] @ [b] @ D ⊢ c)
| ill-deduct-wf (TimesR G a D b P Q) =
    ( ill-deduct-wf P ∧ ill-conclusion P = G ⊢ a ∧
      ill-deduct-wf Q ∧ ill-conclusion Q = D ⊢ b)
| ill-deduct-wf (OneL G D c P) =
    (ill-deduct-wf P ∧ ill-conclusion P = G @ D ⊢ c)
| ill-deduct-wf (OneR) = True
| ill-deduct-wf (LimpL G a D b E c P Q) =
```

```
    ( ill-deduct-wf P ∧ ill-conclusion P = G ⊢ a ∧
      ill-deduct-wf Q ∧ ill-conclusion Q = D @ [b] @ E ⊢ c)
| ill-deduct-wf (LimpR G a D b P) =
    (ill-deduct-wf P ∧ ill-conclusion P = G @ [a] @ D ⊢ b)
| ill-deduct-wf (WithL1 G a b D c P) =
    (ill-deduct-wf P ∧ ill-conclusion P = G @ [a] @ D ⊢ c)
| ill-deduct-wf (WithL2 G a b D c P) =
    (ill-deduct-wf P ∧ ill-conclusion P = G @ [b] @ D ⊢ c)
| ill-deduct-wf (WithR G a b P Q) =
    ( ill-deduct-wf P ∧ ill-conclusion P = G ⊢ a ∧
      ill-deduct-wf Q ∧ ill-conclusion Q = G ⊢ b)
| ill-deduct-wf (TopR G) = True
| ill-deduct-wf (PlusL G a b D c P Q) =
    ( ill-deduct-wf P ∧ ill-conclusion P = G @ [a] @ D ⊢ c ∧
      ill-deduct-wf Q ∧ ill-conclusion Q = G @ [b] @ D ⊢ c)
| ill-deduct-wf (PlusR1 G a b P) =
    (ill-deduct-wf P ∧ ill-conclusion P = G ⊢ a)
| ill-deduct-wf (PlusR2 G a b P) =
    (ill-deduct-wf P ∧ ill-conclusion P = G ⊢ b)
| ill-deduct-wf (ZeroL G D c) = True
| ill-deduct-wf (Weaken G D b a P) =
    (ill-deduct-wf P ∧ ill-conclusion P = G @ D ⊢ b)
| ill-deduct-wf (Contract G a D b P) =
    (ill-deduct-wf P ∧ ill-conclusion P = G @ [!a] @ [!a] @ D ⊢ b)
| ill-deduct-wf (Derelict G a D b P) =
    (ill-deduct-wf P ∧ ill-conclusion P = G @ [a] @ D ⊢ b)
| ill-deduct-wf (Promote G a P) =
    (ill-deduct-wf P ∧ ill-conclusion P = map Exp G ⊢ a)
```

A.2 Stitching Port Graph Interfaces

In Section 5.3.9 we make use of *seqInterfaceEdges* to define port graph sequencing. Our description of that function maps onto the Isabelle definitions as follows:

Isabelle Definition A.2.1 (Gathering edges going into output ports)

```
primrec edgesByOpenTo :: ('s, 'a, 'p) edge list
                         ⇒ (('s, 'a) port, ('s, 'a, 'p) edge list) mapping
 where
  edgesByOpenTo [] = Mapping.empty
| edgesByOpenTo (e#es) =
  ( if place-open (edge-to e)
    then Mapping.map-default
           (place-port (edge-to e)) Nil (Cons e) (edgesByOpenTo es)
    else edgesByOpenTo es)
```

Isabelle Definition A.2.2 (Gathering edges going from input ports)

```
primrec edgesByOpenFrom :: ('s, 'a, 'p) edge list
                         ⇒ (('s, 'a) port, ('s, 'a, 'p) edge list) mapping
 where
  edgesByOpenFrom [] = Mapping.empty
| edgesByOpenFrom (e#es) =
  ( if place-open (edge-from e)
    then Mapping.map-default
           (place-port (edge-from e)) Nil (Cons e) (edgesByOpenFrom es)
    else edgesByOpenFrom es
```

Isabelle Definition A.2.3 (Connect all combinations from two lists of places)

```
primrec allEdges :: ('s, 'a, 'p) place list ⇒ ('s, 'a, 'p) place list ⇒ ('s, 'a, 'p) edge list
 where
  allEdges [] ts = []
| allEdges (f#fs) ts = map (λt. Edge f t) ts @ allEdges fs ts
```

Isabelle Definition A.2.4 (Pair up place lists by shared port and connect them)

```
primrec edgesFromPortMapping :: ('s :: side-in-out, 'a) port list
                              ⇒ (('s, 'a) port, ('s, 'a, 'p) place list) mapping
                              ⇒ (('s, 'a) port, ('s, 'a, 'p) place list) mapping
                              ⇒ ('s, 'a, 'p) edge list
 where
  edgesFromPortMapping [] x y = []
| edgesFromPortMapping (p#ps) x y =
  ( case Mapping.lookup x (portSetSide Out p) of
      None ⇒ []
    | Some xs ⇒ (case Mapping.lookup y (portSetSide In p) of
        None ⇒ []
      | Some ys ⇒ allEdges xs ys))
  @ edgesFromPortMapping ps x y
```

Isabelle Definition A.2.5 (Connect edges by shared interface ports)

```
fun seqInterfaceEdges :: ('s :: side-in-out, 'a, 'p, 'l) port-graph
                       ⇒ ('s, 'a, 'p, 'l) port-graph
                       ⇒ ('s, 'a, 'p) edge list
 where seqInterfaceEdges x y =
 remdups
  ( edgesFromPortMapping (filter (λp. port.side p = Out) (pg-ports x))
    ( Mapping.map-values (λk. map edge-from) (edgesByOpenTo (pg-edges x)))
    ( Mapping.map-values (λk. map edge-to) (edgesByOpenFrom (pg-edges y))))
```

A.3 Conversion of Port Graphs to ELK JSON

In Section 5.3.10 we discuss our ability to visualise port graphs with the Eclipse Layout Kernel by converting them into JSON[1]. The full definitions of the conversions involved are as follows:

Isabelle Definition A.3.1 (Datatype representing port sides that ELK supports)

```
datatype elk-side = UNDEFINED | NORTH | EAST | SOUTH | WEST
```

Isabelle Definition A.3.2 (Converting ELK sides to their representation for JSON)

```
primrec elkSideToString :: elk-side ⇒ String.literal where
  elkSideToString UNDEFINED = STR "UNDEFINED"
| elkSideToString NORTH = STR "NORTH"
| elkSideToString EAST = STR "EAST"
| elkSideToString SOUTH = STR "SOUTH"
| elkSideToString WEST = STR "WEST"
```

Isabelle Definition A.3.3 (Default assignment of input and output to ELK sides)

```
fun sideInOutToELK :: ('s :: side-in-out) ⇒ elk-side
 where sideInOutToELK x =
  (if x = In then WEST else if x = Out then EAST else UNDEFINED)
```

Isabelle Definition A.3.4 (Locale for converting port graphs into ELK JSON)

```
locale portGraphELK =
  fixes portSideToELK :: 's ⇒ elk-side
   and pathToString :: 'p list ⇒ String.literal
   and labelToString :: 'l ⇒ String.literal
```

Isabelle Definition A.3.5 (Converting ground ports into ELK JSON)

```
definition groundPortToJSON :: 'p list ⇒ nat ⇒ ('s, 'a) port
                            ⇒ (String.literal, int) json
 where groundPortToJSON prefix maxIdx p =
 OBJECT [
  (STR "id", STRING (pathToString prefix +
                     elkSideToString (portSideToELK (port.side p)) +
                     String.implode (show (port.index p))))
 , (STR "properties", OBJECT [
    (STR "port.side", STRING (elkSideToString (portSideToELK (port.side p))))
   , (STR "port.index", NUMBER (int (
      if portSideToELK (port.side p) ∈ {WEST, SOUTH}
```

[1] https://www.eclipse.org/elk/documentation/tooldevelopers/graphdatastructure/jsonformat.html

```
        then maxIdx − port.index p
        else port.index p)))
  ])
 , (STR ''width'', NUMBER 10)
 , (STR ''height'', NUMBER 10)]
```

(Note that we may need to invert the index, because ELK numbers ports clockwise while we number them from the top down and from left to right.)

Isabelle Definition A.3.6 (Converting nodes into ELK JSON)

```
definition nodeToJSON :: 'p list ⇒ ('s, 'a, 'p, 'l) node ⇒ (String.literal, int) json
  where nodeToJSON prefix n =
  OBJECT [
    (STR ''id'', STRING (pathToString (prefix @ node-name n)))
  , (STR ''width'', NUMBER 60)
  , (STR ''height'', NUMBER 60)
  , (STR ''properties'', OBJECT [
      (STR ''portConstraints'', STRING STR ''FIXED-ORDER'')
    , (STR ''nodeLabels.placement'',
        STRING STR ''[H_CENTER, V_CENTER, INSIDE]'')])
  , (STR ''labels'', ARRAY [OBJECT [
      (STR ''text'', STRING (labelToString (node-label n)))
    , (STR ''id'', STRING (pathToString (prefix @ node-name n) + STR ''_label''))
    , (STR ''width'', NUMBER 20)
    , (STR ''height'', NUMBER 20)]])
  , (STR ''ports'', ARRAY
      (map (λp. groundPortToJSON
                  (prefix @ node-name n)
                  (length (filter (λx. port.side x = port.side p) (node-ports n)) − 1)
                  p)
           (node-ports n)))]
```

Isabelle Definition A.3.7 (Converting an arbitrary place to its ID)

```
definition groundPlaceToID :: 'p list ⇒ ('s, 'a, 'p) place ⇒ String.literal
  where groundPlaceToID prefix p =
    pathToString (prefix @ place-name p) +
    elkSideToString (portSideToELK (port.side (place-port p))) +
    String.implode (show (port.index (place-port p)))

definition openPlaceToID :: ('s, 'a) port ⇒ String.literal
  where openPlaceToID port =
  STR ''Open'' +
  elkSideToString (portSideToELK (port.side port)) +
  String.implode (show (port.index port))

definition placeToID :: 'p list ⇒ ('s, 'a, 'p) place ⇒ String.literal
```

```
 where placeToID prefix p = ( case p of
   GroundPort - ⇒ groundPlaceToID prefix p
  | OpenPort port ⇒ openPlaceToID port)
```

Isabelle Definition A.3.8 (Converting edges into ELK JSON)

```
definition edgeToJSON :: 'p list ⇒ ('s, 'a, 'p) edge ⇒ (String.literal, int) json
 where edgeToJSON prefix e =
 OBJECT [
  (STR ''id'', STRING (placeToID prefix (edge-from e) +
                       STR ''−'' +
                       placeToID prefix (edge-to e)))
 , (STR ''sources'', ARRAY [STRING (placeToID prefix (edge-from e))])
 , (STR ''targets'', ARRAY [STRING (placeToID prefix (edge-to e))])]
```

Isabelle Definition A.3.9 (Converting open ports into ELK JSON)

```
definition openPortToJSON :: nat ⇒ ('s, 'a) port ⇒ (String.literal, int) json
 where openPortToJSON maxIdx p =
 OBJECT [
  (STR ''id'', STRING (openPlaceToID p))
 , (STR ''properties'', OBJECT [
    (STR ''port.side'', STRING (elkSideToString (portSideToELK (port.side p))))
   , (STR ''port.index'', NUMBER (int (
      if portSideToELK (port.side p) ∈ {WEST, SOUTH}
       then maxIdx − port.index p
       else port.index p)))
  ])
 , (STR ''width'', NUMBER 10)
 , (STR ''height'', NUMBER 10)]
```

Isabelle Definition A.3.10 (Converting port graphs into ELK JSON)

```
definition portGraphToJSON :: 'p list ⇒ ('s, 'a, 'p, 'l) port-graph
                              ⇒ (String.literal, int) json
 where portGraphToJSON prefix G =
 OBJECT [
  (STR ''id'', STRING (pathToString prefix + STR ''Root''))
 , (STR ''layoutOptions'', OBJECT [(STR ''algorithm'', STRING STR ''layered'')])
 , (STR ''children'', ARRAY (
     map (nodeToJSON prefix) (pg-nodes G) @
     map (λp. openPortToJSON
                (length (filter (λx. port.side x = port.side p) (pg-ports G)) − 1)
                p)
         (pg-ports G)))
 , (STR ''edges'', ARRAY (map (edgeToJSON prefix) (pg-edges G)))]
```

(Note that we represent open ports with small nodes, because ELK does not support ports not attached to a node.)

A.4 Processes with Port Graphs

In Chapter 5 we make frequent use of the predicate *pgDefined* to exclude from consideration compositions that make use of non-deterministic or higher-order features, for which we do not define the port graph construction. Here we give the predicate in full in Definition A.4.1. Note that we do not exclude the repeatable executable resource actions *Repeat*, *Close* and *Once*, since those concern the copyable aspect of those resources.

Isabelle Definition A.4.1 (Predicate for processes that have port graphs)

```
primrec pgDefined :: ('a, 'b, 'l, 'm) process ⇒ bool where
  pgDefined (Primitive ins outs l m) = True
| pgDefined (Identity a) = True
| pgDefined (Swap a b) = True
| pgDefined (Seq p q) = (pgDefined p ∧ pgDefined q)
| pgDefined (Par p q) = (pgDefined p ∧ pgDefined q)
| pgDefined (Opt p q) = False
| pgDefined (InjectL a b) = False
| pgDefined (InjectR a b) = False
| pgDefined (OptDistrIn a b c) = False
| pgDefined (OptDistrOut a b c) = False
| pgDefined (Duplicate a) = True
| pgDefined (Erase a) = True
| pgDefined (Represent p) = False
| pgDefined (Apply a b) = False
| pgDefined (Repeat a b) = True
| pgDefined (Close a b) = True
| pgDefined (Once a b) = True
| pgDefined (Forget a) = True
```

A.5 List-Based Process Compositions

To make definitions of larger process compositions more concise, we define functions for composing a list of processes in sequence or in parallel:

Isabelle Definition A.5.1 (Composing a list of processes in sequence)

```
primrec seq-process-list :: ('a, 'b, 'l, 'm) process list ⇒ ('a, 'b, 'l, 'm) process
 where
  seq-process-list [] = Identity Empty
| seq-process-list (x # xs) = (if xs = [] then x else Seq x (seq-process-list xs))
```

Isabelle Definition A.5.2 (Composing a list of processes in parallel)

```
primrec par-process-list :: ('a, 'b, 'l, 'm) process list ⇒ ('a, 'b, 'l, 'm) process
 where
  par-process-list [] = Identity Empty
| par-process-list (x # xs) = (if xs = [] then x else Par x (par-process-list xs))
```

A.6 Contingent Plan for Three Socks

In Section 7.2 we describe a conformant plan for the problem and note the existence of a contingent variant. The Haskell code for the composition representing this contingent plan is shown in Listing A.1.

Listing A.1: Contingent plan for picking three matching socks

```
contingentPlan :: Process LRes () String ()
contingentPlan = seq_process_list
  [ Par (Identity (hidden # hidden)) pick
  , OptDistrIn (hidden # hidden) black white
  , Opt
      (seq_process_list
        [ Swap (hidden # hidden) black
        , Par (Identity (black # hidden)) pick
        , OptDistrIn (black # hidden) black white
        , Opt
            (seq_process_list
              [ Par (Identity black) (Swap hidden black)
              , Par (Identity (black # black)) (Forget hidden)
              , InjectL (black # black # anything) (white # white # anything)
              ])
            (seq_process_list
              [ Par (Identity black) (Swap hidden white)
              , Par (Identity (black # white)) pick
              , OptDistrIn (black # white) black white
              , Opt
                  (seq_process_list
                    [ Par (Identity black) (Swap white black)
                    , Par (Identity (black # black)) (Forget white)
                    , InjectL (black # black # anything) (white # white # anything)
                    ])
                  (seq_process_list
                    [ Swap black (white # white)
                    , Par (Identity (white # white)) (Forget black)
                    , InjectR (black # black # anything) (white # white # anything)
```

```
                ])
            ])
      ])
    (seq_process_list
      [ Swap (hidden # hidden) white
      , Par (Identity (white # hidden)) pick
      , OptDistrIn (white # hidden) black white
      , Opt
          (seq_process_list
            [ Par (Identity white) (Swap hidden black)
            , Par (Identity (white # black)) pick
            , OptDistrIn (white # black) black white
            , Opt
                (seq_process_list
                  [ Swap white (black # black)
                  , Par (Identity (black # black)) (Forget white)
                  , InjectL (black # black # anything) (white # white # anything)
                  ])
                (seq_process_list
                  [ Par (Identity white) (Swap black white)
                  , Par (Identity (white # white)) (Forget black)
                  , InjectR (black # black # anything) (white # white # anything)
                  ])
            ])
          (seq_process_list
            [ Par (Identity white) (Swap hidden white)
            , Par (Identity (white # white)) (Forget hidden)
            , InjectR (black # black # anything) (white # white # anything)
            ])
      ])
]
```

Bibliography

[1] Samson Abramsky. Proofs as processes. *Theoretical Computer Science*, 135(1):5–9, 1994. doi:10.1016/0304-3975(94)00103-0.

[2] Oana Andrei. *A Rewriting Calculus for Graphs: Applications to Biology and Autonomous Systems*. Theses, Institut National Polytechnique de Lorraine - INPL, November 2008. URL: https://theses.hal.science/tel-00337558.

[3] Oana Andrei, Maribel Fernández, Hélène Kirchner, Guy Melançon, Olivier Namet, and Bruno Pinaud. PORGY: Strategy-driven interactive transformation of graphs. *Electronic Proceedings in Theoretical Computer Science*, 48:54–68, February 2011. doi:10.4204/eptcs.48.7.

[4] Robert Atkey. Syntax and semantics of quantitative type theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, page 56–65, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3209108.3209189.

[5] Jeremy Avigad, Johannes Hölzl, and Luke Serafin. A formally verified proof of the central limit theorem. *Journal of Automated Reasoning*, 59:389–423, 2017. doi:10.1007/s10817-017-9404-x.

[6] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998. doi:10.1017/CBO9781139172752.

[7] Franz Baader and Tobias Nipkow. *Universal Algebra*, chapter 3, page 34–57. Cambridge University Press, 1998. doi:10.1017/CBO9781139172752.004.

[8] Andrew Graham Barber. Dual intuitionistic linear logic. Technical Report ECS-LFCS-96-347, University of Edinburgh, 1996. URL: https://www.lfcs.inf.ed.ac.uk/reports/96/ECS-LFCS-96-347/.

[9] Gianluigi Bellin and Philip J. Scott. On the π-calculus and linear logic. *Theoretical Computer Science*, 135(1):11–65, 1994. doi:10.1016/0304-3975(94)00104-9.

[10] Jesper Bengtson. The pi-calculus in nominal logic. *Archive of Formal Proofs*, May 2012. https://isa-afp.org/entries/Pi_Calculus.html, Formal proof development.

[11] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear haskell: practical linearity in a higher-order polymorphic language. *Proc. ACM Program. Lang.*, 2(POPL), December 2018. doi:10.1145/3158093.

[12] Gavin M. Bierman. On intuitionistic linear logic. Technical Report UCAM-CL-TR-346, University of Cambridge, Computer Laboratory, August 1994. doi:10.48456/tr-346.

[13] Jasmin C. Blanchette, Lukas Bulwahn, and Tobias Nipkow. Automatic proof and disproof in Isabelle/HOL. In Cesare Tinelli and Viorica Sofronie-Stokkermans, editors, *Frontiers of Combining Systems*, pages 12–27, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. doi:10.1007/978-3-642-24364-6_2.

[14] Jasmin C. Blanchette, Johannes Hölzl, Andreas Lochbihler, Lorenz Panny, Andrei Popescu, and Dmitriy Traytel. Truly modular (co)datatypes for Isabelle/HOL. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving*, pages 93–110, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-08970-6_7.

[15] Bonnie S. Boardman and Caroline C. Krejci. Simulation of production and inventory control using the computer game Factorio. In *ASEE 2021 Gulf-Southwest Annual Conference*, 2021. doi:10.18260/1-2--36402.

[16] Filippo Bonchi, Pawel Sobocinski, and Fabio Zanasi. Full abstraction for signal flow graphs. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, page 515–526, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2676726.2676993.

[17] Spencer Breiner, Albert Jones, and Eswaran Subrahmanian. Categorical models for process planning. *Computers in Industry*, 112:103124, 2019. doi:10.1016/j.compind.2019.103124.

[18] Joachim Breitner. Visual theorem proving with the incredible proof machine. In Jasmin C. Blanchette and Stephan Merz, editors, *Interactive Theorem Proving*, pages 123–139, Cham, 2016. Springer International Publishing. doi:10.1007/978-3-319-43144-4_8.

[19] Joachim Breitner and Denis Lohner. The meta theory of the incredible proof machine. *Archive of Formal Proofs*, May 2016. https://isa-afp.org/entries/Incredible_Proof_Machine.html, Formal proof development.

[20] Achim D. Brucker. Nano json: Working with json formatted data in isabelle/hol and isabelle/ml. *Archive of Formal Proofs*, July 2022. https://isa-afp.org/entries/Nano_JSON.html, Formal proof development.

[21] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In Paul Gastin and François Laroussinie, editors, *CONCUR 2010 - Concurrency Theory*, pages 222–236, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. doi:10.1007/978-3-642-15375-4_16.

[22] Bob Coecke and Ross Duncan. Interacting quantum observables. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming*, pages 298–310, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. doi:10.1007/978-3-540-70583-3_25.

[23] Thierry Coquand and Gérard Huet. The calculus of constructions. Technical Report RR-0530, INRIA, May 1986. URL: https://inria.hal.science/inria-00076024.

[24] Jeremy E. Dawson and Rajeev Goré. Generic methods for formalising sequent calculi applied to provability logic. In Christian G. Fermüller and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 263–277, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. doi:10.1007/978-3-642-16242-8_19.

[25] Lucas Dixon, Alan Smaill, and Alan Bundy. Verified planning by deductive synthesis in intuitionistic linear logic. In *Workshop on Verification and Validation of Planning and Scheduling Systems: ICALP 2009*, 2009. URL: http://hdl.handle.net/1842/4773.

[26] Sören Domrös, Reinhard von Hanxleden, Miro Spönemann, Ulf Rüegg, and Christoph D. Schulze. The Eclipse Layout Kernel, 2023. doi:10.48550/arXiv.2311.00533.

[27] J. Michael Dunn. *Relevance Logic and Entailment*, pages 117–224. Springer Netherlands, Dordrecht, 1986. doi:10.1007/978-94-009-5203-4_3.

[28] Manuel Eberl, Johannes Hölzl, and Tobias Nipkow. A verified compiler for probability density functions. In Jan Vitek, editor, *Programming Languages and Systems*, pages 80–104, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. doi:10.1007/978-3-662-46669-8_4.

[29] Nneka Chinelo Ene, Maribel Fernández, and Bruno Pinaud. Attributed hierarchical port graphs and applications. *Electronic Proceedings in Theoretical Computer Science*, 265:2–19, February 2018. doi:10.4204/eptcs.265.2.

[30] Maribel Fernández, Hélène Kirchner, and Bruno Pinaud. Strategic Port Graph Rewriting: an Interactive Modelling Framework. *Mathematical Structures in Computer Science*, 29(5):615–662, May 2019. doi:10.1017/S0960129518000270.

[31] Brendan Fong and David I. Spivak. *An Invitation to Applied Category Theory: Seven Sketches in Compositionality*. Cambridge University Press, 2019.

[32] Basil Fürer, Andreas Lochbihler, Joshua Schneider, and Dmitriy Traytel. Quotients of bounded natural functors. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning*, pages 58–78, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-51054-1_4.

[33] Jean-Yves Girard. Linear logic. *Theoretical computer science*, 50(1):1–101, 1987. doi:10.1016/0304-3975(87)90045-4.

[34] Michael J. C. Gordon and Thomas F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, USA, 1993.

[35] Florian Haftmann. *Code generation from Isabelle theories*. https://isabelle.in.tum.de/doc/codegen.pdf.

[36] John Harrison. HOL Light: An overview. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 60–66, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. doi:10.1007/978-3-642-03359-9_4.

[37] Jane Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.

[38] Tony Hoare and Stephan van Staden. The laws of programming unify process calculi. *Science of Computer Programming*, 85:102–114, 2014. Special Issue on Mathematics of Program Construction 2012. doi:10.1016/j.scico.2013.08.012.

[39] Joshua S. Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994. doi:10.1006/inco.1994.1036.

[40] Johannes Hölzl. *Construction and Stochastic Applications of Measure Spaces in Higher-Order Logic*. PhD thesis, Technische Universität München, 2013. URL: https://mediatum.ub.tum.de/1116512.

[41] Johannes Hölzl. Markov chains and markov decision processes in Isabelle/HOL. *Journal of Automated Reasoning*, 59:345–387, 2017. doi:10.1007/s10817-016-9401-5.

[42] Johannes Hölzl and Armin Heller. Three chapters of measure theory in Isabelle/HOL. In Marko van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *Interactive Theorem Proving*, pages 135–151, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. doi:10.1007/978-3-642-22863-6_12.

[43] Johannes Hölzl, Andreas Lochbihler, and Dmitriy Traytel. A formalized hierarchy of probabilistic system types. In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving*, pages 203–220, Cham, 2015. Springer International Publishing. doi:10.1007/978-3-319-22102-1_13.

[44] Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR'93*, pages 509–523, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg. doi:10.1007/3-540-57208-2_35.

[45] Ross Horne. The Sub-Additives: A Proof Theory for Probabilistic Choice extending Linear Logic. In Herman Geuvers, editor, *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*, volume 131 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 23:1–23:16, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.FSCD.2019.23.

[46] Brian Huffman and Ondřej Kunčar. Lifting and Transfer: A modular design for quotients in Isabelle/HOL. In Georges Gonthier and Michael Norrish, editors, *Certified Programs and Proofs*, pages 131–146, Cham, 2013. Springer International Publishing. doi:10.1007/978-3-319-03545-1_9.

[47] Keigo Imai, Shoji Yuen, and Kiyoshi Agusa. Session type inference in Haskell. *Electronic Proceedings in Theoretical Computer Science*, 69:74–91, October 2011. doi:10.4204/eptcs.69.6.

[48] Thomas B. L. Jespersen, Philip Munksgaard, and Ken F. Larsen. Session types for Rust. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*, WGP 2015, page 13–22, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2808098.2808100.

[49] Sara Kalvala and Valeria de Paiva. Mechanizing linear logic in Isabelle. In *In 10th International Congress of Logic, Philosophy and Methodology of Science*, volume 24. Citeseer, 1995.

[50] Aleks Kissinger and Vladimir Zamdzhiev. Quantomatic: A proof assistant for diagrammatic reasoning. In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25*, pages 326–336, Cham, 2015. Springer International Publishing. doi:10.1007/978-3-319-21401-6_22.

[51] Peep Küngas and Mihhail Matskin. Linear logic, partial deduction and cooperative problem solving. In João Leite, Andrea Omicini, Leon Sterling, and Paolo Torroni, editors, *Declarative Agent Languages and Technologies*, pages 263–279, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. doi:10.1007/978-3-540-25932-9_14.

[52] Xavier Leroy et al. *The Objective Caml system – Documentation and user's manual*. http://caml.inria.fr/pub/docs/manual-ocaml/.

[53] Sam Lindley and J. Garrett Morris. Embedding session types in Haskell. In *Proceedings of the 9th International Symposium on Haskell*, Haskell 2016, page 133–145, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2976002.2976018.

[54] Sam Lindley and J. Garrett Morris. Lightweight functional session types. In Simon Gay and António Ravara, editors, *Behavioural Types*, River Publishers Series in Automation Control and Robotics, pages 265–286. Routledge, Denmark, 1 edition, 2017. doi:10.13052/rp-9788793519817.

[55] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer New York, NY, 1998. doi:10.1007/978-1-4757-4721-8.

[56] Areti Manataki, Jacques Fleuriot, and Petros Papapanagiotou. A workflow-driven formal methods approach to the generation of structured checklists for intra-hospital patient transfers. *IEEE Journal of Biomedical and Health Informatics*, 21(4):1156–1162, 2017. doi:10.1109/JBHI.2016.2579881.

[57] John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, Edinburgh, 1969. reprinted in McC90.

[58] Hernán Melgratti and Luca Padovani. An OCaml implementation of binary sessions. In Simon Gay and António Ravara, editors, *Behavioural Types*, River Publishers Series in Automation Control and Robotics, pages 243–264. Routledge, Denmark, 1 edition, 2017. doi:10.13052/rp-9788793519817.

[59] Dale Miller. A multiple-conclusion meta-logic. In *Proceedings Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 272–281, 1994. doi:10.1109/LICS.1994.316062.

[60] Robin Milner. Models of LCF. In *Mathematical Centre Tracts*, volume 82, pages 49–63. Mathematisch Centrum, 1976.

[61] Robin Milner. The polyadic π-calculus: a tutorial. In Friedrich L. Bauer, Wilfried Brauer, and Helmut Schwichtenberg, editors, *Logic and Algebra of Specification*,

pages 203–246, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg. doi:10.1007/978-3-642-58041-3_6.

[62] Robin Milner, Mads Toft, and Robert Harper. *The definition of Standard ML*. MIT Press, Cambridge, Mass., 1990.

[63] Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, pages 625–635, Cham, 2021. Springer International Publishing. doi:10.1007/978-3-030-79876-5_37.

[64] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989. doi:10.1109/5.24143.

[65] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL*. Springer Berlin, Heidelberg, 2002. doi:10.1007/3-540-45949-9.

[66] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the Scala programming language. 2004. URL: http://infoscience.epfl.ch/record/52656.

[67] Dominic Orchard and Nobuko Yoshida. Effects as sessions, sessions as effects. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 568–581, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2837614.2837634.

[68] Petros Papapanagiotou and Jacques Fleuriot. Modelling and implementation of correct by construction healthcare workflows. In Fabiana Fournier and Jan Mendling, editors, *Business Process Management Workshops*, pages 28–39, Cham, 2015. Springer International Publishing. doi:10.1007/978-3-319-15895-2_3.

[69] Petros Papapanagiotou and Jacques Fleuriot. WorkflowFM: A logic-based framework for formal process specification and composition. In Leonardo de Moura, editor, *Automated Deduction – CADE 26*, pages 357–370, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-63046-5_22.

[70] Petros Papapanagiotou, James Vaughan, Filip Smola, and Jacques D. Fleuriot. A real-world case study of process and data driven predictive analytics for manufacturing workflows. In *Proceedings of the 54th Hawaii International Conference on System Sciences 2021*, pages 1001 – 1010, January 2021. 54th Hawaii International Conference on System Sciences, HICSS-54 ; Conference date: 05-01-2021 Through 08-01-2021. doi:10.24251/HICSS.2021.122.

[71] Lawrence C. Paulson. *Isabelle A Generic Theorem Prover*. Lecture Notes in Computer Science, 828. Springer Berlin Heidelberg, Berlin, Heidelberg, 1st ed. 1994. edition, 1994.

[72] Simon Peyton Jones. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. http://www.haskell.org/definition/.

[73] James F. Power and Caroline Webster. Working with linear logic in coq. In *12th International Conference on Theorem Proving in Higher Order Logics*, pages 1–16, 1999. This is the Work-in-progress version of the paper. URL: https://mural.maynoothuniversity.ie/6461/.

[74] Riccardo Pucella and Jesse A. Tov. Haskell session types with (almost) no class. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, Haskell '08, page 25–36, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1411286.1411290.

[75] Gabriel Radanne, Hannes Saffrich, and Peter Thiemann. Kindly bent to free us. *Proc. ACM Program. Lang.*, 4(ICFP), August 2020. doi:10.1145/3408985.

[76] Kenneth N. Reid, Iliya Miralavy, Stephen Kelly, Wolfgang Banzhaf, and Cedric Gondro. The factory must grow: Automation in Factorio. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '21, page 243–244, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3449726.3459463.

[77] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education UK, 2013.

[78] Alceste Scalas and Nobuko Yoshida. Lightweight session programming in Scala. In Benjamin S. Lerner and Shriram Krishnamurthi, editors, *30th European Conference on Object-Oriented Programming, ECOOP 2016*, volume 56 of *30th*

European Conference on Object-Oriented Programming (ECOOP 2016), pages 21:1–21:28. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, July 2016. doi:10.4230/LIPIcs.ECOOP.2016.21.

[79] Dana S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 121(1):411–440, 1993. doi:10.1016/0304-3975(93)90095-B.

[80] Roberto Segala. Probability and nondeterminism in operational models of concurrency. In Christel Baier and Holger Hermanns, editors, *CONCUR 2006 – Concurrency Theory*, pages 64–78, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. doi:10.1007/11817949_5.

[81] Filip Smola and Jacques D. Fleuriot. Deep embedding of intuitionistic linear logic. *Archive of Formal Proofs*, November 2024. https://isa-afp.org/entries/ILL.html, Formal proof development.

[82] Filip Smola and Jacques D. Fleuriot. Linear resources and process compositions. *Archive of Formal Proofs*, November 2024. https://isa-afp.org/entries/ProcessComposition.html, Formal proof development.

[83] Filip Smola and Jacques D. Fleuriot. Linear resources in Isabelle/HOL. *Journal of Automated Reasoning*, 68, 2024. doi:10.1007/s10817-024-09698-2.

[84] Eugene W. Stark. Monoidal categories. *Archive of Formal Proofs*, May 2017. https://isa-afp.org/entries/MonoidalCategory.html, Formal proof development.

[85] Christian Sternagel and René Thiemann. Abstract rewriting. *Archive of Formal Proofs*, June 2010. http://isa-afp.org/entries/Abstract-Rewriting.html, Formal proof development.

[86] Terru Stübinger and Lars Hupel. Extending Isabelle/HOL's code generator with support for the Go programming language. In Andre Platzer, Kristin Yvonne Rozier, Matteo Pradella, and Matteo Rossi, editors, *Formal Methods*, pages 3–19, Cham, 2025. Springer Nature Switzerland. doi:10.1007/978-3-031-71177-0_1.

[87] Terru Stübinger and Lars Hupel. Go code generation for isabelle. *Archive of Formal Proofs*, January 2024. https://isa-afp.org/entries/Go.html, Formal proof development.

[88] The Coq Development Team. The Coq proof assistant, September 2024. doi:10.5281/zenodo.14542673.

[89] René Thiemann and Christian Sternagel. Certification of termination proofs using CeTA. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 452–468, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. doi:10.1007/978-3-642-03359-9_31.

[90] Dmitry Traytel, Andrei Popescu, and Jasmin C. Blanchette. Foundational, compositional (co) datatypes for higher-order logic: Category theory applied to theorem proving. In *2012 27th Annual IEEE Symposium on Logic in Computer Science*, pages 596–605. IEEE, 2012. doi:10.1109/LICS.2012.75.

[91] Jason Vallet, Hélène Kirchner, Bruno Pinaud, and Guy Melançon. A visual analytics approach to compare propagation models in social networks. *Electronic Proceedings in Theoretical Computer Science*, 181:65–79, April 2015. doi:10.4204/eptcs.181.5.

[92] Philip Wadler. Propositions as sessions. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, page 273–286, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2364527.2364568.

[93] Philip Wadler. Propositions as types. *Commun. ACM*, 58(12):75–84, 2015. doi:10.1145/2699407.

[94] Markus Wenzel. Type classes and overloading in higher-order logic. In Elsa L. Gunter and Amy Felty, editors, *Theorem Proving in Higher Order Logics*, pages 307–322, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg. doi:10.1007/BFb0028402.

[95] Markus Wenzel. Isar — a generic interpretative approach to readable formal proof documents. In Yves Bertot, Gilles Dowek, Laurent Théry, André Hirschowitz, and Christine Paulin, editors, *Theorem Proving in Higher Order*

Logics, pages 167–183, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. doi:10.1007/3-540-48256-3_12.

[96] Bruno Xavier, Carlos Olarte, Giselle Reis, and Vivek Nigam. Mechanizing focused linear logic in coq. *Electronic Notes in Theoretical Computer Science*, 338:219–236, 2018. The 12th Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2017). doi:10.1016/j.entcs.2018.10.014.

[97] Ryan Yates and Brent A. Yorgey. Diagrams: a functional EDSL for vector graphics. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Functional Art, Music, Modelling and Design*, FARM 2015, page 4–5, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2808083.2808085.