Type Systems for Coordination Languages

Haoxuan Cai
Statement of Originality

I declare that, other than where clearly stated as referenced, the whole of this thesis is my own work, which has been improved by much feedback, especially:

- the formulae in Chapter 4 are benefited from many discussions with Sophia Drossopoulou, as well as the consistent style of the proofs in Appendix A.

- Appendix B, which was published as [11] with 5 co-authors, is also my work but after many iterations as a result of constant discussions with the co-authors.
Abstract

S-Net is a declarative coordination language rooted in stream processing with a runtime that automatically distributes the computational units among available resources. It is conceived in response to the change of processor development trend, from making the speed faster to embedding more cores. For performance reasons, the S-Net compiler is responsible for generating some additional information, through type inference, which is used by the runtime to make data delivery decisions. This requires the compiler to be supported by a sound type system which can ensure that the program behaviour meets the expectations of the language designers and the programmers.

However, due to S-Net's design principle of ease of use, the S-Net type system was believed to be simple and was only informally documented. As we empirically tested the type inference implementation, we gradually revealed the hidden complexity of the calculus behind the apparently easy-to-use language, which was clearly beyond the capability of the informal type system. We then attempted several formulations of the type system, each addressing more issues we have found, but a complete solution was still missing. S-Net now urgently needs a formal type system with proofs of soundness.

We have identified a major issue which has been making it difficult to design a correct type system, that is the type-semantics interdependency. In this thesis, we present a new design of the S-Net semantics and type system with no type-semantics interdependency, in terms of a new language BL-Net, a reduced S-Net which preserves only the type-related behaviour, which has an operational semantics reflecting that of S-Net, and a type system with the soundness and completeness proof. Our contributions also include a bridging solution to fit the new type system into the existing compiler structure.
Acknowledgements

My supervisor Susan Eisenbach has my deepest gratitude for the very existence of me as a PhD student. She helped me in my PhD study not only academically, but also financially and mentally. She has been believing in my academic competence even during the times I doubted myself, and this has been the main driving force for me to complete the study. Career-wise, she gave me my first professional teaching opportunity, which I enjoyed very much as teaching has been my secret pleasure. She also helped me secure my current fantastic job at Google, for which I cannot thank her enough.

My second supervisor Sophia Drossopoulou has been inspirational. Her lectures enlightened me to the sea of computational theories. In my times of mathematical and computational theoretic immaturity, I was very lucky to have her almost guiding me step by step. When I was most stuck, it was also her hint that I should ‘think out of the box’, that eventually led to a breakthrough, without which this thesis would be an impossibility.

The S-Net designers and fellows at the University of Hertfordshire, Alex Shafarenko, Clemens Grelick, Sven-Bodo Scholz, Frank Penczek and others who worked on the S-Net compiler helped me through the first two years of my PhD study involving the previous type system designs and compiler implementation.

My sincere gratitude also goes to my colleagues at Imperial, including but not limited to Tristan Allwood, Khilan Gudka, Alexis Petrounias, Reuben Rowe, Alex Summers and others I have met through the SLURP group, who did not mind my disturbing with my questions and provided valuable feedback for my study.

Last but not least, I owe my friends and family who have told me not to give up. The motivation was much needed during the final stage of my study.
To VW
## Contents

**1 Introduction**  
1.1 Type-semantics Interdependency ......................................................... 15  
1.2 Previous Type Systems ........................................................................... 16  
1.3 Contributions and Methodology .............................................................. 17  
1.4 Frank Penczek’s Thesis ........................................................................ 18  
1.5 Thesis Structure .................................................................................... 19

**2 Background**  
2.1 Background of S-Net ........................................................................... 21  
2.1.1 Coordination Languages ................................................................... 21  
2.1.2 Stream Processing Languages ........................................................... 22  
2.1.3 S-Net at a Glance ............................................................................ 22  
2.1.4 Comparison: StreamIt, StreamFlex and S-Net ................................. 23  
2.2 S-Net Language Introduction ............................................................... 25  
2.2.1 Boxes and Records ........................................................................... 25  
2.2.2 Program and Networks ..................................................................... 27  
2.2.3 Program Flow and Serial Composition ............................................. 29  
2.2.4 Filters and Tags ................................................................................. 29  
2.2.5 Parallel Composition .......................................................................... 30  
2.2.6 Serial Replication .............................................................................. 31  
2.2.7 Flow Inheritance, Overwriting, Discarding ....................................... 31  
2.2.8 Signed Networks .............................................................................. 32  
2.2.9 Binding Tags Revisited ..................................................................... 33  
2.2.10 Synchrocells ................................................................................. 33  
2.3 Background of S-Net Type System ...................................................... 34  
2.3.1 Type Systems and Type Inference ...................................................... 35  
2.3.2 Types ............................................................................................... 35  
2.3.3 Type Polymorphism ......................................................................... 36  
2.3.4 Soundness and Completeness ............................................................ 37  
2.3.5 Abstract Interpretations ................................................................. 38
# Reducing S-Net

## 3.1 Foundation

- 3.1.1 Label Set Transforming Languages
- 3.1.2 Projection, BL-Net and L-Net

## 3.2 Translation

- 3.2.1 Record Types and Patterns
- 3.2.2 Boxes
- 3.2.3 Serial and Parallel Composition
- 3.2.4 Serial Replication
- 3.2.5 Filters
- 3.2.6 Synchrocells
- 3.2.7 Parallel Replication
- 3.2.8 Unsigned Networks
- 3.2.9 Signed Networks
- 3.2.10 Program

## 3.3 Summary

## 4. L-Net

### 4.1 Example L-Net Programs

### 4.2 L-Net Syntax and Specification

- 4.2.1 Values
- 4.2.2 Program
- 4.2.3 Execution

### 4.3 Type System

- 4.3.1 Functional Representation of Networks
- 4.3.2 Reps
- 4.3.3 Network Types in L-Net and S-Net
- 4.3.4 Type Inference
- 4.3.5 Soundness and Completeness
- 4.3.6 Finiteness and Locality

### 4.4 Semantics for Implementation

- 4.4.1 Attraction
- 4.4.2 Program Execution
- 4.4.3 Compliance
- 4.4.4 Relation with S-Net

### 4.5 Summary
5 BL-Net

5.1 Example BL-Net Programs .................................................. 78
5.2 Binds and the Effects on Values ........................................ 79
  5.2.1 Values with Binds ...................................................... 79
  5.2.2 Partial Order of Values ............................................. 80
  5.2.3 Controlling Branch Selection ...................................... 80
5.3 Sinks and Sunk Executions ................................................. 81
  5.3.1 Definition ............................................................. 81
  5.3.2 Three Forms of Execution ......................................... 82
  5.3.3 Propagation .......................................................... 82
5.4 Serial Replications .......................................................... 83
  5.4.1 Definition ............................................................. 83
  5.4.2 Specification ........................................................ 83
  5.4.3 Modelling ............................................................. 84
5.5 Remaining Specification .................................................... 88
5.6 Type System ................................................................. 89
  5.6.1 Accommodating Sunk Executions .................................. 89
  5.6.2 Reps and Type Inference ........................................... 90
  5.6.3 Serial Replications ................................................. 93
  5.6.4 Soundness and Completeness ..................................... 97
5.7 Semantics for Implementation ............................................. 98
  5.7.1 Attraction ............................................................... 98
  5.7.2 Root Alphabet ........................................................ 98
  5.7.3 Program Execution .................................................. 99
  5.7.4 Compliance ........................................................... 100
5.8 Summary ..................................................................... 100

6 Type System for S-Net .......................................................... 101
6.1 Preliminaries ................................................................. 101
6.2 New Type System ............................................................ 102
  6.2.1 Type Checking S-Net Programs .................................... 102
  6.2.2 Supporting Parallel Compositions ................................ 103
6.3 Comparison with Previous Type System ............................... 104
  6.3.1 Structural Upgrades ................................................. 106
  6.3.2 Record Subtyping and Matching vs. Value Partial Order ...... 106
  6.3.3 Type-semantics Interdependency .................................. 107
  6.3.4 Simulation vs. Function Representation ......................... 108
6.4 Limitations ................................................................. 109
  6.4.1 Initialiser Boxes ...................................................... 109
List of Figures

1.1 S-Net projected and further reduced. .................................................. 17
2.1 Example S-Net Program. ........................................................................... 28
3.1 Example S-Net program (left) translated to BL-Net (right). ..................... 43
3.2 Translating an unsigned S-Net network (left) to BL-Net (right). .............. 51
4.1 Illustration of Example 4.1. ..................................................................... 54
4.2 Illustration of Example 4.2. ..................................................................... 55
4.3 Illustration of Examples 4.3 and 4.4. ......................................................... 55
4.4 Illustration of Example 4.5. ..................................................................... 56
5.1 Illustration of Examples 5.2 and 5.3. ......................................................... 78
5.2 A serial replication. ................................................................................. 84
5.3 Modelling a serial replication. ................................................................. 85
5.4 Serial replication type inference. .............................................................. 97
7.1 Simplified LL(1) grammar of S-Net. ....................................................... 117
B.1 S-NetAx syntax. ..................................................................................... 156
B.2 Signature for route inference structure. .................................................. 163
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Streaming languages comparison</td>
<td>25</td>
</tr>
<tr>
<td>3.1</td>
<td>Language features comparison</td>
<td>42</td>
</tr>
<tr>
<td>6.1</td>
<td>Outputs from both type systems</td>
<td>105</td>
</tr>
<tr>
<td>7.1</td>
<td>Type inference class library</td>
<td>119</td>
</tr>
<tr>
<td>7.2</td>
<td>Type inference functions and methods in TypeInference class</td>
<td>119</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Processor development has changed from making the speed faster to embedding more cores, demanding a complementary revolution in the programming language industry to ease the software development on multi-core platforms. Designed to address such a situation, S-Net [25] is a declarative coordination language rooted in stream processing, which is accompanied by a runtime that automatically distributes the computational units among available resources. S-Net was first published in 2006 and has been under active development by the University of Hertfordshire, UK. A design principle which has not changed since the beginning is the ease of use. Owing to this, S-Net has attracted interest from various industries, and practical applications exist. For example, with the University of Hertfordshire’s help, Thales Research developed a parallel signal processing application for space-time adaptive filtering of radar echoes in S-Net [37], and the University of California tried out a distributed ray tracing solver [38] using Distributed S-Net [26].

S-Net is not alone in the field of stream processing languages. The best known is perhaps StreamIt [48] developed at MIT. In 2007, Vitek et al. designed StreamFlex [46]. S-Net distinguishes itself from them in a variety of ways, providing a solution directly aiming at the aforementioned situation. Moreover, being a coordination language, S-Net actively facilitates code reuse, programmer collaboration, rapid prototyping and large scale design.

StreamIt and StreamFlex both originate from Java, and their stream processing layers do not add to or alter the meaning of types in the Java type system. In contrast, S-Net introduces its own understanding of types: S-Net groups a number of values of the computational language into a record, with each value paired with a label and its type simplified to either integer or pointer, and the S-Net components operate based on the existence or non-existence of certain labels in each input record. In particular, every box (a unit component in S-Net) consumes a defined set of labels per input record, using the values stored under those labels as the arguments to the underlying computational function, and then adds the output values back to the record under another defined set of labels. If the input record does not carry the whole set of required labels, the box cannot correctly invoke the computational function. As another example, every parallel composition (a branching construct in S-Net) routes different records to different branches based on each record’s label combination, and to respect the ease-of-use principle, the type inference
should work out which label combinations match which branch, instead of levying this task onto the programmer. This means that, while a separate type system is not necessary for either StreamIt or StreamFlex, S-Net requires a type system, orthogonal to that of the underlying computational language.

The author of this thesis started cooperating with the S-Net team in 2007 during his MSc individual project, with one of the objectives being the implementation of the S-Net compiler's type inference stage. At that time, it was believed that the S-Net type system should be simple due to S-Net's ease-of-use design principle, and received less attention than the rest of the language design. As the project progressed, we identified many flaws of the existing type system, and gradually discovered the undermined complexity. At the end of the MSc individual project, S-Net urgently required a new type system that could both ensure that the program behaviour meets the expectations of the language designers and the programmers, and support the language's ease of use on a theoretic basis by having certain proven desirable properties. The author's PhD study is an extension to his MSc individual project, with the objective of formulating such a type system.

The design and evolution of the language has gone side by side with the formulation of the type system. While the language design demanded constant update of the type system, the type system formulation process provided valuable feedback to the language designers. The insights to the language, gained during the process, helped discover imperfections in the design, and led to minor tweaks to the original language definition and even loosened requirements.

1.1 Type-semantics Interdependency

The S-Net semantics defines how each component processes input records and produces output records. The S-Net type system has the following two purposes:

1. guarantee that all unit components receive only those records they can process, e.g. a box only receives input records containing all labels needed to invoke the computational function;

2. support the parallel composition's semantics, by providing sufficient information for each parallel branch to allow the runtime to efficiently decide the route for each input record to each parallel composition.

The second purpose of the S-Net type system implies that there is a certain level of mutual dependency between the type system and the semantics of S-Net: while a type system naturally depends on the semantics as the foundation of its correctness, the semantics of S-Net relies on the type system to support a part of its computation. This creates the opportunity for the type system to alter the meaning of S-Net programs while still maintain its soundness, or in other words, the S-Net semantics is unstable. This type-semantics interdependency adds complexity to the formulation process, and was not discussed in detail in any S-Net technical report. In the latest version [25], it is unclear how either the operational semantics or the denotational semantics of the parallel composition retrieve the type information. The current runtime implementation reads the information from a hidden element in the abstract syntax tree, which is populated by the compiler during the type inference stage.
Assume there is an agreed algorithm $M$ to work out the record types that match a parallel branch purely from the semantics of the branch without referring to type inference, as long as the semantics of the branch is stable, i.e. does not change if we swap the type system. The S-Net semantics can be made stable as follows. First of all, if there is no parallel composition, then the second purpose of the type system is not necessary, and the semantics is stable for any program. If parallel composition is available but nesting is not allowed, then for each parallel composition, the parallel branches do not contain other parallel compositions and their semantics are stable, so $M$ can be used to work out the record types that match each branch, and in turn stabilise the semantics of the parallel composition in question. In this way, the semantics of any S-Net program, with all parallel branches' semantics stable, can be stable. By induction, the semantics of an arbitrary S-Net program can be stable.

However, although it should be relatively straightforward to define $M$, it remains undefined in the S-Net technical reports. Moreover, the semantics specified in this way is difficult to implement, unless the record types that the parallel compositions need are readily available at runtime. This is exactly the role of the type inference: it prepares the data needed by the parallel compositions at runtime. Put differently, the existing S-Net semantics are the ‘semantics for implementation’ and provide useful information for implementing the S-Net runtime, but they are not necessarily good foundations for proving the type system's properties.

### 1.2 Previous Type Systems

When the author's MSc individual project commenced, the draft S-Net technical report at that time [24] had a denotational semantics, along with some informal discussions of a type system. Before implementing this type system into the compiler, we tested it manually on some small practical examples, and found that it could not describe a hidden program behaviour, that some components could discard certain labels from the input record. If any such labels are needed later, there is a type error in the program, but the type system could not detect it. To fix this, the author introduced the *discarded label qualifier* in his master's thesis, published as [10]. The paper included the first concrete type system, and its compiler implementation marked the author's first PhD milestone.

Following the first release of the compiler, through internal discussions between the language designers and the author, we found that the discarded labels called for an update to the subtyping definition, affecting the rule of *monotonicity*, with which a subtype is allowed to be used in place of a supertype, while guaranteeing that the outputs are still of the same type or coercible to it. The update to the subtyping definition caused the monotonicity rule to restrict many compositions the programmers would consider valid. To retain S-Net's ease of use, the monotonicity requirement was eventually removed from the compiler implementation.

The type system and its compiler implementation then went through several minor changes, as we empirically discovered more special cases to challenge their correctness. During this process, we constantly encountered a particular kind of issue, that it was apparently impossible to infer a correct signature while maintaining the expected record routing decisions. We believed that a solution could be
to separate type inference from route inference.

Our first separation attempt was a type system with a bottom-up type inference algorithm and a left-to-right route inference algorithm. The route inference attempts to repair the program behaviour by modifying the inferred signatures and even re-requesting type inference using different assumptions. This breaks the modularity which S-Net has always had, and as a result the type system was abandoned.

The next separation attempt, which led to the second published and implemented type system [11], places route inference above type inference: the type inference algorithms are invoked by the route inference algorithms. However, the inference result is still incapable of describing the expected behaviours of some edge case programs, and in response, we had to introduce an additional verification step to ensure the compiler soundness.

Since then we have discussed a couple of more complete but complicated revisions of the type system, but to respect the ease-of-use principle, they were not used.

1.3 Contributions and Methodology

In this thesis, we break the type-semantics interdependency. We introduce the concept that all components use labels, and that the parallel composition favours the branch that potentially uses more labels in the incoming record. It is intuitive to figure out what labels a unit component uses; for example, a box uses the labels from which it finds the arguments to invoke the underlying computational function. Our approach requires that all components specify the used labels, so the parallel composition can use this information to make routing decisions. This is our answer to the algorithm $M$ mentioned in Section 1.1. We then define the operational semantics and the semantics for implementation separately.

However, instead of working on S-Net directly, we will work on a reduced version of the language. Many S-Net features have no impact on the correctness of the type system, such as the record creation and decomposition processes at the boxes, how the records form a stream when the runtime passes them from one component to another, or the number of records in a stream. We borrow the idea of the successful Featherweight Java [31], a reduced Java for compact proofs and studies of extensions of the full Java language, and reduce S-Net to create some smaller languages, allowing us to focus on the type system.

The process is illustrated in Figure 1.3 where S-Net is compared to a 3-dimensional object being
projected onto a plane to yield the projection, the 2-dimensional image of S-Net. The missing dimension contains features unrelated to the correctness of the type system, but the projection still truly represents S-Net's other features that are important to the type system. The projection is then reduced to BL-Net by removing the components whose semantics can be approximated using other components. BL-Net is further reduced to L-Net, but this is only for the purpose of covering the important concepts in manageable batches.

Having broken the type-semantics interdependency, we can now redesign the S-Net type system with the correctness proof. The following list highlights the major contributions of this thesis:

- the language BL-Net and a method to transform S-Net programs into BL-Net;
- the operational semantics of BL-Net as the specification, that does not require any results from the type inference, which features a used value part as part of the execution judgement;
- the type system for BL-Net and the soundness and completeness proof, using the specification as the foundation;
- the semantics for implementation of BL-Net that does use the type inference results, resembling the type-semantics interdependency of S-Net, and the proof that it complies with the specification.

It is worth noting at this point that BL-Net should not be seen as a standalone language; it should instead be treated as a tool to study the type aspect of S-Net. As it currently stands, BL-Net's specification has departed slightly from S-Net's current type-related behaviour, mainly due to the difference in the parallel composition's semantics. However, BL-Net is easier to understand because it lacks the type-semantics interdependency, and has a provably correct type system. In order to be truly easy to use, S-Net's semantics should be updated to match BL-Net's. In fact, this thesis also discusses the following:

- a prototype S-Net compiler using the BL-Net type system, showing that the latter is indeed suitable for S-Net;
- a bridging solution to fit the BL-Net type system into the existing S-Net runtime implementation.

1.4 Frank Penczek's Thesis

Towards the end of his PhD, the author of this thesis was made aware that Frank Penczek's concurrent PhD study about S-Net also included a type system. Penczek has been an active member of the S-Net team at the University of Hertfordshire. His major contributions include participating the implementation of the S-Net runtime, writing up the operational semantics chapter of the technical report [23], extending the runtime to a distributed environment and so on.

In his draft thesis as of the time of writing this section, he defined the semantics of explicitly typed S-Net, wherein the record types supporting the parallel compositions are assumed to be available. He then designed a type system which will first collect all distinct, non-branching routes of the program, and then infer the input and output types of each route, and finally merge the routes back to resemble
the original program, taking care of invalid combinations. Finally, he defined the semantics of implicitly typed S-Net using the results of the type inference. Similarities of the two theses include the use of helper languages, and redefinition of the semantics in terms of the helper languages.

Penczek's approach differs from the method presented in this thesis in the following main aspects. Firstly, the explicitly typed S-Net programs generally exist as intermediate programs created by the compiler, by augmenting the programmer's code with the record types needed by the parallel compositions, and those record types are in turn obtained from the type inference. The programmer's code is most likely in the syntax of implicitly typed S-Net, so the type-semantics interdependency persists. In comparison, BL-Net, although not to be considered a standalone language, has a semantics independent from its type system, providing a better foundation for proving various properties of the type system. On the other hand, the semantics of explicitly typed S-Net is complete with record and stream processing steps, unlike BL-Net which is abstracted away from records and streams.

Another predominant difference is that Penczek's type system decides a fixed route for each type of input records to the program at type inference time. Given an input record type and a parallel composition with all branches accepting said type, if the output record type of a branch could cause a type error later in the program, then the branch is effectively disabled statically for the input record type in question. This means that a small change anywhere in the program can have a potentially drastic effect on the choice of route for the same input record type. S-Net components under this design are not modular: a component's behaviour may be different in different contexts, depending on how it is further composed with other components, but this allows the program to process more types of input records. On the other hand, BL-Net preserves the component modularity. In the aforementioned example, no branches of the parallel composition will be statically disabled: the input record will be processed by a non-deterministically chosen branch, but because the input type may cause a type error later in the program, the BL-Net type system will exclude it from the program's domain to ensure type safety. It has been an ongoing debate whether the S-Net type system should be more permissive or more restrictive. The author of this thesis considers that modularity adds to S-Net's ease of use - there will be no surprise behaviour when the programmer changes a seemingly unrelated part of the program – and should be preserved despite the restriction it causes.

1.5 Thesis Structure

In Chapter 2, we look at some background research on stream processing languages and coordination languages briefly, and then provide an introduction to S-Net and discuss the background of its type system. Having introduced S-Net in more details, in Chapter 3 we revisit the methodology (Section 1.3) with a formal translation algorithm from S-Net to BL-Net. Chapter 4 covers L-Net, and Chapter 5 covers BL-Net, both with full specifications, type systems and correctness proofs, and the semantics for implementation. Chapter 6 describes the method to perform type inference and type checking for S-Net using the BL-Net type system, and compares it with the former, published type system. Then in Chapter 7, we discuss the prototype compiler implementation, and talk about one way to use the new
type system in the existing runtime architecture. Finally, we conclude the thesis with closing remarks and future work in Chapter 8.
Chapter 2

Background

In this chapter, we will skim through the history that created S-Net, then study the S-Net language to some extent, and finally look at why designing a type system of S-Net is a challenging task.

2.1 Background of S-Net

S-Net is both a coordination language and a stream processing language. In this section, we first review the history of coordination languages and stream processing languages, and then briefly compare S-Net with two other stream processing languages.

2.1.1 Coordination Languages

The history of coordination languages can date back to 1992 when Gelernter and Carriero suggested a strict separation of computation model and coordination model in programming languages [23]: the computation model allows building a single computational activity which is a single-threaded, step-at-a-time computation, while the coordination model binds separate activities into an ensemble.

The idea of coordination automatically lends itself to parallelism: two different computational activities, if not affecting or depending on each other, can execute simultaneously if resources permit. This has a certain level of resemblance with the multi-threaded programming model. The implication is that a sequential language within a coordination language is another way to implement concurrent programming, or directly, is a concurrent/distributed programming language itself.

In fact, Imperial has designed a coordination language called Darwin [35] which was mainly used for distributed programming. Darwin is in essence a declarative binding language which can be used to define hierarchic compositions of interconnected components. The research product, the Darwin Framework, compiles a Darwin program into a distributed Java program using RMI. Earlier versions were more heterogeneous and allowed the components to be written in a variety of languages.

As another example of coordination language, Reo [2] is a channel-based exogenous coordination model in which complex coordinators, called connectors, are compositionally built out of simpler ones, the simplest of which are channels with well-defined behaviour supplied by users. Reo can be used as a
language for coordination of concurrent processes.

2.1.2 Stream Processing Languages

The history of stream processing languages starts from Lucid, a data-flow programming language \[3\]. Instead of updating the values of variables through multiple assignments, which conventional procedural languages do, Lucid programs describe the relationships between the current and the next values of variables. Mathematically, each variable now holds a series of values, and using keywords like \texttt{first}, \texttt{next} and \texttt{as soon as}, programmers can travel in such a 'history' of values.

Inspired by Lucid, the LUSTRE declarative language \[14\] further evolved the idea, describing a data-flow program as a diagram, where operators, functions, and also subprograms are the nodes, interconnected with data channels. The series of values a variable holds is now represented by the series of values that pass through the same data channel. This formed the foundation of modern stream processing languages.

A stream processing program describes a collection of independent filters that communicate by the means of unidirectional data channels. The appeal of this is the conceptual simplicity to understand an application as a collection of collaborating coarse-grained components running in parallel.

Many languages support stream processing, including Hume \[27\], Infopipes \[9\], StreamIt \[48\] and StreamFlex \[46\]. Hume focuses on real-time embedded systems and includes components that directly operate on the I/O level. Infopipes come as an extension to a variant of Smalltalk and has a very rich set of operators. StreamIt started as a subset of Java and evolves into its own language and compiler infrastructure. StreamFlex builds on Java, does not modify the language syntax, but provides its own compiler and virtual machine. In general, stream processing languages focus on high responsiveness suitable for real-time applications.

Very recently, Soulé et al. designed Brooklet \[45\], a core calculus for stream processing language, which is designed to model any streaming language and facilitate reasoning about language implementation. It abstracts the language-specific features into wrappers, and exposes core mechanics such as data delivery, component collaboration and non-deterministic executions, which are usually hidden from the users of the streaming languages, as are in the case of S-Net. Brooklet has been demonstrated to successfully model StreamIt, CQL \[1\] and Sawzall \[12\].

We shall present a comparison between StreamIt, StreamFlex and S-Net in Section 2.1.4.

2.1.3 S-Net at a Glance

S-Net, as a coordination language, delegates the computational activities to \textit{box functions} written in full-fledged imperative languages like C or Single-assignment C (SaC) \[14\], and relies on language interfaces to interact between S-Net and the computational language. The box functions are wrapped in \textit{boxes}, a type of unit component in S-Net, which process input data and respond with output data by invoking the box functions. The data unit in S-Net is record; a box extracts values from the input records as the arguments to invoke the box function with, and turns the output values from the box function into output.
records. The S-Net program describes how the boxes and other unit components are interconnected, and is compiled into a separate piece of C code. A consumer program can then fire up the S-Net runtime system, which will build the network topology, establish data channels, or *streams*, between the components, and handle task distribution and perform resource management in a self-adaptive manner.

Like other stream processing systems, S-Net has relatively good real-time responsiveness, but it is achieved indirectly by promoting asynchronous processing. The streams can be seen as buffers between the output of one component and the input of another. Each unit component runs in its own (virtual) processor, and can start processing as soon as a record arrives at its input stream.

To connect the components, S-Net provides a small set of *combinators*: serial composition, parallel composition, serial replication and parallel replication. Apart from the serial composition, the other three combinators will generate splitter-merger pairs in the topology which are not directly available to users. A splitter splits the input stream into multiple output streams, whereas a merger combines the multiple input streams into one output stream. They create branches in the topology, and the records may be processed in different speeds in different branches. Regarding this, the programmer can choose between high throughput and preserved order, by using the non-deterministic and deterministic versions, respectively, of these three combinators. This affects the type of the generated splitter-merger pairs. A non-deterministic merger passes records to the output stream as soon as they arrive in any of its input streams, resulting in high throughput but possibly out-of-order responses. In contrast, a deterministic splitter adds control signals into the streams, so that the paired deterministic merger can rearrange the output records in the order of their corresponding input records.

S-Net is a typed language. The components declare what types of records they accept and produce. These in turn define the types of the streams. A splitter delivers each record into only one of its output streams, according to the stream types. There are subtyping rules to provide flexibility in the components accepting, and the splitters routing, the records.

### 2.1.4 Comparison: StreamIt, StreamFlex and S-Net

Despite sharing the similar roots, StreamIt, StreamFlex and S-Net are designed with totally different emphases. The StreamIt language introduces a more logical and productive programming pattern, and its compiler aims to improve the performance of streaming applications via stream analyses and optimisations. StreamFlex extends the application of stream programming into the real-time domain by targeting high-throughput, low-latency streaming applications with stringent quality-of-service requirements. The S-Net language facilitates collaboration between languages in a similarly logical and productive way as StreamIt does. However, the S-Net emphasis is for automated task distribution and resource management among processors in a self-adaptive manner.

As stream processing languages, these languages all share a similar topology: ‘filters’ handling computational activities are connected by ‘channels’ passing data between them. In StreamIt, filters are written in a language close to Java; while in StreamFlex, filters are programmer-developed derived classes of `Filter`. As a coordination language, S-Net allows the programmers to write their computational codes in languages of their choices, in separate code files, so long as the language interface is
provided in the S-Net package.

The filters in StreamIt have one input channel and one output channel, and the filter codes need to specify the amount of data consumed and produced per filter execution. The situation in StreamFlex is a lot more relaxed: filters can have zero or more input or output channels, and the filter code can take data from any input channels and put data to any output channels at will. In S-Net, filters, or the unit components in S-Net terminologies, which include boxes, filters, and synchrocells, are single-input-single-output, and are restricted to always consume one and only one record from the input stream, but can output zero or more records per execution.

In StreamFlex, filters are connected with each other in a flat, user-defined topology called StreamFlex Graph. In both StreamIt and S-Net, topological units can be connected serially or in parallel to form larger single-input-single-output units, thus allowing modular, hierarchical topology. In all three languages, the channels are implicitly typed by the connected filters’ input and output types. However, in S-Net, a component is allowed to output records of different types, and therefore an S-Net channel, or stream, implicitly has a union type. To accept records of multiple types at the other end of the stream, an S-Net parallel construct, called parallel composition, is allowed to connect two topological units accepting different types.

StreamIt has a loop construct called feedback loop, which on some conditions sends the data back to the same topological unit that produces them. The comparable construct in S-Net is the serial replication, but it simulates a loop structure with new replications of the topological unit, as the name suggests.

One common question in stream processing is, when the streamed data are given alternative routes (i.e. passing through a splitter), how they should be delivered. In StreamFlex, there are no splitters; more specifically, the StreamFlex programmers define how the filters output data among the multiple output channels, and therefore, data routing is explicit. In StreamIt, the splitters either route the input data in a round-robin fashion, or duplicate the input data to all output channels. In S-Net, the splitters route each record to only one output stream, matching the record type to the stream type.

On this note, it is worth mentioning that the record types understood by S-Net is orthogonal to the computational language’s data types. In a record, the computational values are stored as either integers or pointers and referenced by the labels under which the values are stored. The set of labels then form the record type of the record, which is used by S-Net to make routing decisions. In contrast, other languages discussed so far either have trivial type systems (e.g. Infopipe’s polarity design) or reuse the type systems of their base languages (e.g. StreamFlex being a variation of Java). , S-Net assigns types to the data which are already typed in the computational language, and bases the routing decisions on the assigned types.

An overview of this comparison is summarised in Table 2.1.
2.2 S-Net Language Introduction

In this section we will introduce S-Net programming by walking through a toy program that computes $\frac{n!}{10^n}$ for a stream of positive $n$s, using three computational components that perform multiplication, decrement and division. This program demonstrates only those S-Net features important to this thesis. A complete language reference can be found in the language report [25].

2.2.1 Boxes and Records

Before presenting the S-Net program, we shall first define the functions that make up the computational part of the program, which we will write in pseudo-code to avoid over-complication, using a C-like syntax. The function that performs multiplication is as follows.

```c
function mul ( BigNum* p, BigNum* q ) {
    // what this does: p2 = p * q
    BigNum* p2 = BigNum_Mul( p, q );
    SNet_Out( 0, p2 );
}
```

Typically, a computational component in a coordination language will perform a lot more than a simple multiplication. We simulate this by assuming that the incoming data are big numbers, and that we need a library multiplication function to handle them. In S-Net, component-level functions are called box functions, and are used by the boxes. The box that uses the box function above is defined in the S-Net program as follows:

```c
box mul ( (p,q=) -> (p) );
```
This serves as the interface declaration between S-Net runtime and the computational language: the name `mul` tells the box which function to call, and the part inside the outer-most parentheses defines the box's `signature`. Within the signature, the input tuple `(p, q=)` specifies the name and order of the input parameters, and the output tuple `(p)` specifies the name and order of the output parameters. The equal sign after `q` means that the parameter `q` and its value are to be copied to the output, and therefore, although the box declares only one output parameter, there are actually two.

S-Net components process records. A `record` is a set of label-value pairs. The box above processes each input record as follows: it gets the values stored under the labels `p` and `q`, removes only `p` from the record, invokes the function `mul` which produces an output value, stores the output value under label `p` into a copy of the remaining part of the input record (which contains `q`), and sends this new record to the output. Multiple input records are processed sequentially.

Each S-Net component accepts records of certain `record types`. For example, the box above accepts records of type `{p, q}`, where the labels `p` and `q` are called `fields` in this context. Field ordering is not important: the record

```
{ q : 12, p : 34 }
```

is safely accepted by the box `mul`, which will automatically reorder the two parameters before calling the box function. In other words, a box turns a record into a tuple (ordered list) of values, to match the parameter list of the box function for invocation. On the other hand, the data type `BigNum*` of the fields is opaque to S-Net, and S-Net will simply use the generic pointer type (`void*` in C) for all field values.

The box function calls `SNet_Out`, a library function provided by S-Net, to produce outputs. Compared with a return statement which outputs a single value, `SNet_Out` accepts an argument list of variable length, and therefore enables the box function to output multiple values at the same time, i.e. a tuple of values. This tuple of values must become a record when it leaves the box, and this is done by pairing the box function with the box definition, where the output tuple assigns a field name to each value. As a side note, by calling `SNet_Out` multiple times, the box function can produce more than one output record.

The first argument to `SNet_Out` is the output variant index, whose use will become clearer in the following box function.

```c
function dec ( BigNum* q ) {
    // what this does: q2 = q - 1
    BigNum* q2 = BigNum_Dec( q );
    if (BigNum_IsZero( q2 )) {
        SNet_Out( 1 );
    } else {
        SNet_Out( 0, q2 );
    }
}
```

The box function above, which performs decrement, is coupled with the box definition below:
box dec ( (q) -> (q) | (<#z>) );

There are two output variants in the definition above, separated by a vertical bar. The box function specifies which variant to use with the first argument to a call to SNet_Out. In the second variant, we can see a new type of label in a record: a binding tag, whose name is prefixed with a hash sign and then surrounded by angular brackets, to differentiate it from a field. It does not carry any computational data and is therefore excluded from the input and output parameters of the box function. We will come back to the binding tags later.

Finally, the box function and box definition that deal with division are as follows.

```cpp
function div ( BigNum* p, BigNum* q ) {
    // what this does: r = p / q
    BigNum* r = BigNum_Div( p, q );
    SNet_Out( 0, r );
}
```

2.2.2 Program and Networks

Below is the example S-Net program that creates a network to compute $\frac{n^10}{n!}$, and Figure 2.1 illustrates its topology.

```cpp
net program ( {n} -> {r} )
{
    box mul ( (p,q=) -> (p) );
    box dec ( (q) -> (q) | (<#z>) );
    box div ( (p,q) -> (r) );

    net split
    connect [ {n} -> {p=n,q=n,<t=9>} ; {p=n,q=n} ];

    net upper ( {p,q,<t>} -> {p} )
    {
        net inner
        connect [ {<t>} -> {<t=t-1>} ] .. mul;
    }
    connect inner * {<t>} if <t == 0>;
```

---

1 As of writing this section, the latest report indicates that the binding tags carry integral data like the tags (which will be discussed later in this section). An internal meeting has confirmed that they will become purely coordination entities in future releases, meaning that they do not carry any data and are opaque to the box functions, mirroring how the fields are purely computational entities and are opaque to S-Net.
program ( \{n\} -> \{r\} )

split

compute

\{<t>\} -> \{<t=t-1>\}
mul
*
\{<t>\} if <t==0>

upper ( \{p,q,<t>\} -> \{p\} )

mul

\{<#z>\}
->
\{<#z>\}
dec
*
\{<#z>,p\}
-> \{q=p\}

lower

join

div

Figure 2.1: Example S-Net Program
As can be seen, the program consists of a hierarchy of networks. The statement net ... connect defines (wires up) a network. The keyword net is followed by the network name, the optional network signature, and then the optional body which holds the definitions of inner boxes and subnetworks. The network topology is written after the keyword connect, as a topology expression, where the programmer can reference the boxes and networks which have been fully defined before (as marked by the semicolons) in all outer scopes, the current scope, and the outermost scope within the network body attached to the current network definition. The last network definition in the outermost scope defines the topology of the program, and is called the top-level network.

The network topology can contain as few as one box or network reference, one filter, or one synchrocell; it can also describe a composition of any number of them. In the rest of this S-Net introduction, we will follow the flow of the program, as illustrated in Figure 2.1 and introduce the S-Net elements as we come to them.

### 2.2.3 Program Flow and Serial Composition

One idea of the example program is to maximise parallelism, so the numerator $n^{10}$ and the denominator $n!$ can be computed simultaneously. To do this, we first split the input record, which contains just the value of $n$, into two records for calculating the numerator and denominator separately. The two records then flow through the computation network, resulting in two records carrying the results. They will then be joined together and fed into the dividing box, which produces the final result we need.

The topology of the top-level network program, i.e. the last connect clause in the program, agrees with the discussion above. The four components are combined in serial composition (operator `..`), so that the records can pass through each of them sequentially.

### 2.2.4 Filters and Tags

At the start of the program sits the network split, whose topology expression contains the following filter:

$$[\{n\} \rightarrow \{p=n,q=n,\text{<t=9>}\}; \{p=n,q=n\}]$$
A filter is a unit component capable of performing simple computational tasks like renaming, duplicating and removing fields, replicating records and so on. It is a shortcut to writing a box function and box declaration just for these simple tasks. For example, the filter above saves us from the trouble of preparing the following box:

```cpp
function split ( BigNum* n ) {
    BigNum* p = BigNum_Copy( n );
    BigNum* q = BigNum_Copy( n );
    int t = 9;
    SNet_Out( 0, p, q, t );
    BigNum* p2 = BigNum_Copy( n );
    BigNum* q2 = BigNum_Copy( n );
    SNet_Out( 1, p2, q2 );
}

box split ( (n) -> (p,q,<t>) | (p,q) );
```

We have used an imaginative library function `BigNum_Copy` to duplicate the field `n`.

From the code above, we can see one more type of label in a record: the *tag*. A tag is identified by the angular brackets surrounding its name, and holds an integer value (of type `int` in C) which is available to both the box functions and S-Net. This completes the collection of possible label types in a record: field, tag and binding tag.

A filter can initialise and change the tag values with *tag expressions* written inside the angular brackets, where constant integers, tag names from the input record and simple arithmetic operators are available. In the filter above, the tag `t` is written as `<t=9>`, where the tag expression is a single constant, initialising the tag to 9. We will see later that it causes the evaluation `p = p * q` to be done 9 times, with `p` and `q` both initialised to `n`, so that the final `p` contains the value of `n^{10}`.

The filter above creates two records per input, one of type `{p,q,<t>}`, and the other `{p,q}`. They will be delivered by the serial composition into the next component: `compute`.

### 2.2.5 Parallel Composition

The network `compute` is a *parallel composition* (operator `|`) of two subnetworks: `upper` and `lower`. The subnetworks are called the *branches* of the parallel composition. An input record into `compute` will be delivered into either branch, but never both. A branch is chosen when the system deems it more suitable than the other branch to process the incoming record. Suitability is defined by whether the branch can process the record without an error, and also how many labels in the record are potentially used. In case the system cannot decide which branch is more suitable, it will send the record to a non-deterministically chosen branch. A detailed branch selection process can be found in Section 4.2.3.3.

Without knowing the exact details, we shall still see from the illustration that the tag `<t>` is used in `upper`, but not in `lower`, and therefore, a record of type `{p,q,<t>}` is more suitable to be processed by
upper, while a record of type \( \{p, q\} \) can be delivered to lower.

### 2.2.6 Serial Replication

The network upper computes the numerator \( n^{10} \) from the record of type \( \{p, q, <t>\} \). The tag \( t \) is used by S-Net to keep track of the times of multiplications made, which has been initialised to 9 by the filter network split. The field \( p \) accumulates the result whereas \( q \) remains constant as the multiplier, both initialised to \( n \) by split. This also means that the first \( n \) has been included in \( p \) before the computation starts, and hence \( t \) is set to 9 instead of 10. The algorithm is as follows:

\[
\text{repeat until } t = 0: \\
\quad t = t - 1; \\
\quad p = p \times q; \\
\text{end repeat.}
\]

To achieve the repeating effect, we use a *serial replication* (operator \( \ast \)) in the topology of upper. While it might be logical to see it as a loop, a serial replication actually unfolds to a serial composition of an indefinite number of clones of the network, or *replicas*, where records flow strictly unidirectional and never revisit any component.

The terminating criteria are written after the \( \ast \) operator, in the form of record types, which are called the *terminating patterns* here. Each terminating pattern, provided it contains tags, can be accompanied with a further condition, written as the keyword if and a tag expression. A termination check occurs every time a record is about to enter a replica, including the first record and the first replica. A record is released from the serial replication if it *matches* an unconditional terminating pattern, which means it carries all fields and tags the pattern has, or if it matches a conditional terminating pattern and the tag values satisfy the condition. For all other cases, the record continues its journey in the serial replication by entering the next replica.

In the serial replication in the network upper, there is only one terminating pattern, which requires the record to have a tag \( t \) with a value of 0. This will ensure that the field \( p \) stores the result of \( n^{10} \) in the output record from the serial replication.

In theory, a serial replication contains an unlimited number of replicas, but in practise, replica generation is demand-driven. For example, the serial replication in the network upper will have exactly 9 replicas, each generated only as a response to some record awaiting its processing.

### 2.2.7 Flow Inheritance, Overwriting, Discarding

We now go into the serial replication in the network upper. We already know that the input record to this network has the type \( \{p, q, <t>\} \), but the first component it faces, within the serial replication, is a filter expecting a record type of \( \{<t>\} \).

Here, the concept of *flow inheritance* comes into play. A box or filter only uses the fields and tags in its declared input record type. An incoming record can carry more fields and tags than needed,
these extra elements as well as their values will be carried across the box or filter, and reappear in each and every output record.

As a result, the filter \[ \{<t>\} \rightarrow \{<t=t-1>\} \] in the program is just a more compact version of the following filter:

\[ \{p,q,<t>\} \rightarrow \{p,q,<t=t-1>\} \].

Similarly, the record entering the box mul, which still holds the type \( \{p,q,<t>\} \) after the filter above, will have the tag \( t \) and its value flow-inherited to the output. Recall from Section 2.2.1 that the box mul actually produces output records of type \( \{p,q\} \) where \( q \), having a suffix equal sign in the box signature, is actually used by the box function and flow-inherited to the output. This in turn means that all records appearing inside the serial replication in \texttt{upper} are of type \( \{p,q,<t>\} \).

As a side note, a record is unable to store two values under the same name, and a flow-inherited field or tag will be subsumed by an output field or tag, if they share the same name. It is equivalent to saying that the output field or tag overwrites the flow-inherited field or tag. For example, if the input record to the following box contains both field \( x \) and field \( y \), then the original value of \( y \) will be lost after the box:

\[ \text{box } x2y \ ( (x) \rightarrow (y) ) \].

Subsequently, if the record now goes through the following box, the original field \( y \) and its value will not reappear:

\[ \text{box } y2z \ ( (y) \rightarrow (z) ) \].

This creates a discarding effect: the serial composition \( x2y \ .. \ y2z \) discards the field \( y \), even when the overall effect seems to be just \( \{x\} \rightarrow \{z\} \).

### 2.2.8 Signed Networks

The record released from the serial replication in \texttt{upper} is of type \( \{p,q,<t>\} \), but we only need the result which is in the field \( p \). Instead of using a filter to trim off the record, we would like to demonstrate another approach, via the network signature attached to the definition of \texttt{upper}.

A network signature describes the behaviour of the network. The top-level network is required to have a signature for the purposes of documentation, code readability and error detection, the last of which works as follows: the programmer provides the signature of the program to tell the compiler about the expected behaviour, and the compiler gives an error if the signature does not match the inferred behaviour.

A non-top-level network with a signature, called a signed network for short, has the following additional benefits: it creates a boundary for the names of the record fields and tags, demands certain types of input records, and trims the output records. These added features make the network appear to its outside world as how its signature depicts, without the unexpected discarding effect that has been introduced in Section 2.2.7 or additional fields or tags in the output records.
In the example program, the signature of `upper` demands that the input records should be of type `{p,q,<t>}`, which agrees with the actual input record type that is originated from the filter `split`. It also declares that the output records will be of type `{p}`, which is a subset of the real output type `{p,q,<t>}` from the serial replication inside the network. As a consequence, the field `q` and the tag `<t>` will be removed from the output records.

### 2.2.9 Binding Tags Revisited

We now move onto the network that computes the denominator $n!$, `lower`. This computation is also a repetitive process and therefore a serial replication is used. The input record, from the filter `split`, contains the fields `p` and `q`, both initially storing the value of $n$. During each replica, the value stored in `q` is decremented by the box `dec`, and then multiplied into `p`. However, when the result of the decrement is zero, the box `dec` does not reproduce the field `q`, but instead changes the output record type to `{<#z>,p}`, where `<#z>` is a binding tag as introduced in Section 2.2.1. This type of record is unsuitable for processing by the box `mul`, which expects `p` and `q`. As a resolution, we place the box `mul` in parallel with the following filter:

```
[ {<#z>} -> {<#z>} ],
```

which simply expects the binding tag `<#z>`, but effectively does nothing to the record. As can be seen from the illustration, this filter allows the final record of type `{<#z>,p}` to skip the box `mul`, which does not accept it.

In fact, even if the record is of type `{<#z>,p,q}`, the box `mul` still will not accept it. This is because the record and the declared input type of the box do not agree on the binding tags. The box only accepts records with no binding tags and at least two fields, `p` and `q`, whereas the filter above accepts records with one and only one binding tag, `<#z>`, and any number of fields and tags. The underlying principle is that the binding tag set of the record must be exactly the same as the binding tag set of the declared input type of a box or filter, or a terminating pattern in a serial replication, or a matching pattern in a synchrocell (see Section 2.2.10), before the record can be processed by said S-Net component. This creates strong bindings between the record types and the S-Net components, which are easily controllable by the programmers. For example, in the network `lower`, it is obvious that the output type `{<#z>}` from the box `dec` binds to the lower branch of the parallel composition that immediately follows `dec`. Also, when the decrementation results in zero, the output record of type `{<#z>,p}` binds to the unconditional terminating pattern `{<#z>}` of the serial replication.

To complete the picture, the final filter in `lower` prepares the record for the box `div`, by removing the binding tag `<#z>`, and renaming the field `p`, which now holds the value of $n!$, into `q`.

### 2.2.10 Synchrocells

So far, the program is capable of producing the result of $n^{10}$, stored in a record of type `{p}`, and the result of $n!$, stored in a record of type `{q}`, for each input record of type `{n}`. We now need to combine
the two intermediate output records into one, so the box \texttt{div}, which expects \texttt{p} and \texttt{q} in the same record, can be used for the final computation. A \textit{synchrocell} is the only component in S-Net capable of merging multiple records.

The network \texttt{join} is actually a serial replication of the following synchrocell:

\[
\{ [1 \{p\}, \{q\}] \}.
\]

The synchrocell lists two or more record types, called \textit{patterns}, each of which can optionally carry a tag expression as an additional condition of match, like the terminating patterns of a serial replication. An incoming record will be tested against each \textit{unmatched} pattern. A \textit{match} is made if the binding tag sets of the record and the pattern is the same, the record hold all fields and tags mentioned in the pattern, and if a tag expression is present, the evaluation result is true (non-zero). One record may match multiple unmatched patterns simultaneously. When any match is made, the synchrocell stores the values of all the fields and tags which have participated in the match, and disposes of the record, but if there has been no match, the record is passed through as the output with no changes. When all patterns are matched, the synchrocell releases an output record containing all the binding tags, fields and tags listed in the synchrocell, together with their stored values.

Flow inheritance can be observed from the first matching pattern, which is called the \textit{main pattern}. A record matching the previously unmatched main pattern will be stored in full, including any additional fields and tags. However, they are still subject to being overwritten by the fields and tags from other patterns, called the \textit{auxiliary patterns}, which have a higher priority.

After a synchrocell emits a combined record, it does not reset itself to the initial state: all patterns remain matched, and all future incoming records will be passed through. If resetting is required, one can simply wrap the synchrocell in a serial replication, with the combined record type as the single terminating pattern. In this way, a record escaping a used synchrocell will be kept in the serial replication and captured by a fresh synchrocell, possibly generated just in time, further down the chain of replicas. The combination of synchrocell and serial replication is so commonly used that it has its own short name: \textit{sync-star}.

In the example program, because there may be multiple input records of type \texttt{\{n\}}, resulting in multiple pairs of records of types \texttt{\{p\}} and \texttt{\{q\}}, a sync-star is used in the network \texttt{join} to combine the pairs into single records of type \texttt{\{p, q\}}, ready to be processed by the box \texttt{div}, which gives us the final result of type \texttt{\{r\}}.

This concludes the S-Net introduction, which provides a preview of S-Net and sufficient information for the readers to continue onto the rest of this thesis.

\section{Background of S-Net Type System}

Although S-Net is a coordination language, it injects its own understanding of types into the data, i.e. the record types. This makes it comparable to a full language, and the task to model its type system far from trivial. In this section, we go over the history of type systems to better understand the foundations
of the S-Net type system, and in the process, clarify some choices for S-Net which we will make.

2.3.1 Type Systems and Type Inference

A type system is a syntactic method for checking the absence of certain erroneous behaviours by classifying program phrases according to the kinds of values they compute [41]. The type system associates a type for each computed value and, using that information, attempts to ensure that no type errors can occur in a program. What constitutes a type error is usually defined by the type system itself.

Type inference refers to the automatic deduction of the type of an expression without explicit type annotation, using the elements in the expression with known types and the context wherein the expression is used. Type inference helps reduce or eliminate the programmer's burden to define the types in code.

Statically typed programming languages enforce that type information is available at compile time, either explicitly annotated or assigned by the type inference procedure, so that type checking can be done and type errors are caught early before the programs are allowed to execute. It also allows the compiler to apply some optimisations, and even remove all runtime type checking, knowing that type errors are guaranteed not to occur. S-Net is a statically typed language.

The study of statically typed programming languages has traditionally been revolving around a class of typed λ-calculi, including the simply typed λ-calculus [15], System F [18], System F≤ [12, 18], F∧ [39] and so on, which provide the mathematic model for describing and reasoning the languages. An overview of these languages and their type systems can be found in [8]. All these calculi support functions as values and thus assign types to functions.

2.3.2 Types

A type represents the common properties of the values of that type. These properties can be used to check the validity of program phrases involving these values, such as determining whether a function application is applied to a value of the function's expected input type. A type τ' is said to be a subtype of another type τ, if a value of type τ' can be safely used in a context where a value of type τ is expected. A value of both types τ₁ and τ₂ are said to be of the intersection type τ₁ ∧ τ₂ [29], and a union type τ₁ ∨ τ₂ can be given to values of either type τ₁ or τ₂ [5, 6]. Since their introduction, intersection types received many attentions [3, 10, 20, 40, 49].

In set theory, a type can be seen as the set of all values of that type [13]. This perspective can be used to explain many concepts in a simple way: for example, the notion that a value is of some type is treated as that the value exists in the set that represents its type, a subtype is a subset of its supertype, an intersection type τ₁ ∧ τ₂ suits the values residing in the intersection τ₁ ∩ τ₂, and a union type τ₁ ∨ τ₂ is for the values in the union τ₁ ∪ τ₂.

Record types. A record is a set of field-value pairs. A record type is a set of field-type pairs where the type specifies what values are allowed under the associated field. Records are a mathematic model for complex data structures in practical programming languages. The subtype relation on record types
are two-dimensional: a subtype of a record type can be introduced by adding fields, or changing the type of an existing field to a subtype.

In [13], Cardelli et al suggested that the object types in the object-oriented languages with inheritance correspond to the record types, when a class name is seen as the abbreviation of the set of all fields and methods in the class, and subclass as having more fields and methods in the set. The record types in these cases are generally static, but in [22] Gaster et al designed a flexible, polymorphic type system for extensible records with an effective type inference algorithm and compilation methods. This went on to provide the basis for typing dynamic languages like JavaScript [47].

The record types in S-Net are simpler in the sense of the limited choices of data types attached to the labels: at the coordination level, S-Net only recognizes the integer type and the pointer type. In fact, through the reduction process described in Section 1.3 which will be further detailed in Chapter 3, we are able to disregard these data types completely, and turn the record types into label sets. It is therefore easier for us to model the operations on record types as set operations.

2.3.3 Type Polymorphism

Type polymorphism refers to the features in some programming languages that allow a value to appear in different types. This can take many forms. For example, a value of type $\tau'$, which is a subtype of $\tau$, is also of type $\tau$. This form of polymorphism is conventionally referred to as inclusion polymorphism. On the other hand, the identity function, written in $\lambda$-terms as $\lambda x.x$, which accepts a value of any type and returns the same type (in fact the same value), is usually typed as $\forall \alpha. \alpha \rightarrow \alpha$. This is called a polymorphic type, or $\textit{polytype}$ in [26], and a $\textit{type-scheme}$ in [19] where the variable $\alpha$ is defined as a $\textit{type variable}$. The fact that the type can contain parameters contributes to the term for this type of polymorphism: parametric polymorphism. Parametric polymorphism then evolved into more expressive forms, such as bounded quantification ($\forall \alpha \leq \tau$) [13] and qualified types ($\forall \alpha : p(\alpha)$) [22, 35].

In [13], Cardelli et al categorised both inclusion polymorphism and parametric polymorphism as universal polymorphism, because one such polymorphic type can uniformly match infinite number of types having a common property. In comparison, ad-hoc polymorphism refers to overloaded functions and coercion, with a finite choice of matching types, and can normally be treated as different entities of different types sharing the same name or symbol. For this reason, ad-hoc polymorphism is usually excluded in the discussions of the calculi.

Polymorphism in S-Net. An S-Net component can act on input records of different types. This is by definition a kind of polymorphism, but it does not fit nicely in any of the polymorphism kinds covered above.

On introducing record types, Cardelli defines record subtyping as follows: if a record type $r_1$ has more labels than another record type $r_2$, but is otherwise compatible with $r_2$, then $r_1$ is a subtype of $r_2$. Using this definition, if flow inheritance enables an input record of type \{a, b\} to be accepted by an S-Net component \(n\) which otherwise only works for a record type \{a\}, then \{a, b\} has more labels than \{a\} and is a subtype of \{a\}. This scenario seems to fit the inclusion polymorphism.
However, the additional label b has the ability to modify the output record type. Suppose the output of n in response to any input record of type \{a\} is \{x\}, then for any set of labels α which are in addition to (have no duplication with) \{a\}, if n accepts the type \{a\} + α, then the output type will be \{x\} + α, where + combines the two record types by merging the labels. It seems that n has the parametric signature \(\forall α. \{a\} + α \rightarrow \{x\} + α\), which can be instantiated to \{a,b\} \rightarrow \{b,x\} for the previous example. This is a scenario of parametric polymorphism.

Now consider the case where n is actually a parallel composition, where the second branch is triggered, due to a better match with the input record types, only for some special forms of α such as when \{b,c\} ⊆ α, and that instead of outputting records of type \{x\} + α, this branch produces records of a completely different type \{y\}. Only ad-hoc polymorphism can describe this behaviour. Note that this new output type is most probably in the form of \{y\} + β, where β is the part of α that does not contribute to the change of routing decision, e.g. \(β = α\setminus\{b,c\}\).

Finally, think about n as a serial composition combining the previously discussed parallel composition and a filter which accepts only \{x\} (e.g. \{}x\}\(\rightarrow\){x}\). Thanks to flow inheritance, all records of type \{x\} + α can pass through this filter unchanged. But no records of type \{y\} + β is accepted (assuming \(x \notin β\)), meaning that any records originating from the second branch of that parallel composition will cause a type error. To prevent it, we should not allow the second branch to be triggered at all, and the only way – apart from asking the programmer to remove the second branch – is to disallow said special forms of α from occurring at the input of n. In other words, not all subtypes of \{a\} are accepted by n. This means that either Cardelli’s record subtyping definition does not suit S-Net, or that S-Net does not generally exhibit inclusion polymorphism.

The bottom line is that any attempt to fit the special polymorphism that S-Net exhibits into existing frames will in no doubt complicate the thought process. To overcome this, in this thesis we go back to the origin of types, and see a type as the set of values it can represent, and build the S-Net type system on this foundation.

### 2.3.4 Soundness and Completeness

As a logical system, any type system has the properties of soundness and completeness. The soundness of a type system is most famously defined by Milner’s slogan ‘well-typed programs cannot go wrong’ [36], which should be understood as: if the type system declares a program well-typed, then the program cannot reach the special state ‘wrong’, which signifies a runtime error. Wright proposed in [51] the definition of weak soundness as described above, and strong soundness as follows: if the type system declares an expression to be of some type, then in execution the expression evaluates to a value which agrees with the type. Designers of statically typed languages pay particular attention to the soundness of the type systems, because they guarantee that no runtime type errors can ever occur. This guarantee provides the theoretic support for a category of runtime optimisations related to types, such as removing the runtime type checks and minimising the type information stored in the objects’ memory models. Nevertheless, programming language designers frequently weaken the soundness definition to some extent in favour of the languages’ usability. For instance, certain runtime errors which are too difficult to
be captured by the type system are usually excluded from the type system, with a famous example being the division-by-zero error. In this case, the type system can be sound in terms of capturing runtime type errors, but not all runtime errors.

The definition of completeness should logically be the dual of that of soundness, so it can be as follows: if a program cannot reach the ‘wrong’ state, then the type system will declare it well-typed. Equivalently speaking, if the type system declares a program ill-typed, then the program can reach the ‘wrong’ state. Whereas a sound type system accepts only good programs, a complete type system rejects only bad programs. It is generally more tolerated to incorrectly reject good programs than to incorrectly accept bad programs, so the completeness property receives less focus. Examples of research on type system completeness can be found in [7, 21, 30, 50].

For an S-Net type system, the definitions of soundness and completeness require some clarification. In Section 2.2.5 we have mentioned that the parallel composition may non-deterministically choose a branch to process an input record. The exact choice is unknown at compile time. Similarly, some boxes, such as dec on page 27, have several output choices, and S-Net has no means to know the decision made by the opaque computational code — it is non-deterministic to S-Net what output choice or choices will be selected. To guarantee type safety, we let the soundness and completeness definitions to cover all non-deterministic events. The type system is sound if, if it accepts an S-Net program, then a type error will never occur for all potential results of all non-deterministic events during the program execution. The completeness definition is defined dually.

2.3.5 Abstract Interpretations

Cousot demonstrated in [17] that type systems can be seen as abstract interpretations of the programming language, by putting the denotational semantics of the untyped \(\lambda\)-calculus, the Danaë-Milner polymorphic type schemes [19], the Church-Curry monotypes [15], the Hindley principal typing algorithm [28], etc. in a lattice ordered by abstraction. Developing on this idea, Kahrs [34] declared that any abstract interpretation creates a type system. He modelled programming languages and type systems as transition systems with error states (ETS), and discussed the soundness and completeness of the type system in terms of the corresponding properties of a new ETS coupling the ETS for the programming language and the ETS for the type system.

The methodology of this thesis is similar: the reduction process described in Section 1.3 creates BL-Net, which is essentially an abstract interpretation of S-Net for us to focus on S-Net’s type semantics. The type system for BL-Net is used as the type system for S-Net, using BL-Net as a bridge.

2.4 Summary

We have now studied S-Net, including a snapshot of its history. We have clarified some design choices for the type system for S-Net: we use abstract interpretations to understand S-Net’s type semantics, define the soundness and completeness of the S-Net type system around the non-deterministic nature, and use set theory to describe the types. We are now ready to take on the challenge, starting from reducing
S-Net to L-Net.
Chapter 3

Reducing S-Net

Despite being a tiny language, S-Net has so many subtleties that it would be too overwhelming to model a type system for it directly. In order not to be distracted, we shall abstract away the features in S-Net that have no impact on the correctness of the type system. In Section 1.3 we have briefly introduced the reduction process which results in the projection, BL-Net and L-Net. Now that we have covered some details of S-Net in Section 2.2 in this chapter we shall discuss the reduction process in detail, to help understand these languages in the context of a type system for S-Net.

This chapter also contains the full algorithm to translate S-Net programs to BL-Net, skipping the projection, the language sitting inbetween. This is possible because the extra features in the projection compared with BL-Net can all be modelled using BL-Net features.

Some notions and program snippets in BL-Net are inevitable in this chapter. We will preview these BL-Net entities informally, and leave the formal specification in Chapter 5.

3.1 Foundation

S-Net is a stream processing coordination language. An S-Net stream carries a sequence of records to be processed by the component it connects to, one record at a time, and the component produces a stream of output records in response. Each component accepts (can process) a certain set of record types, which do not change midway in an execution. This can be proven straightforwardly from the observation that the only stateful unit component – the synchrocell – accepts all record types.

The first purpose of the S-Net type system (see Section 1.1) implies that we should check that for every component, every potential record in its input stream can be processed, i.e. every potential input record type is accepted. However, because the accepted record types do not change, the individual records can be seen as the inputs to the component in separate executions. Even when several input records are in fact the outputs from the upstream component in response to a single input record of its own, we can consider that the upstream component causes separate executions of the current component. This reduces the notion of streams to individual records.

There are two variants each for the parallel composition, serial replication and parallel replication in
S-Net: non-deterministic (fast throughput) and deterministic (order preserving). Because the notion of streams are reduced, the record order in the stream is no longer relevant. This collapses the two variants for each S-Net combinators mentioned above.

Records store data under fields and tags. The data stored under a field is opaque to S-Net, so it has no relevance to the S-Net type system. The data stored under a tag is always an integer, the only primitive type supported by S-Net, so it needs no special care, either. This reduces a record into a label set. Recall from Section 2.2.4 that a label set can contain fields, tags and binding tags. After the reduction, fields and tags no longer store data, so assuming some preprocessing procedure exists to guarantee that fields and tags have non-overlapping namespaces, their difference becomes irrelevant to the type system. On the contrary, binding tags affect how the records match the parallel branches and terminating patterns, so the difference between a binding tag and a non-binding-tag label should be preserved. This means that the label set after the reduction consists of \textit{binds} (short for binding tags) and \textit{labels} (short for non-binding-tag labels).

### 3.1.1 Label Set Transforming Languages

Through the discussion above we have established that the S-Net type system is unaffected if streams are reduced to discrete label sets. This helps define the category of the projection, BL-Net and L-Net: they are \textit{label set transforming languages}. A program in this category, as well as any component within, takes a label set as the input, and produces another label set as the output. We can see the label sets as the first-class citizens in a label-set transforming language. For this reason, we call the label sets \textit{values}.

We mentioned before that an input to an upstream component may cause separate executions of the current component. Viewing from an arbitrary single execution, we can assume that the upstream component has non-deterministically chosen the intermediate output that resulted in the selected execution. In fact, for these label set transforming languages, we allow a component to non-deterministically decide the form of the output value in response to each input value.

### 3.1.2 Projection, BL-Net and L-Net

Table 3.1 provides a summary of the features of S-Net, the projection, BL-Net and L-Net. S-Net has all relevant features listed, and each language to the right has a reduced set of features of the adjacent language to its left. A ‘√’ means that the language inherits the feature to the left of the tick, ‘(Modelled)’ means that the language models the behaviour of the feature to the left using other features available in the language, a new term suggests that the feature is reduced, and an empty cell indicates that the feature is absent in the language.

The projection is, as the name suggests, a direct projection of S-Net on the space of label set transforming languages. Apart from the reduced features covered in this section, all other S-Net features are preserved. BL-Net further trims the projection by removing the filter, the synchrocell, the parallel replication and the signed network, because their semantics can be modelled using other features in BL-Net. Finally, L-Net is a subset of BL-Net without the bind or the serial replication. The sole purpose of its
existence is to allow us to discuss the concepts in manageable batches.

### 3.2 Translation

In this section we will formalise the translation process to turn S-Net programs into BL-Net programs, skipping the projection, the intermediate language between S-Net and BL-Net. The only difference between BL-Net and the projection is that some components in the projection are modelled in BL-Net. Those components in the projection would look almost identical to their counterparts in S-Net. We would rather translate the S-Net components directly to BL-Net models.

Figure 3.1 shows the example S-Net program on page 27 side by side with two BL-Net programs as the translation result.

**Soundness of translation.** At this point it is worth emphasising the aim of the translation process: to remove irrelevant information from the S-Net programs, while preserving the full behaviour with regard to types, so the type inference result on the translated BL-Net programs can be used directly on the S-Net level. This requires that the translation is sound, which formally means that if the BL-Net type system, which is sound, accepts a program translated from S-Net, then the original S-Net program is type correct. The soundness of translation can also take the form that the translation preserves the S-Net program’s type-related behaviours.

In this thesis, the soundness of translation cannot be proved. Such proof requires that the S-Net semantics is well defined. We mentioned in Section 1.1 that S-Net currently suffers from the type-semantics interdependency, and the most recent type system it depends on [11] is known to have difficulty handling some special cases. There is no solid foundation on which to build the soundness proof. On the other hand, this thesis advocates that S-Net should be updated to match BL-Net’s semantics. If that happens, the soundness of this translation is a tautology.

Nonetheless, in this section we will discuss the validity of this translation as we go through the components.

<table>
<thead>
<tr>
<th>S-Net</th>
<th>Projection</th>
<th>BL-Net</th>
<th>L-Net</th>
</tr>
</thead>
<tbody>
<tr>
<td>Record</td>
<td>Value (label set)</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Field</td>
<td>Label</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Tag</td>
<td>Label</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Binding tag</td>
<td>Bind</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>Box</td>
<td></td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Filter</td>
<td></td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>Synchronous cell</td>
<td></td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>Serial composition</td>
<td></td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Parallel composition</td>
<td></td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Serial replication</td>
<td></td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>Parallel replication</td>
<td></td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>Signed network</td>
<td></td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>Record stream</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Deterministic combinators

Table 3.1: Language features comparison
program : let mul = (box ₀♯pq → ₀♯pq) in let dec = (box ₀♯q → ₀♯q, ½#₀) in let div = (box ₀♯eq → ₀♯eq) in let split = (box ₀♯u → ₀♯u, ₀♯pq) in let upper = (box ₀♯eq → ₀♯eq) in let lower = ((dec ∥ box z#₀ → z#₀)) ∥ box z#₂ → ₀♯q) in let compute = (upper ∥ lower) in let join = (box ₀♯p → ₀♯p, ₀♯pq) ∥ box ₀♯q → ₀♯q) ∥ ₀♯pq; illaume in split ∥ compute ∥ join ∥ div.

upper : let mul = (box ₀♯pq → ₀♯pq) in let inner = (box ₀♯τ → ₀♯τ, ₀♯pq) in inner ∥ ₀♯τ; ₀♯pq.

program : let mul = (box ₀♯pq → ₀♯pq) in let dec = (box ₀♯q → ₀♯q, ½#₀) in let div = (box ₀♯eq → ₀♯eq) in let split = (box ₀♯u → ₀♯u, ₀♯pq) in let upper = (box ₀♯eq → ₀♯eq) in let lower = ((dec ∥ box z#₀ → z#₀)) ∥ box z#₂ → ₀♯q) in let compute = (upper ∥ lower) in let join = (box ₀♯p → ₀♯p, ₀♯pq) ∥ box ₀♯q → ₀♯q) ∥ ₀♯pq; illaume in split ∥ compute ∥ join ∥ div.

3.2.1 Record Types and Patterns

While records exist during the execution of an S-Net program, the closest entity that can be found in code is the record types, used in various places such as the input and output types of a box or filter, the terminating patterns of a serial replication an so on.

Translating an S-Net record type into a BL-Net value is a straightforward process using the reduction process discussed in Section 3.1. The binding tags in S-Net are translated to the binds in BL-Net, and the fields and tags S-Net are translated to the labels in BL-Net.

In BL-Net, binds and labels share the same namespace Label but are put into different positions of a value. A value is formatted as ]ls₁#ls₂, where ls₁ is the set of binds and ls₂ is the set of labels. There is a special element ∗ in Label which should be kept out of the programmers’ reach. In a literal value, we use a small capital letter as a label or bind, and a string of small capital letters as the syntactic sugar for a set of labels or binds. The string can be of one letter to denote a singleton set, but an empty set is still written as 0. As an example, the value ₀♯pq has no binds and two labels ‘p’ and ‘q’.

For the translation, we assume an arbitrary injective mapping Λ from the union of all fields, tags and binding tags in S-Net, to the set Label\{*\} defined for BL-Net. The injective nature ensures that every field, tag or binding tag will be translated to a unique label. For illustration purposes, a record type in S-Net is denoted by the symbol τ, and the translation process is denoted by the function T. The following formula expresses the translation from an S-Net record type to a BL-Net value:

\[ T(\tau) = ]ls₁#ls₂, \text{ where } ls₁ = \text{set of binds } \text{ and } ls₂ = \text{set of labels} \]
\[ T(\tau) \triangleq \{ \Lambda(id) \mid id \text{ is a binding tag in } \tau \} \]
\[ \#\{ \Lambda(id) \mid id \text{ is a field or tag in } \tau \} \]

If \( \tau \) is a pattern used in the filter output, it may contain field initialisers and tag initialisers, but in the translation process they are ignored.

In Figure 3.1 we have used a literal mapping as \( \Lambda \), because the source S-Net program only uses one-letter identifiers for the fields, tags and binding tags. We can see in the figure that the value \( \emptyset \#pq \) is translated from this filter output with field initialisers:

\[ \{p=n,q=n\} \]

We would also like to define the following shortcut for translating a set of record types or patterns:

\[ T(\tau_s) \triangleq \{ T(\tau) \mid \tau \in \tau_s \} \]

3.2.2 Boxes

An S-Net box definition contains the name of the computational function to call, which is not needed in BL-Net, an input record type, and zero or more output variants. As shown in the box \texttt{mul} in the source program, the input record type may contain fields and tags decorated with the equal sign, which means that they are to be copied to every output record. BL-Net has no equivalent notation, so the copied fields and tags need to be explicitly added when translating the output variants.

In the formula below, \( \tau \) denotes the input record type, \( \tau_e \) collects the fields and tags in \( \tau \) with the equal sign, and \( \tau_s \) is the set of all output variants defined in the box.

\[ T(\text{box } \text{id} \ (\tau \rightarrow \tau_s \ )); \}
\[ \triangleq \text{box } T(\tau) \rightarrow \{ T(\tau_0 \cup \tau_e) \mid \tau_0 \in \tau_s \} \]

For example, the following box:

\[ \text{box } \text{dec } (\ (q) \rightarrow (q) \ | \ (<\#z>) \ ) \];

is translated into

\[ \text{box } \emptyset \#q \rightarrow \{ \emptyset \#q,z\#\emptyset \} \].

As a preview, the semantics of the BL-Net box above is as follows. It accepts values with no binds and at least the label \( q \). During a particular execution, it removes \( q \) from the input value, non-deterministically selects an output choice between \( \emptyset \#q \) and \( z\#\emptyset \), combines the remaining of the input value with the selected output choice, and outputs the resulting value. This is equivalent with the semantics of the original S-Net box modulo the effect of reduction, apart from that the BL-Net box must output a value whereas the S-Net box function is not obliged to produce any output. (For the case
where the box function produces multiple output records, recall the discussion in Section 3.1 that we reduce multiple output records into separate executions.) This slight difference plays an important role in the correctness of the type system. Since the box functions are opaque to S-Net, there is no guarantee regarding the number and types of the output records. To fulfil the first purpose of the type system that it should guarantee that the unit components receive only those records they can process, all possible cases must be checked.

Note that if $\tau_s = \emptyset$, the resulting BL-Net component is $\text{box} \, T(\tau) \rightarrow \{\}$. In BL-Net terminology, this is a sink, as it captures input values but does not produce outputs. The S-Net version behaves identically: because there is no valid argument for the box function to invoke $\text{SNet\_Out}$ with, no output records are produced.

### 3.2.3 Serial and Parallel Composition

An S-Net serial or parallel composition can be directly mapped to its BL-Net counterpart. Because they reside in the topology expression after the keyword connect, we shall also translate their operands. The formulae below use the symbol $e$ to denote a part of the S-Net topology expression.

\[
T( \ldots e_2 \ldots ) \triangleq T(e_1) \cdot T(e_2) \\
T( e_1 | e_2 ) \triangleq T(e_1) \parallel T(e_2)
\]

When $e$ binds to just an identifier, i.e. a box or network name, we do not perform any translation. The reference will be resolved using the \textit{let \ldots in} syntax, when we translate the network bearing this topology expression, in Section 3.2.8.

\[
T( \text{id} ) \triangleq \text{id}
\]

The semantics of the serial composition in S-Net and in BL-Net are identical: input records or values are given to the left operand for processing, then the intermediate outputs from the left operand are given to the right operand. The semantics of the parallel composition, however, can differ greatly depending on the actual parallel composition instance, even though they intuitively mean the same: each input record or value is processed by whichever operand that matches it better. In S-Net, computing the match scores requires the branches’ signatures, which may be inferred using the algorithms from some type system, so the parallel composition semantics depends on the type system. In contrast, BL-Net has a specification independent from any type system, defining a fixed behaviour of the parallel composition.

The deterministic variant of the parallel composition in S-Net is written as

\[
e_1 \parallel e_2,
\]

which guarantees that the output records are in the order of their corresponding input records. Because
record order is immaterial in BL-Net, it is translated as a non-deterministic parallel composition:

\[ T(e_1 || e_2) \triangleq T(e_1 || e_2). \]

### 3.2.4 Serial Replication

Like the serial composition, the serial replication in BL-Net is also a direct clone of its counterpart in S-Net modulo the reduction: it lets the operand network process the record or value as long as it does not match a terminating pattern. An S-Net serial replication carries two types of terminating patterns, namely, unconditional and conditional. Whether a conditional terminating pattern can terminate a record depends on the tag values of the record, which does not exist in BL-Net. We can consider that it is a non-deterministic event to BL-Net. In Section 5.4.2 we will see that a conditional terminating pattern of a BL-Net serial replication will non-deterministically decide whether a value should be terminated, and therefore it is fine to map an S-Net serial replication directly to a BL-Net serial replication.

In the formula below, we assume that \( \pi \) holds all the S-Net terminating patterns. A BL-Net serial replication is formatted as \( e^* \pi \) where \( e^* \) and \( \pi \) are separate sets of unconditional and conditional terminating patterns, respectively.

\[
T(e^* \pi) \triangleq T(e) \{ T(\tau) \mid \tau \text{ is an unconditional terminating pattern in } \pi \} ; \{ T(\tau) \mid \tau \text{ is a conditional terminating pattern in } \pi \}
\]

The deterministic, i.e. order preserving variant of the serial replication in S-Net is written as

\[ e^{**} \pi, \]

and translated as a non-deterministic serial replication:

\[ T(e^{**} \pi) \triangleq T(e \* \pi). \]

### 3.2.5 Filters

An S-Net filter is a light-weight, in-line S-Net box, capable of simple computational tasks such as removing and renaming fields and tags, changing the values of tags, duplicating records and so on. Every filter can be rewritten to an S-Net box. As we have demonstrated in Section 2.2.4 the box signature and the box function are both determined by the filter.

We translate an S-Net filter to a BL-Net box by first rewriting it to an S-Net box. In other words, a filter’s semantics can be modelled by a BL-Net box. The formula below shows the translation result.

\[
T( [ \tau \rightarrow \tau s ] ) \triangleq \text{box } T(\tau) \rightarrow T(\tau s)
\]
3.2.6 Synchrocells

In Section 2.2.10 we have seen that a synchrocell waits for two or more records of the specified types to be matched and then merges them into one output record. Flow inheritance is observed only through the main pattern; excess fields and tags from a record matching an auxiliary pattern will not be kept. The synchrocell is the only S-Net component with state: it stores the input records in its state before the merger. Once the merger is done, the synchrocell becomes dead and simply passes all further input records to the output. The stateful nature means that we are unable to describe the exact behaviour of a synchrocell using BL-Net where all components are stateless, but we can create a model to approximate its characteristics with regard to the types.

Consider the synchrocell \([ \{m\}, \{n,q\}, \{p\} \]) where \{m\} is the main pattern and \{n,q\} and \{p\} are the auxiliary patterns. Suppose it first receives a record of type \{m,$\}, where $ denotes one or more fields with unknown names which are not n, p or q. Due to the need to exhibit flow inheritance through the main pattern, the synchrocell stores the record in full, including $ . Ultimately, it releases an output record of type \{m,n,p,q,$\}, which means the following box describes the same behaviour as what is discussed so far, when taking the effect of flow inheritance into consideration:

\[
b_{m1} = \text{box } \emptyset\#M \rightarrow \{\emptyset\#MNPQ\}.
\]

Now assume that it receives another record of type \{m,$\}. Because the main pattern \{m\} has been matched, this record will be passed to the output directly. The following box models this behaviour:

\[
b_{m2} = \text{box } \emptyset\#M \rightarrow \{\emptyset\#M\}.
\]

In real S-Net, the output from \(b_{m2}\) will appear before the output from \(b_{m1}\). However, since record order is immaterial in BL-Net, we consider it non-deterministic whether an input record matching the main pattern triggers a merger or is passed through intact. As a consequence, the two boxes above can be combined into one:

\[
b_{m} = \text{box } \emptyset\#M \rightarrow \{\emptyset\#M, \emptyset\#MNPQ\}.
\]

The first incoming record matching the auxiliary pattern \{n,q\} will be stored in the synchrocell state, which in other words means there will be no output. This is also true if the record triggers a merger, in which case we can see the output record as the delayed response from the box \(b_{m1}\). All subsequent records matching the auxiliary pattern will be passed through intact, as the following box does:

\[
b_{n} = \text{box } \emptyset\#NQ \rightarrow \{\emptyset\#NQ\}.
\]

Once again, it is non-deterministic, to BL-Net, whether such a record is stored in the synchrocell state or passed through, but we can say that the record may produce an output of itself. Following the discussion in Section 3.2.2, the semantics that \(b_{n}\) always produces the output is needed for the type system correctness. Therefore, we use \(b_{n}\) to model the part of synchrocell behaviour relevant to the
auxiliary pattern \{n, q\}. Similarly, the other auxiliary pattern \{p\} can be modelled with box \(b_p\) below:

\[
b_p = \text{box } \emptyset \# p \rightarrow \{\emptyset \# p\}.
\]

One may now think that the apparent complete model is \(b_m \parallel b_n \parallel b_p\). However, doing so would put \(b_n\) at an advantage, because it uses two labels whereas the other two boxes use one. A BL-Net parallel composition favours the branch using more labels, so this model would diverge from the actual synchrocell semantics. Suppose the second auxiliary pattern \{p\} has been matched. A record of type \{z, n, p, q\} should match both remaining patterns simultaneously, causing the merged record of type \{z, n, p, q\} to be released. However, in the model \(b_m \parallel b_n \parallel b_p\), the value \(\emptyset \# MNP\) is always attracted to the branch \(b_n\) only, never producing the merged output.

The workaround is to minimise the number of labels used in the branches describing the auxiliary patterns. We reduce the auxiliary patterns to their roots (keeping only the binding tags), creating the following box for both auxiliary patterns \{n, q\} and \{p\}:

\[
b_r = \text{box } \emptyset \# \emptyset \rightarrow \{\emptyset \# \emptyset\}.
\]

This new box accepts all values \(b_n\) or \(b_p\) accepts, as well as other values with no binds. Accepting these additional values does not invalidate the model, because according to the synchrocell definition, they do not match any patterns in the synchrocell and will be passed to the output, regardless of the synchrocell’s state. The corrected full model for the example synchrocell is therefore

\[
b_m \parallel b_r \parallel b_r,
\]

which is semantically equivalent with

\[
b_m \parallel b_r.
\]

One may now argue that the model does not exhibit the full behaviour. For example, a record of type \{<#a>\} can safely pass through the example synchrocell, but the value \(\lambda \# \emptyset\) is not accepted by the model. In fact, the model only permits a subset of behaviours of the synchrocell intentionally. Consider a syncstar combination, wrapping the example synchrocell with the merged record type as the unconditional terminating pattern:

\[
[ | \{z\}, \{n, q\}, \{p\} | ] \ast \{z, n, p, q\}.
\]

If a record of type \{<#a>\} is allowed into this network, it will pass through all instances of the synchrocell, causing the serial replication to spawn infinitely many new replicas and deplete all available resources. It is therefore reasonable to reject the record types which can never be captured by the synchrocell. In fact, it is fully justifiable for a type inference implementation to assign manually crafted types directly to the branches describing the auxiliary patterns, which accept only the values that can be matched, but uses no labels from them. But for now, we will stick with our modelling approach, and translate all synchrocels into parallel compositions of BL-Net boxes, as follows:
\[ \mathcal{T}( [\tau_0, \tau_1, \ldots, \tau_m ] ) \]

\[ \Delta \quad \text{box } \mathcal{T}(\tau_0) \rightarrow \{ \mathcal{T}(\tau_0), \mathcal{T}(\tau_0 \cup \tau_1 \cup \cdots \cup \tau_m) \} \]

\[ \| \text{box } \mathcal{T}(\tau_1') \rightarrow \{ \mathcal{T}(\tau_1') \} \]

\[ \| \text{box } \mathcal{T}(\tau_2') \rightarrow \{ \mathcal{T}(\tau_2') \} \]

\[ \| \ldots \]

\[ \| \text{box } \mathcal{T}(\tau_m') \rightarrow \{ \mathcal{T}(\tau_m') \} , \]

where \( \{\tau_1', \tau_2', \ldots, \tau_m'\} = \{\text{root of } \tau_i \mid 1 \leq i \leq m\} \).

The root of a record type can be obtained by removing all fields and tags in the record type. Because the number of distinct roots may be smaller than the number of auxiliary patterns, \( m' \) can be less than \( m \) in the formula. This simplifies the resulting parallel composition, as demonstrated above where \( b_m \parallel b_r \parallel b_r \) is simplified to \( b_m \parallel b_r \).

Another example which demonstrates how binding tags are handled in this translation is given below.

\[ \mathcal{T}( [\{<#a>, m\}, \{<#b>, n\}, \{p\}, \{<#b>, q\} ] ) \]

\[ = \text{box } A \# M \rightarrow \{A\# M, A#MN#PQ\} \]

\[ \| \text{box } B \# \emptyset \rightarrow \{B \# \emptyset\} \]

\[ \| \text{box } \emptyset \# \emptyset \rightarrow \{\emptyset \# \emptyset\} \]

3.2.7 Parallel Replication

A parallel replication is an S-Net component not mentioned in Section 2.2, which has the following syntax:

\[ e! <id>, \]

where \( id \) is a tag name. It creates a parallel composition of indefinitely many replicas of \( e \), demands the tag \( <id> \) to exist in every incoming record, and delivers it to the replica indexed by the value of the tag. In practise, the parallel replication is a useful component to introduce simple load balancing to the program, however, in theory we can simply see this component as just \( e \) with the added requirement of the tag.

To model a parallel replication, we can place a component in front of \( e \) which demands the tag, but does not change the incoming records. The following filter is one example:

\[ [ \{<id> \} \rightarrow \{<id>\} ], \]

which is suitable for records with no binding tags. The operand network \( e \) may accept different combinations of binding tags. To correctly require the tag in the records \( e \) can accept, we use the root alphabet function \( \mathcal{R} \) (Definition 5.8 on page 86), which takes a BL-Net network and returns all distinct roots of the values it accepts, and translate a parallel replication using the following formula:
\( T( e ! <id> ) \)
\[ \triangleq (\text{box } v_1 \rightarrow \{v_1\} \parallel \text{box } v_2 \rightarrow \{v_2\} \parallel \cdots \parallel \text{box } v_m \rightarrow \{v_m\}) \cdot T(e), \]
where \( \{v_1, \ldots, v_m\} = \{v_0 + T(\{<id>\}) \mid v_0 \in \mathbb{R}(T(e))\} \).

The parallel replication also has a deterministic variant:

\( e !! <id>, \)

which again is translated as its non-deterministic version:

\[ T( e !! <id> ) \triangleq T( e ! <id> ). \]

### 3.2.8 Unsigned Networks

An unsigned network is an S-Net network as defined using the keyword \texttt{net} which does not carry a signature. It is merely a shortcut definition so the other networks can refer to the whole topology by a name. To see how it can be translated to BL-Net, we will first look at an unsigned network with a body.

In the body of a network, the programmer can define additional boxes and networks. Each definition is identified by a name that immediately follows the keyword \texttt{box} or \texttt{net}. (The name of a box doubles as the box function name.) Immediately after the definition, the name can be used by other definitions in the body or the enclosing network's topology expression (the expression following the keyword \texttt{connect}) to reference the defined box or network. The following helper function \( T_{id} \) retrieves the name of a box or network definition. For succinctness, we use an ellipsis to denote that the rule matches the box or net definition of any form.

\[ T_{id}( \text{box } id \ldots ; ) \triangleq id \]
\[ T_{id}( \text{net } id \ldots ; ) \triangleq id \]

We represent the network body as a list of box and network definitions. We use \([\ ]\) to match the empty list and define `:' as the list concatenation operation, both of whose operands can be either single definitions or lists of them, so \( D;D' \) matches a non-empty list where \( D \) is assigned the first definition and \( D' \) the sublist after \( D \). To make the name of a definition available for later definitions and the final topology expression, we use the recursive helper function \( T_{let} \) below to translate a list of definitions using the \texttt{let}...\texttt{in} syntax. Each iteration binds the name of a definition to its translation. The \texttt{let}...\texttt{in} syntax is assumed to unfold only at the BL-Net level, so the name defines a variable in the translation of the rest of the network. This answers the question why translating a reference is an identity operation \((T(id) = id)\) in Section 3.2.3. The parenthesis pair enclosing the translation keeps any nested \texttt{let}...\texttt{in} constructs local, and is also to be brought verbatim into the translation result.

\[ T_{let}( D;\overline{D}, n ) \triangleq \texttt{let } T_{id}(D) = \left( T(D) \right) \texttt{ in } T_{let}( \overline{D}, n ) \]
\[ T_{let}( [\ ], n ) \triangleq n \]
Figure 3.2: Translating an unsigned S-Net network (left) to BL-Net (right).

An unsigned network with or without body can now be translated as follows.

\[ T(\text{net id} \{ D \} \text{connect } e; ) \equiv T_{\text{let}}(D, T(e)) \]

\[ T(\text{net id} \text{connect } e; ) \equiv T(e) \]

Figure 3.2 demonstrates how this translation keeps the definitions in scope. In line 4 of the S-Net network, the inner network \( b \) is not yet defined (it is defined only after the semicolon in line 4), so the identifier \( b \) can only refer to the box \( b \). In line 5, the box \( b \) is out of reach: the topology expression can only reference the definitions in the outermost scope of the body, so the identifier \( b \) can only refer to the network \( b \). In the translation result, the two references are resolved correctly.

### 3.2.9 Signed Networks

As introduced in Section 2.2.8, a signed network is a non-top-level network that carries a signature. The signature provides the exported interface of the network, so when it is used in other networks, it is trusted to behave as its signature describes.

The type system establishes said trust by type-checking these signed networks separately. The easiest way to do so is to treat them as standalone programs. When isolating a signed network into a program, the box and network definitions referenced by the topology expression should also be copied. The isolation process is easily done by hand, so a formal definition is omitted. The following S-Net program is the result of the isolation of the signed network upper in the source program, which allows us to type check upper as a standalone program.

```s-net
box mul ( (p,q=) -> (p) );
net upper ( {p,q,<t>} -> {p} )
{
    net inner
        connect [ {<t>} -> {<t=t-1>} ] .. mul;
    }
    connect inner * {<t>} if <t == 0>;
}
```

In S-Net, forward declaration is impossible, so a reference to a box or network definition must appear after the definition. Therefore, when processing an S-Net program in the lexical order, at the definition of a signed network, all referenced boxes and networks will have been available, and the isolation and type checking processes can be done on the spot, before the first reference of said signed network.
Assuming that the type check has been successful, we can model a signed network's behaviour on the consuming side using its network signature. The signature can contain one or more mappings, each of which resembles a box signature: it defines the input record type and zero or more output variants. The network behaviour is expected to be identical to a parallel composition of multiple boxes, whose box signatures match the mappings. The following formula models a signed network with a parallel composition of multiple boxes, where the symbol Σ is the set of all mappings in the network signature.

\[
T( \text{net id} ( \Sigma ) \text{ connect e; } ) \\
\triangleq \ b_1 \parallel b_2 \parallel \cdots \parallel b_m,
\]

where \( \{b_1, \ldots, b_m\} = \{\text{box } T(\tau) \rightarrow T(\tau_s) \mid (\tau \rightarrow \tau_s) \in \Sigma\} \)

Translating a signed network with body is no different on the consuming side; the body is relevant only on the type-checking side.

\[
T( \text{net id} ( \Sigma ) \{ B \} \text{ connect e; } ) \\
\triangleq \ T( \text{net id} ( \Sigma ) \text{ connect e; } )
\]

### 3.2.10 Program

At the outermost scope of an S-Net program, a programmer may include one or more box and network definitions, the last of which must be a network definition and will become the top-level network that defines the whole program. All other outermost-scope definitions are available in the topology expression of the top-level network, as well as the definitions in the outermost scope of its body.

Our purpose of translating a program is to see if it behaves as its signature claims, so we should treat it as an unsigned network. The translation for signed networks is not applicable to the top-level network. We use a new function \( T_p \) to translate an S-Net program. In the formulae below, \( \vec{D}:(\text{net } \ldots) \) matches the whole program as a list of definitions, where \( \vec{D} \) is assigned the sublist before the last, top-level network, and \( T_{let} \), defined in Section 3.2.8 builds the \text{let} \ldots \text{in} constructs to resolve the references that might occur in the translation of the top-level network.

\[
T_p( \vec{D}:( \text{net id} ( \Sigma ) \{ \vec{D'} \} \text{ connect e; } ) ) \\
\triangleq \ T_{let}( \vec{D}, T( \text{net id} \{ \vec{D'} \} \text{ connect e; } ) )
\]

\[
T_p( \vec{D}:( \text{net id} ( \Sigma ) \text{ connect e; } ) ) \\
\triangleq \ T_{let}( \vec{D}, T( \text{net id connect e; } ) )
\]

The readers can now verify that the function \( T_p \) translates the source program into the program \text{program} in Figure 3.1, where the signed network \text{upper} is isolated and translated into \text{upper}.
3.3 Summary

In this chapter we have detailed the methodology set out in Section 1.3 and the rationale behind it: to perform type inference and type check on S-Net programs without having to take care of the S-Net features unrelated to the types. We have briefly introduced the languages reduced from S-Net, namely the projection, BL-Net and L-Net, and their features. They are label set transforming languages, which process one label set per execution. Multiple outputs in S-Net are modelled as separate, non-deterministically chosen executions.

We have also defined an algorithm to translate S-Net programs to BL-Net for type inference and type checking. During the process we have introduced all BL-Net components informally. Some S-Net concepts and components have matching counterparts in BL-Net, while the others must be modelled using other components in BL-Net. The soundness of this translation cannot be proven, because the current S-Net semantics depends on a flawed type system, and cannot be used as the foundation of the proof. Nevertheless, we have strived to make the translation valid and logical.

The translation process serves as a motivation to discuss S-Net's type-related behaviour in terms of BL-Net. After introducing L-Net and BL-Net, in Chapter 6 we will complete the picture with a type system for S-Net. S-Net and BL-Net still share a great resemblance, and in Chapter 7 we will provide an implementation demonstrating a shortcut to assign types to S-Net components directly, bypassing the translation process.
Chapter 4

L-Net

We have previously established the relation between S-Net and L-Net. See Figure 1.3, Section 3.1 and Table 3.1 for a quick review. This chapter covers L-Net, which we use to study a manageable portion of the concepts in BL-Net.

L-Net is a label set transforming language. An L-Net program takes a set of labels as input, and outputs a modified set of labels. The modification adheres to the rules set out by the program topology, which is a network consisting of boxes composed in serial and parallel. The program, as well as every component of it, outputs one and only one label set in response to the input. In case of multiple applicable choices, a decision will be made non-deterministically.

In this chapter, we define the L-Net language formally, including the specification and a type system with the correctness proof. We will also see how this type system helps with efficient implementation of L-Net. Because the ultimate goal of inventing L-Net, as well as other label set transforming languages, is to study the type-related behaviour of S-Net, we will establish the relations between the L-Net concepts and their S-Net counterparts.

4.1 Example L-Net Programs

Example 4.1. A single-box program.

\[ \text{box } a \rightarrow \{ b, cd \} \]

Figure 4.1: Illustration of Example 4.1
Example 4.2. A serial composition of two boxes.

\[
\text{let } b_1 = \text{box } A \rightarrow \{B, CD\} \text{ in} \\
\text{let } b_2 = \text{box } BE \rightarrow \{F\} \text{ in} \\
b_1 \cdot b_2
\]

![Figure 4.2: Illustration of Example 4.2](image)

Example 4.3. A simple parallel composition.

\[
\text{let } b_1 = \text{box } A \rightarrow \{X\} \text{ in} \\
\text{let } b_2 = \text{box } B \rightarrow \{Y\} \text{ in} \\
b_1 \parallel b_2
\]

![Figure 4.3: Illustration of Examples 4.3 and 4.4](image)
Example 4.4. Another simple parallel composition.

\[\text{let } b_1 = \text{box } AB \rightarrow \{x\} \text{ in} \]
\[\text{let } b_2 = \text{box } ABCD \rightarrow \{y\} \text{ in} \]
\[b_1 \parallel b_2 \]

<table>
<thead>
<tr>
<th>Input</th>
<th>Selected Branch</th>
<th>Output</th>
<th>(u)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A)</td>
<td>-</td>
<td>stuck</td>
<td>-</td>
</tr>
<tr>
<td>(AB)</td>
<td>(b_1)</td>
<td>(x)</td>
<td>(AB)</td>
</tr>
<tr>
<td>(ABC)</td>
<td>(b_1)</td>
<td>(CX)</td>
<td>(AB)</td>
</tr>
<tr>
<td>(ABCD)</td>
<td>(b_2)</td>
<td>(Y)</td>
<td>(ABCD)</td>
</tr>
<tr>
<td>(ABCDE)</td>
<td>(b_2)</td>
<td>(EV)</td>
<td>(ABCD)</td>
</tr>
</tbody>
</table>

Example 4.5. Routing complexity.

\[\text{let } b_1 = \text{box } AB \rightarrow \{x\} \text{ in} \]
\[\text{let } b_2 = \text{box } A \rightarrow \{M,N\} \text{ in} \]
\[\text{let } b_3 = \text{box } MBC \rightarrow \{y\} \text{ in} \]
\[\text{let } b_4 = \text{box } N \rightarrow \{z\} \text{ in} \]
\[b_1 \parallel (b_2 \cdot (b_3 \parallel b_4)) \]

![Figure 4.4: Illustration of Example 4.5.](image)

<table>
<thead>
<tr>
<th>(v) (Input)</th>
<th>Selected Branch</th>
<th>(v'')</th>
<th>(v') (Output)</th>
<th>(u)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(ABC)</td>
<td>(b_2 \cdot (b_3 \parallel b_4))</td>
<td>(MBC)</td>
<td>(Y)</td>
<td>(ABC)</td>
</tr>
<tr>
<td>(ABC)</td>
<td>(b_2 \cdot (b_3 \parallel b_4))</td>
<td>(NBC)</td>
<td>(ZBC)</td>
<td>(A)</td>
</tr>
</tbody>
</table>

4.2 L-Net Syntax and Specification

4.2.1 Values

\[v, u, f, d, a \in Value \quad ::= \quad ls\]
\[l \in Label \quad ::= \quad A \mid B \mid C \mid \cdots\]

In L-Net, labels are literal tokens, represented by small capital letters in the examples. A value is a set of labels which can be empty, but for conciseness we represent it by a string of small capital letters, while maintaining the properties of sets, e.g. \(AB = BA = ABA\). In order to differentiate the label sets from other kinds of sets, we use the symbols +, -, \times for the union, difference and intersection operations, respectively, and \(\leq\) for the subset relation, on values. For other kinds of sets, we still use the conventional
symbols $\cup, \setminus, \cap, \subseteq$. Furthermore, to reduce the use of parentheses, we assume that the set operations $+, -, \times, \cup, \setminus, \cap$ are all left associative. The symbol $\times$ still denotes a Cartesian product in syntax and function type declarations. Empty values will be represented by the empty set symbol $\emptyset$.

**Relation with S-Net.** A value in L-Net corresponds to a record type in S-Net. A label in L-Net corresponds to a field or tag in S-Net.

We assume that the number of literal tokens for use as labels is infinite, but the set $\text{Label}$ is always finite, as a collection of all labels the programmer has used in a program. Consequently, all named sets in L-Net, which are based on $\text{Label}$, are finite.

**Variable naming conventions.** The following variable naming conventions apply throughout the thesis. Appending the suffix $\times$ to a variable makes the resulting variable range over the powerset of the original range. For example, the variable $\times$ will be used to denote an element in the set $P(\text{Value})$, or literally, $\times$ is a set of values. One or more prime symbols $'$ and digital subscripts $0, 1, 2, \ldots, 9$ will be added after the main variable and all its suffixes, if any, to distinguish it from other variables of the same range. Letter subscripts $i, j, k$ or $m$ in italics are always integer variables, defined in the context, which resolve the affected symbol to another variable with one or more digital subscripts. An underscore '_' matches with any non-syntactic element (implicit $\exists$).

### 4.2.2 Program

We will skip the syntactic sugar `let ... in` that allows us to name a component and then refer to it by the name (see Example 4.2) and present the syntax of L-Net programs as follows:

$$n \in \text{Net} ::= b \mid n \cdot n \mid n \parallel n$$

$$b \in \text{Box} ::= \text{box } v \rightarrow \times$$

An L-Net program is a hierarchical network, at the bottom of which lie boxes, which are composed through serial composition ($\cdot$) and parallel composition ($\parallel$). The set $\times$ in a box definition is required to be non-empty.

### 4.2.3 Execution

To execute a program, one feeds an input value to it. The judgement

$$n, v \rightsquigarrow s u, f, v'$$

expresses that the network $n$, in response to the input value $v$, produces the output $v'$, using the labels in $u$ from $v$, and *flow-inheriting* the labels in $f$ from $v$ to $v'$. We will define use and flow inheritance with the specification of boxes. An instance of the judgement above denotes a *non-stuck execution*, and the label set $u$ is called the *used value part* and $f$ the *flow-inherited value part*. 

57
The judgement

\[ n, v \not\rightsquigarrow_{S} \]

denotes a stuck execution. In this case there is no used or flow-inherited value parts.

**Relation with S-Net.** Let \( v \) correspond to the S-Net record type \( \tau_1 \), and \( v' \) to \( \tau_2 \). A non-stuck execution \( n, v \rightsquigarrow_{S} u, f, v' \) denotes a class of executions in S-Net wherein the S-Net network corresponding to \( n \) takes an input record \( r \) of type \( \tau_1 \) and produces an output record \( r' \) of type \( \tau_2 \), among possibly multiple output records. Throughout the whole process of creating \( r' \), the labels (fields and tags) in \( u \) that originally existed in \( r \) have been actually used by the unit components in the network, and the labels in \( f \) which exist in both \( r \) and \( r' \) are untouched, i.e. flow-inherited from \( r \) to \( r' \). A stuck execution \( n, v \not\rightsquigarrow_{S} \) expresses the fact that the S-Net network corresponding to \( n \) cannot process an input record of type \( \tau_1 \), because a box or a filter somewhere in the network is given an input record it does not accept.

In this section, we will define both forms of execution per network type.

### 4.2.3.1 Boxes

In a box definition (box \( v_0 \rightarrow v_{S0} \)), \( v_0 \) is the *declared input* and \( v_{S0} \) is the *declared output set*. In Example 4.1, the declared input is \( a \) and the declared output set is \( \{b, cd\} \). The box behaves as follows.

On given an input value \( v \), called the *actual input*, the box removes from \( v \) the labels also existing in \( v_0 \), and adds back the labels in \( v_1 \), a non-deterministically chosen member of \( v_{S0} \), then outputs the modified value. On adding an output label, because the value is treated as a label set, if the same label is found to remain in the value, the output label replaces the remaining one.

Example 4.1 demonstrates 5 cases of execution. Cases 1 and 2, as well as cases 3 and 4, show two different outputs from the same program and input, because the chosen declared output, as seen in the column ‘(Chosen)’, are different. Remember that the choice occurs non-deterministically and is not controlled by the programmer. This means, taking cases 3 and 4 for example, we have

\[
\text{box } a \rightarrow \{b, cd\}, \text{ ACE } \rightsquigarrow_{S} a, ce, bce
\]

and

\[
\text{box } a \rightarrow \{b, cd\}, \text{ ACE } \rightsquigarrow_{S} a, e, cde
\]

simultaneously, due to non-determinism.

The box *uses* the labels which it removes from the actual input. Those labels not used and not replaced by the output labels due to duplication are *flow-inherited* from the input to the output. The opposite of being flow-inherited is being *deleted*; a deleted label is either used or discarded due to duplication with an output label. The remaining columns in the table of Example 4.1 show the used value part \( u \), the flow-inherited value part \( f \), and the *deleted value part* \( d \), corresponding to each execution.
The whole process is formulated as the following rule. The label prefix is denotes ‘L-Net semantics’.

\[
\frac{v_0 \leq v \quad v_1 \in v s_0}{\text{box} \ v_0 \rightarrow v s_0, \ v \rightsquigarrow_v v_0, \ v - v_0 - v_1, \ v - v_0 + v_1} \ (\text{lsBox})
\]

For a box to process an input value, all labels in the declared input must be present in the actual input. The execution is stuck if this requirement cannot be met. For the last case in Example 4.1, the execution is stuck because the label \( a \) in the declared input is not found in the actual input \( b \). This is formulated in Section 4.2.3.4 together with the stuck executions for other L-Net constructs.

**Relation with the S-Net box.** The syntax part ‘\( v_0 \rightarrow v s_0 \)’ of a BL-Net box corresponds to the box signature in S-Net. The corresponding S-Net box removes the labels in \( v_0 \) from an input record, as part of the process to retrieve data stored under them to invoke the box function. The labels in \( v_1 \), one of the output variants of the S-Net box, may contain labels present in the remaining part of the input record. If so, when the box function outputs a record of this variant, the data stored under any label \( l \in v_1 \) in the input record will be replaced by the box function output, so the actual flow-inherited labels exclude those in \( v_1 \).

### 4.2.3.2 Serial Composition

In a serial composition, the two operand networks are called the *left operand* and the *right operand*, respectively. A serial composition lets the left operand process the input value, and delivers the intermediate output to the right operand for processing.

The serial composition uses the labels in the input value which are either used by the left operand, or flow-inherited through the left operand and subsequently used by the right. This implies that the labels produced by the left operand and then used by the right operand do not constitute to the overall used value part. The labels flow-inherited through both operands are flow-inherited through the serial composition.

The semantics of serial composition is defined below.

\[
\frac{n_1, v \rightsquigarrow_v u_1, f_1, v_1 \quad n_2, v_1 \rightsquigarrow_v u_2, f_2, v_2}{n_1 \cdot n_2, v \rightsquigarrow_v u_1 + (f_1 \times u_2), f_1 \times f_2, v_2} \ (\text{lsSer})
\]

Example 4.2 shows a serial composition of two boxes and lists 6 cases of execution. The choices of the output by box \( b_1 \) are not shown but can be easily deduced from \( v_1 \). The table also shows the used and flow-inherited value parts for the execution of individual boxes as well as for the composition.

The execution of a serial composition is stuck, if the input value is stuck inside the left operand, or any intermediate value from the left operand is stuck inside the right operand. The example demonstrates three cases of stuck executions. Note that Cases 2 is a non-stuck execution, while Case 3, using the same
input, is stuck. This indicates that the two forms of execution can coexist due to non-determinism:

\[(b_1 \cdots b_2, \text{AE } \rightarrow_s \text{AE}, \emptyset, f) \land (b_1 \cdots b_2, \text{AE } \not\rightarrow_s).

Relation with the S-Net serial composition. The L-Net serial composition is a straightforward copy of the S-Net version.

4.2.3.3 Parallel Composition

The following definition of \textit{acceptance} is required to describe the specification of a parallel composition. A network \(n\) accepts a value \(v\), iff \(v\) can \textit{never} be stuck in \(n\):

\textbf{Definition 4.6.} A network \(n\) accepts a value \(v\), denoted as \(n \triangleright v\), iff \(\neg(n, v \not\rightarrow_s)\).

For instance, the serial composition in Example 4.2 accepts \(\text{abe}\), but does not accept \(\text{ae}\).

In a parallel composition, the two operand networks are called the \textit{branches} of the composition. A parallel composition lets either branch process the input value, whichever is \textit{more attractive} than the other branch. A branch \(n_j\) is more attractive to an input value than the other branch \(n_k\), if \(n_j\) accepts the value, and the value is either not accepted by \(n_k\), or eligible for a non-stuck execution of \(n_j\) which uses more labels than any eligible non-stuck executions of \(n_k\), or equivalently, the largest used value part in \(n_j\) is larger than that in \(n_k\). In case the two branches are equally attractive, a branch is selected non-deterministically. Execution is stuck if neither branch accepts the input value.

Examples 4.3 and 4.4 provide a preview of the specification. The figure shows that the two branches are placed side by side, and a diamond shape carries out the branch selection process, which splits the input arrow to two. Also, the outputs from both branches share the same output arrow. In Example 4.3, the first two cases of executions demonstrate when only one branch accepts the input, and the next two cases show the non-deterministic selection of branches, when the largest used value parts of both branches have the same size. In Example 4.4, however, we see that the branch \(b_2\) is selected as long as the input value is accepted, due to the larger used value part.

Note that the branch selecting process does not determine the execution; i.e. the execution in the selected branch \(n_j\) which uses the labels in \(u_j\) does not necessarily become the final execution. It can be any execution of the selected branch that defines the execution of the parallel composition. Example 4.5 demonstrates this issue. With regard to the input \(\text{abc}\), the upper branch will use two labels \(\text{ab}\), whereas the lower branch may use all three, through the route \(b_2\) (outputting \(\text{mbc}\)) \(\rightarrow b_3\), and therefore the lower branch is selected. However, it is uncertain that \(b_2\) will produce \(\text{mbc}\), and if it emits \(\text{nbc}\) instead, the overall used value part will be \(\text{a}\), even smaller than what the upper branch could offer.

The process is formulated as follows:
Relation with the S-Net parallel composition. The L-Net parallel composition is almost a straightforward copy of the S-Net version, except that the semantics in L-Net does not refer to any type inference results. The semantics in S-Net indicates that an input record is sent to the branch ‘whose type signature’s input type is best matched by the record’s type’ [25, p19]. We shall understand its purpose as to try to make the most use of the labels available in the record, hence the search of the largest used value part in the L-Net semantics. We also guard against the potential stuck executions in the branch with the largest used value part, by requiring that a branch do not attract the values it does not accept.

4.2.3.4 Stuck Executions

Stuck executions for individual constructs have been mentioned informally in their corresponding sections. We now formulate them in one definition.

\[
\begin{align*}
n, v \not\Rightarrow_s  \iff & \begin{cases} \text{n = box } v_0 \Rightarrow \_ & \land \ v_0 \not\leq v; \text{ or} \\
n = n_1 \cdot n_2 & \land \ (n_1, v \not\Rightarrow_s \lor \\
& (\exists v_1, n_1, v \Rightarrow_s \_\_\_v_1 \land n_2, v \not\Rightarrow_s)); \text{ or} \\
n = n_1 \parallel n_2 & \land \ n_1, v \not\Rightarrow_s \land n_2, v \not\Rightarrow_s. \end{cases} 
\end{align*}
\]

4.2.3.5 Properties of Execution

We present the following properties of execution as lemmas which will aid the proofs of the type system properties discussed later.

**Lemma 4.7.** A box accepts a value if it contains the whole of the box’s declared input; a serial composition accepts a value if the left operand accepts it, and all intermediate values the left operand produces are accepted by the right operand; a parallel composition accepts a value if either branch does. Formally, \( n \ll v \), iff:

\[
\begin{align*}
n = \text{box } v_0 \Rightarrow \_ & \land \ v_0 \leq v; \text{ or} \\
n = n_1 \cdot n_2 & \land \ n_1 \ll v \land \\
& \forall v_1, n_1, v \Rightarrow_s \_\_\_v_1 \implies n_2 \ll v_1; \text{ or} \\
n = n_1 \parallel n_2 & \land \ (n_1 \ll v \lor n_2 \ll v).
\end{align*}
\]
Proof. By Definition 4.6 and negating lsStuck.

Lemma 4.8. If a network \( n \) accepts a value \( v \), then there is at least one non-stuck execution with regard to \( n \) and \( v \):

\[ n \bowtie v \implies n, v \rightarrow s, s, s, s. \]

Proof. By structural induction on \( n \). In all cases, \( n \bowtie v \) is rewritten as more useful terms by Lemma 4.7, fulfilling the premises of the non-stuck execution rules lsBox, lsSer and lsPar. See Section A.1 on page 128 for a detailed proof.

Because acceptance is the negation of the stuck execution, another meaning of Lemma 4.8 is that there is at least one kind of execution available per network per value:

\[ n, v \not\rightarrow s \lor n, v \rightarrow s, s, s, s. \]

This means that the specification of L-Net is complete.

Lemma 4.9. For any non-stuck execution \( n, v \rightarrow s u, f, v' \), the following holds:

- \( u \leq v \);
- \( f \leq v \land f \leq v' \);
- \( u \times f = \emptyset \).

Proof. By structural induction on \( n \). The three conditions above mean that the used labels and the flow-inherited labels all come from the input and are disjoint, and that the flow-inherited labels all appear in the output. The box rule lsBox provides these guarantees at the base case. The serial composition rule lsSer preserves them, although showing this requires some set-based computations. The parallel composition uses the full non-stuck execution from one of the branches, so these guarantees are naturally preserved. The full proof is in Section A.2 on page 129.

4.3 Type System

The purpose of the L-Net type system is to detect and guard against stuck executions for a given L-Net program. This corresponds to the first purpose of the S-Net type system, mentioned in Section 1.1 on page 15. However, executing an L-Net program requires an input value, which is not available at compile time. The purpose can still be fulfilled, if there is an algorithm to find all values the program accepts, so that an input can be quickly tested for safety when it becomes available. We call the set of these values the domain of the program.

Definition 4.10. The domain of a network, denoted as \( \text{dom}(n) \), is the set of all values it accepts:

\[ \text{dom}(n) = \{ v \mid n \bowtie v \}. \]
Because a program is itself a network, the definition above can also be used to retrieve the domain of a program.

The program domain is the complement of the set of all values that can cause stuck executions. From the rule `lsStuck` we can see that, to detect stuck executions for a serial composition, it is a requirement to predict the output values of its left operand. This in turn means that the task can be done only with a mechanism to capture the full behaviour of every network.

In this section, we introduce a set-based type for the networks, which is based on the concept to represent networks by functions.

### 4.3.1 Functional Representation of Networks

Consider a network $n$ and a partial function $F : \text{Value} \rightarrow \mathcal{P}(\text{Value} \times \text{Value} \times \text{Value})$. If the following both hold:

$$\text{dom}(F) = \text{dom}(n),$$

$$\forall v, v \in \text{dom}(n) \implies F(v) = \{(u, f, v') \mid n, v \Rightarrow_u u, f, v'\},$$

then we say that $F$ represents $n$.

For example, one can easily show that the following partial function represents the box in Example 4.1 on page 54:

$$F(v) = \{(a, v - a - b, v + a + b), (a, v - a - cd, v + a + cd)\}, \quad \text{if } a \leq v.$$

We choose to work with such partial functions in the type system, because it is more beneficial than working with the specifications directly. The most important benefit is that we can now discuss the network behaviours without the need to know the topology. It is agreeable that two networks, regardless of their topologies, are equivalent if they have the same domain and behave identically, i.e. produce the same results for each value in the domain. Note though that the two networks can behave differently in response to input values outside their domains; for example, all executions are stuck for one network while for the other there are a couple of non-stuck executions. However, for the purpose of the type system, it is sufficient to know that the concerned inputs are outside their domains, i.e. can cause stuck executions. In other words, the information encapsulated in a representing function is sufficient for the purpose of the type system.

Another benefit lies in the fact that one such function produces an aggregated output for each input, grouping together all non-deterministic responses. In this way, as soon as one output value leads to a stuck execution, we can directly declare the concerned input unfeasible, wasting less time to go through the rest of the options.

To compute the function that represents a program, the type system works bottom-up. Firstly the functions representing the boxes are constructed relatively straightforwardly, as seen above. Then, level by level, the functions representing the compositions are constructed using the results for the boxes and the subnetworks at lower levels. Once the whole program is given a function, the domain of that function
can be used as the domain of the program.

### 4.3.2 Reps

We adopt the following set structure $Rep$ which helps describe the functions. This allows us to construct the functions using set operations, which simplifies the formulation of the algorithm.

$$
\begin{align*}
\rho & \in Rep = \mathcal{P}(Case) \\
\kappa & \in Case = Subdom \times \mathcal{P}(Out) \\
\sigma & \in Subdom = \mathcal{P}(Value) \\
\omega & \in Out = Value \times Value \times Value
\end{align*}
$$

For simplicity, we call a $Rep$ instance a rep. A rep is divided into several cases, each having its own subdomain not overlapping with those for other cases. Each case carries a set of output choices represented by instances of $Out$. An output choice is a triple of values, but instead of $(u, f, v')$ as found in the output of the representation functions, it is in fact $(u, d, a)$: the three elements in an output choice are the used, deleted and added value parts, respectively. The different elements in the two triples have the following relations:

$$
\begin{align*}
f &= v - d, \\
v' &= f + a = v - d + a,
\end{align*}
$$

where $v$ is the input value. The deleted value part has been briefly introduced in Section 4.2.3.1 and the added value part is the difference between the output value and the flow-inherited value part, which according to Lemma 4.9 is a subset of the output value.

The reason of using the $(u, d, a)$ triple is that we can encapsulate many more executions in one case than using $(u, f, v')$, due to that $f$ and $v'$ depend on $v$ as seen above. Take Example 4.1 for instance. If the $(u, f, v')$ triple is used, we need two cases to encapsulate the 4 non-stuck executions in the example (labels are given for reading convenience):

$$(\sigma: \{\lambda\}, \omega: ((u:A, f:\emptyset, v':b), (u:A, f:\emptyset, v':cd))),$$

$$(\sigma: \{ace\}, \omega: ((u:A, f:ce, v':bce), (u:A, f:e, v':cde))).$$

However, using $(u, d, a)$, the case

$$(\sigma: \{v \mid a \leq v\}, \omega: ((u:A, d:ab, a:b), (u:A, d:acd, a:cd)))$$

will not only encapsulate all 4 non-stuck executions in the example, but also all non-stuck executions not listed. Therefore, only one case is needed for the rep that represents the box in Example 4.1

$$\left\{ (\{v \mid a \leq v\}, \{a, ab, b\}, a:acd, c:cd) \right\}.$$

One may state now, for example, that the expression for the output value $v'$ reconstructed from the first output choice above, $(a, ab, b)$, which is $v - ab + b$, has an extra label $b$ in the term $-ab$ compared
with the original function in Section 4.3.1 where it was \( v - a + b \). The argument is as follows. Firstly, the two expressions are equivalent; secondly, the extra term conveys the operational information that the label \( b \), if found in the input, will be deleted, because it will be replaced by the same label which is the actual product of the box execution. The deleted value part is a superset of both the used value part and the added value part, and may contain additional labels. For instance, the 4th execution listed in Example 4.2 on page 55 is a result of the following triple:

\[(uae, d:abef, a:t)\]

where the label \( b \), not found in \( u \) or \( a \), is discarded by the left operand due to duplication with the intermediate output.

To make the expressions more concise, we define the following extractor functions as suffix operators:

\[
\begin{align*}
.s &: \text{Case} \rightarrow \text{Subdom}, \quad \kappa.s \triangleq \kappa \downarrow_1; \\
o &: \text{Case} \rightarrow \mathcal{P}(\text{Out}), \quad \kappa.o \triangleq \kappa \downarrow_2; \\
u &: \text{Out} \rightarrow \text{Value}, \quad \omega.u \triangleq \omega \downarrow_1; \\
d &: \text{Out} \rightarrow \text{Value}, \quad \omega.d \triangleq \omega \downarrow_2; \\
a &: \text{Out} \rightarrow \text{Value}, \quad \omega.a \triangleq \omega \downarrow_3;
\end{align*}
\]

The wellformedness of a rep is defined as follows.

**Definition 4.11.** A rep is well formed iff:

- the subdomains of its cases do not overlap, and
- for each of its output choices, the deleted value part contains all labels of the added value part.

Formally, \( \rho \) iff

\[
(\forall \kappa_1, \kappa_2. \kappa_1, \kappa_2 \in \rho \land \kappa_1 \neq \kappa_2 \implies \kappa_1.s \cap \kappa_2.s = \emptyset) \\
\land (\forall \kappa, \omega. \kappa \in \rho \land \omega \in \kappa.o \implies \omega.a \leq \omega.d).
\]

Note that besides the two constraints above, the type inference algorithm discussed later will also guarantee the following, when constructing the reps:

- all cases have non-empty subdomains, and
- for each output choice, the deleted value part also contains all labels of the used value part.

However, for proving the desired properties of the type system, we only need the constraints listed in the definition.

To use a well formed rep as a function of type \( \text{Value} \rightarrow \mathcal{P}(\text{Value} \times \text{Value} \times \text{Value}) \), we define the
domain and the application operators for reps as follows:

\[
\text{dom} : \text{Rep} \rightarrow \mathcal{P}(\text{Value}), \\
\text{dom}(\rho) \triangleq \bigcup_{\kappa \in \rho} \kappa.s; \\
'(\cdot) : \text{Rep} \rightarrow \text{Value} \rightarrow \mathcal{P}(\text{Value} \times \text{Value} \times \text{Value}), \\
v \in \text{dom}(\rho) \implies \\
\rho(v) \triangleq \{(\omega.u, v - \omega.d, v - \omega.d + \omega.a) \\
| \kappa \in \rho \land v \in \kappa.s \land \omega \in \kappa.o\}.
\]

Note the formula in the application operator definition, where the outputs are produced from as many cases as there are whose subdomains include the input value. However, for a well-formed rep, an input value will only match at most one case. The formula above is thus constructed in order to simplify the proofs.

An empty rep represents a network not accepting any values, because its domain is empty by definition. To differentiate an empty rep from other empty sets, we introduce a symbol of a different shape, \( \emptyset \in \text{Rep} \), as opposed to \( \emptyset \).

### 4.3.3 Network Types in L-Net and S-Net

A rep can represent all networks exhibiting the behaviour the rep encapsulates, and therefore it is reasonable to use the rep as the type of all these networks. Note that we do not actually need to assign types to networks, because they are not first-class in L-Net. However, we do need the rep in order to compute the output value types when the input value type is known. The notion of network type is therefore a convenient option. Put differently, the network type can be seen as a type scheme from which to deduce the output types of a network in response to a given input type.

Note that a rep encapsulates one behaviour, so all networks typed as the same rep are in fact equivalent at the L-Net level. At the S-Net level, however, the S-Net networks whose corresponding L-Net networks are typed as the same rep can exhibit many different behaviours. Consider the following extremely restrictive rep:

\[
\left\{ \begin{array}{c}
(\sigma : \{A\}, \ \omega s : \{(A, AB, B)\}), \\
(\sigma : \{M\}, \ \omega s : \{(M, M, M), (M, MN, N)\})
\end{array} \right\}
\]

which if used as a function contains only two mappings:

\[
\begin{align*}
A & \mapsto \{B\}, \\
M & \mapsto \{M, MN\}.
\end{align*}
\]
This expands to a complex S-Net type as follows:

\[
(a) \rightarrow \text{seq}(b))
\land (\{m\} \rightarrow \text{seq}(\{m\} \lor \{m,n\}))
\]

where \{a\}, \{m,n\} etc. are record types, ‘seq’ creates a sequence (stream) type, \land introduces an intersection type and \lor constructs a union type. This network type describes that the network takes a record of type \{a\} and responds with a stream of records of type \{b\}, and also accepts a record of type \{m\} and outputs a stream of records, each of type either \{m\} or \{m,n\}.

In general, a rep \(\rho\) expands to the S-Net network type

\[
\bigwedge_{\tau \in \text{dom}(\rho)} (\tau \rightarrow \text{seq}(\bigvee_{\tau' \in \rho(\tau)} \tau')),
\]

where we assume that the L-Net values can be used directly as S-Net record types, for convenience. We are able to capture this complex type in a relatively simple structure, thanks to the reduction process.

### 4.3.4 Type Inference

The function \(\mathcal{R} : \text{Net} \rightarrow \text{Rep}\) constructs a rep for every network. This process is called type inference. It is foreseeable at this point that the definition of \(\mathcal{R}\) will be rather lengthy, so we split it up into three cases, each serving a different form of network, and will define the subfunctions \(\mathcal{R}^b, \mathcal{R}^s\) and \(\mathcal{R}^p\), for boxes, serial compositions and parallel compositions, respectively, separately in subsections.

Following the discussion in Section [4.3.1](#431) that the topology is insignificant to the type system, we will decouple the algorithm from the topology as early as possible. In the definition below, we feed the subfunctions \(\mathcal{R}^s\) and \(\mathcal{R}^p\) with the reps for the composition operands instead of the operands themselves.

**Definition 4.12.** The type inference algorithm \(\mathcal{R}\) is defined as follows.

\[
\mathcal{R} : \text{Net} \rightarrow \text{Rep},
\]

\[
\mathcal{R}(n) \triangleq \begin{cases} 
\mathcal{R}^b(b), & \text{if } n = b; \\
\mathcal{R}^s(\mathcal{R}(n_1), \mathcal{R}(n_2)), & \text{if } n = n_1 \cdot n_2; \\
\mathcal{R}^p(\mathcal{R}(n_1), \mathcal{R}(n_2)), & \text{if } n = n_1 \parallel n_2.
\end{cases}
\]

The sub-algorithms will be defined in their own sections.

### 4.3.4.1 Boxes

The rep for any box has only one case and is easily defined by mirroring \(\text{lsBox}\) as follows.

**Definition 4.13.** (Type inference sub-algorithm for a box.)

\[
\mathcal{R}^b : \text{Box} \rightarrow \text{Rep},
\]

\[
\mathcal{R}^b(\text{box } v_0 \rightarrow v_{S_0}) \triangleq \left\{ \left\{ v \mid v_0 \leq v \right\}, \left\{ (v_0, v_0 + v_1, v_1) \mid v_1 \in v_{S_0} \right\} \right\}.
\]
4.3.4.2 Serial Composition

Treating its two arguments—which we will call $\rho_1$ and $\rho_2$—as functions, the function $R^s : Rep \times Rep \rightarrow Rep$ for serial compositions is similar to a function composition: the outputs from $\rho_1$ are taken as the inputs to $\rho_2$. We will build the definition bit by bit, starting with the finest case.

**Base Case.** If $\rho_1 = ((\sigma, \{\omega\}))$ and $\rho_2 = \{\kappa_2\}$, i.e. informally, both $\rho_1$ and $\rho_2$ are single-case reps, and the case in $\rho_1$ has a single output choice, then all outputs from $\rho_1$ will be described by $\omega$, and subsequently fed into $\kappa_2$. According to l1.accept, for an input value to be accepted by the serial composition, any intermediate output values from the left operand must be accepted by the right operand. The output values from $\rho_1$ must fall within $\kappa_2$. We know that an output value from $\rho_1$ can be computed by

$$v - \omega.d + \omega.a,$$

given $v \in \sigma$. This leads to a new domain suitable for the returned rep:

$$\sigma' = \{v \mid v \in \sigma \land (v - \omega.d + \omega.a) \in \kappa_2.s\}.$$

If $\sigma'$ turns out to be empty, no values will be accepted by the serial composition. We can then simply declare that the function $R^s$ returns $\emptyset$. Otherwise, we move on to computing the used, discarded, and added value parts for each combined output.

For each output choice $\omega_2 \in \kappa_2.o$, the labels in $\omega_2.u$ are either provided by the generated labels $\omega.a$, or otherwise flow-inherited from the input value. According to s1sSer, the latter should be included in the used value part of the combined output. As a result, the used value part of the combined output is

$$u' = \omega.u + (\omega_2.u - \omega.a).$$

The combined deleted value part contains the labels deleted by either $\omega$ or $\omega_2$:

$$d' = \omega.d + \omega_2.d.$$

Finally, the combined added value part contains the labels added by $\omega$ which have survived through $\omega_2$, as well as the labels added by $\omega_2$:

$$a' = \omega.a - \omega_2.d + \omega_2.a.$$

This covers the case where $\rho_1$ is single-case and single-output, and $\rho_2$ is single-case. For reusability, we sum up the information in a helper function $R^s_3$.

**Definition 4.14.** (Type inference sub-algorithm for the part of a serial composition involving a subdomain-
output choice pair \((\sigma, \omega)\) from the left operand and a case \((\kappa_2)\) from the right operand.)

\[
R_3^s : \text{Subdom} \times \text{Out} \times \text{Case} \rightarrow \text{Rep},
\]

\[
R_3^s(\sigma, \omega, \kappa_2) \triangleq \text{let } \sigma' = \{ v \in \sigma \mid (v - \omega.d + \omega.a) \in \kappa_2.s \} \text { in }
\begin{cases}
\emptyset, & \text{if } \sigma' = \emptyset, \\
\{(\sigma', \{(\omega.u + (\omega_2.u - \omega.a), \omega.d + \omega_2.d, \\
\quad \omega.a - \omega_2.d + \omega_2.a) \mid \omega_2 \in \kappa_2.o\})\}, & \text{otherwise}.
\end{cases}
\]

**Multiple Cases in** \(\rho_2\). If \(\rho_2\) has multiple cases, then an output value from the single-case single-output \(\rho_1\) can be delivered to the accepting case in \(\rho_2\), if any. To obtain the overall result, we simply need to try out every case in \(\rho_2\) using \(R_3^s\), letting the latter detect the non-accepting cases and return \(\emptyset\) for them. This process is formulated in the following helper function.

**Definition 4.15.** (Type inference sub-algorithm for the part of a serial composition involving a subdomain-output choice pair \((\sigma, \omega)\) from the left operand and the whole of the right operand \((\rho_2)\).)

\[
R_2^s : \text{Subdom} \times \text{Out} \times \text{Rep} \rightarrow \text{Rep},
\]

\[
R_2^s(\sigma, \omega, \rho_2) \triangleq \bigcup_{\kappa_2 \in \rho_2} R_3^s(\sigma, \omega, \kappa_2).
\]

**Empty** \(\rho_2\). We define that the big union operation on an nonexistent range results in an empty set. Therefore, if \(\rho_2 = \emptyset\), the rep returned from the function above will be \(\emptyset\), reflecting the fact that the serial composition cannot accept any value.

**Multiple Output Choices in** \(\rho_1\). When the single-case \(\rho_1\) provides multiple output choices, an accepted input value will non-deterministically generate different output values, carrying different labels. We can use \(R_2^s\) for each output choice, but the returned reps, one per output choice, may impose different extra requirements on the input values. Due to the non-determinism in choosing the output choice from \(\rho_1\), to guarantee acceptance, an input value must then satisfy all of them. The challenge here is that these requirements are represented by the domains of the reps returned by \(R_3^s\), which may be partitioned into subdomains in a number of cases.

As an example, consider the case when the single case in \(\rho_1\) carries two output choices, \(\omega_1\) and \(\omega_2\), and the function \(R_3^s\) produces a two-case rep for each output choice. Assume that the two subdomains in the rep produced from \(\omega_1\) are \(\sigma_{11}\) and \(\sigma_{12}\), and those from \(\omega_2\) are \(\sigma_{21}\) and \(\sigma_{22}\). Then, if \(\omega_1\) is chosen, the input value must reside in \((\sigma_{11} \cup \sigma_{12})\), and if \(\omega_2\), \((\sigma_{21} \cup \sigma_{22})\). To guarantee acceptance, the intersection of them is the overall domain, which expands to four smaller intersections as follows:

\[
\sigma_{11} \cap \sigma_{21} \cup \sigma_{12} \cap \sigma_{21} \cup \sigma_{11} \cap \sigma_{22} \cup \sigma_{12} \cap \sigma_{22},
\]

each of which can be tested for emptiness individually. Then, the non-empty intersections become the subdomains of the resulting rep.
The algorithm captured in the helper function $R^s_1$ below uses the same technique, going through all intersections of the subdomains from the rep returned by $R^s_2$, one per output choice.

**Definition 4.16.** (Type inference sub-algorithm for the part of a serial composition involving a case $(\kappa)$ from the left operand and the whole of the right operand $(\rho_2)$.)

$$R^s_1 : \text{Case} \times \text{Rep} \rightarrow \text{Rep},$$

$$R^s_1(\kappa, \rho_2) \triangleq \text{let } \{\omega_i \mid i \in 1..|\kappa.o|\} = \kappa.o \text{ in}$$

$$\left\{ (\sigma', \bigcup_{i=1}^{\kappa.o} k'_i.o) \mid (\forall i \in 1..|\kappa.o| \implies k'_i \in R^s_2(\kappa.s, \omega_i, \rho_2)) \right\}$$

$$\land \sigma' = \bigcap_{i=1}^{\kappa.o} k'_i.s \land \sigma' \neq \emptyset \right\}.$$

In the formula above, the let-expression $\{\omega_i \mid i \in 1..|\kappa.o|\} = \kappa.o$ assigns an arbitrary indexing to the output choices in $\kappa.o$. The variables $k'_i$ are free in the set comprehension formula (not bound by the $\forall$-quantifier), and each of them can be one of the cases in the result of $R^s_2(\kappa.s, \omega_i, \rho_2)$, as if the corresponding output choice $\omega_i$ were chosen. This causes the resulting rep to enumerate through all combinations of the $k'_i$ values to find the correct subdomains as per the discussion prior to this definition.

**Multiple Cases in $\rho_1$.** When $\rho_1$ is multi-case, an input value is eligible for the case in $\rho_1$ whose subdomain includes it. Having chosen the valid case in $\rho_1$, the rest of the process is described by $R^s_1$.

This is very similar with the case where $\rho_2$ is multi-case.

**Definition 4.17.** (Type inference sub-algorithm for a serial composition.)

$$R^s : \text{Rep} \times \text{Rep} \rightarrow \text{Rep},$$

$$R^s(\rho_1, \rho_2) \triangleq \bigcup_{\kappa \in \rho_1} R^s_1(\kappa, \rho_2).$$

Note that the formula above can also cope with the case where $\rho_1 = \emptyset$. This completes the definition of $R^s$ for serial compositions.

### 4.3.4.3 Parallel Compositions

The rep for a parallel composition is almost as straightforward as the union of the two argument reps to the function $R^p$. However, for the overlapping parts of the two domains, we shall conform to the rule of branch selection as set out in s.l.sPar: the branch or branches with the largest used value part will be selected. This selection can be easily done because the used value parts are readily available in the reps.

Firstly, for each possible assignment of $j$ and $k$ satisfying $j, k \in \{1, 2\} \land j \neq k$, and every case $\kappa_j$ in $\rho_j$, we shrink its subdomain so that for any case $\kappa_k$ in $\rho_k$, either the subdomain of $\kappa_k$ does not overlap with that of $\kappa_j$, or there is an output choice in $\kappa_j$ with a used value part strictly larger than all in $\kappa_k$ — when the largest used value parts are equal in size, we would like to use a separate case to include the outputs from both cases. The process is formulated as follows, which also combines the results originating from
both branches into one set:

\[ \rho' = \left\{ (\sigma, \kappa_j, o) \mid j, k \in \{1, 2\} \land j \neq k \land \kappa_j \in \rho_j \land \sigma = \{ v \mid v \in \kappa_j, s \land (\forall k'. \kappa' \in \rho_k \land v \in \kappa'. s) \implies \exists w_j, w_j \in \kappa_j, o \land \forall \omega', \omega' \in \kappa', o \implies |w_j, u| > |\omega', u| \} \right\} \]

\[ \wedge \sigma \neq \emptyset \right\}. \]

The result above can handle every input value eligible for just one branch. We now add the cases where both branches can be chosen non-deterministically, by combining the cases from the two argument reps where the largest used value parts are equal in size:

\[ \rho'' = \left\{ (\kappa_1.s \cap \kappa_2.s, \kappa_1.o \cup \kappa_2.o) \mid \kappa_1 \in \rho_1 \land \kappa_2 \in \rho_2 \land \kappa_1.s \cap \kappa_2.s \neq \emptyset \land \exists \omega_1, \omega_2, \omega_1 \in \kappa_1.o \land \omega_2 \in \kappa_2.o \implies |\omega_1, u| = |\omega_2, u| \land \forall \omega, \omega \in \kappa_1.o \cup \kappa_2.o \implies |\omega, u| \geq |\omega, u| \right\}. \]

Then, \( \rho' \cup \rho'' \) is the result we need. The full definition is presented below.

**Definition 4.18.** (Type inference sub-algorithm for a parallel composition.)

\[ \mathcal{R}^p : \text{Rep} \times \text{Rep} \rightarrow \text{Rep}, \]

\[ \mathcal{R}^p(\rho_1, \rho_2) = \rho' \cup \rho'', \quad \text{where} \]

\[ \rho' = \left\{ (\sigma, \kappa_j, o) \mid j, k \in \{1, 2\} \land j \neq k \land \kappa_j \in \rho_j \land \sigma = \{ v \mid v \in \kappa_j, s \land (\forall k'. \kappa' \in \rho_k \land v \in \kappa'. s) \implies \exists w_j, w_j \in \kappa_j, o \land \forall \omega', \omega' \in \kappa', o \implies |w_j, u| > |\omega', u| \} \right\} \]

\[ \wedge \sigma \neq \emptyset \right\}. \]

\[ \rho'' = \left\{ (\kappa_1.s \cap \kappa_2.s, \kappa_1.o \cup \kappa_2.o) \mid \kappa_1 \in \rho_1 \land \kappa_2 \in \rho_2 \land \kappa_1.s \cap \kappa_2.s \neq \emptyset \land \exists \omega_1, \omega_2, \omega_1 \in \kappa_1.o \land \omega_2 \in \kappa_2.o \implies |\omega_1, u| = |\omega_2, u| \land \forall \omega, \omega \in \kappa_1.o \cup \kappa_2.o \implies |\omega, u| \geq |\omega, u| \right\}. \]

### 4.3.5 Soundness and Completeness

To discuss the soundness and completeness of the L-Net type system, we must first present the properties of the algorithm \( \mathcal{R} \) as follows.

**Theorem 4.19.** The algorithm \( \mathcal{R} \) creates only well formed reps. That is, for any network \( n \), \( \mathcal{R}(n) \) is well formed.

**Proof.** Rep wellformedness (Definition 4.11) includes (1) non-overlapping subdomains, and (2) inclusion of the added value part in the deleted value part. The proof is split into two subproofs for these two
propositions. For (1), \( R^b \) creates a single-case rep, and \( R^s \) only reduces existing subdomains, so the focus is on \( R^p \). For (2), \( R^b \) guarantees the property at the base level, and \( R^p \) simply reuses the outputs from the operand reps, so the focus is on \( R^s \). The detailed proof is in A.3 on page 131.

**Theorem 4.20.** For any network \( n \), \( R(n) \), when used as a function, represents \( n \):

- \( \text{dom}(n) = \text{dom}(R(n)) \);
- \( \forall v. v \in \text{dom}(n) \implies n, v \rightarrow s u, f, v' \iff (u, f, v) \in (R(n))(v) \).

**Proof.** The discussions in this section accompanying the introducing of the algorithm \( R \) should suffice as an informal proof. See A.4 on page 136 for a formal proof.

At the beginning of Section 4.3 we mentioned that the reps are used as the network types, so that when an input value is known, we can quickly deduce whether it is type safe. Intuitively, we deduce that an input value \( v \) is type safe for a network \( n \) if \( v \in \text{dom}(R(n)) \).

The definition of soundness of the L-Net type system can now be clarified: the type system is sound iff any value \( v \) deducible as type safe for any network \( n \) is indeed accepted by the network \( n \). Dually, the type system is complete iff any value \( v \) accepted by any network \( n \) is deducible as type safe for \( n \). Note that the definition of acceptance (Definition 4.6) requires that the L-Net semantics can never deduce a stuck execution, which is taking into account all non-deterministically chosen executions.

It follows intuitively from Theorem 4.20 that the L-Net type system is sound and complete. We formalise this in the two theorems below.

**Theorem 4.21.** The type system for L-Net is sound. That is, given any network \( n \) and an input value \( v \), if \( v \in \text{dom}(R(n)) \), then \( n \triangledown v \).

**Proof.** By Theorem 4.20.

**Theorem 4.22.** The type system for L-Net is complete. That is, given any network \( n \) and an input value \( v \), if \( n \triangledown v \), then \( v \in \text{dom}(R(n)) \).

**Proof.** By Theorem 4.20.

Observe that Theorem 4.20 is stronger than the combined effect of soundness and completeness, with the additional information that the network rep correctly predicts all possible output values and only those. The theorem is designed as such to be able to prove \( \text{dom}(n) = \text{dom}(R(n)) \), which in turn implies the soundness and completeness properties. As a side note, Theorem 4.20 contributes to the strong soundness [51] of the L-Net system.

### 4.3.6 Finiteness and Locality

The set-based type \( Rep \) for networks is implicit in the program specification. It is a finite set, because it is ultimately based on the finite set \( Label \). Because the type inference algorithm only enumerates the sets non-recursively, the finiteness of \( Rep \) guarantees that the type inference algorithm can terminate.
An issue of the algorithm is the label locality and compositionality. In Definition 4.13 for $R^b$, we see that the subdomain is constructed as $\{v \mid v_0 \leq v\}$, which implicitly expands to $\{v \mid v \in \mathcal{P}(Label) \land v_0 \leq v\}$.

When the subdomain is computed, a smaller, local $Label$ set may have been used which includes merely the labels used in the box. If later the box participates in a composition, where the other operand uses some additional labels, it would seem that the subdomain above has to be recomputed. However, in Chapter 7, we will discuss a method to encapsulate the subdomains with only the labels in the local $Label$ set, which can automatically adapt to any larger $Label$ set, thus showing that compositionality is present.

### 4.4 Semantics for Implementation

The specification in Section 4.2.3 provides the standard which any implementation should follow, but the rule lsPar requires traversing inside the two parallel branches and exhausting all possible executions to detect the stuck cases and look for the used value parts. In this section, we will discuss how we can improve the runtime efficiency, by delegating these tasks elsewhere.

In Section 4.3, we have shown that the rep given to the network $n$, by the function $R(n)$, encapsulates all relevant behaviours of $n$, i.e. its domain and the results per value in the domain. The rep is naturally the place to look into, if we need to understand a network without going through its topology. In other words, the rep given to a parallel branch provides a snapshot of the branch, allowing the runtime to quickly detect the stuck executions and find the used value parts. Incidentally, we have done something similar in the formula of $R^p$ in Definition 4.18, constructing the rep for the parallel composition using only the reps for the branches.

A benefit of using the reps is that they are always available, because by the time the compiler finishes preparing a program for execution, the type system has given each network, and therefore each branch, a rep. The only effort required now is to expose the relevant information in the reps to the runtime.

#### 4.4.1 Attractions

As Definition 4.18 suggests, to execute a program, the L-Net runtime needs only a fraction of the information contained in the reps for all parallel branches, namely the subdomains and the size of the used value parts. We therefore build the attractions as the interface between the type system and the runtime, exposing only the interesting portions of the reps. The $Attr$ structure and the relation between reps and attractions are defined as follows, where each subdomain is paired with its rank, the largest size of the used value parts related to that subdomain in the rep.

$$\alpha \in Attr = \text{Subdom} \times \mathbb{Z}$$
\[ \mathcal{A} : \text{Net} \rightarrow \mathcal{P}(\text{Attr}), \]
\[ \mathcal{A}(n) \triangleq \{(\sigma, \max(ks)) \mid (\sigma, \omega s) \in \mathcal{R}(n)\} \]
where \( ks = \{||\omega \cdot u|| \mid \omega \in \omega s\}. \]

\[ .s : \text{Attr} \rightarrow \text{Subdom}, \quad \alpha.s \triangleq \alpha \downarrow_1; \]
\[ .r : \text{Attr} \rightarrow \mathbb{Z}, \quad \alpha.r \triangleq \alpha \downarrow_2. \]

The subexpression \( \max(ks) \) in the function \( \mathcal{A} \) is defined, because of the following property.

**Lemma 4.23.** For any network \( n \), every case in \( \mathcal{R}(n) \) contains at least one output choice:

\[ \forall \kappa. \kappa \in \mathcal{R}(n) \implies \exists \omega. \omega \in \kappa. \omega. \]

**Proof.** The syntax requirement that the declared output set of a box cannot be empty indirectly proves the base case for the lemma. The rest of the algorithm \( \mathcal{R} \) preserves this property. The formal proof is in [A.5 on page 147](#).

Then, for any branch \( n \), the runtime can use its attraction set \( \mathcal{A}(n) \) to quickly check if a potential input value is accepted, and if so, what size the largest possible used value part is.

### 4.4.2 Program Execution

The specification includes the operations on the used value parts and the flow-inherited value parts. However, the only place the used value parts are utilised is in the rule \( \text{lsPar} \), for deciding which branch to take, and the only purpose of the flow-inherited value parts is to help compute the resulting used value part in the rule \( \text{lsSer} \). The availability of the ranks in the attractions renders it unnecessary to keep track of these two kinds of value parts during an execution.

We therefore define the semantics for implementation using the following non-stuck execution judgement:

\[ n, v \rightsquigarrow_1 v', \]

and the following stuck execution judgement:

\[ n, v \not\rightsquigarrow_1. \]

For a box, a serial composition, and all the stuck executions, the rules resemble their counterparts in the specification (the label prefix \( li \) denotes 'L-Net semantics for implementation'):

\[
\begin{align*}
\text{box} & \quad v_0 \leq v, \quad v_1 \in v s_0 \quad \text{(liBox)} \\
& \quad v_0 \rightarrow v s_0, \quad v \rightsquigarrow_1 v - v_0 + v_1 \\
\text{liSer} & \quad n_1, v \rightsquigarrow_2 v_1, n_2, n_1 \rightsquigarrow_2 v_2 \rightarrow n_1 \cdot n_2, \quad v \rightsquigarrow_1 v_2
\end{align*}
\]
\[ n, v \not\Rightarrow_i \text{ iff } \begin{cases} n = \text{box } v_0 \rightarrow _\_ \land v_0 \not\in v; \text{ or} \\ n = n_1 \cdot n_2 \land (n_1, v \not\Rightarrow_i \lor \\ (\exists v_1. n_1, v \Rightarrow_i v_1 \land n_2, v_1 \not\Rightarrow_i)); \text{ or} \\ n = n_1 \parallel n_2 \land n_1, v \not\Rightarrow_i \land n_2, v \not\Rightarrow_i. \end{cases} \]

\[ \text{(liStuck)} \]

For a parallel composition, the rule uses the attraction sets of the two branches, obtained using the function \( A \):

\[
\exists \alpha_j, \alpha_j \in A(n_j) \land v \in \alpha_j.s \\
(\forall \alpha_k, \alpha_k \in A(n_k) \land v \in \alpha_k.s \implies \alpha_j.r \geq \alpha_k.r) \\
\frac{n_j, v \Rightarrow_i v'}{n_1 \parallel n_2, v \Rightarrow_i v'} \quad \text{(liPar)}
\]

4.4.3 Compliance

**Theorem 4.24.** The semantics for implementation complies with the specification. I.e. \( \forall n, v, v' : \)

- \( n, v \not\Rightarrow_s \iff n, v \not\Rightarrow_i ; \)
- \( n, v \Rightarrow_s \_ \_ , v' \iff n, v \Rightarrow_i v'. \)

**Proof.** The two semantics \( \Rightarrow_i \) and \( \Rightarrow_s \) share great similarity. Most cases can be proven trivially. For the parallel composition (liPar and liPar), we focus on proving the equivalence of their premises, which has been informally covered when introducing the attractions. See A.6 on page 148 for the details. \( \square \)

4.4.4 Relation with S-Net

As discussed in Section 1.1, the S-Net semantics for parallel composition depends on the type system. While this causes inconvenience in formalising the type system, it is perfect as the guideline to implement the S-Net runtime. Indeed, in the existing S-Net implementation, the compiler generated code for a parallel composition includes the type information, which is read by the runtime for routing records.

The L-Net specification does not depend on its type system, and the parallel composition specification is hard to implement as is. The semantics for implementation in this section reintroduces the type-semantics interdependency, using the attractions as the interface which provide the type information required to implement the parallel composition.

4.5 Summary

In this chapter, we have introduced L-Net, a label-set transforming language with a limited set of constructs derived from S-Net, with which a programmer can build a hierarchical network: boxes lie at
the bottom of the hierarchy, and remove and add labels to the input values; serial compositions link boxes and subnetworks in sequential order, feeding the output values of a subnetwork to the input of another; parallel compositions align subnetworks side by side and deliver different values to different subnetworks depending on the attractions. L-Net models a subset of the S-Net’s type-related behaviour.

We have also provided a sound and complete type system for L-Net, which can capture the exact behaviours of L-Net programs. The network types – reps – are simple structures describing very complex types at the S-Net level. They also help simplify the implementation of the language by providing snapshots of the networks. L-Net and its type system provide us the foundation on which we will gradually add the features in S-Net and build up to a full-fledged type system for S-Net. Next step, BL-Net.
Chapter 5

BL-Net

We have defined L-Net and its sound and complete type system in Chapter 4, which serves as a prelude to BL-Net, which is closer to S-Net (see Figure 1.3). This chapter extends L-Net to BL-Net. According to Table 3.1, BL-Net has the following new features compared with L-Net: the binds, the sinks, and the serial replication.

The binds are the new elements in a BL-Net value. They affect how the boxes accept the values, and in doing so bind a specific category of values to a specific route in the program topology. In effect, they provide the programmers with a much easier way than in L-Net to control the branch selection process in the parallel compositions.

The sink is another kind of unit component, which looks like a box, accepts certain values like a box, but does not produce an output. Put differently, it removes the values from the network.

The serial replication resembles a loop in conventional programs but implemented iteratively, where an input value is processed by replicas of the same network indefinitely many times, until the outcome meets the predefined criteria. Although the values flow strictly unidirectional in L-Net as well as in BL-Net, the loop-like structure allows a BL-Net value to appear to move backwards and redo a part of the program when necessary. This makes BL-Net a lot more useful than L-Net.

Any serial replication can be modelled by other BL-Net constructs, which simplifies the specification and the type system. To model a serial replication, we need the box, serial composition and parallel composition, as well as the sink.

In this chapter, the new features in BL-Net will be formalised first, and then the concepts with no or little change from L-Net are added to complete the BL-Net specification. Afterwards, the type system for L-Net and the semantics for efficient L-Net implementation will be extended to serve BL-Net. We shall consider L-Net to have served its purpose of easing us into BL-Net, and 'deprecate' all syntaxes, variables, operators and functions for L-Net from this point on. We will redefine all of them using the same names and symbols, but still allow referencing back to their predecessors by number (e.g. Theorem 4.19 for the wellformedness theorem in L-Net) and by label (e.g. IsBox for the box semantics in L-Net). The variable naming conventions described in Section 4.2.1 still apply in this chapter, as well as everywhere else in this thesis.
Many discussions in Chapter 4 for L-Net can be shared in this chapter for BL-Net, including the idea that BL-Net exists to help study the type-related behaviour of S-Net, the relation between BL-Net components and their S-Net counterparts, the use of BL-Net reps as S-Net network types and so on. In this chapter, we will omit most of these discussions to avoid duplication, and to focus on the new elements in BL-Net.

5.1 Example BL-Net Programs

Example 5.1. A single-box program.

\[
\text{box } AB\#BC \rightarrow \{D\#E\}
\]

<table>
<thead>
<tr>
<th>(v) (Input)</th>
<th>(v') (Output)</th>
<th>(u)</th>
<th>(f)</th>
<th>(d)</th>
<th>(a)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A#BC</td>
<td>stuck</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>AB#BC</td>
<td>D#E</td>
<td>AB#BC</td>
<td>###</td>
<td>AB#BC</td>
<td>D#E</td>
</tr>
<tr>
<td>AB#BCDE</td>
<td>D#DE</td>
<td>AB#BC</td>
<td>##D</td>
<td>AB#BCE</td>
<td>D#E</td>
</tr>
<tr>
<td>ABC#BC</td>
<td>stuck</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Example 5.2. Failing to deliver \(bcd\) to \(b_3\). Program is in L-Net.

let \(b_1 = \text{box } A \rightarrow \{bc,bcd\}\),

\[
\begin{align*}
  b_2 &= \text{box } bc \rightarrow \{x\}, \\
  b_3 &= \text{box } d \rightarrow \{y\} \text{ in} \\
  b_1 \cdot (b_2 || b_3)
\end{align*}
\]

![Figure 5.1: Illustration of Examples 5.2 and 5.3](image)

<table>
<thead>
<tr>
<th>(v) (Input)</th>
<th>(v'')</th>
<th>Selected Branch</th>
<th>(v') (Output)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>BC</td>
<td>(b_2)</td>
<td>X</td>
</tr>
<tr>
<td>A</td>
<td>BCD</td>
<td>(b_2)</td>
<td>DX</td>
</tr>
</tbody>
</table>

Example 5.3. Controlling branch selection using binds.

let \(b_1 = \text{box } \emptyset\#A \rightarrow \{p\#bc,q\#bcd\}\),

\[
\begin{align*}
  b_2 &= \text{box } p\#bc \rightarrow \{\emptyset\#x\}, \\
  b_3 &= \text{box } q\#d \rightarrow \{\emptyset\#y\} \text{ in} \\
  b_1 \cdot (b_2 || b_3)
\end{align*}
\]

<table>
<thead>
<tr>
<th>(v) (Input)</th>
<th>(v'')</th>
<th>Selected Branch</th>
<th>(v') (Output)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\emptyset#A</td>
<td>p#bc</td>
<td>(b_2)</td>
<td>\emptyset#x</td>
</tr>
<tr>
<td>\emptyset#A</td>
<td>q#bcd</td>
<td>(b_3)</td>
<td>\emptyset#bcy</td>
</tr>
</tbody>
</table>
5.2 Binds and the Effects on Values

In Chapter 3 we have seen the reduction process from S-Net to BL-Net, where the binding tag in S-Net record types is translated to the bind in BL-Net. This section describes the bind in detail.

5.2.1 Values with Binds

A bind comes from the same set Label as an ordinary label, but it is its placement in a value that makes it special. A value in BL-Net consists of two label sets separated by ‘#’, and the set to the left contains binds. To reduce confusion, the term label will now be reserved for a non-bind label. As a result, a value in BL-Net consists of a bind set and a label set, as opposed to in L-Net where a value has only a label set. Equivalently speaking, the bind set for all L-Net values are empty.

The formulation below also defines a symbol ⋆, which is a special bind only available in binary code, and cannot be used by a programmer.

\[ v, u, f, d, a \in Value ::= ls#ls \]
\[ l \in Label ::= A | B | C | \cdots | ⋆ \]

The symbol ‘#’ also serves as the constructor, allowing us to mathematically create a value with a bind set and a label set. The two functions defined below do the opposite, extracting the bind set and label set of the input value.

\[ B : Value \rightarrow \mathcal{P}(Label), \quad L : Value \rightarrow \mathcal{P}(Label), \]
\[ B(ls_1#ls_2) \triangleq ls_1, \quad L(ls_1#ls_2) \triangleq ls_2. \]

For succinctness of the expressions, when applying either function above to a value expression consisting of a single variable, the parentheses are omitted, e.g. \( B \ v_1 \) and \( L \ v_2 \). We shall now augment the definitions of addition, difference and intersection operations on Value. In L-Net, they are defined as simply the corresponding label set operations, and in BL-Net, the bind sets undergo the same operation, as seen below. Note that the definitions all fall back to their predecessors when all bind sets are empty.

\[ v_1 + v_2 \triangleq (B v_1 \cup B v_2)#(L v_1 \cup L v_2) \]
\[ v_1 - v_2 \triangleq (B v_1 \setminus B v_2)#(L v_1 \setminus L v_2) \]
\[ v_1 \times v_2 \triangleq (B v_1 \cap B v_2)#(L v_1 \cap L v_2) \]

Once again, we assume that the three operators above are left associative. Also note that we have reverted to using \( \cup, \cap, \setminus \) for operations on the subsets of Label, because +,−,× are for the members of Value. We also define the size of a value as the size of its label set:

\[ |v| \triangleq |L v|. \]
5.2.2 Partial Order of Values

**Definition 5.4.** The partial order `$\leq$` on `Value` is defined such that the following holds:

$$v_1 \leq v_2 \iff Bv_1 = Bv_2 \land Lv_1 \subseteq Lv_2.$$  

The partial order above requires two values to have exactly the same bind set to form an ordered relation. Values with different bind sets are unrelated, and therefore there is no greatest or least value under this order. However, if the set `Value` is partitioned by the bind set, then in each part, we will be able to find a minimum value `$ls_1 \# \emptyset$`, where `$ls_1$` is the shared bind set of that part, such that every value `$v$` in the part, including this minimum, satisfies `$ls_1 \# \emptyset \leq v$`. We call this minimum the `root` of any value in this part. It is not difficult to see that the root of any given value `$v_0$` is `(B$v_0$)\#\emptyset`.

It is also observable that this partial order reduces to a simple subset relation on the label set, if all bind sets are the same. Recall that the set `Value` for L-Net can be seen as a subset of `Value` for BL-Net where all bind sets are empty. This implies that the same partial order has existed since L-Net. In fact, it has been used in the rule `lsBox`: the actual input and the declared input must have matching bind sets (both empty), and all labels in the declared input must exist in the actual input.

Below we have the specification of a box in BL-Net, which is identical to `lsBox`, indicating that a box in BL-Net behaves the same as a box in L-Net: if the partial order is satisfied, the box removes the equivalent of the whole declared input from the actual input, leaving no binds and possibly some leftover labels in the value, and then adds the binds and labels found in a nondeterministically chosen value from the declared output set. See also Example 5.1 and note how the binds and labels do not interfere with each other. The deleted value part `$d$` and the added value part `$a$` is also included per execution for reference.

$$(v_0 \leq v \quad v_1 \in vs_0 \quad box\ v_0 \rightarrow vs_0, \quad v \rightarrow_k v_0, \quad v = v_0 - v_1, \quad v = v_0 + v_1) (bsBox)$$

The label prefix `bs` denotes `BL-Net semantics`.

5.2.3 Controlling Branch Selection

In L-Net, Parallel compositions are useful in two occasions: switching, e.g. `$\lambda$` to branch one and `$\beta$` to branch two, and specialisation, e.g. `$\kappa$` to branch one but `$\kappa\beta$` to branch two. For the latter, it suffices to let the branch selection algorithm in the parallel composition specification `lsPar` do the job, and the expected outcome can be observed. For the former occasion, however, the same algorithm may cause unexpected behaviours, when the input value contains both sets of labels.

Example 5.2 which is written in L-Net, demonstrates the issue with a parallel composition of the switching type. Suppose we want to deliver the second output of `$b_1, bcd$`, to `$b_3$`. Both branches accept `$bcd$`, and the upper branch attracts it better because of a larger used value part. In order to divert `$bcd$`
to the lower branch, we can create a pass-through box that uses the two flow-inherited labels \( b \) and \( c \), like so:

\[
b_4 = \text{box } bc \rightarrow \{ bc \},
\]

and then add this in front or behind \( b_3 \), like so:

\[
b_1 \cdot ( b_2 \parallel ( b_3 \cdot b_4 ) ),
\]

or we can simply let \( b_3 \) use and regenerate the two labels, if we are allowed to modify its definition. This creates a larger used value part, \( bcd \), for the lower branch, and the output \( bcd \) will now be diverted to \( b_3 \).

This and other similar tricks have several drawbacks. First of all, it introduces unnecessary complexities into the code, and secondly, the modification may cause side effects. For example, the first output of \( b_1, bc \), when carrying a flow-inherited label \( d \), will also be diverted to the lower branch. Controlling branch selection in L-Net is not always applicable.

A more feasible way is to let the value be accepted into only one branch, and this can be easily done with the help of the binds. Example 5.3 upgrades the previous example to BL-Net, and introduces two binds, \( p \) and \( q \), in the two outputs of \( b_1 \) and the inputs of \( b_2 \) and \( b_3 \). Due to the partial order \( \leq \) (Definition 5.4), the first output of \( b_1, p \# bc \), is unaccepted by \( b_3 \), and the second output \( q \# bcd \) is unaccepted by \( b_2 \). In effect, the first output is now bound to \( b_2 \), and the second output to \( b_3 \), regardless of whether there are additional flow-inherited labels. In other words, by using the binds, we have created two virtual routes in the topology, and despite that a part of the routes are shared, i.e. the arrow labelled \( v'' \) in Figure 5.1, the values travelling through the shared part still stay in the virtual routes we have defined for them.

Therefore, in BL-Net, programmers are recommended to use different bind sets in different branches, for the parallel compositions of the switching type, so that the choice of branch is defined by the bind set of the input value. The virtual route effect helps improve code readability and minimise errors.

### 5.3 Sinks and Sunk Executions

#### 5.3.1 Definition

In L-Net, the declared output set of a box is required to be non-empty. Such constraint is lifted in BL-Net. A box definition with an empty declared output set defines a sink. It accepts the same type of values as a box, but produces no output. We say that these values are sunk by the sink.

\[
\frac{v_0 \leq v}{\text{box } v_0 \rightarrow \emptyset, \ v \rightsquigarrow_s \text{nil} \ (\text{bsSink})}
\]
5.3.2 Three Forms of Execution

The rule bsSink presents a new form of execution: the sunk execution, expressed with

\[ n, v \rightsquigarrow_s \text{nil}, \]

which means that \( v \) will not produce an output that comes out from \( n \), either because of a sink or as the result of an infinite serial replication execution (see Section 5.4.2). The keyword \text{nil} is part of the syntax of the judgment.

Therefore, in BL-Net, there are three types of execution. Besides the sunk execution, there is the normal execution:

\[ n, v \rightsquigarrow_s u, f, v', \]

and the stuck execution:

\[ n, v \not\rightsquigarrow_s ; \]

both of them mean the same as in the L-Net specification. In comparison, the lack of \( u \) and \( f \) in a sunk execution implies that it is not declared to use or flow-inherit any binds or labels. The term non-stuck execution now means a normal or sunk execution.

While the definition of acceptance remains unchanged, with the new form of execution come the updated lemmas describing the properties of execution in BL-Net.

**Definition 5.5.** A network \( n \) accepts a value \( v \), denoted as \( n \prec v \), iff \( \neg(n, v \not\rightsquigarrow_s) \).

**Lemma 5.6.** If a network \( n \) accepts a value \( v \), then there is at least one non-stuck execution with regard to \( n \) and \( v \):

\[ n \prec v \implies n, v \rightsquigarrow_s u, f, v' \lor n, v \rightsquigarrow_s \text{nil}. \]

**Lemma 5.7.** For any normal execution \( n, v \rightsquigarrow_s u, f, v' \), the following holds:

- \( u \leq v \);
- \( f \leq (\emptyset \# L v) \land f \leq (\emptyset \# L v') \);
- \( u \times f = \emptyset \# \emptyset \).

It is worth noting that the lemma above essentially restricts \( u \) to contain all binds in \( v \), and \( f \) to have an empty bind set.

The proofs for these lemmas are omitted for conciseness. They can be easily constructed, after we have the complete BL-Net specification, by consulting the proofs for their predecessors, Lemma 4.8 (Section A.1 on page 128) and Lemma 4.9 (Section A.2 on page 129), respectively.

5.3.3 Propagation

A sunk execution occurring in an inner network causes a sunk execution of the outer network. The following rules help propagate the sunk executions.

\[ 82 \]
\[
\frac{n_1, v \Rightarrow s \textbf{nil}}{n_1 \cdot n_2, v \Rightarrow s \textbf{nil}} \quad (\text{bsSerSunk1})
\]

\[
\frac{n_1, v \Rightarrow s \cdot \cdot \cdot \cdot v_1 \quad n_2, v_1 \Rightarrow s \textbf{nil}}{n_1 \cdot n_2, v \Rightarrow s \textbf{nil}} \quad (\text{bsSerSunk2})
\]

\[
j, k \in \{1, 2\} \quad j \neq k \quad n_j \triangleleft v
\]

\[
n_k, v \not\Rightarrow_s \lor (\forall u_k. n_k, v \Rightarrow_s u_k, \cdot\cdot\cdot \Rightarrow \exists u_j, n_j, v \Rightarrow_s u_j, \cdot\cdot\cdot \land |u_j| \geq |u_k|) \Rightarrow n_j, v \Rightarrow_s \textbf{nil}
\]

\[
\frac{n_1 \parallel n_2, v \Rightarrow_s \textbf{nil}}{n_1 \parallel n_2, v \Rightarrow_s \textbf{nil}} \quad (\text{bsParSunk})
\]

In the second line of the premises of the rule bsParSunk above, the \(\exists\) quantifier is nested within the \(\forall\) quantifier, which in effect requires that \(n_j\) can perform a normal execution only when \(n_k\) is also able to. As a consequence, when only sunk executions are available for both branches, \(n_j\) is still allowed to process \(v\) and result in the overall sunk execution. Meanwhile, if all preconditions of the rule bsPar (see Section 5.5) hold, as well as \(n_j, v \Rightarrow_s \textbf{nil}\), then both bsPar and bsParSunk are applicable, resulting in either a normal execution or a sunk execution. This agrees with the specification that the selected branch, here \(n_j\), can process the input value in whatever way available.

### 5.4 Serial Replications

#### 5.4.1 Definition

A serial replication is defined with the following syntax:

\[n \ast vs_1; vs_2.\]

In this definition, \(n\) is the operand network, \(vs_1\) is a set of \textit{unconditional terminating patterns}, and \(vs_2\) is a set of \textit{conditional terminating patterns}. Either set could be empty, but not both simultaneously. An actual value \(v\) is said to \textit{match} a pattern \(v_0\) in either set of patterns if \(v_0 \leq v\). In this context, we say that the binds and labels in \(v\) which are also found in \(v_0\) \textit{contribute to the match}.

#### 5.4.2 Specification

\textit{Replicas}, which are exact copies of \(n\), are composed serially one after another to form a \textit{chain}. An input value to the serial replication will be processed by this chain. However, before it enters the first replica, or any intermediate value produced by a previous replica enters the next one, a termination check takes place. If the value matches one of the unconditional terminating patterns, it exits the chain and
becomes the output of the serial replication; otherwise if the value matches a conditional terminating pattern, it either exits the chain or continues through the rest of the chain, and the decision is chosen non-deterministically. Failing the termination check, the value will continue through the rest of the chain.

The chain stretches as long as needed to guarantee the availability of additional replicas to process any intermediate values not exiting the chain. It is possible that the chain needs to contain an infinite number of replicas, because a particular execution never terminates, and as a result, there will never be an output value. We call this an infinite execution, and perceive it as a sunk execution.

For a normal execution of a serial replication, the used value part consists of all binds and labels in the original input value which are either used by a replica in the chain, or have contributed to a match during any termination check, including the conditional termination checks where the values are decided to be kept in the chain. The flow-inherited value part consists of those labels flow-inherited through all replicas, if any, during the execution, without contributing to any matches.

Figure 5.2 illustrates the internal structure of a serial replication. The filled diamonds represent the sites of termination checks.

**Relation with the S-Net serial replication.** The BL-Net serial replication is a true representation of the S-Net serial replication, up to the limitation of BL-Net, that is, the absence of tag values. In S-Net, a conditional terminating pattern carries a tag expression which S-Net is able to evaluate at runtime, because the integers stored under the tags are visible to S-Net. The reduction process (see Chapter 3) removes the integer values from the tags during the process of translating fields and tags into labels, so it is impossible for the BL-Net semantics to decide whether a value matches a conditional terminating pattern. We resort to non-determinism, as we have been for multiple outputs of the box, and make it a non-deterministic event whether a conditional terminating pattern actually causes a value to terminate.

**5.4.3 Modelling**

A serial replication can be modelled with other network constructs of BL-Net, which greatly simplifies the specification formulation. From Figure 5.2 we can see that the structure is periodic, and should be representable by serially composing infinite copies of a subnetwork $I$, in which $n$ is a component. Within one $I$, there are two parallel routes: the upper one is part of the chain and contains $n$, and the lower one is solely an exit route to pass the output values to the far right of the figure. For preparation, we assume that there are copies of a subnetwork $P$ placed on the exiting route, one per occurrence of $n$, for delivering the exiting values. Figure 5.3(a) illustrates the discussion so far.

A parallel composition is suitable to model $I$, but this causes the two routes to merge into one at the end of $I$, and split again to two at the beginning of the next $I$. In order to prevent an exiting value
from re-entering the chain, we apply the virtual route technique introduced in Section 5.2.3 and let \( P \) accept only the values with the special bind \( * \). The task to tag all exiting values with \( * \) naturally goes to the filled diamonds, which originally carry out the termination checks. They are now modelled by \( T \).

Because they produce two types of output values, one of which unaccepted by \( n \), the composition \( T \cdot n \) will be invalid. Therefore, we model the first diamond as a lone \( T \), and couple every \( n \) with the diamond after it to form \( n \cdot T \). This results in Figure 5.3(b).

Finally, at the end of the sequence of \( I \), the special bind \( * \) should be removed from the output values, and the values without \( * \) should be sunk to model the infinite executions. We delegate these tasks to a subnet work \( F \). The serial replication is now modelled with

\[
T \cdot I \cdots I \cdot I \cdots F,
\]

where \( I = (n \cdots) \parallel P \), as illustrated in Figure 5.3(c). Note that the first subnet work \( T \) does not accept any values with \( * \), and that the last subnet work \( F \) removes all occurrences of \( * \). This guarantees that the special bind is local to the current serial replication and does not propagate outwards. As a result, if another serial replication exists within \( n \), it is safe to model that instance using the same technique here and reusing the same special bind \( * \), without causing any conflicts.

### 5.4.3.1 Helper Subnetworks

It should be obvious that the helper subnetworks \( T \), \( P \) and \( F \) depend on the original serial replication, not only because of the operand network \( n \), but also the terminating patterns \( v_{s1} \) and \( v_{s2} \). Therefore, we will now define each of them as a function that takes the serial replication as the argument and builds the helper subnetwork.

The tagger \( T \) should check if any value entering it can exit the chain, use the labels in the values that contribute to any match, and add the special bind \( * \) to the value if it is exiting. Note the symmetry between an input value \( v \) matching a pattern \( v_0 - v_0 \leq v \) and a box declaring \( v_0 \) as the input accepting an actual input \( v - v_0 \leq v \), and also note that a box uses the labels in the actual input (see \( \text{bsBox} \)). This makes a box the perfect choice to model a match: for an unconditional terminating pattern \( v_1 \), the
corresponding box definition is
\[
\text{box } v_1 \rightarrow \{(v_1 + *\emptyset)\};
\]
and for a conditional terminating pattern \(v_2\), provided it does not itself match an unconditional terminating pattern, we include two outputs in the box definition to model the nondeterministic behaviour:
\[
\text{box } v_2 \rightarrow \{v_2, (v_2 + *\emptyset)\}.
\]

Additionally, the tagger should pass through all values not matching any terminating patterns, without using any labels in them. Using roots, i.e. values with empty label sets, guarantees that no labels will be used. As discussed in Section 5.2.2 there is no least element in the set \(\text{Value}\), but many roots. To pass through all values, we will need as many boxes as there are roots. However, because these values will be fed into the operand network \(n\), those with a root not in the root alphabet of \(n\), which we will define below, will never be accepted. Therefore, we can simply omit such roots, and thus let the affected values rejected in advance by \(T\).

**Definition 5.8.** The root alphabet of a network \(n\), denoted as \(\mathcal{R}(n)\), consists of the roots of all values it accepts:
\[
\mathcal{R}(n) \triangleq \{(B v)\emptyset | n \not\triangleleft v\}.
\]

Now, the tagger \(T\) can be defined as a parallel composition of as many boxes as necessary, some performing termination checks, and the others passing through the values not matching any terminating patterns.

**Definition 5.9.** The **tagger** for a serial replication is defined as
\[
T(n * vs_1; vs_2) \triangleq b_{11} \parallel b_{12} \parallel \cdots \parallel b_{1i}
\]
\[
\parallel b_{21} \parallel b_{22} \parallel \cdots \parallel b_{2j}
\]
\[
\parallel b_{31} \parallel b_{32} \parallel \cdots \parallel b_{3k},
\]
where \(\{b_{11}, \cdots, b_{1i}\} = \{\text{box } v_1 \rightarrow \{(v_1 + *\emptyset)\} | v_1 \in vs_1\};\)
\(\{b_{21}, \cdots, b_{2j}\} = \{\text{box } v_2 \rightarrow \{v_2, (v_2 + *\emptyset)\}
\]
\[
| v_2 \in vs_2 \land (\exists v_1, v_1 \in vs_1 \land v_1 \leq v_2)\};\)
\(\{b_{31}, \cdots, b_{3k}\} = \{\text{box } v_3 \rightarrow \{v_3\}
\]
\[
| v_3 \in \mathcal{R}(n) \land (\exists v_0, v_0 \in vs_1 \cup vs_2 \land v_0 \leq v_3)\}.
\]

The **passer** \(P\) is responsible for passing through the exiting values tagged with the special bind *. Comparing with \(T\), it is much easier to construct \(P\), because the forms of exiting values are provided in \(vs_1\) and \(vs_2\).
Definition 5.10. The **passer** for a serial replication is defined as

\[ P(n \ast vs_1; vs_2) \triangleq b_1 \parallel b_2 \parallel \cdots \parallel b_i, \]

where \( \{b_1, \cdots, b_i\} = \{\text{box } (v_0 + \ast\#\emptyset) \rightarrow \{v_0\} \mid v_0 \in vs_1 \cup vs_2\} \).

The **finaliser** \( F \) takes away the special bind \( \ast \) from the exiting values, and sinks other values without it to simulate an infinite execution. It is natural to think that the values it sinks are the outputs from \( n \), and we will need another type of root alphabet – the output alphabet – of \( n \). However, during a real infinite execution, these output values will become inputs to the next replica, and if any of these values are not accepted by \( n \), it would be a stuck execution instead. The root alphabet of \( n \) will suffice.

Definition 5.11. The **finaliser** for a serial replication is defined as

\[ F(n \ast vs_1; vs_2) \triangleq b_{11} \parallel b_{12} \parallel \cdots \parallel b_{1i} \]

\[ \parallel b_{21} \parallel b_{22} \parallel \cdots \parallel b_{2j}, \]

where \( \{b_{11}, \cdots, b_{1i}\} = \{\text{box } (v_0 + \ast\#\emptyset) \rightarrow \{v_0\} \mid v_0 \in vs_1 \cup vs_2\} \);

\( \{b_{21}, \cdots, b_{2j}\} = \{\text{box } v_2 \rightarrow \emptyset \mid v_2 \in \aleph(n)\} \).

5.4.3.2 Semantics Using Finite-length Model

In order to describe a serial replication, which is an infinite chain, we start with a finite-length model where an input value will be processed by a chain of limited replicas, and sunk at the end of the chain if it is unable to produce an output. The notion \( n^m vs_1; vs_2 \), where \( m \) is a non-negative integer, is a short notation for a finite-length serial replication with \( m \) replicas. It is defined using the three helper subnetworks as follows.

Definition 5.12. For a serial replication \( n \ast vs_1; vs_2 \), the finite-length model with \( m \) replicas, where \( m \in \mathbb{N} \), is defined as

\[ n^m vs_1; vs_2 \triangleq T(n \ast vs_1; vs_2) \cdot I \cdot I \cdots I \cdot I \cdots I \cdot F(n \ast vs_1; vs_2), \]

where \( I = (n \ast T(n \ast vs_1; vs_2)) \parallel P(n \ast vs_1; vs_2) \) stands for an iteration.

If an input value to this model, with a certain \( m \), is able to produce an output, we can be sure that the same value will produce the same output if processed by the infinite model, i.e. the full serial replication:

\[ m \in \mathbb{N} \quad \frac{(n^m vs_1; vs_2), v \sim_s u, f, v'}{(n \ast vs_1; vs_2), v \sim_s u, f, v'} \text{(bsStar)} \]

On the other hand, if the input value is sunk by the finite-length model, it may be due to either a sunk execution within \( n \), or the finaliser \( F \). If the former case is true, no matter how many more replicas are provided, the result will still be a sunk execution; whereas for the latter, we should check whether it would be an infinite execution, by providing infinitely more replicas. The two cases can be merged into
one rule:
\[
\forall m', m' \geq m \implies (n' vs_1; vs_2), v \rightsquigarrow_s nil \quad (\text{bsStarSunk})
\]

An input value causes a stuck execution, if it or any resulting intermediate output is stuck within the chain. This means that the finaliser is not involved in a stuck execution. To help the formulation of the stuck execution, we define the finaliser-less model as follows.

**Definition 5.13.** For a serial replication \( n * vs_1; vs_2 \), the finaliser-less model of length \( m \), where \( m \in \mathbb{N} \), is defined as
\[
n^m_{c,P} vs_1; vs_2 \triangleq T(n * vs_1; vs_2) \cdot I \cdot \ldots \cdot I_{m \text{ times}},
\]
where \( I = (n \cdot T(n * vs_1; vs_2)) \parallel P(n * vs_1; vs_2) \).

**Lemma 5.14.** For any serial replication \( n * vs_1; vs_2 \),
\[
n^m vs_1; vs_2 = (n^m_{c,P} vs_1; vs_2) \cdot F(n * vs_1; vs_2).
\]

**Proof.** By Definitions 5.12 and 5.13

The stack execution for serial replications, formulated using the finaliser-less model, is defined along with other stuck executions under the shared rule \( \text{bsStuck} \) on the following page.

### 5.5 Remaining Specification

Having introduced the new features in BL-Net, we will now complete the specification by listing the definitions, lemmas and rules largely unchanged from L-Net. The only observable changes here are due to the introduction of the serial replication.

\[
n \in \text{Net} \quad ::= \quad b \mid n \cdot n \mid n \parallel n \mid n * vs; vs
\]

\[
b \in \text{Box} \quad ::= \quad \text{box} \; v \rightarrow vs
\]

\[
n_1, v \rightsquigarrow_s u_1, f_1, v_1 \quad n_2, v_1 \rightsquigarrow_s u_2, f_2, v_2
\]

\[
n_1 \cdot n_2, v \rightsquigarrow_s u_1 + (f_1 \times u_2), f_1 \times f_2, v_2 \quad (\text{bsSer})
\]

\[
j, k \in \{1, 2\} \quad j \neq k \quad n_j < v
\]

\[
n_k, v /\rightsquigarrow_s \; \vee \; (\forall u_k, n_k, v \rightsquigarrow_s u_k, \_ , \_ \implies \\
\exists u_j, n_j, v \rightsquigarrow_s u_j, \_ , \_ \wedge |u_j| \geq |u_k|)
\]

\[
n_j, v \rightsquigarrow_s u, f, v'\quad n_1 \parallel n_2, v \rightsquigarrow_s u, f, v'\quad (\text{bsPar})
\]

88
$$n, v \not\xrightarrow{s}$$ iff

$$\left\{ \begin{array}{l}
n = \text{box } v_0 \rightarrow_0 \land v_0 \not\leq v; \text{ or } \\
n = n_1 \cdot n_2 \land (n_1, v \not\xrightarrow{s} v) \lor \\
(\exists v_1, n_1, v \not\xrightarrow{s} \_ \vdash_0 v_1 \land n_2, v \not\xrightarrow{s})); \text{ or } \\
n = n_1 \parallel n_2 \land n_1, v \not\xrightarrow{s} \land n_2, v \not\xrightarrow{s}; \text{ or } \\
n = n_0 \ast vs_1; vs_2 \land (\exists m, m \in \mathbb{N} \land (n_0^m, v, vs_1; vs_2), v \not\xrightarrow{s}).
\end{array} \right.$$ (bsStuck)

**Lemma 5.15.** $n \triangleleft v$, iff:

$$\left\{ \begin{array}{l}
n = \text{box } v_0 \rightarrow_0 \land v_0 \leq v; \text{ or } \\
n = n_1 \cdot n_2 \land n_1 \triangleleft v \land \\
\forall v_1, n_1, v \not\xrightarrow{s} v_1 \implies n_2 \leq v_1; \text{ or } \\
n = n_1 \parallel n_2 \land (n_1 \triangleleft v \lor n_2 \triangleleft v); \text{ or } \\
n = n_0 \ast vs_1; vs_2 \land \\
\forall m, m \in \mathbb{N} \implies (n_0^m, vs_1; vs_2) \triangleleft v.
\end{array} \right.$$

**Proof.** by Definition 5.5 and the inverse of the rule bsStuck.

**Definition 5.16.** The domain of a network, denoted as $\text{dom}(n)$, is the set of all values it accepts:

$$\text{dom}(n) = \{ v \mid n \triangleleft v \}.$$  

### 5.6 Type System

In this section, the type system for L-Net, discussed in Section 4.3, will be extended for use as a BL-Net type system. The purpose of the type system is unchanged: to provide the runtime with the domain of the program. Thanks to the symmetry of the specifications and execution properties of the two languages, no big change is necessary. The definitions and the type inference algorithm will be reintroduced here, with explanations for the minor modifications. The relation between this type system and the types in S-Net is as discussed in 4.3.3, so we will omit the duplicated discussion.

#### 5.6.1 Accommodating Sunk Executions

We would still like to use the same functional representation of networks as before: regarding a network $n$ and partial function $G : \text{Value} \rightarrow \mathcal{P}(\text{Value} \times \text{Value} \times \text{Value})$, if the following both hold:

$$\text{dom}(G) = \text{dom}(n),$$

$$\forall v, v \in \text{dom}(n) \implies G(v) = \{ (u, f, v') \mid n, v \not\xrightarrow{s} u, f, v' \},$$

89
then \( G \) represents \( n \). However, before we continue, a discussion about the sunk executions is necessary.

The definition above hides some of the sunk executions. If \( n \not< v_0 \) but \( v_0 \) can only result in sunk executions in \( n \), then \( G(v_0) = \emptyset \) as per the second line of the definition above. However, if \( n \not< v_0 \), and \( v_0 \) can cause some normal as well as sunk executions, then there will be no sign of any sunk executions in the result of \( G(v_0) \). Sunk executions are by definition not stuck; it is the other outputs in the result of \( G(v_0) \) that are prone to causing stuck executions, if they are fed into another network composed serially after. Therefore, the fact that the sunk executions may be trivialised by the normal executions does not impair the ability for the functional representation approach to serve the purpose of the type system.

In conclusion, we do not need to specially accommodate the sunk executions.

5.6.2 Reps and Type Inference

\[
\begin{align*}
\rho & \in \text{Rep} = \mathcal{P}(\text{Case}) \\
\kappa & \in \text{Case} = \text{Subdom} \times \mathcal{P}(\text{Out}) \\
\sigma & \in \text{Subdom} = \mathcal{P}(\text{Value}) \\
\omega & \in \text{Out} = \text{Value} \times \text{Value} \times \text{Value}
\end{align*}
\]

The \text{Rep} structure above is identical to that in L-Net: a rep is divided into several cases, each having its own subdomain and a set of output choices. An output choice is a triple of values: the used, deleted and added value parts. The accessor operators and the definition of rep wellformedness is repeated below.

\[
\begin{align*}
.s : \text{Case} & \rightarrow \text{Subdom}, \quad \kappa.s & \triangleq \kappa \downarrow_1; \\
.o : \text{Case} & \rightarrow \mathcal{P}(\text{Out}), \quad \kappa.o & \triangleq \kappa \downarrow_2; \\
.u : \text{Out} & \rightarrow \text{Value}, \quad \omega.u & \triangleq \omega \downarrow_1; \\
.d : \text{Out} & \rightarrow \text{Value}, \quad \omega.d & \triangleq \omega \downarrow_2; \\
a : \text{Out} & \rightarrow \text{Value}, \quad \omega.a & \triangleq \omega \downarrow_3;
\end{align*}
\]

Definition 5.17. A rep is well formed iff:

- the subdomains of its cases do not overlap, and
- for each of its output choices, the deleted value part contains all labels of the added value part.

Formally, \( \rho \circ \) iff

\[
(\forall \kappa_1, \kappa_2, \kappa_3, \kappa_4 \in \rho \land \kappa_1 \neq \kappa_2 \implies \kappa_3.s \cap \kappa_4.s = \emptyset)
\]

Formally, \( \rho \circ \) iff

\[
\land (\forall \kappa, \omega \in \rho \land \omega \in \kappa.o \implies L.\omega.a \subseteq L.\omega.d).
\]

The wellformedness definition above has the same wording as, but a different formulation to, its predecessor (Definition 4.11). The difference is merely due to the introduction of the binds. Like in L-Net, the type inference algorithm guarantees the following extra properties:

- there will be no cases with an empty subdomain, and
- for each output choice \( \omega \), \( \omega.u \leq \omega.d \).
The following definitions are identical to those in L-Net:

\[ \text{dom} : \text{Rep} \rightarrow \mathcal{P}(\text{Value}) \]
\[ \text{dom}(\rho) \triangleq \bigcup_{\kappa \in \rho} \kappa. \]

\[ '() : \text{Rep} \rightarrow \text{Value} \rightarrow \mathcal{P}(\text{Value} \times \text{Value} \times \text{Value}) \]
\[ v \in \text{dom}(\rho) \implies \\
\rho(v) \triangleq \{ (\omega.\upsilon, v - \omega.\upsilon, v - \omega.\upsilon + \omega.\alpha) \\
\mid \kappa \in \rho \land v \in \kappa.s \land \omega \in \kappa.0 \}. \]

\[ \emptyset \in \text{Rep}; \quad \emptyset = \{ \}. \]

The type inference algorithm is encapsulated in the function \( R \) below, which gives a rep to each network as its type.

**Definition 5.18.** The type inference algorithm \( R \) is defined as follows.

\[
R : \text{Net} \rightarrow \text{Rep}, \\
R(n) \triangleq \\
\begin{cases} 
R^b(b), & \text{if } n = b; \\
R^n(R(n_1), R(n_2)), & \text{if } n = n_1 \cdot n_2; \\
R^p(R(n_1), R(n_2)), & \text{if } n = n_1 \parallel n_2; \\
R^s(n_0, v_1, v_2), & \text{if } n = n_0 \ast v_1, v_2.
\end{cases}
\]

The definition above sees the introduction of \( R^s \), a type inference function for serial replications. Section 5.6.3 discusses it in detail.

**5.6.2.1 Boxes and Sinks**

**Definition 5.19.** (Type inference sub-algorithm for a box.)

\[
R^b : \text{Box} \rightarrow \text{Rep}, \\
R^b(\text{box } v_0 \rightarrow v_0) \triangleq \left\{ \left( v \mid v_0 \leq v \right), \left( \{ v_0, v_0 + (\emptyset \# L v_1), v_1 \mid v_1 \in v_0 \} \right) \right\}.
\]
5.6.2.2 Serial Compositions

**Definition 5.20.** (Type inference sub-algorithm for the part of a serial composition involving a subdomain-output choice pair \((\sigma, \omega)\) from the left operand and a case \((\kappa_2)\) from the right operand.)

\[
\mathcal{R}_3^3 : \text{Subdom} \times \text{Out} \times \text{Case} \rightarrow \text{Rep},
\]

\[
\mathcal{R}_3^3(\sigma, \omega, \kappa_2) \triangleq \text{let } \sigma' = \{ v \mid v \in \sigma \land (v - \omega.d + \omega.a) \in \kappa_2.s \} \text{ in}
\]

\[
\begin{cases}
0, & \text{if } \sigma' = \emptyset, \\
\{ \{ \sigma', \{ \omega.u + (\omega_2.u - \omega.a), \omega.d + \emptyset \# \mathcal{L}(\omega_2.d), \omega.a - \omega_2.d + \omega_2.a \mid \omega_2 \in \kappa_2.o \} \}, & \text{otherwise}.
\end{cases}
\]

This definition does not apply to sinks from the left operand, because they cannot supply an output choice \(\omega\). The algorithm covers left operand sinks in Definition 5.22. However, right operand sinks are covered, where \(\kappa_2.o = \emptyset\), so the rule bsSerSunk2 is covered.

The formula above computes the combined deleted value part as \(\omega.d + \emptyset \# \mathcal{L}(\omega_2.d)\), as opposed to \(\omega.d + \omega_2.d\) in Definition 4.14 on page 68. This is to exclude any binds in the deleted value part from the right operand to maintain the optional but elegant property of a well-formed rep that the used value part precedes the added value part by the partial order \((\omega'.u \leq \omega'.d)\).

**Definition 5.21.** (Type inference sub-algorithm for the part of a serial composition involving a subdomain-output choice pair \((\sigma, \omega)\) from the left operand and the whole of the right operand \((\rho_2)\).)

\[
\mathcal{R}_3^5 : \text{Subdom} \times \text{Out} \times \text{Rep} \rightarrow \text{Rep},
\]

\[
\mathcal{R}_3^5(\sigma, \omega, \rho_2) \triangleq \bigcup_{\kappa_2 \subseteq \rho_2} \mathcal{R}_3^3(\sigma, \omega, \kappa_2).
\]

**Definition 5.22.** (Type inference sub-algorithm for the part of a serial composition involving a case \((\kappa)\) from the left operand and the whole of the right operand \((\rho_2)\).)

\[
\mathcal{R}_3^1 : \text{Case} \times \text{Rep} \rightarrow \text{Rep},
\]

\[
\mathcal{R}_3^1(\kappa, \rho_2) \triangleq \text{let } \{ \omega_i \mid i \in 1..|\kappa.o| \} = \kappa.o \text{ in}
\]

\[
\begin{cases}
k' \mid (\kappa.o = \emptyset \implies k' = \kappa) \land (\kappa.o \neq \emptyset \implies \\
\{
\forall i. i \in 1..|\kappa.o| \implies k'_i \in \mathcal{R}_3^3(\kappa.s, \omega_i, \rho_2))
\}
\end{cases}
\]

\[
\land k' = (\bigcap_{i=1}^{k'.s, \bigcup_{i=1}^{\kappa.o} k'_i, o} k' \neq \emptyset).
\]

In the definition above, when \(\kappa.o = \emptyset\), i.e. the case from the left operand only encapsulates some sunk executions, there will be no valid \(i\), and the aggregated operations \(\bigcap, \bigcup\) on \(i\) are undefined. In order to cover the rule bsSerSunk1, when \(\kappa.o = \emptyset\), we let the resulting case be identical to \(\kappa\), propagating the sunk executions. It can be easily shown that when \(\kappa.o \neq \emptyset\), this definition is equivalent with its predecessor, Definition 4.16.
Definition 5.23. (Type inference sub-algorithm for a serial composition.)

\[ R^s : \text{Rep} \times \text{Rep} \rightarrow \text{Rep}, \]
\[ R^s(\rho_1, \rho_2) \triangleq \bigcup_{\kappa \in \rho_1} R^s_1(\kappa, \rho_2). \]

5.6.2.3 Parallel Compositions

Definition 5.24. (Type inference sub-algorithm for a parallel composition.)

\[ R^p : \text{Rep} \times \text{Rep} \rightarrow \text{Rep}, \]
\[ R^p(\rho_1, \rho_2) \triangleq \rho' \cup \rho'', \text{ where } \]
\[ \rho' = \left\{ (\sigma, \kappa_j, o) \mid j, k \in \{1, 2\} \land j \neq k \land \kappa_j \in \rho_j \land \sigma = \right. \]
\[ \left. \{ v \mid v \in \kappa_j, s \land (\forall \kappa', \kappa' \in \rho_k \land v \in \kappa'. s \implies \exists \omega_j, \omega_j \in \kappa_j, o \land \right. \]
\[ \left. \forall \omega'. \omega' \in \kappa'. o \implies |\omega_j, u| > |\omega', u| \} \right\}, \]
\[ \rho'' = \left\{ (\kappa_1, s \cap \kappa_2, s, \kappa_1, o \cup \kappa_2, o) \mid \right. \]
\[ \left. \kappa_1 \in \rho_1 \land \kappa_2 \in \rho_2 \land \kappa_1, s \cap \kappa_2, s \neq \emptyset \land \right. \]
\[ \left. (\kappa_1, o = \kappa_2, o = \emptyset \lor \right. \]
\[ \left. \exists \omega_1, \omega_2, \omega_1 \in \kappa_1, o \land \omega_2 \in \kappa_2, o \land |\omega_1, u| = |\omega_2, u| \land \right. \]
\[ \left. \forall \omega. \omega \in \kappa_1, o \cup \kappa_2, o \implies |\omega_1, u| > |\omega, u| \} \right\}. \]

The definition above differs from its predecessor, Definition 4.18, by the addition of \( \kappa_1, o = \kappa_2, o = \emptyset \lor \) in the sub-formula for \( \rho'' \), which handles the scenario where both sets of output choices are empty, due to that the chosen cases \( \kappa_1 \) and \( \kappa_2 \) for both branches cover sunk executions only. In this situation, \( \rho'' \) contains a case with no output choices for the overlapping part of the subdomains of \( \kappa_1 \) and \( \kappa_2 \), in line with the rule \text{bsParSunk}. For \( \rho' \), the algorithm design in the former definition already supports sunk executions.

5.6.3 Serial Replications

All serial replications can be modelled by boxes, sinks, serial compositions and parallel compositions, as shown in Section 5.4.3. It should therefore naturally follow that the type inference algorithm for a serial replication can simply reuse the algorithms for other network constructs. The model as defined in Section 5.4.3.2 is a serial composition of a tagger, then a number of iterations, then the finaliser in the end. The type inference follows the same order, including a number of iterations before adding the finaliser. The question remains is how many iterations should be included. We now discuss a method to detect a point where the computation can end, i.e. we can stop adding more iterations.
5.6.3.1 Fixed Points

For any serial replication, because the choice for \( m \in \mathbb{N} \) is infinite, there are infinite finaliser-less models. However, the observable behaviours, i.e. the executions, of these models are finite. This is ultimately because the set \( \text{Label} \) is finite, containing only the binds and labels the programmer has used, plus the special bind \( \star \). As a consequence, the set \( \text{Value} \), being essentially \( \mathcal{P}(\text{Label}) \times \mathcal{P}(\text{Label}\{\star\}) \), is also finite. This in turn limits the number of different instances of all three forms of execution the model can perform.

Therefore, as the length of the model increases, at a certain point we will have observed all executions available to any finaliser-less model of any length. We call the length at this point the fixed point of the finaliser-less model of the serial replication, or simply the fixed point of the serial replication. By the discussion above, a fixed point always exists.

**Definition 5.25.** A fixed point of a serial replication \( n \ast vs_1; vs_2 \) is a natural number \( m_0 \) that satisfies the following:

\[
\forall m', v, u, f, v'. \ m' \in \mathbb{N} \land m' > m_0 \land ((n^{m'}_{\text{F}} vs_1; vs_2), v \rightsquigarrow_s u, f, v') \\
\implies \exists m''. \ m'' \in \mathbb{N} \land m'' \leq m_0 \land ((n^{m''}_{\text{F}} vs_1; vs_2), v \rightsquigarrow_s u, f, v');
\]

\[
\forall m', v, m' \in \mathbb{N} \land m' > m_0 \land ((n^{m'}_{\text{F}} vs_1; vs_2), v \neq \rightsquigarrow_s)
\implies \exists m''. \ m'' \in \mathbb{N} \land m'' \leq m_0 \land ((n^{m''}_{\text{F}} vs_1; vs_2), v \neq \rightsquigarrow_s).
\]

We have excluded the sunk executions from the definition above. They are downplayed in compliance with the discussion in Section 5.6.1. Some of them are still detectable; for example, a value not stuck in the network but with no normal executions available is eligible for a sunk execution. It is only these lone sunk executions that are important to type inference.

It is easily deducible that, if \( m_0 \) is a fixed point of a serial replication, all natural numbers greater than \( m_0 \) are fixed points of the same serial replication.

**Detection.** Definition 5.25 still requires looking into an infinite number of models beyond the current length, i.e. the number of iterations. In fact, fixed point can be detected by looking only one iteration further.

**Theorem 5.26.** Given a serial replication \( n \ast vs_1; vs_2 \) and a natural number \( m_0 \), if

\[
\forall v, u, f, v'. \ (n^{m_0+1}_{\text{F}} vs_1; vs_2), v \rightsquigarrow_s u, f, v' \implies \exists m'. \ m' \in \mathbb{N} \land m' \leq m_0 \land (n^{m'}_{\text{F}} vs_1; vs_2), v \rightsquigarrow_s u, f, v', \text{ and}
\]

\[
\forall v. \ (n^{m_0+1}_{\text{F}} vs_1; vs_2), v \neq \rightsquigarrow_s \implies \exists m'. \ m' \in \mathbb{N} \land m' \leq m_0 \land (n^{m'}_{\text{F}} vs_1; vs_2), v \neq \rightsquigarrow_s,
\]

then \( m_0 \) is a fixed point of \( n \ast vs_1; vs_2 \).
Proof. We rewrite Definition 5.25 into the following:

\[
\forall p, p \in \mathbb{N}^+ \implies \\
\left( \forall m', v, u, f, v', m' \in \mathbb{N} \land m' > m_0 \land (n_{\text{IF}}^{m'} v_{s1}; v_{s2}) \nleq v \land (n_{\text{IF}}^{m'} v_{s1}; v_{s2}), v \leadsto_s u, f, v' \right) \\
\implies \exists m'', m'' \in \mathbb{N} \land m'' \leq m_0 \land (n_{\text{IF}}^{m''} v_{s1}; v_{s2}), v \leadsto_s u, f, v'', \forall m', v, m' \in \mathbb{N} \land m' > m_0 \land (n_{\text{IF}}^{m'} v_{s1}; v_{s2}), v \nleq_s u, f, v''; \\
\forall m', v, m' \in \mathbb{N} \land m' > m_0 \land (n_{\text{IF}}^{m'} v_{s1}; v_{s2}), v \leadsto_s u, f, v''; \\
\implies \exists m'', m'' \in \mathbb{N} \land m'' \leq m_0 \land (n_{\text{IF}}^{m''} v_{s1}; v_{s2}), v \nleq_s u, f, v'', \forall m', v, m' \in \mathbb{N} \land m' > m_0 \land (n_{\text{IF}}^{m'} v_{s1}; v_{s2}), v \leadsto_s u, f, v''; \\
\implies \exists m'', m'' \in \mathbb{N} \land m'' \leq m_0 \land (n_{\text{IF}}^{m''} v_{s1}; v_{s2}), v \nleq_s u, f, v'', \forall m', v, m' \in \mathbb{N} \land m' > m_0 \land (n_{\text{IF}}^{m'} v_{s1}; v_{s2}), v \nleq_s u, f, v''; \\
\implies \exists m'', m'' \in \mathbb{N} \land m'' \leq m_0 \land (n_{\text{IF}}^{m''} v_{s1}; v_{s2}), v \leadsto_s u, f, v'', \forall m', v, m' \in \mathbb{N} \land m' > m_0 \land (n_{\text{IF}}^{m'} v_{s1}; v_{s2}), v \nleq_s u, f, v''; \\
\implies \exists m'', m'' \in \mathbb{N} \land m'' \leq m_0 \land (n_{\text{IF}}^{m''} v_{s1}; v_{s2}), v \nleq_s u, f, v'', \forall m', v, m' \in \mathbb{N} \land m' > m_0 \land (n_{\text{IF}}^{m'} v_{s1}; v_{s2}), v \nleq_s u, f, v''; \\
\implies \exists m'', m'' \in \mathbb{N} \land m'' \leq m_0 \land (n_{\text{IF}}^{m''} v_{s1}; v_{s2}), v \leadsto_s u, f, v'', \forall m', v, m' \in \mathbb{N} \land m' > m_0 \land (n_{\text{IF}}^{m'} v_{s1}; v_{s2}), v \nleq_s u, f, v''; \\
\implies \exists m'', m'' \in \mathbb{N} \land m'' \leq m_0 \land (n_{\text{IF}}^{m''} v_{s1}; v_{s2}), v \leadsto_s u, f, v'', \forall m', v, m' \in \mathbb{N} \land m' > m_0 \land (n_{\text{IF}}^{m'} v_{s1}; v_{s2}), v \nleq_s u, f, v'';
\]

and the theorem becomes easily provable by mathematic induction on \(p\), with the help of bsSer and bsStuck. Details are omitted.  \(\square\)

Theorem 5.26 suggests an iterative method to construct a fixed point model for a serial replication: starting with a lone tagger \(T\), repeatedly append the iteration subnet \(I\) in serial composition, and record the observable normal and sunk executions. When no new normal or sunk executions are observed, stop appending \(I\) and finalise the model with the finaliser \(F\). The resulting model is finite in length and covers all executions of the original serial replication.

### 5.6.3.2 Early Fixed Points

For an input value which may cause both normal executions and stuck executions in a serial replication, its fixed point model exhibits the same behaviour. However, by definition such input value is outside the domain of the serial replication, and should be declared unsafe by the type inference. During the process of finding a fixed point, a stuck execution observed regarding an input value indicates that we no longer need to collect all normal executions for the same input. We incorporate this idea into the definition of the early fixed point below.

**Definition 5.27.** An early fixed point of a serial replication \(n \star v_{s1}; v_{s2}\) is a natural number \(m_0\) that satisfies the following:

\[
\forall m', v, u, f, v', m' \in \mathbb{N} \land m' > m_0 \land \\
(n_{\text{IF}}^{m'} v_{s1}; v_{s2}) \nleq v \land (n_{\text{IF}}^{m'} v_{s1}; v_{s2}), v \leadsto_s u, f, v' \\
\implies \exists m'', m'' \in \mathbb{N} \land m'' \leq m_0 \land (n_{\text{IF}}^{m''} v_{s1}; v_{s2}), v \leadsto_s u, f, v''; \\
\forall m', v, m' \in \mathbb{N} \land m' > m_0 \land (n_{\text{IF}}^{m'} v_{s1}; v_{s2}), v \nleq_s u, f, v''; \\
\implies \exists m'', m'' \in \mathbb{N} \land m'' \leq m_0 \land (n_{\text{IF}}^{m''} v_{s1}; v_{s2}), v \nleq_s u, f, v''; \\
\implies \exists m'', m'' \in \mathbb{N} \land m'' \leq m_0 \land (n_{\text{IF}}^{m''} v_{s1}; v_{s2}), v \nleq_s u, f, v''; \\
\implies \exists m'', m'' \in \mathbb{N} \land m'' \leq m_0 \land (n_{\text{IF}}^{m''} v_{s1}; v_{s2}), v \nleq_s u, f, v''; \\
\implies \exists m'', m'' \in \mathbb{N} \land m'' \leq m_0 \land (n_{\text{IF}}^{m''} v_{s1}; v_{s2}), v \nleq_s u, f, v'';
\]

Similarly, for a serial replication, if \(m_0\) is an early fixed point, then all natural numbers greater than \(m_0\) are early fixed points. An early fixed point can be detected by looking one iteration further.

**Theorem 5.28.** Given a serial replication \(n \star v_{s1}; v_{s2}\) and a natural number \(m_0\), if

\[
\forall v, u, f, v'. (n_{\text{IF}}^{m_0+1} v_{s1}; v_{s2}) \nleq v \land (n_{\text{IF}}^{m_0+1} v_{s1}; v_{s2}), v \nleq_s u, f, v' \implies \\
\exists m', m' \in \mathbb{N} \land m' \leq m_0 \land (n_{\text{IF}}^{m'} v_{s1}; v_{s2}), v \nleq_s u, f, v', and \\
\forall v. (n_{\text{IF}}^{m_0+1} v_{s1}; v_{s2}), v \nleq_s u, f, v' \implies \\
\forall v. (n_{\text{IF}}^{m_0+1} v_{s1}; v_{s2}), v \nleq_s u, f, v'.
\]
\( \exists m', m' \in \mathbb{N} \land m' \leq m_0 \land (n_{m'};v_1;v_2), v \not\sim_s, \)

then \( m_0 \) is an early fixed point of \( n \circ v_1;v_2. \)

The proof can be constructed using a similar technique as that for Theorem 5.26.

The early fixed point is as useful as the fixed point, since the input values outside the domain of the serial replication need not be taken into account. The following theorem formulates this.

**Theorem 5.29.** If \( m_0 \) is an early fixed point of \( n \circ v_1;v_2, \) then \( n^{m_0};v_1;v_2 \) has the same domain as \( n \circ v_1;v_2, \) and all input values within the domain exhibit the same normal executions in both networks.

Equivalently,

\[
\forall v. (n^{m_0};v_1;v_2), v \not\sim_s \iff (n \circ v_1;v_2), v \not\sim_s, \text{ and } \\
\forall v, u, f, v'. (n^{m_0};v_1;v_2) \sim v \implies (n^{m_0};v_1;v_2), v \sim u, f, v'.
\]

**Proof.** Relatively straightforward by the connection between the early fixed point and the model of the serial replication (Definition 5.12) established by Theorem 5.28. See A.7 on page 150 for a detailed proof.

The early fixed point model \( n^{m_0};v_1;v_2 \) is therefore suitable for type inference to obtain the type of the original serial replication.

### 5.6.3.3 Algorithm

We are now prepared for the type inference algorithm for a serial replication. In the following definition, we construct \( \rho \) using the model \((T \cdots I \cdots \cdots I \cdots F)\) from left to right, and use a loop to add iterations one by one. We detect the early fixed point of the finaliser-less model by means of Theorem 5.28, keeping track of \( \delta, \) the domain of \( \rho \) after each iteration, and \( \beta, \) the input-output pairs observed so far. The change of \( \delta \) signals some new stuck executions, and the change of \( \beta \) signals some new normal executions. Once \( \delta \) stops changing and \( \beta \) stops growing, an early fixed point is detected. We then include the finaliser to complete the process.

**Definition 5.30.** The type inference sub-algorithm for a serial replication is defined as function \( R^* : Net \times P(Value) \times P(Value) \rightarrow Rep, \) which follows the procedure in Figure 5.4

This algorithm always terminates, and an information reasoning is given as follows. First of all, observe that the type inference for serial composition, \( R^* \), always produces a rep whose domain is a subset of the left operand’s domain. This is ultimately due to how a serial composition accepts values (see Lemma 5.15). This implies that the domain of \( \rho \) in the step ‘set \( \rho := R^*(\rho, \rho') \)' can only become smaller as the number of iterations increase. Also, because the set \( Label \) is finite, the initial domain of \( \rho \) is finite, and there are finite elements the set \( \beta \) can hold, so the growth of \( \beta \) is bounded. Then, within a finite number of iterations, the domain of \( \rho \) will stop shrinking and the set \( \beta \) will stop growing, so the algorithm terminates.
function $R^*(n, vs_1, vs_2)$:
set $\rho := R(T(n * vs_1; vs_2))$;
let $\rho' := R(I)$, where $I = (n \cdots T(n * vs_1; vs_2)) \parallel P(n * vs_1; vs_2)$;
set $\delta \in \mathcal{P}(Value) := \text{dom}(\rho)$;
set $\beta \in \mathcal{P}(Value \times \text{Out}) := \{(v, \omega) \mid \kappa \in \rho \land v \in \kappa.s \land \omega \in \kappa.o\}$;
repeat:
set $\rho := R^*(\rho, \rho')$;
if $\text{dom}(\rho) = \delta \land \{(v, \omega) \mid \kappa \in \rho \land v \in \kappa.s \land \omega \in \kappa.o\} \subseteq \beta$ then:
exit repeat;
else:
set $\delta := \text{dom}(\rho)$;
set $\beta := \beta \cup \{(v, \omega) \mid \kappa \in \rho \land v \in \kappa.s \land \omega \in \kappa.o\}$;
end if;
until exited;
return $R^*(\rho, R(F(n * vs_1; vs_2)))$;
end function.

Note: The keyword let defines a constant whereas the keyword set initialises or modifies a mutable variable.

Figure 5.4: Serial replication type inference.

5.6.4 Soundness and Completeness

Like in Section 4.3.5, the soundness and completeness of the BL-Net type system benefits from the following properties of $R$.

**Theorem 5.31.** The algorithm $R$ creates only well formed reps. That is, for any network $n$, $R(n)$ is well formed.

*Proof.* By structural induction on $n$ like the proof for Theorem 4.19, its counterpart in L-Net, in A.3 on page 131. The sub-algorithm $R^*$ reuses $R^b$, $R^s$ and $R^p$ to build the result, so it is a trivial inductive case. For other functions, consult A.3.\[\square\]

**Theorem 5.32.** For any network $n$, $R(n)$, when used as a function, represents $n$:

- $\text{dom}(n) = \text{dom}(R(n))$;
- $\forall v. \, v \in \text{dom}(n) \implies n, v \sim_s u, f, v' \iff (u, f, v') \in (R(n))(v)$.

*Proof.* Similar to the proof for Theorem 4.20, its counterpart in L-Net, in A.4 on page 136. The new elements in BL-Net require special attention. For binds, the upgraded value operations $+,-,\times$ simplifies the process to upgrade the computation part of the proof A.4. The discussions after Definition 5.22 and Definition 5.22 informally cover the correctness of sunk execution representation. Finally, because the serial replication is modelled using other BL-Net constructs, the correctness of representation is almost automatic. See A.8 on page 153 for a more detailed discussion on upgrading the former proof.\[\square\]

The soundness and completeness of the BL-Net type system are directly upgraded from their L-Net counterparts.

**Theorem 5.33.** The type system for BL-Net is sound. That is, given any network $n$ and an input value $v$, if $v \in \text{dom}(R(n))$, then $n \vdash v$. 97
Theorem 5.34. The type system for BL-Net is complete. That is, given any network \( n \) and an input value \( v \), if \( n \triangleright v \), then \( v \in \text{dom}(\mathcal{R}(n)) \).

Proof. By Theorem 5.32.

5.7 Semantics for Implementation

Naturally, the semantics for implementing L-Net efficiently in Section 4.4 can be extended for use in BL-Net. Comparing to L-Net, the reps generated by the BL-Net type inference algorithm help in two ways: providing the attractions for the parallel compositions as before, and also giving a quicker way to obtain the root alphabet of a network \( \mathcal{N}(n) \) for constructing the helper networks of the serial replications.

5.7.1 Attractions

It is preferable that the parallel composition semantics remains unchanged from liPar, which only uses the attractions to select the branches. However, due to the existence of sunk executions, which may cause some cases in the reps to have no output choices (and therefore there is no equivalent property in BL-Net as Lemma 4.23, we cannot fully reuse the definition of the function \( A \) in Section 4.4.1, where the subexpression \( \max(ks) \) will now be undefined when \( ks = \emptyset \). As a workaround, we let the set \( ks \) contain at the minimum \(-1\), so a subdomain paired with no output choices is still recorded in the attraction set, but with a rank easily superseded by a used value part of any size.

\[
\alpha \in Attr = \text{Subdom} \times \mathbb{Z}
\]

\[
A : \text{Net} \rightarrow \mathcal{P}(Attr),
\]

\[
A(n) \triangleq \{(\sigma, \max(ks)) \mid (\sigma, \omega s) \in \mathcal{R}(n)\}
\]

where \( ks = \{-1\} \cup \{ \lvert \omega \rvert \mid \omega \in \omega s \} \).

\[
.s : Attr \rightarrow \text{Subdom}, \quad \alpha.s \triangleq \alpha \downarrow 1;
\]

\[
.r : Attr \rightarrow \mathbb{Z}, \quad \alpha.r \triangleq \alpha \downarrow 2.
\]

5.7.2 Root Alphabet

Using the rep of a network, its root alphabet can be easily extracted.

Theorem 5.35. For any network \( n \),

\[
\mathcal{N}(n) = \{ Bv \# \emptyset \mid v \in \text{dom}(\mathcal{R}(n)) \}.
\]
5.7.3 Program Execution

The three forms of executions defined in Section 5.3.2 are translated as follows:

- Normal execution: \( n, v \rightsquigarrow_{n} v' \);
- Sunk execution: \( n, v \rightsquigarrow_{n} \text{nil} \);
- Stuck execution: \( n, v \not\rightsquigarrow_{n} \).

The following rules describe the semantics for efficient implementation, using the attractions for parallel compositions, and implicitly using the root alphabet computation (Theorem 5.35) to aid the serial replications. The label prefix bi denotes ‘BL-Net semantics for implementation’.

\[
\frac{v_0 \leq v}{\text{box } v_0 \Rightarrow v_{s_0}, v \rightsquigarrow_{v} v - v_0 + v_1} \quad \text{(biBox)}
\]

\[
\frac{v_0 \leq v}{\text{box } v_0 \Rightarrow \emptyset, v \rightsquigarrow_{n} \text{nil}} \quad \text{(biSink)}
\]

\[
\frac{n_1, v \rightsquigarrow_{s_1} v_1, n_2, v \rightsquigarrow_{s_2} v_2}{n_1 \cdot n_2, v \rightsquigarrow_{n} v_1 \cdot v_2} \quad \text{(biSer)}
\]

\[
\frac{n_1, v \rightsquigarrow_{s_1} \text{nil}, n_2, v \rightsquigarrow_{s_2} \text{nil}}{n_1 \cdot n_2, v \rightsquigarrow_{n} \text{nil}} \quad \text{(biSerSunk1)}
\]

\[
\frac{n_1, v \rightsquigarrow_{s_1} \text{nil}, n_2, v \rightsquigarrow_{s_2} \text{nil}}{n_1 \cdot n_2, v \rightsquigarrow_{n} \text{nil}} \quad \text{(biSerSunk2)}
\]

\[
\frac{j, k \in \{1, 2\}, j \neq k}{\exists \alpha_j, \alpha_j \in A(n_j) \land v \in \alpha_j.s \land (\forall \alpha_k, \alpha_k \in A(n_k) \land v \in \alpha_k.s \Rightarrow \alpha_j.r \geq \alpha_k.r) \quad \text{(biPar)}}
\]

\[
\frac{j, k \in \{1, 2\}, j \neq k}{\exists \alpha_j, \alpha_j \in A(n_j) \land v \in \alpha_j.s \land (\forall \alpha_k, \alpha_k \in A(n_k) \land v \in \alpha_k.s \Rightarrow \alpha_j.r \geq \alpha_k.r) \quad \text{(biParSunk)}}
\]

\[
\frac{m \in \mathbb{N}}{(n^{m}v_{s_1}; v_{s_2}), v \rightsquigarrow_{n} v'} \quad \text{(biStar)}
\]
\[
m \in \mathbb{N} \quad \forall m', m' \geq m \implies (n^{m'}; v_{s1}; v_{s2}), v \rightsarrow \text{nil} \quad (\text{biStarSunk})
\]

\[
\begin{cases}
n = \text{box} \quad v_0 \rightarrow \_ \quad \land \quad v_0 \not\in v; & \text{or} \\
n = n_1 \cdots n_2 \quad \land \quad (n_1, v \not\rightsarrow \_ \lor \ldots) & \text{or} \\
n = n \parallel n_2 \quad \land \quad n_1, v \not\rightsarrow \_ \land n_2, v \not\rightsarrow \_ & \text{or} \\
n = n * v_{s1}; v_{s2} \quad \land \quad (\exists m. m \in \mathbb{N} \land (n^{m}_{0}; v_{s1}; v_{s2}), v \not\rightsarrow \_).
\end{cases}
\]

\[n, v \not\rightsarrow \_ \iff n, v \not\rightsarrow \_ \quad \land \quad v_0 \not\in v; \quad \text{or}
\]

\[n, v \not\rightsarrow \text{nil} \iff n, v \not\rightsarrow \text{nil}.
\]

5.7.4 Compliance

**Theorem 5.36.** The semantics for implementation complies with the specification. I.e. \( \forall n, v, v' : \)

- \( n, v \not\rightsarrow s \iff n, v \not\rightsarrow s; \)
- \( n, v \rightsarrow s, \_, v' \iff n, v \rightsarrow s, v'; \)
- \( n, v \rightsarrow s \text{nil} \iff n, v \rightsarrow s \text{nil} \)


5.8 Summary

In this chapter, we have introduced BL-Net, a label set transforming language with all features of L-Net plus the following new language elements: binds, sinks and serial replications. When discussing the binds, a partial order of values formerly hidden in L-Net is made clear. Proper use of the binds and the partial order can give the programmers the power to easily control routing at the parallel compositions. The sinks are a special type of boxes, and together with binds, it allows us to model the serial replications by other network constructs of BL-Net.

The previous type system for L-Net has been extended to cover the new features in BL-Net. Its soundness and completeness are preserved during the process. As before, the reps calculated per network by the type inference can help with efficient implementation of BL-Net.

Like L-Net, BL-Net exists to model S-Net’s type-related behaviour. We have omitted most discussions about the relation between BL-Net and S-Net; similar discussions have been covered in Chapter 4 for L-Net. The full context is now given for the readers to understand the reduction process in Chapter 3 to turn S-Net programs to BL-Net. We can now use the BL-Net type system to type-check S-Net programs.
Chapter 6

Type System for S-Net

In the previous chapters, we have proposed a method to reduce S-Net in order to study S-Net’s type-related behaviour with minimum noise. This includes the specification of a new language BL-Net with a sound and complete type system (Chapter 5) and a translation algorithm turning S-Net programs to BL-Net programs (Chapter 3). In this chapter, we will introduce the type system for S-Net in terms of BL-Net, completing the big picture.

We call this type system for S-Net the new type system, as opposed to the previous type system documented in the published technical report [25] and included in this thesis as Appendix B. We will also test them side by side using an example S-Net program.

This new type system is not without its limitations. Two major limitations are discussed at the end of this chapter.

6.1 Preliminaries

We will now revisit some important ideas as a preparation to the type system for S-Net.

S-Net type system purposes. In Section 1.1 we have listed the S-Net type system purposes. We rephrase them using the proper terminology for S-Net as follows. The S-Net type system should:

1. guarantee that every box or filter receives only those records it can process, which means that the type of every input record matches the type it is declared to accept;

2. provide sufficient information for every parallel branch, to allow the runtime to efficiently decide the correct route for each input record to each parallel composition.

The translation process. In Section 3.2 we have defined the translation algorithm $T$ (entry point $T_p$, Section 3.2.10) to turn an S-Net program to a BL-Net program. However, Section 3.2.9 explains that a signed network in an S-Net program needs to be type checked separately as a stand-alone S-Net program isolated from the original one. So to type check an S-Net program, we must also type check as many subprograms as there are signed networks in it.
Type inference in BL-Net. Section 5.6.2 introduces the BL-Net type inference algorithm $R$, which computes a rep per BL-Net network. The rep has the same domain as that of the network, and represents the network’s full behaviour for all values in the domain. This domain helps us deduce straightforwardly whether an arbitrary input value is type safe in the network.

BL-Net reps as S-Net network types. The relation between an L-Net rep and an S-Net network type is established in Section 4.3.3. Because L-Net is a temporary language as a prelude to BL-Net, this relation can also apply to a BL-Net rep and an S-Net network type. In short, a BL-Net rep can describe the type-related behaviour of the corresponding S-Net program or program fragment.

Attractions and the semantics for implementation. The algorithm $A$ to extract the attractions of a BL-Net network is defined in Section 6.7.1, which collects information from the type of the network, the rep. The attractions are then used by the BL-Net semantics for implementation (Section 5.7), which has the same type-semantics interdependency as S-Net but is more efficient to implement.

6.2 New Type System

We are now ready to define the type system for S-Net driven by the two purposes. Following the introduction at the beginning of this chapter, we call it the new type system.

6.2.1 Type Checking S-Net Programs

The first purpose of the type system maps directly to the definition of acceptance of BL-Net (Definition 5.5 on page 82). A stuck execution in BL-Net models the situation where an S-Net box or filter receives a record it cannot process. Acceptance guarantees that this never occurs in all possible executions. The first purpose of the type system is now reduced to checking whether the program accepts its inputs.

An S-Net program must carry a top-level network signature. The S-Net runtime guarantees that every input record to the program will bear one of the record types declared in the signature. If the record actually contains more fields and tags than listed in the type, the runtime will put them in a protected space (prevent them from being visible in the program). To type check a program $P$, we first turn it into BL-Net using the translation algorithm $T_p$, then compute the rep of the resulting BL-Net program using the type inference algorithm $R$. We then check whether the rep accepts the translation of each declared input record type $\tau$ in the signature, like so:

$$T(\tau) \in \text{dom}(R(T_p(P))).$$  \hspace{1cm} (1)

The program signature also documents the possible output record types per input type. This is particularly important when type checking a signed network as an isolated stand-alone program, because the signed network will be trusted to behave as its signature describes. Recall from Section 2.2.8 on page 32 that the signature has the additional use of trimming the output records, so it is allowed that an actual (predicted) output to have more fields and tags than the declared type. In the BL-Net terminology,
the declared output value precedes the actual (predicted) output value as per \( \leq \) (Definition 5.4 on page 80).

Before continuing with the discussion, we shall define a function to help predict the output values (output record types) in response to an input using a rep, which essentially simplifies the rep application operator \( '(' \) defined on page 91 to return only the output values without the used and flow-inherited value parts.

**Definition 6.1.** The following function evaluates a value using the given rep:

\[
\mathcal{E} : \text{Rep} \rightarrow \text{Value} \rightarrow \mathcal{P}(\text{Value}) \cup \{\text{wrong}\},
\]

\[
\mathcal{E}(\rho, v) \triangleq \begin{cases} 
\text{wrong}, & \text{if } v \notin \text{dom}(\rho); \\
\{v' \mid (\_\,, \_, v') \in \rho(v)\}, & \text{otherwise}.
\end{cases}
\]

We let the variable `vs` range over \( \mathcal{P}(\text{Value}) \cup \{\text{wrong}\} \), and \( vs = \mathcal{E}(R(T_p(P)), T(\tau)) \). If \( vs \neq \text{wrong} \), then (1) above holds, and \( vs \) is the set of predicted output values for the input record type \( \tau \). Suppose \( \tau \) is coupled with the set of output record types \( \tau_s \) in the program signature, i.e. the mapping \( \tau \rightarrow \tau_s \) exists in the signature, then this mapping is valid if

\[
\forall v. \ v \in vs \implies \exists v'. \ v' \in T(\tau_s) \land v' \leq v.
\]

We are now ready to present the full definition of a well typed S-Net program.

**Definition 6.2.** An S-Net program \( P \) with the signature \( \Sigma \) is well typed, iff:

\[
\forall \tau, \tau_s. \ (\tau \rightarrow \tau_s) \in \Sigma \implies \\
vs \neq \text{wrong} \land \forall v. \ v \in vs \implies \exists v'. \ v' \in T(\tau_s) \land v' \leq v
\]

where \( vs = \mathcal{E}(R(T_p(P)), T(\tau)) \), and all signed networks in \( P \), if any, are well typed when isolated as stand-alone S-Net programs.

Assuming the soundness of the translation algorithm is proven, then by Theorem 5.33 on page 97 a well typed S-Net program does not go wrong.

### 6.2.2 Supporting Parallel Compositions

We now discuss the second purpose of the type system: to provide sufficient information for every parallel branch, to allow the runtime to efficiently decide the correct route for each input record to each parallel composition.

It should be clear from the translation algorithm \( T \) that, if an S-Net program \( P \) translates to a BL-Net program \( n \), then every subexpression of every topology expression in \( P \) corresponds to a subnetwork in \( n \). (We assume that there are no unused definitions in \( P \), so all topology expressions are reachable by \( T \).) We also assume that the variable \( n \) can be updated silently to mean the translation of a subprogram.
isolated from $P$, if a topology expression is only reachable from a signed network.) This implies that every parallel branch has a corresponding translation in BL-Net.

During the type check of an S-Net program, the corresponding BL-Net network for each parallel branch will be assigned a rep. We can then use the algorithm $A$ (Section 5.7.1) to extract the attractions for the parallel branch. This allows the runtime to efficiently decide, for each parallel composition, to which branch to send the individual input records, as per the rule biPar on page 99 in the BL-Net semantics for implementation.

The second purpose of the type system is therefore fulfilled.

6.3 Comparison with Previous Type System

Having defined the new type system, we would like to compare it with the previous type system (see Appendix B). The following S-Net program, specially tailored to highlight the differences between the two type systems, will guide us through the various aspects of the comparison.

```plaintext
net compare ( {a,c,e} -> {a,c,e} )
{
  net trouble
  {
    box m1 ( (x) -> (a) );
    box m2 ( (c) -> (d) );
    net m connect m1 | m2;
    box n1 ( (a) -> (b) );
    box n2 ( (e) -> (x) );
    net n connect n1 | n2;
    box p ( (x) -> (y) );
  }
  connect m .. n .. p;
}
connect trouble | [ {a} -> {a} ];
```

When the new type system type checks the program, it assigns a rep to each network component, and a set of attractions per parallel branch for efficient execution. As a final step, it compares the signature of the top-level network $\text{compare}$ against the calculated rep for its translation in BL-Net. Similarly, the previous type system performs route inference on the program, giving each component, except $\text{compare}$, a $\text{sig}$ and each parallel branch the routing information. Its final step of type checking the program is a simulation of the type transformations of the input type $\{a, c, e\}$ through the network $\text{compare}$.
<table>
<thead>
<tr>
<th>New type system</th>
<th>Previous type system</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{R}(m_1) = {x/\emptyset \rightarrow {x,xa,a}}$</td>
<td>$m_1 : {x \rightarrow a[x]}$</td>
</tr>
<tr>
<td>Attractions of $m_1$: ${(x/\emptyset, 1)}$</td>
<td>Rt. info. of $m_1$: ${x}$</td>
</tr>
<tr>
<td>$\mathcal{R}(m) = {x/c \rightarrow {x,ax,a}}$</td>
<td>$m : {x \rightarrow a[x]; c \rightarrow d[c]}$</td>
</tr>
<tr>
<td>$\mathcal{R}(n) = {a/e \rightarrow {a,ab,b}}$</td>
<td>$n : {a \rightarrow b[a]; e \rightarrow x[e]}$</td>
</tr>
<tr>
<td>$\mathcal{R}(m \cdot n) = {x/ce \rightarrow {x,abx,b}}$</td>
<td>$m \cdot n : {x \rightarrow b[ax]; ex \rightarrow b[ax], ax[ex]; ac \rightarrow bd[ac]; ce \rightarrow dx[ce]}$</td>
</tr>
<tr>
<td>$\mathcal{R}(trouble) = {ce/ax \rightarrow {ce,cde,dy}}$</td>
<td>$trouble : {ce \rightarrow dy[ce]}$</td>
</tr>
<tr>
<td>Attractions of trouble: ${(ce/ax, 2)}$</td>
<td>Rt. info. of trouble: ${ce}$</td>
</tr>
<tr>
<td>$\mathcal{R}(compare) = {ce/ax \rightarrow {ce,cde,dy}}$</td>
<td>Rt. info. of filter: ${a}$</td>
</tr>
<tr>
<td>$\mathcal{L}(\mathcal{R}(compare), ace) = {ace}$.</td>
<td>Simulation for input ace: $</td>
</tr>
<tr>
<td>$\mathcal{L}(\mathcal{R}(compare), ace) = {ace}$.</td>
<td>$M \vdash ace \sim {ade}$</td>
</tr>
<tr>
<td>Type check passed.</td>
<td>$N \vdash ade \sim {bde, adx}$</td>
</tr>
<tr>
<td></td>
<td>bde cannot pass $P$.</td>
</tr>
</tbody>
</table>

Table 6.1: Outputs from both type systems

The computation processes are omitted, while some of the results are shown in Table 6.1. For all boxes in the program, the reps, sigs, attraction sets and routing information are all trivial; the table lists them for only one box as an example and omits the rest. For readability, the following syntactic sugars are applied to the outputs of the new type system. Firstly, because no binding tags are used in the program, we would omit the prefix $\emptyset#$ of all the values. To reduce the use of brackets, a case $(\sigma, \{\omega_1, \omega_2, \ldots\})$ is written as `$\sigma \rightarrow \omega_1, \omega_2, \ldots$' and is separated from adjacent cases of the same rep by semicolons. We also use a new syntax `$v_1/ls_2$' as the shortcut to the subdomain $\{v : v_1 \leq v \land ls_2 \cap \mathcal{L}(v) = \emptyset\}$, which informally means `$v_1$ with any extra labels except those in $ls_2$'. This syntax will be formally introduced in Section 7.1. The previous type system constructs found in Table 6.1 will be explained on demand in the following sections.
6.3.1 Structural Upgrades

In the new type system, the type inference algorithm \( \mathcal{R} \) is used to give each network a rep, which is a set of cases. Each case establishes the connection between a subdomain – a set of values – and a set of output choices. An output choice is a triple of used, deleted and added value parts. The term value comes from BL-Net, which models S-Net’s type-related behaviour. The translation algorithm \( \mathcal{T} \) establishes the link between an S-Net record type and a BL-Net value.

This is very similar to the previous type system, which uses the syntax \( n : \Sigma \) backed by inference rules to give the network \( n \) a sig \( \Sigma \). A sig is not to be confused with a signature, the latter written by the programmer and attached to an S-Net network in the code. A sig consists of individual maps, each of them maps an input type to a set of output variants. An output variant is a pair of an output type and a discarded field set, formatted in Table 6.1 as \( v[d] \) where \( v \) is the output type and \( d \) is the discarded field set. The input and output types are both S-Net record types.

Observably, the case structure is richer than the map structure, in that it can encapsulate more information. Firstly, the implicit subdomain of a map, assumed to be all the subtypes (see Section 6.3.2) of the input type, is now explicit in a case, which allows more flexible customisation. Secondly, the previous output variant is upgraded to the output choice, with the output type renamed as the added value part, the discarded field set turned into the deleted value part, and a used value part added to the tuple. Previously, the input type in a map doubles as the storage of useful labels (equivalent of the used value part) which is shared among all output variants of the map. As can be seen from the left half of line 10 in Table 6.1 the used value parts are not always the same as the minimum required labels of the subdomain; the latter is \( \{x\} \) but the first output choice uses only \( x \). This is the case when an input of type \( \{e,x\} \) goes through box \( n_1 \), becoming \( \{e,a\} \) where \( e \) is flow-inherited, and then goes through box \( n_1 \), becoming \( \{e,x\} \) where \( e \) is still flow-inherited. The right half of line 10 shows the output of the previous type system, which incorrectly indicates that the field \( e \) is used for the first output variant.

The deleted value part is required to contain all labels in the added value part (see Definition 5.17 on page 90), as opposed to the requirement of the discarded field set to cover all useful labels. However, this difference is trivial, because in actual computation the labels in the used value part (or input type) and the added value part (or output type) are both removed; the two type systems tackle this in different but equivalent ways.

6.3.2 Record Subtyping and Matching vs. Value Partial Order

In the original S-Net design, a record type can be subtyped by adding more fields. If record type \( r_1 \) is a subtype of \( r_2 \), we write \( r_1 \subseteq r_2 \). The previous type system directly uses this definition, whereas the new type system defines it as a partial order on values (see Definition 5.4 on page 80). If \( r_1 \) translates to the value \( v_1 \) (\( \mathcal{T}(r_1) = v_1 \)) and \( r_2 \) translates to \( v_2 \), the record subtyping relation and value partial ordering has the following equivalence:

\[
r_1 \subseteq r_2 \iff v_2 \leq v_1.
\]

An input record is said to match a declared type, if the type of the input record is a subtype of the
declared type. Matching is central to several S-Net operations, for example, a box can only process an input record if it matches the box’s declared input type, a synchrocell stores an input record if it matches one of its unmatched patterns, and a serial replication stops processing an intermediate output record if it matches one of the terminating patterns. In the new type system, all matching operations are replaced with the value partial order.

In essence, there is no difference between record subtyping and the value partial order, but the word ‘subtype’ is tightly related to subtype polymorphism, which causes the language users to assume a subtype safe to be used where a supertype is expected. In S-Net this does not generally hold. For example, line 19 in Table 6.1 shows the single case for the network inferred by the new type system, and the single map for the same network inferred by the previous type system. By seeing the input type of the map, one may think that the network accepts all subtypes of ce, but according to the subdomain of the case, if the field a or x is present in the record, the execution may be stuck. Indeed, an input record of type \{a,c,e\} may go through m2 and then n1, becoming \{b,d,e\}, which is stuck at box p for having no field x.

6.3.3 Type-semantics Interdependency

With regard to how parallel compositions distribute the input records, the original semantics, still documented in [25], uses the definition of better matching: a match is better than another, if the declared type used in the match has a greater number of fields and tags than the one used in the other match. The original semantics of a parallel composition says that an input record will be delivered to a branch, if the signature of the branch, either provided by the programmer or inferred by the type system, contains a declared input type which the type of the input record matches the best, among all declared input types in both branch signatures. In case of a tie between the two branches, the runtime chooses one non-deterministically for processing the input record.

This semantics of parallel compositions creates a strong mutual dependency between the type system and the operational semantics: the type system needs to infer the branch signatures which can summarise the semantic behaviours of the branches, but the latter depend on the signatures of the subnetworks of the branches, if there are any parallel compositions among them. We have discussed this type-semantics interdependency in Section 1.1 and various other places throughout the thesis. Although this is not a cyclic dependency – one way to unfold it is discussed in Section 1.1 – it still causes difficulty in formulation of S-Net and construction of the proofs.

Ideally, the type system and the operational semantics should exist on their own, so that the operational semantics is well defined and that the type system can be checked for correctness against the operational semantics. When designing the previous type system, we attempted to remove part of the mutual dependency, by introducing the idea of the unit network components using the labels and redefining better matching as more labels being used from the input record. To further decouple the operational semantics from the type system, we also facilitated the redesign of the system, so now the runtime looks in a dedicated internal language construct – the routing information – for the input types of the branch signatures, provided by the type system.
However, the way the previous type system models the runtime still reflects the original operational semantics of the parallel compositions. For example, the right half of lines 3-4 in Table 6.1 is the inferred sig for the network \( m \), a parallel composition of two boxes, which has two maps. When simulating an input record of type \( \{c,x\} \) through this network, we still apply the original operational semantics and judge that both maps are eligible and therefore the output may be \( \{a,c\} \) or \( \{d,x\} \).

In the new type system we aim to remove the interdependency completely. We have redefined the operational semantics in the form of the BL-Net program specification, which directly incorporates the used value parts as well as the flow-inherited value parts, eliminating its dependency on the type system. Meanwhile, to completely decouple the type system from the operational semantics, we have disambiguated the reps, so that an input value matches at most one case, by requiring that the subdomains do not overlap each other. For example, at lines 3-4 in Table 6.1 the input types of the two maps by the previous type system overlap with each other, but the new type system moves the overlapping part to a separate case at line 5, which records both outputs from the two original cases. The cases in lines 8 and 13-18 are similarly created.

6.3.4 Simulation vs. Function Representation

The differences between the two type systems discussed so far contribute vitally to the correctness of the new type system. Especially, the use of explicit subdomains has led to the discovery of important details we could not find previously. Line 19 in Table 6.1 perfectly illustrates this. The previous type system assumes that all subtypes of \( \{c,e\} \) are accepted into the network \( \text{trouble} \), and assigns the routing information (lines 20-21) such that the input record of type \( \{a,c,e\} \) is incorrectly delivered to \( \text{trouble} \) due to a better match. The new type system corrects this by showing that the accepted types should not contain field \( a \) or \( x \).

In fact, it was the very same network as \( \text{trouble} \) in the example program that has helped discover the flaw of the previous type system. To make the type system safe, the additional simulation step was added. The judgement form \( n \vdash r \Rightarrow rs \) found on the right of lines 23 and 24 of Table 6.1 denotes the simulation, meaning that the network \( n \) transforms the input type \( r \) into any of the output types in \( rs \). Instead of a direct calculation using the sig of network \( n \), the simulation follows a partial semantics of S-Net which is similar to the complete BL-Net semantics for implementation in Section 5.7. Simulating an input type through the program is therefore like performing the complete execution at the BL-Net level. Lines 22 to 25 briefly list the important simulation steps, which lead to the conclusion at line 26 that the program does not pass the type check, under the previous type system.

Using the new type system supported by the soundness proof for BL-Net, and assuming that the S-Net semantics will be fixed so that it matches BL-Net, we are guaranteed that the whole program is correctly represented by a function, which in turn is represented by the inferred rep. To obtain the possible output types from a given input type, we only need to apply the function. At lines 22-23 on the left side of Table 6.1 we have the inferred rep of the whole program. The evaluation function \( E \) (Definition 6.1) uses the rep as a function on the declared input type \( \{a,c,e\} \), and yields the output value \( \text{ace} \) (line 25), which matches the declared output type \( \{a,c,e\} \) according to the well-typedness
6.4 Limitations

The new type system has two major limitations, which we will discuss briefly in this section. They are more related to the design limitations of S-Net than of BL-Net.

6.4.1 Initialiser Boxes

The latest S-Net language design has one more network unit construct – the initialiser box. Comparing to a normal box, an initialiser box does not accept any input records, but will emit zero or more output records, like a box, when it is initialised in the network at runtime. It is in a sense the dual of the sink, and can be used to pre-populate the network with some records.

The initialiser box introduces another form of execution and therefore requires a relatively large change to the whole system. Moreover, whether the initialiser boxes in a serial replication are initialised once or as many times as there are replicas is still unclear, the decision of which will likely influence how the system should be changed. As a result, the new type system does not yet support the initialiser box.

6.4.2 Synchrocell Progressiveness

In Section 3.2.6 we have demonstrated the way to approximate a synchrocell with other network constructs. We are unable to model it precisely, because of its stateful nature as opposed to the statelessness of all other S-Net components. The approximation helps the type system perform type inference and type check for the networks where synchrocells are used, which can guarantee type safety, but not progressiveness of the program. For example, if at runtime one of the synchrocell patterns are never matched, and the synchrocell is wrapped in a serial replication whose only terminating pattern is the combined output type of the synchrocell (a sync-star construct), then it effectively becomes a sink, blocking the record flow.

It is possible to design a type system that helps check for progressiveness, but this requires multiplicity built into the language. The current box signature describes what types of records the box may output, but it is unknown how many records will actually be emitted. A box signature with multiplicity information will document the number of records emitted per output type, thus providing the type system with crucial information as what types of records are guaranteed to exist at a certain location of the network. Only with this information can progressiveness be inferred and checked by the type system.

6.5 Summary

In this chapter, we have defined the new type system for S-Net in terms of BL-Net. The S-Net type system has two purposes: to type check S-Net programs, and to provide the runtime with sufficient data supporting the routing algorithm for the parallel composition. These purposes are fulfilled using the sound and complete BL-Net type system, by translating S-Net programs to BL-Net.
We have also compared the new type system with the previous one, discovering that the new type system correctly handles the situations where the previous has failed. This is thanks to the redesign of the operational semantics and the rep structure, which removes the type-semantics interdependency, enabling a formalisation provable of correctness.

There are limitations of the new type system, namely the lack of support for the initialiser box, and that synchrocell progressiveness cannot be checked. They are the consequences of the restrictive design of the S-Net language itself.

What remains is to put the theory to practise. We do this by implementing the new type system in a prototype compiler.
Chapter 7

Implementation

In Chapter 6, we have built a type system for S-Net in terms of BL-Net and its sound and complete type system, which involves translating S-Net programs to BL-Net and isolating the signed networks into subprograms. This new type system has not been implemented in the official S-Net compiler, which uses the previous type system (in Appendix B) as of the time of writing this thesis.

A major challenge in implementing the type inference algorithm is the subdomains, each of which is a set of values depending on the labels available in the set Label, which in theory requires that the implementation pre-scans the whole program to discover all labels in use, resulting in bad performance. We discuss our method to compress the subdomains to make the operations on them efficient.

We then introduce a prototype implementation in C# to demonstrate the implementability of the new type system. This prototype compiler takes an S-Net program in a simplified grammar as input, and tells whether it passes the type check. It performs all type inference and type checking in place, without an explicit translation or subprogram isolation phase.

The compiler is only half of the S-Net picture, the other half being the runtime. The structural difference of the data supporting the parallel composition's operation (routing information in the previous type system, attractions in the new type system) makes it difficult to use the new type system in the existing compiler-runtime architecture. We include an informal discussion on one way to bridge the difference.

7.1 Subdomain Compression

Implementing the new type system involves translating the Rep structure and the type inference algorithm $\mathcal{R}$ as defined in Section 5.6.2 on page 90 into code. This is relatively straightforward, because the Rep structure is based on finite sets, and the main way the algorithms construct the instances of Rep and the inner sets therein is by enumeration.

The only exception to this is the Subdom substructure. The subdomains are constructed by $\mathcal{R}^b$ (Definition 5.19 on page 91) using a set comprehension syntax, which requires going through the set Value, whose size is exponential to that of Label. There are two problems if this is implemented
by enumeration: a subdomain potentially contains a very large number of values, and the set Label 
may still be incomplete if the compiler has only walked through a part of the program. For a more 
efficient implementation, it is better to be able to compress the subdomains without the knowledge 
of the complete set Label. Moreover, the type system algorithms may perform any combination of 
the following operations on the subdomains: intersection (e.g. in Definition 5.22 on page 92), difference 
(hidden in Definition 5.24 on page 93), and refinement (as seen in Definition 5.20 on page 92). Preferably, 
the compression method should allow these operations to be done on the compressed subdomains.

We now present one solution that enables efficient implementation of the subdomains using only the 
local knowledge of labels. This has the added benefit of addressing the label locality and compositionality 
issue discussed in Section 4.3.6. We will establish the solution on BL-Net.

7.1.1 Extended Values

The extended value is defined as a BL-Net value with an additional label set stating which labels are 
unwanted.

\[ V \in X_{value} ::= v/ls \]

In other words, an extended value consists of a bind set, a label set and an unwanted set, in that order. 
Let \( B, L \) and \( U \) be the functions of type \( X_{value} \rightarrow \mathcal{P}(\text{Label}) \) which return the three sets, respectively, 
of the given extended value. Also let the BL-Net value addition + be augmented onto \( X_{value} \), i.e.

\[ v_1/ls_1 + v_2/ls_2 \equiv (v_1 + v_2)/(ls_1 \cup ls_2). \]

An extended value captures values. That a value \( v \) is captured by an extended value \( V \) is written as 
\( v \in V \) and is defined below:

\[ \forall v \in V \quad \text{iff} \quad (B(V) \#(L(V)) \leq v \land (U(V) \cap (L V) = \emptyset), \]

where \( \leq \) is the partial order defined in Definition 5.2.2 on page 80. Informally, if a value \( v \) is captured 
by an extended value \( V \), then they have the same bind set, the label set of \( v \) is a superset of the label 
set of \( V \), and no labels in \( v \) are in the unwanted set of \( V \). The fact that an extended value captures a 
number of values is the key of the compression. We can expand an extended value into a subdomain 
using the definition of capturing, as the function below shows:

\[ \mathcal{X} : X_{value} \rightarrow \text{Subdom}, \]

\[ \mathcal{X}(V) \equiv \{ v \mid v \in V \}. \]

If there is any common label in the label set and the unwanted set, by definition the extended value 
cannot capture any values because it is self-conflicting: it demands that a label is required and unwanted
at the same time. Naturally, a self-conflicting extended value expands to an empty subdomain.

We are now ready to discuss the construction of, and the three operations on, the subdomains in terms of the extended values.

7.1.2 Construction

All subdomains are initially constructed in Definition 5.19 on page 91 as \( \{ v \mid v_0 \leq v \} \), where \( v_0 \) is the box’s declared input. This subdomain requires that all binds and labels in \( v_0 \) are present, but does not require any label to be absent. It is then easy to see that the extended value \( v_0/\emptyset \) expands to the same subdomain.

Proposition 7.1. (Extended value construction.) \( \mathcal{X}(v_0/\emptyset) = \{ v \mid v_0 \leq v \} \).

7.1.3 Intersection

Consider two extended values \( V_1, V_2 \) with the same bind set: \( B V_1 = B V_2 \), and assume that they expand to \( \sigma_1 \) and \( \sigma_2 \), respectively: \( \sigma_1 = \mathcal{X}(V_1) \), \( \sigma_2 = \mathcal{X}(V_2) \). To compress the subdomain intersection \( \sigma_1 \cap \sigma_2 \), we simply need to combine the requirements, stating that the labels from both label sets are required, and that the labels from both unwanted sets are unwanted. This results in the extended value \( V_1 + V_2 \).

Note that if one extended value requires a label which is unwanted by the other extended value, in which case we say that the two extended values conflict with each other, the resulting extended value is self-conflicting and expands to an empty subdomain. Moreover, when the two extended values have different bind sets, they capture different sets of values and also have an empty subdomain intersection.

We summarise these into the following proposition.

Proposition 7.2. (Intersection of extended values.) If \( B V_1 = B V_2 \) and \( L V_1 \cap U V_2 = L V_2 \cap U V_1 = \emptyset \), then \( \mathcal{X}(V_1) \cap \mathcal{X}(V_2) = \mathcal{X}(V_1 + V_2) \), otherwise \( \mathcal{X}(V_1) \cap \mathcal{X}(V_2) = \emptyset \).

7.1.4 Refinement

The subdomain refinement operation is defined as the following calculation in Definition 5.20 on page 92 which involves one of the left operand’s subdomains \( \sigma \), one output choice \( \omega \) associated with \( \sigma \), and the subdomain of a case \( \kappa_2 \) in the right operand’s rep:

\[
\sigma' = \{ v \mid v \in \sigma \wedge (v - \omega \cdot d + \omega \cdot a) \in \kappa_2 \cdot s \}.
\]

Assume that \( \sigma \) is compressed into an extended value \( V_1 \) and \( \kappa_2 \cdot s \) into \( V_2 \). We are to compute the extended value for \( \sigma' \). The criteria of \( v \) imply \( v \in V_1 \) and \( v'' \in V_2 \), where \( v'' = v - \omega \cdot d + \omega \cdot a \). It is easy to see that \( v'' \) will contain all binds and labels in

\[
v' = (B V_1) \# (L V_1) - \omega \cdot d + \omega \cdot a,
\]
but no labels in
\[ ls' = U \cup V \cup L(\omega.d) \setminus L(\omega.a). \]

In other words, the range of \( v'' \) can be compressed to the extended value \( v'/ls' \). To narrow the range so it further satisfies \( v'' \in V_s \), we perform an intersection operation between \( v'/ls' \) and \( V_s \). The empty intersection check which takes place during the operation can predict whether \( \sigma' = \emptyset \). During the operation, the labels in \( L V_s \setminus L v' \) are added to the label set, and the labels in \( U V_s \setminus ls' \) are added to the unwanted set of \( v'/ls' \). Had they been in place since \( V_1 \), we would not need any changes to make \( v'' \in V_s \) hold.

The process described above sums up to the following proposition, which also defines the refinement operation on extended values.

**Proposition 7.3.** *(Extended value refinement.)* Consider some \( V_1, V_s \) and \( \omega \). Let

\[
\begin{align*}
\sigma' &= \{ v \mid v \in X(V_1) \land (v - \omega.d + \omega.a) \in X(V_2) \}, \\
v' &= (B V_1 \# L V_1) - \omega.d + \omega.a, \\
ls' &= U \cup V \cup L(\omega.d) \setminus L(\omega.a).
\end{align*}
\]

If \( B v' = B V_s \land L v' \cap U V_s = L V_s \cap ls' = \emptyset \), then \( \sigma' = X(V') \) where

\[ V' = V_1 + \emptyset \# (L V_2 \setminus L v') / (U V_2 \setminus ls') \]

otherwise, \( \sigma' = \emptyset \).

### 7.1.5 Difference

The subdomain difference operation is hidden in Definition 5.24 on page 93 at the following step of calculation:

\[ \sigma = \{ v \mid v \in \kappa_j.s \land (\forall \kappa', \omega' \land v \in \kappa'.s \land \omega' \in \kappa'.o) \land \omega' \in \kappa'.o \} \]

which informally means that \( \sigma \) is \( \kappa_j.s \) excluding the parts that overlap with the subdomain of any \( \kappa' \) from \( \rho_k \) with a used value part \( (\omega', u \ \omega' \in \kappa'.o) \) containing at least the same number of labels as the largest used value part in \( \kappa_j \). We can also calculate \( \sigma \) by \( \kappa_j.s \setminus \kappa_j.s \setminus \kappa_j.s \ldots \), where \( \kappa_j \), \( \kappa_j' \), etc. meet the aforementioned description. The difference operation on subdomains is now revealed.

Let \( \sigma_1 = X(V_1) \) and \( \sigma_2 = X(V_2) \). We will now attempt to find \( \sigma_1 \setminus \sigma_2 \) in terms of \( V_1 \) and \( V_2 \). Apparently, if \( V_1 \) and \( V_2 \) have different bind sets or conflict with each other, then their intersection is empty and \( \sigma_1 \setminus \sigma_2 = X(V_1) \); and if \( V_2 \) captures all values that \( V_1 \) does, i.e. \( (B V_2 \# L V_2) \leq (B V_1 \# L V_1) \land U V_2 \subseteq U V_1 \), then \( \sigma_1 \setminus \sigma_2 = \emptyset \).

The remaining cases are trickier. Take \( V_1 = \emptyset \# 0/0 \) (all values with no binds) and \( V_2 = \emptyset \# a/b/c/d/ \) for example. Their difference is effectively the complement of \( V_2 \) in the universe of values with no binds. Multiple kinds of values fit in the complement of \( V_2 \); those without the label \( \lambda \), or those with a but
follows holds:

\[ \emptyset \# \emptyset / A, \emptyset \# A / B, \emptyset \# A B C / \emptyset, \emptyset \# A B D / C. \]

It is easy to verify that these extended values conflict with each other. This means that the resulting subdomain is partitioned into four parts, each compressible to an extended value.

In general, representing the result of a difference operation between \( V_1 \) and \( V_2 \) requires at most as many extended values as there are labels in the label set and the unwanted set of \( V_2 \). There are at least two ways to accommodate this in the implementation: we can use a set of extended values to compress every subdomain and adapt all three operations to work on sets of extended values, or, since the extended values in this set conflict with each other, implying non-overlapping subdomains, we can use multiple cases with the same output to carry the different parts of the compressed subdomain. Either way, the difference operation potentially enlarges the \( \text{Rep} \) structure, but only linearly to the size of \( \text{Label} \).

We will now formulate the process of finding the extended values that form a partition of the difference between two subdomains compressible to \( V_1 \) and \( V_2 \). It is easy to see that the set of extended values as the result of a difference operation depend on the order of labels in the operand extended values. To allow the calculations below to easily fit into the type inference algorithm \( R \) in Section 5.6.2 on page 90 which have a functional nature, we shall first establish an arbitrary but deterministic ordering on \( \text{Label} \), so the operation below produces the same result every time it is used. In practise, however, the ordering is required not to change only throughout one difference operation. Let \( \text{ls}[i] \), where \( 1 \leq i \leq |\text{ls}| \), denote a singleton set containing just the \( i \)th label in \( \text{ls} \) as per the ordering. We also define \( \text{ls}[i : j] \), where \( 1 \leq \{i, j\} \leq |\text{ls}| \), as the result of \( \bigcup_{i \leq k \leq j} \text{ls}[k] \). Note that if \( i > j \) then \( \text{ls}[i : j] = \emptyset \). The proposition that shows the calculation of the difference between two extended values is as follows.

**Proposition 7.4.** (Difference of extended values.) Consider some \( V_1, V_2 \), and \( \sigma = \mathcal{X}(V_1) \setminus \mathcal{X}(V_2) \). The following holds:

\[
\begin{align*}
&\text{if } B V_1 \neq B V_2 \lor \mathcal{L} V_1 \cap \mathcal{U} V_2 \neq \emptyset \lor \mathcal{L} V_2 \cap \mathcal{U} V_1 \neq \emptyset, \text{ then } \sigma = \mathcal{X}(V_1); \\
&\text{if } B V_2 \# \mathcal{L} V_2 \leq B V_1 \# \mathcal{L} V_1 \land \mathcal{U} V_2 \subseteq \mathcal{U} V_1, \text{ then } \sigma = \emptyset; \\
&\text{otherwise, } \sigma = \bigcup_{V \in V_s} \mathcal{X}(V), \text{ where } \\
&V_s = \text{let } \text{ls}_1 = \mathcal{L}(V_2) \setminus \mathcal{L}(V_1), \text{ ls}_2 = \mathcal{U}(V_2) \setminus \mathcal{U}(V_1) \text{ in } \\
&\{V_1 + \emptyset \# \text{ls}_1[i : i - 1] / \text{ls}_1[i] \mid 1 \leq i \leq |\text{ls}_1|\} \\
&\cup\{V_1 + \emptyset \# (\text{ls}_1 \cup \text{ls}_2[j]) / \text{ls}_2[1 : j - 1] \mid 1 \leq j \leq |\text{ls}_2|\},
\end{align*}
\]

having \( \forall V_3, V_4, V_3, V_4 \in V_s \land V_3 \neq V_4 \implies \mathcal{X}(V_3) \cap \mathcal{X}(V_4) = \emptyset \).

### 7.1.6 Benefits

By replacing the subdomain element in the \( \text{Case} \) structure with an extended value, and adapting the type inference algorithm so the construction of subdomains and the intersection, refinement and difference
operations are substituted with the calculations shown in the four propositions above, we will make the whole \( \text{Rep} \) structure constructable by simple enumeration, enabling efficient implementation of the type system.

Furthermore, as can be seen from the propositions, all three operations on the extended values have the ability to check whether the result is empty before doing the actual calculations. The type system algorithm always checks if a subdomain operation results in an empty set (examples can be found in Definitions 5.22 and 5.24), so it is an added benefit that the extended values provide a more convenient way to detect the empty results.

Most importantly, by using extended values in place of subdomains, we have shown that the BL-Net type inference algorithms actually do not depend on the universal Label set: all computations can be done with a local knowledge of the labels, and as the smaller networks are composed in larger ones, the additional labels are automatically covered thanks to the way the extended values expand to the subdomains.

### 7.2 Prototype Compiler

The prototype compiler takes an S-Net program in a simplified grammar as the input, parses it to an AST, and performs type inference directly on the tree, giving each box definition, topology subexpression and network definition a rep. The compiler type checks each signed network in place before assigning it the inferred rep. A type error is defined as an empty rep, which by definition does not accept any values, indicating that no input record types can be type safe for the corresponding S-Net component. The compiler terminates and reports the type error as soon as one is found, otherwise if the program passes the type check, the compiler halts normally.

#### 7.2.1 Implementation Choices

\( \text{C}^\# \). The type inference algorithm performs many set operations. For quick prototyping, \( \text{C}^\# \) is the chosen language to write the prototype compiler, to take advantage of the comprehensive support for sets in the .NET library and the LINQ language features. For example, the class \( \text{LabelSet} \) supporting the bind set and the label set in a value is backed by a \( \text{SortedSet}\langle\text{char}\rangle \). As another example, to implement the evaluation function \( \mathcal{E} \) (Definition 6.1) in the class \( \text{Rep} \) supporting the rep structure, the following method evaluates an input value with the receiver instance, and returns \( \text{null} \) if the value is outside the domain, or an enumeration of the output values:

```csharp
public IEnumerable<Value> Eval(Value input)
{
    var selectedCases = this.Where(
        kase => kase.Subdomain.Captures(input));
    if (!selectedCases.Any())
    {
        return null;
    }
    // More code here...
}
```
Definition non-terminals:
Program \rightarrow Def^* 
Def \rightarrow \{ \text{Box, Net} \} 
BoxDef \rightarrow
\text{`box' ID (\text{`Map'} ID )' 1:;} 
Map \rightarrow \text{Val} 1:\rightarrow \text{[OutVal]} 
NetDef \rightarrow
\text{`net' ID [Sig] [NetBody]} 
\text{`connect'} \text{Topology (';')}
Sig \rightarrow (\text{Map MoreMap* })' 
MoreMap \rightarrow \text{','} \text{Map} 
NetBody \rightarrow \{ \text{Def*} \}'

Value non-terminals:
Val \rightarrow \text{Labels [HashLabels]} 
Labels \rightarrow \{ \text{Empty, NonEmpty} \} 
EmptyLabels \rightarrow \text{'}0\text{'}
NonEmptyLabels \rightarrow \text{[A-Za-z]*} 
HashLabels \rightarrow \text{#'} \text{Labels} 
OutVal \rightarrow \text{Val MoreOutVal*} 
MoreOutVal \rightarrow \text{'}1\text{' Val} 
Pattern \rightarrow \text{Val [Condition]} 
Condition \rightarrow \text{`if' 1:...'}

Topology non-terminals:
Topology \rightarrow \text{Branch MoreBranch*} 
MoreBranch \rightarrow \text{'}1\text{' Branch} 
Branch \rightarrow \text{Serial MoreSerial*} 
MoreSerial \rightarrow \text{'}...\text{' Serial} 
Serial \rightarrow \text{Unit [Replication]} 
Unit \rightarrow \{ \text{Ref, Filt, Sync, Sub} \} 
RefUnit \rightarrow \text{ID} 
FiltUnit \rightarrow
\text{[FiltOut] Val'} 1: \text{[FiltOut] Val'} 
FiltOut \rightarrow \text{Val MoreFiltOut*} 
MoreFiltOut \rightarrow \text{'}1\text{' Val} 
SyncUnit \rightarrow
\text{[Pattern MorePattern* ]'} 
MorePattern \rightarrow \text{'}1\text{' Pattern} 
SubUnit \rightarrow \text{[\text{Topology} ']'} 
Replication \rightarrow \{ \text{Star, Ex} \} 
StarReplication \rightarrow
\text{'}*\text{' Pattern MorePattern*} 
ExReplication \rightarrow \text{'}1\text{' [A-Za-z-]}

Figure 7.1: Simplified LL(1) grammar of S-Net.

} 
return from kase in selectedCases 
from omega in kase.OutValues 
select input - omega.Deleted + omega.Added;
}

The extension method where which applies to all IEnumerable<T> instances and the from ... select syntax are LINQ features helping to reduce code.

**Simplified grammar.** As a proof of concept, the prototype compiler reads S-Net programs in a simplified grammar, which differs from the full version mainly by the removal of constructs only important at runtime, such as the tag expression. Other changes include a record type syntax closer to the BL-Net value, and the pass-through qualifier '=' for the box input type, as seen in Section 2.2.1 on page 25.

The grammar is shown in Figure 7.1, which is an LL(1) grammar, as required by the parser used in the prototype compiler. A grammar rule in the form

\[ A \rightarrow \{ B, C, \ldots \} \]

introduces choices of the nonterminal \( A \) which are named \( BA, CA \) and so on, i.e. the name of \( A \) is used as the suffix of the choices. Equivalently, the rule above is short for

\[ A \rightarrow BA | CA | \ldots . \]
On the right side of a grammar rule, a nonterminal is surrounded in brackets if it is optional, and is decorated with an asterisk if there can be a list of zero or more elements of the nonterminal in that position. Terminals, other than \textit{ID} for identifiers, are presented in typewriter font. Literal tokens are wrapped in quotation marks, and others in regular expressions.

The simplification can be found in the nonterminal \textit{Val}, which resembles a BL-Net value of the form $ls_1\#ls_2$ rather than an S-Net record type, thus has no means to differentiate a field from a tag, and also in the nonterminal \textit{Pattern}, which can be conditional if followed with the \textit{if} part, but the tag expression is replaced by an ellipsis.

The simplification is in line with the motivation behind the reduction process which turns S-Net programs to BL-Net (see Chapter 3). Because the prototype compiler will not generate code to run the S-Net program, entities important only at runtime can be ignored.

\textbf{In-place type inference.} The prototype compiler performs type inference in-place in the AST that encodes the input S-Net program, rather than by first translating it to several BL-Net programs as discussed in Section 6.2.1. An informal reasoning of the feasibility is given as follows.

The type inference for the unit components — boxes, filters and synchrocells — can be done by composing the type inference algorithm $\mathcal{R}$ on the translation algorithm $\mathcal{T}$, and assigning the inferred reps to the unit components without storing the translations. For each composite component, $\mathcal{T}$ merely maps it to the equivalent BL-Net construct, and $\mathcal{R}$ requires not the operand components, but only their reps (see Definition 5.18), which will be available if the S-Net program is processed in lexical order. This also applies to the parallel replication (see Section 3.2.7), for which $\mathcal{T}$ creates a serial composition whose left operand depends on the root alphabet of the operand component (see Definition 5.8). Theorem 5.35 shows that the root alphabet can be extracted from the rep of the operand component which is available, so the type inference for the parallel replication can also be done in place.

The type inference for the signed network is comprised of the following steps: isolate it to a separate program, type check that program, and then infer the rep using its signature. Type checking a program is in turn comprised of: treat the program as an unsigned network and infer a rep for its topology expression, and check whether the program signature complies with the rep (see Definition 6.2). However, as mentioned in Section 3.2.9 if the S-Net program is processed in lexical order, all definitions required in the isolated program will have been processed. This implies that the topology expression of the signed network will have been assigned a rep, ready to be checked against its signature. As a result, the type inference for the signed network can be simplified to the following steps: process the S-Net program in lexical order up to the topology expression of the signed network to obtain a rep for it, then check if the signature of the signed network complies with said rep, and if so, infer a new rep for the signed network as per the translation defined in Section 3.2.9. These can all be done in place, without having to actually isolate the signed network to a separate program.

The sample outputs of the prototype compiler in Appendix C.2 record the type inference process, and can indirectly show that only one pass is needed to type check the whole program, confirming the possibility to perform in-place type inference.
Table 7.1: Type inference class library.

<table>
<thead>
<tr>
<th>Function</th>
<th>Definition</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R^0 )</td>
<td>5.19</td>
<td>Box()</td>
</tr>
<tr>
<td>( R^1 )</td>
<td>5.20</td>
<td>Serial3()</td>
</tr>
<tr>
<td>( R^2 )</td>
<td>5.21</td>
<td>Serial2()</td>
</tr>
<tr>
<td>( R^3 )</td>
<td>5.22</td>
<td>Serial1(). CartesianProduct()</td>
</tr>
<tr>
<td>( R^4 )</td>
<td>5.23</td>
<td>Serial()</td>
</tr>
<tr>
<td>( R^5 )</td>
<td>5.24</td>
<td>Parallel(). ParallelCasesFromBranchI(). Reduce(). ParallelCasesFromBothBranches()</td>
</tr>
<tr>
<td>( R^6 )</td>
<td>5.30</td>
<td>Star()</td>
</tr>
<tr>
<td>( R^7 )</td>
<td>5.8</td>
<td>Root()</td>
</tr>
<tr>
<td>( R^8 )</td>
<td>5.9</td>
<td>Tagger()</td>
</tr>
<tr>
<td>( R^9 )</td>
<td>5.10</td>
<td>Passer()</td>
</tr>
<tr>
<td>( R^{10} )</td>
<td>5.11</td>
<td>Finalizer()</td>
</tr>
<tr>
<td>( (Direct\ from\ translation) )</td>
<td></td>
<td>Sync(), Ex(), Net()</td>
</tr>
</tbody>
</table>

Table 7.2: Type inference functions and methods in TypeInference class.

### 7.2.2 Algorithm

The prototype compiler has a standard architecture: a lexer reads and tokenises the input S-Net program, a parser builds the AST from the tokens, and a type checker traverses the AST in a preorder fashion to process the program in lexical order. We shall then omit the implementation details of these standard components.

The type checker is supported by a library of classes implementing the type inference algorithm \( \mathcal{R} \) and the rep structure. Table 7.1 lists all classes in the library, matching each of them to the BL-Net definition that it implements. Note that the subdomain structure Subdom in BL-Net has been replaced by the extended value Xvalue, introduced in Section 7.1. A sample of class members is also included in the table.

In order to produce sorted outputs, the class LabelSet wraps a sorted set of characters, Case contains a sorted set of OutValues, and Rep is a sorted set of Cases. Except TypeInference which is a utility class, other classes implement the interfaces IComparable\<T\> and IComparable\<T\>, wherever appropriate, to enable sorting, and IEnumerable\<T\>, to support enumerating the elements. However, now implementing the extended value, the class Subdomain is not an enumerable of Values. Coding these classes are straightforward, so we will skip their details for succinctness.
The utility class `TypeInference` contains the implementation of all subfunctions used by the type inference algorithm $R$. Table 7.2 maps each subfunction to its implementation in the class. Taking into account that the subdomains can be replaced by the extended values, these subfunctions mainly involve simple calculations and enumeration on sets, with the exception of $R^*$ which is presented in pseudo-code. This implies that it is also relatively easy to translate the definitions to code, taking advantage of the powerful .NET library and the LINQ feature. The complete code of this class is presented in Appendix C.1 for reference.

### 7.3 On Routing Information

The new type system assigns attractions to each parallel branch, which are subdomain-integer pairs, or can be implemented as extended value-integer pairs. The integer, called the `rank`, denotes the level of attractiveness of the subdomain or extended value to a candidate input record. However, the S-Net runtime, designed at the same time as the previous type system, only recognises routing information as a set of simple record types, and uses the concept of better matching. In other words, the unwanted sets in the extended values, and the ranks, do not exist in the routing information. In an ideal situation, the runtime should also be upgraded to recognise the attraction structure, but in this section, we will informally discuss some methods to craft the routing information so that the runtime behaves correctly as if it can read the attractions.

#### 7.3.1 Unwanted Labels

If the extended value of an attraction has a non-empty unwanted set, the subdomain which it represents excludes the values with any number of unwanted labels. Some of these values may have to be delivered to the other branch according to the attraction sets of both branches. For them, we can insert virtual attractions to the attraction set of the other branch, without changing the runtime behaviour.

Take, for example, the S-Net program in Section 6.3 which is a parallel composition of trouble and a filter. As listed in Table 6.1, the attraction set of trouble is \{`(ce/ax, 2)`\}, and the attraction set of the filter is \{(\(\lambda/\emptyset\), 1)`\}. Once again, the prefix `\#` denoting the empty bind set is omitted for succinctness. The extended value `ce/ax` does not capture the values in \(\{v \mid ace \leq v\}\), but these values can be accepted into the filter, which attracts \(\lambda/\emptyset\) albeit with a lower rank. We can then insert a virtual attraction \(\alpha_1 = (\text{ace}/\emptyset, 3)\) into the attraction set of the filter, so that the values in question will be delivered to the filter, even if the unwanted sets are ignored.

For the label `x`, we could also add a virtual attraction `(ace\#x/\emptyset, 4)` for the filter, where the label `\text{a}` is present as the filter’s own requirement. However, this is unnecessary because its purpose is covered by the previously added virtual attraction \(\alpha_1\).

When all extended values with unwanted labels are processed as above, the unwanted sets are no longer necessary. We now continue onto removing the ranks from the attractions.
7.3.2 Ranks

Due to Lemma 5.7 on page 82, the rank of an attraction, which is in fact the size of the used value part, will never be greater than the number of labels in the label set of the coupled extended value. For a typical attraction, the rank equals to the size of the label set. If this is the case, all labels are used, and the BL-Net value part of the extended value can be directly used as an input type in the routing information. When the rank is less than the size of the label set, we can again use virtual attractions to achieve the expected behaviours.

In particular, if said attraction attracts all the values captured by its subdomain better than the other branch, albeit with a smaller rank, we can simply pretend that all labels in the label set are used. On the other hand, if the other branch attracts all these values better, then said attraction can be removed. For every other situation, there is at least the brute-force way to insert as many virtual attractions as necessary to both branches’ attraction sets, covering every value individually, but cleverer algorithms do exist.

As an example, consider a parallel composition whose left branch is assigned the attraction set \(\alpha_2 = \{ (abc/\emptyset, 2) \} \) and right branch \(\alpha_3 = \{ (bcd/\emptyset, 3) \} \). The values in \(\{ v \mid abc \leq v \} \) should be attracted to the right branch, but if the ranks are ignored, they would be also eligible for the left branch. To correctly route these values, we add the virtual attraction \( (abc/\emptyset, 4) \) to \(\alpha_3 \). It is easy to verify that the following pair of attraction sets can replace the original pair without changing the routing behaviour:

\[ \alpha_2' = \{ (abc/\emptyset, 3) \}, \alpha_3' = \{ (bcd/\emptyset, 3), (abcd/\emptyset, 4) \}. \]

After the modifications above targeting the unwanted labels and the ranks, the additional elements of attractions, comparing to the routing information, can all be safely removed from the data that supports the parallel composition’s operations. What remains are a set of BL-Net values per parallel branch, resembling the data structure the existing S-Net runtime can handle. By using the sets of BL-Net values as the routing information per parallel branch, the current runtime can correctly route the input records for all parallel compositions as per the new semantics biPar on page 99. It is worth noting that these modifications are all local to individual parallel compositions; they do not affect how the outer network accepts or attracts the input records.

7.4 Summary

In this chapter, we have introduced a prototype implementation of the new type system. We do this by first proposing a method to compress the subdomains in the rep structure, which in the native BL-Net type system is constructed by set comprehension, requiring the knowledge of the complete Label set. The compression requires extending the value with an unwanted set. The extended value creates a correct representation of the original subdomain with only the local knowledge of the Label set, and turns set comprehension to simple enumeration. This makes the type inference algorithm implementation-friendly.

We have also confirmed the feasibility to skip the translation step which turns an S-Net program to a number of BL-Net programs, and work on the S-Net program directly. This allows for a prototype
compiler implementation which type checks an S-Net program in a single pass.

The new design, however, has created a gap between formalisation and implementation: the existing S-Net runtime does not understand the attraction structure which supports the parallel composition’s semantics. For this, we have discussed a method involving virtual attractions, for the existing runtime to operate using the new semantics.
Chapter 8

Conclusion

The previous type system for S-Net needs correction, it cannot always fulfill its main purpose: to detect 'stuck' executions in S-Net programs. There has been a number of attempts to fix it, but the type-semantics interdependency in S-Net since its infancy has been making the task difficult. The type-semantics interdependency can be found in the semantics of the parallel composition, which routes its input records based on the information given by the type inference. Because of this, the S-Net semantics is unstable, and cannot be used as the foundation for proofs of properties of the type system.

As an answer, in this thesis we have introduced a new type system for S-Net in terms of a new language called BL-Net, which is reduced from S-Net by removing the features unrelated to type safety. Such features include the data stored under the labels in the records, multiple output records from a component, ordering of these records and so on. We have defined a formal translation process from S-Net to BL-Net.

The specification of BL-Net is stable; no algorithm depends on the type inference. We have removed the type-semantics interdependency using the concept that components use labels, and that the parallel composition should favor the branch potentially using more labels from the input record. However, the parallel composition semantics defined in this way becomes difficult to implement. We have compensated this by introducing the semantics for implementation, which reads information from the types.

We have designed the BL-Net type system with soundness and completeness proofs. The type system uses the representative function, or rep, as the network type. A representative function captures a BL-Net network's full behavior for all values it accepts. A rep is a structure to encode a representative function. Using the rep, we can efficiently detect stuck executions, thus type check a BL-Net program. Thanks to the soundness and completeness of the type system, the semantics for implementation can also be proved correct.

As a side note, a big portion of the BL-Net concepts are introduced in terms of L-Net, a more abstract language with fewer features than BL-Net. In particular, the bind, the sink and the serial replication are absent in L-Net. We have used L-Net as a stepping stone to understand BL-Net.

We have compared the new type system for S-Net, which is defined using BL-Net concepts and its sound and complete type system, with the previous, published version. The new type system produces
slightly more complex, but most importantly correct, type inference results. As a proof of concept, we have also created a prototype compiler implementing the new type system. The compiler works on the S-Net program directly, without first translating it to BL-Net.

There is a second purpose of the type system: to support the parallel composition’s semantics at runtime. The BL-Net semantics for implementation is a perfect guideline, but it uses a structure incompatible with the existing runtime architecture. We have included a discussion to bridge the incompatibility, allowing the new type system to be used in the existing runtime.

8.1 Future Work

It has been the case that the S-Net language and the type system design evolve together, so the future work for this thesis naturally follows the same goal. In Section 6.4 we discussed that the new type system did not include a new member of S-Net components: the initialiser box, and could not describe the full semantics of the synchrono cell, and that the latter limitation could be tackled with multiplicity. We now briefly discuss the possible directions for these tasks.

8.1.1 Incorporating Initialiser Boxes

An initialiser box is defined using a syntax similar to a box but without the input type:

\[ \text{box init } ( \rightarrow (a,b)) \].

Unlike other components which are executed only in response to incoming records, an initialiser box is executed automatically after the runtime creates an instance in the topology, or in response to the initialisation control record (carrying signals instead of data). The computational function, \( \text{init} \) in this example, is invoked to produce the output record or records of type \( \{a,b\} \), which will then go through the normal record delivery and processing procedures. The type system should make sure that these records cannot cause any type error.

For a component with one or more initialiser boxes in its topology, if not all output records from the initialiser boxes are sunk within the component, we will observe that the component itself reacts to the initialisation signal and produces output records. It is therefore logical to consider every component to have the ability to produce zero or more records on initialisation. To correctly perform type check for these records, we need to store their record types in the component’s type inference result, i.e. the representative function, or the rep structure. We can consider adding a special mapping in the representative function, that maps the special input, the initialisation control record, to a set of record types which the component may produce on initialisation. We tentatively call the latter the initialisation response, or IR set, of the component. By definition, the IR set of a box, a filter, or a synchrocell is empty, and the IR set of an initialiser box is its output record type set.

The type inference then performs largely similar to the algorithm outlined in this thesis. The IR set of a serial composition is the union of the IR set of the right operand, and the image of the left operand’s IR set under the right operand’s representative function. The IR set of a parallel composition is the
union of the IR sets of the two branches. The IR set of a serial or parallel replication can be constructed using the model of the replication. If an IR set cannot be inferred using these rules, there is a type error.

From the discussion above, it is clear that incorporating the initialiser boxes to the type system is relatively easy. However, this does involve a new type of execution, and the operational semantics needs updating, as well as the proofs.

### 8.1.2 Multiplicity

The current type system, and in fact the language syntax, allows us to describe the record types, but not their quantity. Multiplicity is the extension to the type system with which we can specify how many records of a certain type is produced by a component. Depending on its complexity, the extension can improve the expressiveness of the component signatures to a variable extent. For example, the following box

```plaintext
box A ( (a) -> (b)^1 );
```

declares that the computational function A will only produce one record of type {b} per input record of type {a}. The following box

```plaintext
box B ( (a) -> (b)^1 | (c)^0..2 );
```

says that the function B will either produce one record of type {b}, or zero to two records of type {c}. The filter

```
[ {a,b,c} -> {a};{a};{b};{c} ]
```

will have an inferred signature

```
{a,b,c} -> {a}^2;{b}^1;{c}^1
```

with its self-explanatory meaning, and finally, if the extension covers input multiplicity, the sync-star

```
[ | {a}, {b} ] * {a,b}
```

can have an inferred signature

```
{a};{b} -> {a,b}^1
```

describing that the inputs of type {a} and type {b} will be paired to produce one {a,b} per pair.

It is easy to see that the last example above could allow us to infer the network progressiveness by checking whether both input types exist for the synchrocell, but the complexity introduced is beyond a simple extension to the type system in this thesis.
8.2 Closing Remarks

This thesis has demonstrated the importance of having a calculus proven to be correct to support a programming language, no matter how small it is. Originally S-Net was designed and partially implemented (with the type inference part left blank) without a formal type system, and the design flaws and the resulting unintended program behaviours only started to emerge when we began the type system formulation. Thankfully, as the formulation process went side by side with the language’s development, many design flaws were rectified timely. Now that we have a correct type system for S-Net, provided the missing compiler implementation is done, we can be confident that the language will be well behaved, as demonstrated by the mathematical proofs.

However, this correctness comes with a price. We have made many compromises on the language features in favour of a correct mathematical model, and it seems that we have slightly departed from the original design principle of S-Net – ease of use. The idea of record types with flow inheritance that S-Net adds to the data turns out to be a double-edged sword: the convenience that the components need not process the whole incoming record is somewhat cancelled by the necessity of a complex type system. Although the new type system in this thesis will always predict the correct resulting record types for each component, if any such result is unexpected by the programmer, they will need to know exactly how S-Net operates before being capable of fixing the program. The author feels that the complexity in the record routing by type, which is only properly formulated in this thesis, will create a steep learning curve for potential S-Net users. If the type system were formulated at an early stage of the language design, we may have been able to consider alternatives to record types and flow inheritance, making S-Net truly easy to learn and easy to use.
Appendix A

Proofs

This appendix contains the detailed proofs of the lemmas and theorems in the thesis body with proof summaries that redirect the readers here. Each proof in this appendix is accompanied with a reprise of the lemma or theorem, which may be a trivial logical equivalence, so that the premises given in the lemma or theorem can be numbered. In a proof body, a line may begin with one of these items:

- \([\text{reason, reason...}]\)
  The statement that follows the square-bracketed list is deduced from the reasons in the list. A reason may be a number referring to a previous statement, a number qualified by chapter referring to a definition, lemma or theorem, a semantic rule name, or a short phrase in natural language describing other reasons.

- THEN
  The following statement is deduced from the previous statement.

- LET var, var...
  The listed variables are instantiated with certain values that make the statement following the colon hold. The context will make it clear that such values are guaranteed to exist. Note that when the variables need to be instantiated from an \(\exists\) quantified statement with no or little change to the statement itself, the LET line is implicit. For example, when \(\exists x. p(x)\) is deducible from the context, the line may read \(p(x)\) to implicitly mean \(\exists x. p(x)\) followed by LET \(x: p(x)\).

- ASSUME
  The following statement is assumed to hold.

- AND
  The following statement share the same context (a reason list, THEN, LET or ASSUME) as the previous statement, and they both hold.

- OR
  The following statement and the previous one, together with those in the OR lines that follow
immediately, form a disjunctive statement set in the same context, among which at least one will hold at a time. This signals the start of a subproof by cases.

- **(heading)**
  The line marks the beginning of a subproof. If the heading is about a case of a structural induction, the statement that follows shows the structure of the inductive variable. If it is about a case in a subproof by cases, the corresponding case mentioned in the heading is assumed to hold (implicit ASSUME line).

- IH
  The line contains an inductive hypothesis.

- other text
  The line is the start of a paragraph of explanations in natural language.

Statements which need to be referenced in the reason lists are numbered uniquely within a proof. The numbers are bracketed and placed to the right of the statements. If a statement holds under an assumption, the current statement number will be followed by ‘/’ and the assumption statement number. Multiple assumptions are represented by stacking the numbers; for example, (35/24/13) means that the current statement, numbered 35, is under the assumption numbered 24, which in turn is under the assumption numbered 13.

### A.1 Lemma 4.8

Given \( n \lessdot v \),

\[ (1) \]

to show \( n, v \leadsto_\cdot \cdot \cdot \).

\[ \text{(Goal)} \]

**Proof.** By structural induction on \( n \).

**Base Case** \( n = \text{box } v_0 \rightarrow v_{s_0} \)

\[ (2) \]

\[ 1, 2, \text{Lem 4.7} \quad v_0 \leq v \]  \[ (3) \]

\[ 2, \text{L-Net box requirement} \quad v_1 \in v_{s_0} \]  \[ (4) \]

\[ 3, 4, \text{IsBox} \quad n, v \leadsto_\cdot \cdot \cdot \]  \[ \text{(Goal)} \]

**Inductive Case 1** \( n = n_1 \cdots n_2 \)

\[ (5) \]

\[ \text{IH} \quad n_1 \lessdot v \implies n_1, v \leadsto_\cdot \cdot \cdot \]  \[ (6) \]

\[ \text{IH} \quad n_2 \lessdot v_1 \implies n_2, v_1 \leadsto_\cdot \cdot \cdot \]  \[ (7) \]

\[ 1, 5, \text{Lem 4.7} \quad n_1 \lessdot v \]  \[ (8) \]

\[ \text{AND} \quad \forall v_1. \quad n_1, v \leadsto_\cdot \cdot \cdot , v_1 \implies n_2 \lessdot v_1 \]  \[ (9) \]

\[ 8, 6 \quad n_1, v \leadsto_\cdot \cdot \cdot , v_1 \]  \[ (10) \]
\[ n_1 \parallel n_2 \quad (\text{Goal}) \]

**Inductive Case 2** \( n = n_1 \parallel n_2 \)

IH \( n_1 \not\models v \Rightarrow n_1, v \not\models s, u, v \quad (i \in \{1, 2\}) \) \( (11) \)

\[ n_1 \not\models v \Rightarrow n_1 \not\models v \quad (i \in \{1, 2\}) \quad (12) \]

Lem 4.7 \( n_1 \not\models v \lor n_2 \not\models v \quad (13) \)

THEN \( j, k \in \{1, 2\} \land j \neq k \land n_j \not\models v \land n_k, v \not\models s \)

OR \( n_1 \not\models v \land n_2 \not\models v \quad (14a) \)

**Case 14a** \( n_j, v \not\models s, u, v \quad (15) \)

\( n_1 \not\models v \lor n_2 \not\models v \quad (14b) \)

**Case 14b** \( n_i, v \not\models s, u, v \quad (16) \)

THEN \( u_{s_i} = \{ u \mid n_i, v \not\models s, u, v \} \neq \emptyset (i \in \{1, 2\}) \) \( (17) \)

\( u_{\text{max}_i}, u_{\text{max}_i} \in u_{s_i} (i \in \{1, 2\}) \) \( (18) \)

\( \forall u_i, n_i, v \not\models s, u_{\text{max}_i}, v \quad (i \in \{1, 2\}) \) \( (19) \)

\( \land \exists u_j, n_j, v \not\models s, u_j, v \land |u_{\text{max}_j}| \geq |u_i| (i \in \{1, 2\}) \) \( (20) \)

\( \land u_{\text{max}_j}, u_{\text{max}_j} \in u_{s_j} (j \in \{1, 2\}) \) \( (21) \)

\( \land j, k \in \{1, 2\} \land j \neq k \land \exists u_j, n_j, v \not\models s, u_j, v \land |u_j| \geq |u_k| \) \( (22) \)

\( \land u_k, n_k, v \not\models s, u_k, v \quad (23) \)

\( \land \exists u_j, n_j, v \not\models s, u_j, v \land |u_j| \geq |u_k| \) \( (24) \)

\( \land u_k, n_k, v \not\models s, u_k, v \quad (25) \)

(\text{Goal}) \( (14a, 14b, 16, 25) \)

**A.2 Lemma 4.9**

Given \( n, v \not\models s, u, f, v' \),

\[ u \leq v \land f \leq v \land f \leq f' \land u \times f = \emptyset. \quad (\text{Goal}) \]

**Proof.** By structural induction on \( n \).

**Base Case** \( n = \text{box} \ v_0 \Rightarrow v s_0 \)

\[ \quad (2) \]

[2, lsBox] \( v_1 \in v s_0 \)

129
\[
\begin{align*}
\text{AND } & v_0 \leq v \\
\text{AND } & u = v_0 \\
\text{AND } & f = v - v_0 - v_1 \\
\text{AND } & v' = v - v_0 + v_1 \\
\text{[3, 4]} & u \leq v \\
\text{[5]} & f \leq v \\
\text{[5, 6]} & f = v' - v_1 \\
\text{THEN } & f \leq v' \\
\text{[4, 5]} & u \times f = \emptyset \\
\text{[7, 8, 9, 10]} & u \leq v \wedge f \leq v \wedge f \leq v' \wedge u \times f = \emptyset \tag{Goal}
\end{align*}
\]

**Inductive Case 1** \( n = n_1 \cdots n_2 \)

\[
\begin{align*}
\text{IH } & n_1, v \leadsto u_1, f_1, v_1 \implies u_1 \leq v \wedge f_1 \leq v \wedge f \leq v_1 \wedge u_1 \times f_1 = \emptyset \\
\text{IH } & n_2, v_1 \leadsto u_2, f_2, v' \implies u_2 \leq v_1 \wedge f_2 \leq v_1 \wedge f \leq v' \wedge u_2 \times f_2 = \emptyset \\
\text{[11, IsSeq]} & n_1, v \leadsto u_1, f_1, v_1 \\
\text{AND } & n_2, v_1 \leadsto u_2, f_2, v' \\
\text{AND } & u = u_1 + (f_1 \times u_2) \\
\text{AND } & f = f_1 \times f_2 \\
\text{[14, 12]} & u_1 \leq v \\
\text{AND } & f_1 \leq v \\
\text{AND } & f \leq v_1 \\
\text{AND } & u_1 \times f_1 = \emptyset \\
\text{[15, 13]} & u_2 \leq v_1 \\
\text{AND } & f_2 \leq v_1 \\
\text{AND } & f_2 \leq v' \\
\text{AND } & u_2 \times f_2 = \emptyset \\
\text{[19]} & f_1 \times u_2 \leq v \\
\text{[18, 26, 16]} & u \leq v \\
\text{[17, 19]} & f \leq v \\
\text{[17, 24]} & f \leq v' \\
\text{[21]} & u_1 \times f_1 \times f_2 = \emptyset \\
\text{[25, set intersection equivalence]} & u_2 \times f_2 \times f_1 \times f_1 = \emptyset
\end{align*}
\]
\[ (u_1 + (f_1 \times u_2)) \times (f_1 \times f_2) = \emptyset \]  
(32)

\[ u \times f = \emptyset \]  
(33)

\[ u \le v \land f \le v' \land u \times f = \emptyset \]  
(Goal)

(Inductive Case 2) \( n = n_1 \parallel n_2 \)  
(34)

IH \( n_j, v \leadsto_s u, f, v' \implies u \le v \land f \le v' \land u \times f = \emptyset \) \( (j \in \{1, 2\}) \)  
(35)

\[ u \le v \land f \le v \land f \le v' \land u \times f = \emptyset \]  
(36)

\[ u \le v \land f \le v \land f \le v' \land u \times f = \emptyset \]  
(Goal)

\[ \Box \]

A.3 Theorem 4.19

To prove \( R(n) \) is well formed for every \( n \), we split the proof into two parts, each covering one wellformedness constraint.

**Subdomain non-intersection.** To show \( \forall \kappa_1, \kappa_2, \kappa_1, \kappa_2 \in R(n) \land \kappa_1 \neq \kappa_2 \implies \kappa_1.s \cap \kappa_2.s = \emptyset \) (Goal)

**Proof.** By structural induction on \( n \).

(Base Case) \( n = \text{box} \ v_0 \to v_{s_0} \)  
(1)

[Def 4.12, 1, Def 4.13] \( R(n) \) has only one case  
(2)

THEN \( \neg \exists \kappa_1, \kappa_2, \kappa_1, \kappa_2 \in R(n) \land \kappa_1 \neq \kappa_2 \)  
(3)

THEN \( \forall \kappa_1, \kappa_2, \kappa_1, \kappa_2 \in R(n) \land \kappa_1 \neq \kappa_2 \implies \kappa_1.s \cap \kappa_2.s = \emptyset \)  
(Goal1)

(Inductive Case 1) \( n = n_1 \cdot n_2 \)  
(5)

The variables used in this part of the proof can have up to 3 integral subscripts. The first subscript indicates the variable’s relation with either \( \kappa_1 \) or \( \kappa_2 \) in the lemma. The second subscript indicates the variable’s relation with either \( n_1 \) or \( n_2 \) in (5). The third subscript differentiates different variables when the first two subscripts are the same. When a subscript must be fixed while its preceding subscripts are free, 0 is used as the prefix.

IH \( \forall \kappa_{011}, \kappa_{012}, \kappa_{011}, \kappa_{012} \in R(n_1) \land \kappa_{011} \neq \kappa_{012} \implies \kappa_{011}.s \cap \kappa_{012}.s = \emptyset \)  
(6)

IH \( \forall \kappa_{021}, \kappa_{022}, \kappa_{021}, \kappa_{022} \in R(n_2) \land \kappa_{021} \neq \kappa_{022} \implies \kappa_{021}.s \cap \kappa_{022}.s = \emptyset \)  
(7)

ASSUME \( \kappa_1, \kappa_2 \in R(n) \land \kappa_1 \neq \kappa_2 \)  
(8)

THEN \( \kappa_i \in R(n) \ (i \in \{1, 2\}) \)  
(9/8)

Note: if a statement is followed by \( (i \in \{1, 2\}) \), the statement will be true for both \( i \).
AND $\kappa_1 \neq \kappa_2$ (10/8)

[Def 4.12 5, Def 4.17 9] $\kappa_{i1} \in \mathcal{R}(n_1) \land \kappa_i \in \mathcal{R}_i^2(\kappa_{i1}, \mathcal{R}(n_2))$ ($i \in \{1, 2\}$) (11/8)

[tautology] $\kappa_{i1} = \kappa_{21}$ (12a/8)

OR $\kappa_{i1} \neq \kappa_{21}$ (12b/8)

(Case 12a)

[11, 12a, Def 4.16] $\kappa_i = (\bigcap_{j=1}^{\lvert \kappa_{i1}\rvert} \kappa_{ij}^{\prime}, s, \bigcup_{j=1}^{\lvert \kappa_{i1}\rvert} \kappa_{ij}^{\prime}, o) \ (i \in \{1, 2\})$ (13/12a/8)

AND $\forall j. j \in 1..\lvert \kappa_{i1}\rvert \implies \kappa_{ij}^{\prime} \in \mathcal{R}_2^3(\kappa_{i1}, s, \omega_{ij}, \mathcal{R}(n_2))$ ($i \in \{1, 2\})$ (14/12a/8)

[13, 10] $j \in 1..\lvert \kappa_{i1}\rvert$ (15/12a/8)

AND $\kappa_{ij}^{\prime1} \neq \kappa_{ij}^{\prime2}$ (16/12a/8)

[15, 14, Def 4.15] $\kappa_{i2} \in \mathcal{R}(n_2)$ ($i \in \{1, 2\}$) (17/12a/8)

AND $\kappa_{ij}^{\prime} \in \mathcal{R}_2^3(\kappa_{i1}, s, \omega_{ij}, \kappa_{i2})$ ($i \in \{1, 2\}$) (18/12a/8)

[16, 18, Def 4.14] ($\mathcal{R}_3^3$ returns a rep with at most 1 case) $\kappa_{i2} \neq \kappa_{22}$ (19/12a/8)

[19, 17, 7] $\kappa_{12} = \emptyset$ (20/12a/8)

[20, function property] $\{ v \mid f_0(v) \in \kappa_{12}, s \} \cap \{ v \mid f_0(v) \in \kappa_{22}, s \} = \emptyset$ (21/12a/8)

where $f_0(v) = v - \omega_{ij}, d + \omega_{ij}, a$

[21, 18, Def 4.14] $\kappa_{ij}^{\prime1} \cap \kappa_{ij}^{\prime2} = \emptyset$ (22/12a/8)

THEN $(\kappa_{ij}^{\prime1} \cap s_1) \cap (\kappa_{ij}^{\prime2} \cap s_1) = \emptyset$ (23/12a/8)

(Case 12b)

[11, 12b, 6] $\kappa_{i1} \cap \kappa_{21} = \emptyset$ (24/12b/8)

[11, Def 4.16] $\kappa_i = (\bigcap_{j=1}^{\lvert \kappa_{i1}\rvert} \kappa_{ij}^{\prime}, s, \bigcup_{j=1}^{\lvert \kappa_{i1}\rvert} \kappa_{ij}^{\prime}, o) \ (i \in \{1, 2\})$ (25/12b/8)

AND $\forall j. j \in 1..\lvert \kappa_{i1}\rvert \implies \kappa_{ij}^{\prime} \in \mathcal{R}_2^3(\kappa_{i1}, s, \omega_{ij}, \kappa_{21})$ ($i \in \{1, 2\}$) (26/12b/8)

[26, Def 4.15] $\forall j. j \in 1..\lvert \kappa_{i1}\rvert \implies \kappa_{ij}^{\prime} \in \mathcal{R}_3^3(\kappa_{i1}, s, \omega_{ij}, \kappa_{21})$ ($i \in \{1, 2\}$) (27/12b/8)

[27, Def 4.14] $\forall j. j \in 1..\lvert \kappa_{i1}\rvert \implies \kappa_{ij}^{\prime} = \kappa_{i1} \cap s$ ($i \in \{1, 2\}$) (28/12b/8)

THEN $\forall j. j \in 1..\lvert \kappa_{i1}\rvert \implies \kappa_{ij}^{\prime} \subseteq \kappa_{i1}$ ($i \in \{1, 2\}$) (29/12b/8)

[29, 25] $\kappa_{i1} \subseteq \kappa_{i1}$ ($i \in \{1, 2\}$) (29/12b/8)

[29, 24] $\kappa_{i1} \cap \kappa_{22} = \emptyset$ (30/12b/8)

[12a, 12b, 30] $\kappa_{i1} \cap \kappa_{21} = \emptyset$ (31/8)

[8, 31] $\kappa_1, \kappa_2 \in \mathcal{R}(n) \land \kappa_1 \neq \kappa_2 \implies \kappa_1 \cap \kappa_2 = \emptyset$
THEN \( \forall \kappa_1, \kappa_2, \kappa_1, \kappa_2 \in \mathcal{R}(n) \land \kappa_1 \neq \kappa_2 \implies \kappa_1.s \cap \kappa_2.s = \emptyset \)  

(\textbf{Inductive Case 2}) \( n = n_1 \parallel n_2 \)  

The subscript scheme in this part of the proof is as follows. For a case or output choice, the same 3-tier subscripts as in Inductive Case 1 are used. For integer variables \( j \) and \( k \), the first subscript makes the connection with \( \kappa_1 \) or \( \kappa_2 \), the variable itself establishes the relation with \( n_1 \) or \( n_2 \) (\( j \) the chosen branch and \( k \) the other branch), and a second subscript is to distinguish between different variables with the same name and first subscript.

IH \( \forall \kappa_{011}, \kappa_{012}, \kappa_{011}, \kappa_{012} \in \mathcal{R}(n_1) \land \kappa_{011} \neq \kappa_{012} \implies \kappa_{011}.s \cap \kappa_{012}.s = \emptyset \)  

IH \( \forall \kappa_{021}, \kappa_{022}, \kappa_{021}, \kappa_{022} \in \mathcal{R}(n_2) \land \kappa_{021} \neq \kappa_{022} \implies \kappa_{021}.s \cap \kappa_{022}.s = \emptyset \)  

\( \text{ASSUME} \ k_1, k_2 \in \mathcal{R}(n) \land k_1 \neq k_2 \)  

[35, Def 4.18] \( k_1, k_2 \in \rho' \) where \( \rho' \) is as in Def 4.18  

OR \( k_1, k_2 \in \rho'' \) where \( \rho'' \) is as in Def 4.18  

OR \( k_1 \in \rho' \land k_2 \in \rho'' \)  

OR \( k_1 \in \rho'' \land k_2 \in \rho' \)  

(\textbf{Case 36a/35})

\( \text{LET} \ j_1, k_1, k_{1j1}, j_2, k_2, k_{2j2}; j_1, k_1 \in \{1, 2\} \land j_1 \neq k_1 \)  

\( \text{AND} \ k_{1j1} \in \mathcal{R}(n_{j1}) \)  

\( \text{AND} \ k_1.s = \{v \mid v \in k_{1j1}.s \land (\forall \kappa', \kappa' \in \mathcal{R}(n_{k_1}) \land v \in k'.s \implies \exists \omega_{1j1}. \omega_{1j1}, k_{1j1}, o \land \forall \omega', \omega' \in k'.o \implies |\omega_{1j1}.u| > |\omega'.u| \} \neq \emptyset \)  

\( \text{AND} \ j_2, k_2 \in \{1, 2\} \land j_2 \neq k_2 \)  

\( \text{AND} \ k_{2j2} \in \mathcal{R}(n_{j2}) \)  

\( \text{AND} \ k_2.s = \{v \mid v \in k_{2j2}.s \land (\forall \kappa', \kappa' \in \mathcal{R}(n_{k_2}) \land v \in k'.s \implies \exists \omega_{2j2}. \omega_{2j2}, k_{2j2}, o \land \forall \omega', \omega' \in k'.o \implies |\omega_{2j2}.u| > |\omega'.u| \} \neq \emptyset \)  

[tautology] \( j_1 = j_2 \)  

OR \( j_1 \neq j_2 \)  

(\textbf{Case 43a/36a/35})

\( [43a, 37, 40] k_1 = k_2 \)  

\( [43a, 44, 39, 42, 35] k_{1j1} \neq k_{2j2} \)  

\( [45, 38, 41, 33, 34] k_{1j1}.s \cap k_{2j1}.s = \emptyset \)  

\( [46, 39, 42] k_1.s \cap k_2.s = \emptyset \)
(Case 43b/36a/35)

\[ j_1 = k_2 \land j_2 = k_1 \]  
\[ \text{ASSUME } v \in \kappa_1.s \land v \in \kappa_2.s \]

\[ \forall \kappa'. \kappa' \in R(n_{k1}) \land v \in \kappa'.s \implies \exists \omega_{j1}. \omega_{j2} \in \kappa_1.o \land \]
\[ \forall \omega'. \omega' \in \kappa'.o \implies |\omega_{j1}.u| > |\omega'.u| \]

\[ v \in \kappa_{2j2}.s \]

\[ \forall \omega'. \omega' \in \kappa_{2j2}.o \implies |\omega_{j1}.u| > |\omega'.u| \]

\[ \omega_{j1} \in \kappa_{1j}.o \]

\[ \forall \omega'. \omega' \in \kappa_{2j2}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ |\omega_{j2}.u| > |\omega_{j1}.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j1}.u| > |\omega'.u| \]

\[ \omega_{j1} \in \kappa_{1j}.o \]

\[ \omega_{j2} \in \kappa_{2j2}.o \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ 0 \leq |\omega_{j2}.u| \leq |\omega_{j1}.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ \omega_{j1} \in \kappa_{1j}.o \]

\[ \omega_{j2} \in \kappa_{2j2}.o \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]

\[ \forall \omega'. \omega' \in \kappa_{1j}.o \implies |\omega_{j2}.u| > |\omega'.u| \]
∀ω', ω' ∈ κ'.o \implies |ω_{1j}.u| > |ω'.u| \} \neq \emptyset \quad (69/36c/35)

AND κ_{2k} ∈ \mathcal{R}(n_k) \quad (70/36c/35)

AND κ_{2j}.s ∩ κ_{2k}.s \neq \emptyset \quad (71/36c/35)

AND \exists ω_j, ω_{2k}. ω_{2j} ∈ κ_{2j}.o \land ω_{2k} ∈ κ_{2k}.o \land |ω_{2j}.u| = |ω_{2k}.u|\land
∀ω, ω ∈ κ_{2j}.o ∪ κ_{2k}.o \implies |ω_{2j}.u| ≥ |ω.u| \quad (72/36c/35)

AND κ_{2j}.s = κ_{2j}.s ∩ κ_{2k}.s \quad (73/36c/35)

[tautology] κ_{1j} \neq κ_{2j} \quad (74a/36c/35)

OR κ_{1j} = κ_{2j} \quad (74b/36c/35)

(Case 74a/36c/35)

[74a, 68, if \ j = 1 then 33 else 34] κ_{1j}.s ∩ κ_{2j}.s = \emptyset

THEN (κ_{1j}.s ∩ _) ∩ (κ_{2j}.s ∩ _) = \emptyset \quad (75/74a/36c/35)

[75, 69, 73] κ_{1s} ∩ κ_{2s} = \emptyset \quad (76/74a/36c/35)

(Case 74b/36c/35)

ASSUME v ∈ κ_{1s} ∧ v ∈ κ_{2s} \quad (77/74b/36c/35)

[77, 69, 73, 74b] v ∈ κ_{2j}.s \quad (78/74b/36c/35)

AND ∀κ', κ' ∈ \mathcal{R}(n_k) \land v ∈ κ'.s \implies \exists ω_{2j}. ω_{2j} ∈ κ_{2j}.o \land
∀ω', ω' ∈ κ'.o \implies |ω_{2j}.u| > |ω'.u| \quad (79/74b/36c/35)

AND v ∈ κ_{2k}.s \quad (80/74b/36c/35)

[70, 80, 79] ω_{2j} ∈ κ_{2j}.o \quad (81/74b/36c/35)

AND ∀ω', ω' ∈ κ_{2k}.o \implies |ω_{2j}.u| > |ω'.u| \quad (82/74b/36c/35)

[72] ω_{2k} ∈ κ_{2k}.o \quad (83/74b/36c/35)

AND ∀ω, ω ∈ κ_{2j}.o ∪ κ_{2k}.o \implies |ω_{2k}.u| ≥ |ω.u| \quad (84/74b/36c/35)

[83, 82] |ω_{2j}.u| > |ω_{2k}.u| \quad (85/74b/36c/35)

[81, 84] |ω_{2k}.u| ≥ |ω_{2j}.u| \quad (86/74b/36c/35)

[77, 85, 86] \neg \exists v. v ∈ κ_{1s} ∧ v ∈ κ_{2s} \quad (87/74b/36c/35)

THEN κ_{1s} ∩ κ_{2s} = \emptyset \quad (88/36c/35)

(Case 36d/35)

[Similar to Case 36c/35] κ_{1s} ∩ κ_{2s} = \emptyset \quad (89/36d/35)

[36a, 60, 36b, 66, 36c, 88, 36d, 89] κ_{1s} ∩ κ_{2s} = \emptyset \quad (90)

[35, 90] ∀κ_1, κ_2. κ_1, κ_2 ∈ \mathcal{R}(n) ∧ κ_1 \neq κ_2 \implies κ_{1s} ∩ κ_{2s} = \emptyset \quad (Goal)
All added labels in deleted value part. The goal is to show $\forall \kappa, \omega. \kappa \in \mathcal{R}(n) \land \omega \in \kappa \implies \omega. a \leq \omega. d$.

A complete proof would be by structural induction on $n$. However, by observing all type inference sub-algorithms (Definitions 4.12, 4.13, 4.15, 4.16, 4.17, 4.18), one can see that the only places where individual output choices are constructed are Definition 4.13 for boxes and Definition 4.14 for the base case of serial composition, which means that the inductive cases for other sub-algorithms are all trivial.

We shall then focus on showing that the goal holds for these two sub-algorithms.

**For Definition 4.13**. Given $\kappa \in \mathcal{R}^b(\text{box } v_0 \rightarrow v_{s_0}) \land \omega \in \kappa.o,$

(1) to show $\omega.a \leq \omega.d$.

**Proof.** [1, Def 4.13] $\exists v_1. v_1 \in v_{s_0} \land \omega.a = v_1 \land \omega.d = v_0 + v_1$

THEN $\omega.a \leq \omega.d$

(Goal)

**For Definition 4.14**. Given $\kappa \in \mathcal{R}^f(\_, \omega_1, \kappa_2) \land \omega \in \kappa.o,$

(1) with IH $\omega_1.a \leq \omega_1.d,$

(2) and IH $\forall \omega_2. \omega_2 \in \kappa_2.o \implies \omega_2.a \leq \omega_2.d,$

(3) to show $\omega.a \leq \omega.d$.

**Proof.** [1, Def 4.14] $\omega_2 \in \kappa_2.o$

(4) AND $\omega.a = \omega_1.a - \omega_2.d + \omega_2.a$

(5) AND $\omega.d = \omega_1.d + \omega_2.d$

(6) [4, 3] $\omega_2.a \leq \omega_2.d$

(7) [2, 7] $\omega_1.a + \omega_2.a \leq \omega_1.d + \omega_2.d$

THEN $\omega_1.a - \omega_2.d + \omega_2.a \leq \omega_1.d + \omega_2.d$

(8) [5, 6, 8] $\omega.a \leq \omega.d$

(Goal)

**A.4 Theorem 4.20**

To prove $\mathcal{R}(n)$ represents $n$ for every $n$. 
Before starting the actual proof, we shall first modify the goals into the equivalents which are easier
to prove. Applying the definition of network domains, as well as the domain operator ‘dom’ and the
application operator ‘()’ defined for reps, the goals are transformed to the following:

\[
\begin{align*}
\text{• } \{ v \mid n \triangleleft v \} &= \{ v \mid \exists \kappa. \kappa \in \mathcal{R}(n) \land v \in \kappa.s \}; \\
\text{• } \forall v. n \triangleleft v &\implies \{ (u, f, v') \mid n, v \rightsquigarrow_s u, f, v' \} = \{(\omega.u, v - \omega.d, v - \omega.d + \omega.a) \mid \kappa \in \mathcal{R}(n) \land v \in \kappa.s \land \omega \in \kappa.o \}.
\end{align*}
\]

Both goals involve determining equivalence of two sets defined using the set comprehension syntax. This
will be equivalent with testing the logical equivalence of the membership criteria, so the goals can be
further written as the following, where the second goal above is expanded to two subgoals, for clarity
reasons:

\[
\begin{align*}
\text{• } \forall v. n \triangleleft v &\iff \exists \kappa. \kappa \in \mathcal{R}(n) \land v \in \kappa.s; \quad (\text{Goal 1}') \\
\text{• } \forall v, u, f, v'. n \triangleleft v \land (n, v \rightsquigarrow_s u, f, v') &\implies \\
&\exists \kappa, \omega. \kappa \in \mathcal{R}(n) \land v \in \kappa.s \land \omega \in \kappa.o \land \\
&u = \omega.u \land f = v - \omega.d \land v' = v - \omega.d + \omega.a; \quad (\text{Goal 2})
\end{align*}
\]

\[
\begin{align*}
\text{• } \forall v, \kappa, \omega. n \triangleleft v \land \kappa \in \mathcal{R}(n) \land v \in \kappa.s \land \omega \in \kappa.o &\implies \\
&n, v \rightsquigarrow_s \omega.u, v - \omega.d, v - \omega.d + \omega.a; \quad (\text{Goal 3'})
\end{align*}
\]

However, the left-to-right direction of Goal 1’ can be implied from Goal 2 as the following steps show.
Assume \( n \triangleleft v \). From Lemma 4.8, \( n, v \rightsquigarrow_s \_ \_ \_ \_ \_ \_ \_ \_ \_ \). This completes the left of \( \implies \) in Goal 2, and therefore
the right hand side holds, which can imply \( \exists \kappa. \kappa \in \mathcal{R}(n) \land v \in \kappa.s \), the right hand side of Goal 1’. As
a result, we only need to show the right-to-left direction of Goal 1’, which can be rewritten into the
following equivalent form:

\[
\begin{align*}
\text{• } \forall v, \kappa, \kappa \in \mathcal{R}(n) \land v \in \kappa.s \implies n \triangleleft v. \quad (\text{Goal 1})
\end{align*}
\]

In addition, Goal 3’ can also be simplified, because among the four premises in the proposition, the
second and third imply the first, using Goal 1:

\[
\begin{align*}
\text{• } \forall v, \kappa, \omega. \kappa \in \mathcal{R}(n) \land v \in \kappa.s \land \omega \in \kappa.o &\implies \\
&n, v \rightsquigarrow_s \omega.u, v - \omega.d, v - \omega.d + \omega.a. \quad (\text{Goal 3})
\end{align*}
\]

Therefore, this theorem will be proven when Goals 1, 2 and 3 are proven.

We will construct the full proof by structural induction on \( n \), showing all three goals hold simulta-
neously. For readability reasons only, we will present the proofs in separate subsections, one per goal,
but an inductive hypothesis used for proving Goal 1 may come from Goal \( j \) where \( j \neq i \), because these
subsections are meant to be restructured into a big proof.
Goal 1  Given: \( \kappa \in \mathcal{R}(n) \),
and \( v \in \kappa.s \),
to show \( n \triangleleft v \).  

Proof. By structural induction on \( n \).

(Base Case) \( n = \text{box} \) \( v_0 \rightarrow v s_0 \)  
[3, Def 4.13] \( \kappa.s = \{ v \mid v_0 \leq v \} \)  
[4, 2] \( v_0 \leq v \)  
[5, 3, Lem 4.7] \( n \triangleleft v \)  

(Inductive Case 1) \( n = n_1 \cdot n_2 \)

IH \( \forall \kappa_1, v. \kappa_1 \in \mathcal{R}(n_1) \land v \in \kappa_1.s \implies n_1 \triangleleft v \)  
IH \( \forall \kappa_2, v_1. \kappa_2 \in \mathcal{R}(n_2) \land v_1 \in \kappa_2.s \implies n_2 \triangleleft v_1 \)  
IH (from Goal 2) \( \forall v, v'. n_1 \triangleleft v \land (n_1, v \leadsto_{s, \omega} v_1) \implies \exists \kappa_1, \omega_1. \kappa_1 \in \mathcal{R}(n_1) \land v \in \kappa_1.s \land \omega_1 \in \kappa_1.o \land v_1 = v - \omega_1.d + \omega_1.a \)

[1, 6, Def 4.17] \( \kappa_{11} \in \mathcal{R}(n_1) \)  
AND \( \kappa \in \mathcal{R}_1^1(\kappa_{11}, \mathcal{R}(n_2)) \)  
[11, Def 4.16] \( \kappa'_{11} \in \mathcal{R}_2^2(\kappa_{11}, s, \omega) \)  
AND \( \kappa.s \subseteq \kappa'_{11} \)  
[12, Def 4.15] \( \kappa'_{11} \in \mathcal{R}_2^2(\kappa_{11}, s, \omega) \)  
[14, Def 4.14] \( \kappa'_{11}.s \subseteq \kappa_{11}.s \)  
[2, 13, 15] \( v \in \kappa_{11}.s \)  
[10, 16, 7] \( n_1 \triangleleft v \)  
ASSUME \( n_1, v \leadsto_{s, \omega} v_1 \)  
[17, 18, 9] \( \kappa_1 \in \mathcal{R}(n_1) \)  
AND \( v \in \kappa_1.s \)  
AND \( \omega_1 \in \kappa_1.o \)  
AND \( v_1 = v - \omega_1.d + \omega_1.a \)  
[19, 10, 20, 16, Thm 4.19] \( \kappa_1 = \kappa_{11} \)  
[23, 11] \( \kappa \in \mathcal{R}_1^1(\kappa_1, \mathcal{R}(n_2)) \)  
[24, 21, Def 4.16] \( \kappa'_1 \in \mathcal{R}_2^2(\kappa_{11}, \omega_1, \mathcal{R}(n_2)) \)  
AND \( \kappa.s \subseteq \kappa'_1.s \)
∃κ2, κ2 ∈ R(n2) ∧
κ′1.s = { v | v ∈ κ1.s ∧ (v − ω1.d + ω1.a) ∈ κ2.s }  (27/18)

∃κ2, κ2 ∈ R(n2) ∧ v1 ∈ κ2.s    (28/18)

n2 ∈ v1     (29/18)

∀v1, n1, v ⇝ s, v1 =⇒ n2 ∈ v1    (30)

n = n1 || n2    (31)

IH ∀κ1, κ1 ∈ R(n1) ∧ v ∈ κ1.s =⇒ n1 ∈ v    (32)

IH ∀κ2, κ2 ∈ R(n2) ∧ v ∈ κ2.s =⇒ n2 ∈ v    (33)

κ ∈ ρ′ where ρ′ is as in Def 4.18    (34a)

κ ∈ ρ″ where ρ″ is as in Def 4.18    (34b)

n1 ∈ v ∨ n2 ∈ v    (37/34a)

n1 ∈ v ∧ n2 ∈ v    (38/34b)

n1 ∈ v ∨ n2 ∈ v    (40/34b)

n1 ∈ v ∨ n2 ∈ v    (41)

∃κ, ω, κ ∈ R(n) ∧ v ∈ κ.s ∧ ω ∈ κ.o ∧
   u = ω.u ∧ f = v − ω.d ∧ v′ = v − ω.d + ω.a.    (Goal)

Proof. By structural induction on n.
(Base Case) $n = \text{box } v_0 \rightarrow v_{s_0}$

$[3, 2, \text{lsBox}] v_0 \leq v$ AND $v_1 \in v_{s_0}$ AND $u = v_0$ AND $f = v - v_0 - v_1$ AND $v' = v - v_0 + v_1$

$[3, \text{Def 4.12 Def 4.13}] \mathcal{R}(n) = \{ \kappa \}$ AND $\kappa.s = \{ v \mid v_0 \leq v \}$ AND $\forall v_1, v_1 \in v_{s_0} \implies (v_0, v_0 + v_1, v_1) \in \kappa.o$

$[9] \kappa \in \mathcal{R}(n)$ AND $\forall v \in \kappa.s$

$[10, 4] v \in \kappa.s$

$[11, 5] (v_0, v_0 + v_1, v_1) \in \kappa.o$

$[12, 13, 14, 6, 7, 8] \exists \kappa, \omega, \kappa \in \mathcal{R}(n) \land v \in \kappa.s \land \omega \in \kappa.o \land u = \omega.u \land f = v - \omega.d \land v' = v - \omega.d + \omega.a$. (Goal)

(Inductive Case 1) $n = n_1 \cdots n_2$

IH $\forall v, u_1, f_1, v_1, n_1 \triangleleft v \land (n_1, v \rightsquigarrow s, u_1, f_1, v_1) \implies$

$\exists \kappa_1, \omega_1, \kappa_1 \in \mathcal{R}(n_1) \land \kappa_1.s \land \omega_1 \in \kappa_1.o \land u_1 = \omega_1.u \land f_1 = v - \omega_1.d \land v_1 = v - \omega_1.d + \omega_1.a$ (16)

IH $\forall v, u_2, f_2, v', n_2 \triangleleft v_1 \land (n_2, v_1 \rightsquigarrow s, u_2, f_2, v') \implies$

$\exists \kappa_2, \omega_2, \kappa_2 \in \mathcal{R}(n_2) \land \kappa_2.s \land \omega_2 \in \kappa_2.o \land u_2 = \omega_2.u \land f_2 = v_1 - \omega_2.d \land v' = v_1 - \omega_2.d + \omega_2.a$ (17)

IH (from Goal 3) $\forall v, \kappa_1, \omega_1', \kappa_1 \in \mathcal{R}(n_1) \land v \in \kappa_1.s \land \omega_1' \in \kappa_1.o \implies$

$n_1, v \rightsquigarrow s, v - \omega_1'.d + \omega_1.a$ (18)

$[1, 15, \text{Lem 4.17}] n_1 \triangleleft v$ (19)

AND $\forall v_1, n_1, v \rightsquigarrow s, v_1 \implies n_2 \triangleleft v_1$ (20)

$[2, 15, \text{lsSer}] n_1, v \rightsquigarrow s, u_1, f_1, v_1$ (21)

AND $n_2, v_1 \rightsquigarrow s, u_2, f_2, v'$ (22)

AND $u = u_1 + (f_1 \times u_2)$ (23)

AND $f = f_1 \times f_2$ (24)

$[19, 21, 16] \kappa_1 \in \mathcal{R}(n_1)$ (25)

AND $v \in \kappa_1.s$ (26)
AND $\omega_1 \in \kappa_1.o$

AND $u_1 = \omega_1.u$

AND $f_1 = v - \omega_1.d$

AND $v_1 = v - \omega_1.d + \omega_1.a$

[25, 26, 18] $\forall \omega'_1, \omega'_1 \in \kappa_1.o \implies n_1, v \sim_s \omega'_1, v - \omega'_1.d + \omega'_1.a$

[31, 20] $\forall \omega'_1, \omega'_1 \in \kappa_1.o \implies n_2 \sim v - \omega'_1.d + \omega'_1.a$

[32, Lem 4.8, 17] $\forall \omega'_1, \omega'_1 \in \kappa_1.o \implies$

$$\exists \kappa'_2, \kappa'_2 \in \mathcal{R}(n_2) \land v - \omega'_1.d + \omega'_1.a \in \kappa'_2.s$$

[33, 26, Def 4.14] $\forall \omega'_1, \omega'_1 \in \kappa_1.o \implies \exists \kappa', \kappa'_2 \in \mathcal{R}(n_2) \land$

$$\mathcal{R}_3^{s}(\kappa_1.s, \omega'_1, \kappa'_2) = \{\kappa'_1 \land v \in \kappa'.s\}$$

[34, Def 4.15] $\forall \omega'_1, \omega'_1 \in \kappa_1.o \implies$

$$\exists \kappa', \kappa' \in \mathcal{R}_3^{s}(\kappa_1.s, \omega'_1, \mathcal{R}(n_2)) \land v \in \kappa'.s$$

[35, Def 4.16] $\kappa \in \mathcal{R}_1^{s}(\kappa_1, \mathcal{R}(n_2)) \land v \in \kappa.s$

AND $\forall \omega'_1, \omega'_1 \in \kappa_1.o \implies \exists \kappa', \kappa' \in \mathcal{R}_3^{s}(\kappa_1.s, \omega'_1, \mathcal{R}(n_2)) \land$

$$v \in \kappa'.s \land \kappa'.o \subseteq \kappa.o$$

[36, 25, Def 4.17] $\kappa \in \mathcal{R}(n) \land v \in \kappa.s$

[37, 27] $\kappa' \in \mathcal{R}_3^{s}(\kappa_1.s, \omega'_1, \mathcal{R}(n_2)) \land v \in \kappa'.s$

AND $\kappa'.o \subseteq \kappa.o$

[39, Def 4.15] $\exists \kappa'_2, \kappa'_2 \in \mathcal{R}(n_2) \land \kappa' \in \mathcal{R}_3^{s}(\kappa_1.s, \omega'_1, \kappa'_2) \land v \in \kappa'.s$

[41, Def 4.14] $\kappa'_2 \in \mathcal{R}(n_2) \land v - \omega_1.d + \omega_1.a \in \kappa'_2.s$

AND $\forall \omega'_2, \omega'_2 \in \kappa'_2.o \implies$

$$(\omega_1.u + (\omega'_2.u - \omega_1.a), \omega_d + \omega'_2.d, \omega_1.a - \omega'_2.d + \omega'_2.a) \in \kappa'.o$$

[43, 40] $\forall \omega'_2, \omega'_2 \in \kappa'_2.o \implies$

$$(\omega_1.u + (\omega'_2.u - \omega_1.a), \omega_d + \omega'_2.d, \omega_1.a - \omega'_2.d + \omega'_2.a) \in \kappa.o$$

[21, 20] $n_2 \in v_1$

[45, 22, 17] $\kappa_2 \in \mathcal{R}(n_2) \land v_1 \in \kappa_2.s$

AND $\omega_2 \in \kappa_2.o$

AND $u_2 = \omega_2.u$

AND $f_2 = v_1 - \omega_2.d$

AND $v' = v_1 - \omega_2.d + \omega_2.a$

[46, 30, 42, Thm 4.19] $\kappa'_2 = \kappa_2$

LET $\omega.u = \omega_1.u + (\omega_2.u - \omega_1.a)$

AND $\omega.d = \omega_1.d + \omega_2.d$
AND $\omega.a = \omega_1.a - \omega_2.d + \omega_2.a$  \hfill (54)

[51, 44, 47, 52, 53, 54] $\omega \in \kappa.0$  \hfill (55)

[25, 27, Thm 4.19] $\omega_1.a \leq \omega_1.d$

**THEN** $(v - \omega_1.d) \times \omega_1.a = \emptyset$  \hfill (56)

[22, Lem 4.9] $u_2 \leq v_1$  \hfill (57)

[57, 48, 30] $\omega_2.u \leq (v - \omega_1.d) + \omega_1.a$  \hfill (58)

[56, 58] $\omega_2.u \times (v - \omega_1.d) = \omega_2.u - \omega_1.a$  \hfill (59)

[23, 28, 29, 48] $u = \omega_1.u + ((v - \omega_1.d) \times \omega_2.u)$  \hfill (60)

[60, 59, 52] $u = \omega.u$  \hfill (61)

[24, 29, 49] $f = (v - \omega_1.d) \times (v_1 - \omega_2.d)$  \hfill (62)

[62, 30] $f = (v - \omega_1.d) \times (v - \omega_1.d + \omega_1.a - \omega_2.d)$

**THEN** $f = v - \omega_1.d - \omega_2.d$  \hfill (63)

[63, 53] $f = v - \omega.d$  \hfill (64)

[50, 30] $v' = v - \omega_1.d + \omega_1.a - \omega_2.d + \omega_2.a$

**THEN** $v' = v - \omega_1.d - \omega_2.d + (\omega_1.a - \omega_2.d + \omega_2.a)$  \hfill (65)

[65, 53, 54] $v' = v - \omega.d + \omega.a$  \hfill (66)

[38, 55, 61, 64, 66] $\exists \kappa, \omega, \kappa \in \mathcal{R}(n) \land \omega \in \kappa.0 \land
\omega = \omega.u \land f = v - \omega.d \land v' = v - \omega.d + \omega.a$
\hspace{1.5em} \text{(Goal)}  \hfill (67)

(Inductive Case 2) $n = n_1 \parallel n_2$

IH $\forall v, u_0, f_0, v', n_1 \triangleleft v \land (n_i, v \rightsquigarrow_s u_0, f_0, v') \implies
\exists x_0, x_0, x_0, x_0 \in \mathcal{R}(n_i) \land v \in x_0.0 \land
u_0 = \omega_0.u \land f_0 = v - \omega_0.d \land v' = v - \omega_0.d + \omega_0.a, (i \in \{1, 2\})$  \hfill (68)

IH (from Goal 1) $\forall x_0, x_0, x_0, x_0 \in \mathcal{R}(n_i) \land v \in x_0.0 \implies n_i \triangleleft v \land (i \in \{1, 2\})$  \hfill (69)

IH (from Goal 3) $\forall x_0, x_0, x_0, x_0 \in \mathcal{R}(n_i) \land v \in x_0.0 \land x_0 \in x_0.0 \implies
n_i, v \rightsquigarrow_s x_0.0, u_0, v, v' \land (i \in \{1, 2\})$  \hfill (70)

[1, 67, Lem 4.7] $n_1 \triangleleft v \lor n_2 \triangleleft v$

**THEN** $n_1 \triangleleft v \lor \neg(n_2 \triangleleft v)$  \hfill (71a)

OR $n_2 \triangleleft v \lor \neg(n_1 \triangleleft v)$  \hfill (71b)

OR $n_1 \triangleleft v \land n_2 \triangleleft v$  \hfill (71c)

(Case 71a)

[67, 71a, 2, IsPar] $n_1, v \rightsquigarrow_s u, f, v'$  \hfill (72/71a)
[71a, 72, 68] \( \exists \kappa_1, \omega_1. \kappa_1 \in \mathcal{R}(n_1) \land v \in \kappa_1.s \land \omega_1 \in \kappa_1.o \land \\
\quad u = \omega_1.u \land f = v - \omega_1.d \land v' = v - \omega_1.d + \omega_1.a \)  
(73/71a)

[71a, 69] \( \neg \exists \kappa_2. \kappa_2 \in \mathcal{R}(n_2) \land v \in \kappa_2.s \)  
(74/71a)

[73, 74] \( \exists \kappa, \omega. \kappa \in \rho' \land v \in \kappa.s \land \omega \in \kappa.o \land \\
\quad u = \omega.u \land f = v - \omega.d \land v' = v - \omega.d + \omega.a \)  
where \( \rho' \) is as in Def 4.18 using \( \mathcal{R}(n_1) \) as \( \rho_1 \) and \( \mathcal{R}(n_2) \) as \( \rho_2 \)  
(75/71a)

[75, 67, Def 4.12 Def 4.18] \( \exists \kappa, \omega. \kappa \in \mathcal{R}(n) \land v \in \kappa.s \land \omega \in \kappa.o \land \\
\quad u = \omega.u \land f = v - \omega.d \land v' = v - \omega.d + \omega.a \)  
(76/71a)

(Case 71b)

[see Case 71a] \( \exists \kappa, \omega. \kappa \in \mathcal{R}(n) \land v \in \kappa.s \land \omega \in \kappa.o \land \\
\quad u = \omega.u \land f = v - \omega.d \land v' = v - \omega.d + \omega.a \)  
(77/71b)

(Case 71c)

[71c, Lem 4.8] \( n_i, v \leadsto_{s \_ \_ \_ \_ \_ \_ \_ \_} (i \in \{1, 2\}) \)

LET \( u_1, u_2 : n_i, v \leadsto_{s \_ \_ \_ \_ \_ \_ \_ \_} (i \in \{1, 2\}) \)  
(78/71c)

AND \( \forall w_0. n_i, v \leadsto_{s \_ \_ \_ \_ \_ \_ \_ \_} \implies |u_i| \geq |w_0| (i \in \{1, 2\}) \)  
(79/71c)

[71c, 78, 68] \( \kappa_i \in \mathcal{R}(n_1) \land v \in \kappa_i.s \land \omega_i \in \kappa_i.o \land \omega_i.u = u_i (i \in \{1, 2\}) \)  
(80/71c)

[80, 70, 79] \( \forall \omega_i', \omega_i' \in \kappa_i.o \implies |\omega_i.u| \geq |\omega_i'.u| (i \in \{1, 2\}) \)  
(81/71c)

[tautology] \( |u_1| = |u_2| \)  
(82a/71c)

OR \( |u_1| \neq |u_2| \)  
(82b/71c)

(Case 82a/71c)

[82a, 80] \( |\omega_1.u| = |\omega_2.u| \)  
(83/82a/71c)

[80, 83, 81] \( \kappa \in \rho'' \land v \in \kappa.s \land \kappa.o = \kappa_1.o \cup \kappa_2.o \)

where \( \rho'' \) is as in Def 4.18 using \( \mathcal{R}(n_1) \) as \( \rho_1 \) and \( \mathcal{R}(n_2) \) as \( \rho_2 \)  
(84/82a/71c)

[84, 67, Def 4.12 Def 4.18] \( \kappa \in \mathcal{R}(n) \land v \in \kappa.s \land \kappa.o = \kappa_1.o \cup \kappa_2.o \)  
(85/82a/71c)

[67, 71c, 78, 79, 82a, lsPar] \( j \in \{1, 2\} \land n_j, v \leadsto_{s \_ \_ \_ \_ \_ \_ \_ \_} u, f, v' \)  
(86/82a/71c)

[71c, 86, 68] \( \kappa_j' \in \mathcal{R}(n_j) \land v \in \kappa_j'.s \land \omega_j' \in \kappa_j'.o \land \\
\quad u = \omega_j'.u \land f = v - \omega_j'.d \land v' = v - \omega_j'.d + \omega_j'.a \)  
(87/82a/71c)

[87, 80, Thm 4.19] \( \kappa_j' = \kappa_j \)  
(88/82a/71c)

[88, 87, 85] \( \exists \kappa, \omega. \kappa \in \mathcal{R}(n) \land v \in \kappa.s \land \omega \in \kappa.o \land \\
\quad u = \omega.u \land f = v - \omega.d \land v' = v - \omega.d + \omega.a \)  
(89/82a/71c)

(Case 82b/71c)

LET \( j, k : j, k \in \{1, 2\} \land j \neq k \land |u_j| > |u_k| \)  
(90/82b/71c)

[81, 80, 90] \( \forall \omega', \omega' \in \kappa_k.o \implies |\omega_j.u| > |\omega'.u| \)  
(91/82b/71c)
\[ \forall \kappa'. \kappa' \in \mathcal{R}(n_k) \land v \in \kappa'.s \implies \kappa' = \kappa_k \]  
\hfill (92/82b/71c)

\[ \forall \omega'. \omega' \in \kappa'.o \implies \omega_j, u > |\omega'.u| \]  
\hfill (93/82b/71c)

\[ \kappa \in \rho' \land v \in \kappa.s \land \kappa.o = \kappa_j.o \]  
where \( \rho' \) is as in Def 4.18 using \( \mathcal{R}(n_1) \) as \( \rho_1 \) and \( \mathcal{R}(n_2) \) as \( \rho_2 \)  
\hfill (94/82b/71c)

\[ \exists \kappa, \omega, \kappa \in \mathcal{R}(n) \land v \in \kappa.s \land \omega \in \kappa.o \land \]  
\[ u = \omega, u \land f = v - \omega, d \land v' = v - \omega, d + \omega.a \]  
\hfill (98/82b/71c)

\[ \exists \kappa, \omega, \kappa \in \mathcal{R}(n) \land v \in \kappa.s \land \omega \in \kappa.o \land \]  
\[ u = \omega, u \land f = v - \omega, d \land v' = v - \omega, d + \omega.a \]  
\hfill (99/82b/71c)

\[ \exists \kappa, \omega, \kappa \in \mathcal{R}(n) \land v \in \kappa.s \land \omega \in \kappa.o \land \]  
\[ u = \omega, u \land f = v - \omega, d \land v' = v - \omega, d + \omega.a \]  
\hfill (100/71c)

\[ \exists \kappa, \omega, \kappa \in \mathcal{R}(n) \land v \in \kappa.s \land \omega \in \kappa.o \land \]  
\[ u = \omega, u \land f = v - \omega, d \land v' = v - \omega, d + \omega.a \]  
\hfill (Goal)

\[ \Box \]

**Goal 3**  
Given: \( \kappa \in \mathcal{R}(n) \), \( v \in \kappa.s \), and \( \omega \in \kappa.o \), to show \( n, v \leadsto_s \omega.u, v - \omega.d, v - \omega.d + \omega.a \).  
\hfill (Goal)

**Proof.** By structural induction on \( n \).

**(Base Case)** \( n = \text{box} \ v_0 \rightarrow v_{s_0} \)

\[ [4, 1, \text{Def } 4.12 \text{ Def } 4.13] \kappa.s = \{ v \mid v_0 \leq v \} \]  
\hfill (5)

AND \( \kappa.o = \{ (v_0, v_0 + v_1, v_1) \mid v_1 \in v_{s_0} \} \)

\[ [5, 2] v_0 \leq v \]  
\hfill (7)

\[ [6, 3] v_1 \in v_{s_0} \land \omega = (v_0, v_0 + v_1, v_1) \]  
\hfill (8)

\[ [7, 8, 4, \text{IsBox}] n, v \leadsto_s v_0, v - v_0 - v_1, v - v_0 + v_1 \]  
\hfill (9)

\[ [8, 9] n, v \leadsto_s \omega.u, v - \omega.d, v - \omega.d + \omega.a \]  
\hfill (Goal)

**(Inductive Case 1)** \( n = n_1 \cdots n_2 \)  
\hfill (10)
\( \text{IH } \kappa_1 \in \mathcal{R}(n_1) \wedge v \in \kappa_1.s \wedge \omega_1 \in \kappa_1.o \implies 
_1, v \rightsquigarrow_s \omega_1.u, v - \omega_1.d, v - \omega_1.d + \omega_1.a \) \hfill (11)

\( \text{IH } \kappa_2 \in \mathcal{R}(n_2) \wedge v_1 \in \kappa_2.s \wedge \omega_2 \in \kappa_2.o \implies 
_2, v_1 \rightsquigarrow_s \omega_2.u, v_1 - \omega_2.d, v_1 - \omega_2.d + \omega_2.a \) \hfill (12)

[10, Def 4.12, Def 4.17] \( \kappa_1 \in \mathcal{R}(n_1) \) \hfill (13)

\( \text{AND } \kappa \in \mathcal{R}_k'(\kappa_1, \mathcal{R}(n_2)) \) \hfill (14)

[14, Def 4.16] 2, 3 \( \omega_1 \in \kappa_1.o \) \hfill (15)

\( \text{AND } \kappa'_1 \in \mathcal{R}_3(k_1.s, \omega_1, \mathcal{R}(n_2)) \wedge v \in \kappa'_1.s \wedge \omega \in \kappa'_1.o \) \hfill (16)

[16, Def 4.15] \( \kappa_2 \in \mathcal{R}(n_2) \) \hfill (17)

\( \text{AND } \kappa'_1 \in \mathcal{R}_3(k_1.s, \omega_1, \kappa_2) \wedge v \in \kappa'_1.s \wedge \omega \in \kappa'_1.o \) \hfill (18)

\( \text{LET } v_1 : v_1 = v - \omega_1.d + \omega_1.a \) \hfill (19)

[18, Def 4.14] \( v \in \kappa_1.s \) \hfill (20)

\( \text{AND } v_1 \in \kappa_2.s \wedge \omega_2 \in \kappa_2.o \) \hfill (21)

\( \text{AND } \omega = (\omega_1.u + (\omega_2.u - \omega_1.a), \omega_1.d + \omega_2.d, \omega_1.a - \omega_2.d + \omega_2.a) \) \hfill (22)

[11, 12, 13, 20, 15, 19] \( n_1, v \rightsquigarrow_s \omega_1.u, v - \omega_1.d, v_1 \) \hfill (23)

[12, 17, 21] \( n_2, v_1 \rightsquigarrow_s \omega_2.u, v_1 - \omega_2.d, v_1 - \omega_2.d + \omega_2.a \) \hfill (24)

[10, 23, 24, IsSer, 19, 22, see computation steps 56-66 in Inductive Case 1 in Goal 2 from page 142] \( n, v \rightsquigarrow_s \omega.u, v - \omega.d, v - \omega.d + \omega.a \) \hfill (Goal)

(Inductive Case 2) \( n = n_1 \parallel n_2 \) \hfill (25)

\( \text{IH } \kappa_i \in \mathcal{R}(n_i) \wedge v \in \kappa_i.s \wedge \omega_i \in \kappa_i.o \implies 
_i, v \rightsquigarrow_s \omega_i.u, v - \omega_i.d, v - \omega_i.d + \omega_i.a \quad (i \in \{1, 2\}) \) \hfill (26)

\( \text{IH (from Goal 1) } \kappa_i \in \mathcal{R}(n_i) \wedge v \in \kappa_i.s \implies n_i \circ v \quad (i \in \{1, 2\}) \) \hfill (27)

\( \text{IH (from Goal 2) } n_i \circ v \wedge (n_i, v \rightsquigarrow_s u_i, \_ , \_) \implies 
\exists \kappa_i, \omega_i, \kappa_i \in \mathcal{R}(n_i) \wedge v \in \kappa_i.s \wedge \omega_i \in \kappa_i.o \wedge u_i = \omega_i.u \quad (i \in \{1, 2\}) \) \hfill (28)

[25, Def 4.12, Def 4.18] \( \kappa \in \rho' \) where \( \rho' \) is as in Def 4.18 with \( \rho_1 = \mathcal{R}(n_1) \wedge \rho_2 = \mathcal{R}(n_2) \) \hfill (29a)

\( \text{OR } \kappa \in \rho'' \) where \( \rho'' \) is as in Def 4.18 with \( \rho_1 = \mathcal{R}(n_1) \wedge \rho_2 = \mathcal{R}(n_2) \) \hfill (29b)

(Case 29a)

[29a, 2, 3] \( j, k \in \{1, 2\} \wedge j \neq k \) \hfill (30/29a)

\( \text{AND } \kappa_j \in \mathcal{R}(n_j) \wedge v \in \kappa_j.s \wedge \omega \in \kappa_j.o \) \hfill (31/29a)

\( \text{AND } \forall \kappa'. \kappa' \in \mathcal{R}(n_k) \wedge v \in \kappa'.s \implies \exists \omega_j, \omega_j \in \kappa_j.o \wedge 
\forall \omega', \omega' \in \kappa'.o \implies |\omega_j.u| > |\omega'.u| \) \hfill (32/29a)
[31, 27] \( n_j \circ v \)  

[31, 26] \( n_j, v \leftarrow_s \omega, u, v - \omega, d, v - \omega, d + \omega, a \)  

**ASSUME** \( n_k \circ v \)  

[35, Lem 4.7] 28 \[ \forall u_k, n_k, v \leftarrow_s u_k, - , - \implies \exists \kappa_k, \omega_k, \kappa_k \in \mathcal{R}(n_k) \land v \in \kappa_k, s \land \omega_k \in \kappa_k, o \land u_k = \omega_k, u \]

[36, 32] \[ \forall u_k, n_k, v \leftarrow_s u_k, - , - \implies \exists \omega_j, \omega_j \in \kappa_j, o \land |\omega_j, u| > |u_k| \]

[37, 31, 26] \[ \forall u_k, n_k, v \leftarrow_s u_k, - , - \implies \exists \omega_j, v \leftarrow_s u_j, - , - \land |\omega_j| > |u_k| \]

[35, 38, Def 4.6] \[ n_k, v \not\leftarrow_s \exists \forall u_k, n_k, v \leftarrow_s u_k, - , - \implies \exists \omega_j, v \leftarrow_s u_j, - , - \land |\omega_j| > |u_k| \]

[25, 30, 33, 39, 34, lsPar] \( n, v \leftarrow_s \omega, u, v - \omega, d, v - \omega, d + \omega, a \)

(Goal)

(Case 29b)

[29b, 2, 3] \( j, k \in \{1, 2\} \land j \neq k \)

**AND** \( \kappa_j \in \mathcal{R}(n_j) \land v \in \kappa_j, s \)

**AND** \( \kappa_k \in \mathcal{R}(n_k) \land v \in \kappa_k, s \)

**AND** \( \omega \in \kappa_j, o \)

**AND** \( \omega_j \in \kappa_j, o \)

**AND** \( \forall \omega', \omega' \in \kappa_j, o \cup \kappa_k, o \implies |\omega_j, u| > |\omega', u| \)

[42, 27] \( n_j \circ v \)

[42, 44, 26] \( n_j, v \leftarrow_s \omega, u, v - \omega, d, v - \omega, d + \omega, a \)

[42, 45, 26] \( n_j, v \leftarrow_s \omega_j, u, - , - \)

[43, 27] \( n_k \circ v \)

[50, Lem 4.7] 28 \[ \forall u_k, n_k, v \leftarrow_s u_k, - , - \implies \exists \kappa', \omega', \kappa' \in \mathcal{R}(n_k) \land v \in \kappa', s \land \omega' \in \kappa', o \land u_k = \omega', u \]

[51, 43, Thm 4.19] \[ \forall u_k, n_k, v \leftarrow_s u_k, - , - \implies \exists \omega', \omega' \in \kappa_k, s \land u_k = \omega', u \]

[52, 46] \[ \forall u_k, n_k, v \leftarrow_s u_k, - , - \implies |\omega_j, u| > |u_k| \]

[53, 49] \[ \forall u_k, n_k, v \leftarrow_s u_k, - , - \implies |\omega_j, u| > |u_k| \]

[54, 25, 41, 47, 54, 48, lsPar] \( n, v \leftarrow_s \omega, u, v - \omega, d, v - \omega, d + \omega, a \)

[29a, 40, 29b, 55] \( n, v \leftarrow_s \omega, u, v - \omega, d, v - \omega, d + \omega, a \)
A.5 Lemma 4.23

To show: \( \forall \kappa \in \mathcal{R}(n) \Rightarrow \exists \omega \in \kappa. \) (Goal)

Proof. By structural induction on \( n \).

**Base Case** \( n = \text{box } v_0 \rightarrow v_{s_0} \)

1. [L-Net box requirement] \( v_1 \in v_{s_0} \)
2. [Def 4.12, 1, Def 4.13] \( \mathcal{R}(n) = \{ \kappa_0 \} \land \kappa_0 = (\_ , \{ \_ | v_1 \in v_{s_0} \}) \)
3. [2, 3] \( \exists \omega \in \kappa_0 \)
4. [3, 4] \( \forall \kappa \in \mathcal{R}(n) \Rightarrow \exists \omega \in \kappa. \) (Goal)

**Inductive Case 1** \( n = n_1 \cdot n_2 \)

IH \( \forall \kappa_1, \kappa_1 \in \mathcal{R}(n_1) \Rightarrow \exists \omega_1, \omega_1 \in \kappa_1. \) (6)

IH \( \forall \kappa_2, \kappa_2 \in \mathcal{R}(n_2) \Rightarrow \exists \omega_2, \omega_2 \in \kappa_2. \) (7)

Assume \( \kappa \in \mathcal{R}(n) \) (8)

[Def 4.12 5, Def 4.17 8] \( \kappa_1 \in \mathcal{R}(n_1) \) (9/8)

AND \( \kappa \in \mathcal{R}_1^1(\kappa_1, \mathcal{R}(n_2)) \) (10/8)

[9, 6] \( \omega_1 \in \kappa_1. \) (11/8)

[10, Def 4.16 11] \( \kappa_1' \in \mathcal{R}_2^1(\kappa_1, s_1, \omega_1, \mathcal{R}(n_2)) \) (12/8)

AND \( \kappa_1'. \subseteq \kappa. \) (13/8)

[11, Def 4.15] \( \kappa_2 \in \mathcal{R}(n_2) \) (14/8)

AND \( \kappa_1' \in \mathcal{R}_3^1(\kappa_1, s_1, \omega_1, \kappa_2) \) (15/8)

[14, 7] \( \omega_2 \in \kappa_2. \) (16/8)

[15, Def 4.14] \( \kappa_1' = (\_ , \{ \_ | \omega_2 \in \kappa_2. \}) \) (17/8)

[16, 17] \( \exists \omega \in \kappa_1'. \) (18/8)

[18, 13] \( \exists \omega \in \kappa. \) (19/8)

[8, 19] \( \kappa \in \mathcal{R}(n) \Rightarrow \exists \omega \in \kappa. \) (Goal)

**Inductive Case 2** \( n = n_1 \parallel n_2 \)

IH \( \forall \kappa_1, \kappa_1 \in \mathcal{R}(n_1) \Rightarrow \exists \omega_1, \omega_1 \in \kappa_1. \) (21)
IH $\forall \kappa_2, \kappa_2 \in \mathcal{R}(n_2) \implies \exists \omega_2, \omega_2 \in \kappa_2.o \quad (22)$

ASSUME $\kappa \in \mathcal{R}(n)$

[Def 4.12 20, Def 4.18 23] $\kappa \in \rho'$ where $\rho'$ is as in Def 4.18

OR $\kappa \in \rho''$ where $\rho''$ is as in Def 4.18

(Case 24a)

THEN $\kappa.o = \kappa_j.o \land \kappa_j \in \mathcal{R}(n_j) \land (j = 1 \lor j = 2)$

[25, if $j = 1$ then 21 else 22] $\exists \omega. \omega \in \kappa_j.o$ (25/24a/23)

[25, 26] $\exists \omega. \omega \in \kappa.o$ (27/24a/23)

(Case 24b)

THEN $\kappa.o = \kappa_1.o \cup \_ \land \kappa_1 \in \mathcal{R}(n_1)$

[28, 21] $\exists \omega. \omega \in \kappa_1.o$ (29/24b/23)

[28, 29] $\exists \omega. \omega \in \kappa.o$ (30/24b/23)

[24a, 27, 24b, 30, 23] $\kappa \in \mathcal{R}(n) \implies \exists \omega. \omega \in \kappa.o$

THEN $\forall \kappa. \kappa \in \mathcal{R}(n) \implies \exists \omega. \omega \in \kappa.o$ (Goal)

\[ \square \]

A.6 Theorem 4.24

To prove $\sim_1/\sim_1$ complies with $\sim_s/\sim_s$. A complete proof would be by structural induction on $n$, covering:

1. $n$ is a box;

2. $n$ is a serial composition; and

3. $n$ is a parallel composition.

For each case, two goals need to be proven: one for stuck executions and the other for non-stuck executions. However, the close resemblance between lsBox and liBox, between lsSer and liSer, and between lsStuck and liStuck makes it trivial to prove the parts of cases covered by them. What remains is the goal for non-stuck executions regarding the third case:

$$(n_1 \parallel n_2), v \sim_s \_ \_ \_ , v' \iff (n_1 \parallel n_2), v \sim_1 v'.$$

To prove the above, we only need to show that the premises of lsPar and liPar are logically equivalent. This further simplifies the goal to the following:
\[
\left( \exists \alpha_j. \alpha_j \in \mathcal{A}(n_j) \land v \in \alpha_j.s \land \forall \alpha_k. \alpha_k \in \mathcal{A}(n_k) \land v \in \alpha_k.s \implies \alpha_j.r \geq \alpha_k.r \right) \\
\iff \left( n_j \bowtie v \land (n_k, v \not\bowtie s \lor \forall u_k. n_k, v \bowtie s u_k, -r \implies \exists u_j. n_j, v \bowtie s u_j, -r \land |u_j| \geq |u_k|) \right)
\]

A.6.0.1 Left-to-right Subgoal

Given \( j, k \in \{1, 2\} \land j \neq k \), and \( \alpha_j \in \mathcal{A}(n_j) \), and \( v \in \alpha_j.s \), and \( \forall \alpha_k. \alpha_k \in \mathcal{A}(n_k) \land v \in \alpha_k.s \implies \alpha_j.r \geq \alpha_k.r \), to show \( n_j \bowtie v \), and \( n_k, v \not\bowtie s \lor \forall u_k. n_k, v \bowtie s u_k, -r \implies \exists u_j. n_j, v \bowtie s u_j, -r \land |u_j| \geq |u_k| \).

**Proof.** [2, 3, definition of \( \mathcal{A} \), \( \kappa_j \in \mathcal{R}(n_j) \), \( v \in \kappa_j.s \), \( \alpha_j.r = \max\{\|\omega.u\| \mid \omega \in \kappa_j.o\} \), \( n_j \bowtie v \), \( n_j, v \bowtie s u_j, -r \land |u_j| = \alpha_j.r \), \( n_k, v \not\bowtie s \lor \forall u_k. n_k, v \bowtie s u_k, -r \implies \exists u_j. n_j, v \bowtie s u_j, -r \land |u_j| \geq |u_k| \), \( |u_k| \geq |u'| \), \( \forall \omega'. \omega' \in \kappa_k.o \implies |\omega_k.u| \geq |\omega'.u| \), \( |u_j| \geq |u_k| \), \( |u| \geq |u'| \), \( |u_j| \geq |u'| \), \( |u_j| \geq |u'| \), \( n_k, v \not\bowtie s \lor \forall u_k. n_k, v \bowtie s u_k, -r \implies \exists u_j. n_j, v \bowtie s u_j, -r \land |u_j| \geq |u_k| \).
A.6.0.2 Right-to-left Subgoal

Given $j, k \in \{1, 2\} \land j \neq k$, and $n_j \triangleleft v$, and $n_k \neq v$ \forall
\[\forall u_k. n_k, v \triangleright u_k, -, - \implies \exists u_j. n_j, v \triangleright u_j, -, - \land |u_j| \geq |u_k|,\]
to show $\exists \alpha_j, \alpha_j \in A(n_j) \land v \in \alpha_j, s \land \forall \alpha_k, \alpha_k \in A(n_k) \land v \in \alpha_k, s \implies \alpha_j, r \geq \alpha_k, r$.

\textbf{Proof.} \[2, \text{Lem 4.7}\] $n_j, v \triangleright u, -, -$

LET $u_j : n_j, v \triangleright u_j, -, -$

AND $\forall u'. n_j, v \triangleright u', -, - \implies |u_j| \geq |u'|$

\[2, 4, \text{Thm 4.20}\] $\kappa_j \in R(n_j) \land v \in \kappa_j, s$

AND $\omega_j \in \kappa_j, o \land u_j = \omega_j, u$

\[5, 6, \text{Thm 4.19} \& \text{def of } A, 7\] $\alpha_j \in A(n_j) \land v \in \alpha_j, s \land |u_j| = |u_j|$

ASSUME $\alpha_k \in A(n_k) \land v \in \alpha_k, s$

\[10, \text{def of } A\] $\kappa_k \in R(n_k) \land v \in \kappa_k, s$

AND $\omega_k \in \kappa, o \land |\omega_k, u| = \alpha_k, r$

\[11, \text{Thm 4.20}\] $n_k \triangleleft v$

\[11, 12, \text{Thm 4.20}\] $n_k, v \triangleright u_k, -, -$

\[13, \text{Def 4.6} \& \text{def of } A, 14\] $\exists u'_j, n_j, v \triangleright u'_j, -, - \land |u'_j| \geq |\omega_k, u|$

\[15, 5\] $|u_j| \geq |\omega_k, u|$

\[16, 9, 12\] $\alpha_j, r \geq \alpha_k, r$

\[10, 17\] $\forall \alpha_k, \alpha_k \in A(n_k) \land v \in \alpha_k, s \implies \alpha_j, r \geq \alpha_k, r$

\[9, 18\] $\exists \alpha_j, \alpha_j \in A(n_j) \land v \in \alpha_j, s \land \forall \alpha_k, \alpha_k \in A(n_k) \land v \in \alpha_k, s \implies \alpha_j, r \geq \alpha_k, r$

\textbf{A.7 Theorem 5.29}

To prove that the early fixed point model ($n^{m_0} v s_1; v s_2$) of a serial replication ($n^{*} v s_1; v s_2$) has the same domain and the same normal executions within its domain as the serial replication. The proof has 4 subgoals to reach:
1. \( \forall v. (n \ast vs_1; vs_2), v \not\rightarrow_s \implies (n^{m_0} vs_1; vs_2), v \not\rightarrow_s; \)

2. \( \forall v. (n^{m_0} vs_1; vs_2), v \not\rightarrow_s \implies (n \ast vs_1; vs_2), v \not\rightarrow_s; \)

3. \( \forall v, u, f, v'. (n \ast vs_1; vs_2) \triangleleft v \land (n \ast vs_1; vs_2), v \not\rightarrow_s u, f, v' \implies (n^{m_0} vs_1; vs_2), v \not\rightarrow_s u, f, v'; \)

4. \( \forall v, u, f, v'. (n^{m_0} vs_1; vs_2) \triangleleft v \land (n^{m_0} vs_1; vs_2), v \not\rightarrow_s u, f, v' \implies (n \ast vs_1; vs_2), v \not\rightarrow_s u, f, v'. \)

Proving subgoal 4 is straightforward by directly applying bsStar. The following subsections detail the steps to show the remaining three subgoals.

**Subgoal 1**

Given: \( m_0 \) is an early fixed point of \( n \ast vs_1; vs_2 \) (1) and \( (n \ast vs_1; vs_2), v \not\rightarrow_s \), (2) to show \( (n^{m_0} vs_1; vs_2), v \not\rightarrow_s \). (Goal)

**Proof.** [2, bsStuck] \( n^{m_0} vs_1; vs_2, v \not\rightarrow_s \) (3)

[tautology] \( m > m_0 \) (4a)

OR \( m \leq m_0 \) (4b)

(Case 4a)

[4, 1, Def 5.27, 3] \( m'' \leq m_0 \land (n^{m''} vs_1; vs_2), v \not\rightarrow_s \) (5/4a)

[5, Def 5.13, bsStuck for \( m_0 - m'' \) times]

\( (n^{m''} vs_1; vs_2), v \not\rightarrow_s \) (6/4a)

(Case 4b)

[see step 6] \( (n^{m_0} vs_1; vs_2), v \not\rightarrow_s \) (7/4b)

[4a, 6, 4b, 7] \( (n^{m_0} vs_1; vs_2), v \not\rightarrow_s \) (8)

[8, Lem 5.14, bsStuck] \( (n^{m_0} vs_1; vs_2), v \not\rightarrow_s \). (Goal)

**Subgoal 2**

Given: \( m_0 \) is an early fixed point of \( n \ast vs_1; vs_2 \) (1) and \( (n^{m_0} vs_1; vs_2), v \not\rightarrow_s \), (2) to show \( (n \ast vs_1; vs_2), v \not\rightarrow_s \). (Goal)

**Proof.** ASSUME \( (n^{m_0} vs_1; vs_2) \triangleleft v \) (3)

[2, Lem 5.14, bsStuck, 3] \( (n^{m_0} vs_1; vs_2), v \not\rightarrow_s u, v_1 \) (4/3)

AND \( F(n \ast vs_1; vs_2), v_1 \not\rightarrow_s \) (5/3)
Observe from Definition 5.11 that the domain of $F(n \ast vs_1; vs_2)$ covers the root alphabet of $n$, which means it is larger than (a superset of) the domain of $n$. That $F(n \ast vs_1; vs_2)$ does not accept $v_1$ indicates that $n$ does not accept $v_1$ as well. Moreover, the domain of $F(n \ast vs_1; vs_2)$ also covers the domain of $P(n \ast vs_1; vs_2)$. The detailed reasoning steps are omitted.

\[\text{[5]}\ n, v_1 \not\in \mathcal{N}\]

AND $P(n \ast vs_1; vs_2), v_1 \not\in \mathcal{N}$

\[\text{[6]}\ \text{bsStuck}]\ n \cdot \_ , v_1 \not\in \mathcal{N}\]

\[\text{[8, 7]}\ \text{bsStuck}]\ (n \cdot \_ ) \parallel P(n \ast vs_1; vs_2), v_1 \not\in \mathcal{N}\]

\[\text{[4, 9, Def 5.13]}\ \text{bsStuck}]\ (n^{m_0+1}_1 vs_1; vs_2), v \not\in \mathcal{N}\]

\[\text{[3, 10]}\ m_0\ \text{is not an early fixed point of } n \ast vs_1; vs_2\]

\[\text{[3, 1, 11, Def 5.5]}\ (n^{m_0}_1 vs_1; vs_2), v \not\in \mathcal{N}\]

\[\text{[12, bsStuck]}\ (n \ast vs_1; vs_2), v \not\in \mathcal{N}\]

\[\text{Goal}\]

\[\text{Subgoal 3} \quad \text{Given: } m_0\ \text{is an early fixed point of } n \ast vs_1; vs_2\]

and $(n \ast vs_1; vs_2) \circ v$.

and $(n \ast vs_1; vs_2), v \leadsto u, f, v'$

to show $(n^{m_0} vs_1; vs_2), v \leadsto u, f, v'$.

\[\text{Goal}\]

\[\text{Proof.}\ [3, \text{bsStar}]\ m \in \mathbb{N} \land (n^{m_0} vs_1; vs_2), v \leadsto u, f, v'\]

\[\text{[4, Lem 5.14]}\ \text{bsSer}]\ (n^{m_0}_1 vs_1; vs_2), v \leadsto u_1, f_1, v_1\]

AND $F(n \ast vs_1; vs_2), v_1 \leadsto u, f_2, v'$

AND $u = u_1 + (f_1 \times u_2)$

AND $f = f_1 \times f_2$

[tautology] $m = m_0$

OR $m < m_0$

OR $m > m_0$

(Case 9a)

\[\text{[9a, 4]}\ (n^{m_0} vs_1; vs_2), v \leadsto u, f, v'\]

(Case 9b)

\[\text{[9b, Def 5.13]}\ n^{m_0}_1 vs_1; vs_2 = n^{m_0}_1 vs_1; vs_2 \cdots \underbrace{I \cdots I}_m \cdot \underbrace{\cdots I}_{m_0-m \times m_0} \times \cdots \times I\]

where $I = (n \cdot T(n \ast vs_1; vs_2)) \parallel P(n \ast vs_1; vs_2)$

\[\text{[6, Def 5.11]}\ \text{bsPar, bsSink}]\ v_0 \in vs_1 \cup vs_2\]

\[\text{Goal}\]

\[\text{Goal}\]

\[\text{Goal}\]
AND (\(\text{box} \ (v_0 + \#\emptyset) \rightarrow \{v_0\}\)) \(v_1 \sim_s u_2, f_2, v'\)
\(\text{(13/9b)}\)

Note: if there are multiple possibilities of \(v_0\) in (12) above, let \(v_0\) be the largest one as per \(\leq\) (Definition 5.4).

\[13, \text{box}] \ (v_0 + \#\emptyset) \leq v_1 \quad \text{\(\text{(14/9b)}\)}
\]

\[\text{AND} \ u_2 = v_0 + \#\emptyset \quad \text{\(\text{(15/9b)}\)}
\]

\[\text{AND} \ f_2 = v_1 - \#\emptyset - v_0 \quad \text{\(\text{(16/9b)}\)}
\]

\[\text{AND} \ v' = v_1 - \#\emptyset \quad \text{\(\text{(17/9b)}\)}
\]

\[14, \ast \text{ is unavailable in } n \] \(n, v_1 \not\sim_s\)
\(\text{\(\text{(18/9b)}\)}
\]

\[18, \text{boxStuck} \ (n \cdots T(n \ast vs_1; vs_2)), v_1 \not\sim_s\)
\(\text{\(\text{(19/9b)}\)}
\]

\[12, \text{Def 5.10, boxPar, 15, 16} \; P(n \ast vs_1; vs_2) \circ v_1 \quad \text{\(\text{(20/9b)}\)}
\]

\[\text{AND} \ P(n \ast vs_1; vs_2), v_1 \sim_s u_2, f_2, v_1 \quad \text{\(\text{(21/9b)}\)}
\]

\[19, 20, 21, \text{boxPar, 11} \; I, v_1 \sim_s u_2, f_2, v_1 \quad \text{\(\text{(22/9b)}\)}
\]

\[5, 22, \text{boxSer, 7, 8} \; (n^{m_0 \ast vs_1; vs_2} \cdots I \cdots I \cdots I), v \sim_s u, f, v_1 \quad \text{\(\text{(23/9b)}\)}
\]

\[23, 6, 7, 8, 11 \; (n^{m_0 \ast vs_1; vs_2}), v \sim_s u, f, v' \quad \text{\(\text{(24/9b)}\)}
\]

(Case 9c)

\[2, \text{Lem 5.15} \; (n^{m \ast vs_1; vs_2}) \circ v \quad \text{\(\text{(25/9c)}\)}
\]

\[1, \text{Def 5.27, 9c, 25} \; m'' \in \mathbb{N} \land m'' \leq m_0 \quad \text{\(\text{(26/9c)}\)}
\]

\[\text{AND} \; (n^{m'' \ast vs_1; vs_2}), v \sim_s u, f, v' \quad \text{\(\text{(27/9c)}\)}
\]

\[26 \; m'' = m_0 \quad \text{\(\text{(28a/9c)}\)}
\]

\[\text{OR} \; m'' < m_0 \quad \text{\(\text{(28b/9c)}\)}
\]

(Case 28a/9c)

\[\text{[see Case 9a]} \; (n^{m_0 \ast vs_1; vs_2}), v \sim_s u, f, v' \quad \text{\(\text{(29/28a/9c)}\)}
\]

(Case 28b/9c)

\[\text{[see Case 9b]} \; (n^{m_0 \ast vs_1; vs_2}), v \sim_s u, f, v' \quad \text{\(\text{(30/28b/9c)}\)}
\]

\[28a, 29, 28b, 30 \; (n^{m_0 \ast vs_1; vs_2}), v \sim_s u, f, v' \quad \text{\(\text{(31/9c)}\)}
\]

\[9a, 10, 9b, 24, 9c, 31 \; (n^{m_0 \ast vs_1; vs_2}), v \sim_s u, f, v' \quad \text{(Goal)}
\]

\[\square\]

A.8 Theorem 5.32

To prove \(R(n)\) represents \(n\) for every \(n\).

153
We use a similar process as the proof for Theorem 4.20 in A.4 on page 136 where Goal 1:

\[ \forall v. n \triangleleft v \iff \exists \kappa. \kappa \in \mathcal{R}(n) \land v \in \kappa.s \]

is simplified using Goal 2 and Lemma 4.8 to Goal 1:

\[ \forall v, \kappa. \kappa \in \mathcal{R}(n) \land v \in \kappa.s \Rightarrow n \triangleleft v, \]

which is equivalent with the right-to-left direction of Goal 1'. However, due to the sunk executions, Lemma 5.6 (the successor of Lemma 4.8 in BL-Net) cannot help us deduce Goal 1 from Goal 1' and Goal 2, so we need to add a Goal 4, to tackle the sunk executions:

\[ \forall v, n \triangleleft v \land n, v \leadsto \text{nil} \Rightarrow \exists \kappa. \kappa \in \mathcal{R}(n) \land v \in \kappa.s. \]

If we temporarily exclude the serial replications, proving this additional goal is easy, using the sunk execution specifications and following the discussions after Definitions 5.22 \((R^1_s)\) and 5.22 \((R^p)\) regarding the sunk execution. The rest of the proof in A.4 can be upgraded with minor changes for the other goals.

What remains is the case covering the serial replication.

The serial replication type inference \(R^*\) (Definition 5.30) executes the procedure in Figure 5.4 on page 97 which essentially does a type inference for the model \(T \cdot I \cdot \ldots \cdot I \cdot F\), with \(m\) instances of \(I\), where \(m\) is the number of times the \texttt{repeat} block is executed. According to the discussion prior to this definition, \(m - 1\) is an early fixed point of the serial replication, and so is \(m\), indicating that the model is an early fixed point model, and Theorem 5.29 applies. The remaining steps that bridge Theorem 5.29 to Theorem 5.32 are trivial. The type inference algorithm can then be easily proven to be correct.
Appendix B

Previous S-Net Type System

The type system presented in this appendix is adapted from the type system chapter in [25] and matches the latest implementation in the S-Net compiler as of the time of writing this thesis. It also features a set-based design; the set assigned to a network as its type has a structure inspired by the multi-mapping signature of the signed network (Sections 2.2.8, 3.2.9).

B.1 Introduction

The record types in S-Net not only specify what should be input and what may be output, they also influence the decision of routing at the parallel compositions for each input record. The programmer provides the box and subnet network signatures, from which the compiler should gather the routing information for each parallel composition, for use by the runtime to deliver the records accordingly. Assuming the routing information exists, the complete behaviour of the program as handled by the runtime is known, so we can check whether the program is type safe.

To populate the routing information, the compiler performs inference on each parallel branch to find out what record types are useful. The resulting routing information for the branch is a set of types, indicating to the runtime that they should be attracted into the branch. However, the procedure cannot guarantee that all attracted types are safe within the branch, and the compiler will perform a type check immediately after the route inference.

We first present the type check procedure on an abstract interpretation of an S-Net program populated with the routing information, and then introduce the route inference procedure.

B.2 S-Net Abstract Interpretation

The type system does not deal with individual records, and using the full semantics is an overkill. In this section, we present an abstract interpretation of S-Net programs adapted to the purposes of the type system. We call the resulting language S-Net_A.
\[ P \in \text{Program} ::= N \]
\[ N \in \text{Net} ::= \text{net} (\sigma) \text{ connect } e \]
\[ e \in \text{TopoExpr} ::= \text{box} (r \to \tau) \quad \text{(Box)} \]
\[ \quad \text{Dotdot} \]
\[ \quad \text{Bar} \]
\[ \quad \text{Star} \]
\[ \quad \text{Ex} \]
\[ \quad N \quad \text{(Subnet)} \]

\[ l \in \text{Label} = \text{Field} \cup \text{BTag} \]
\[ r, v \in \text{Rtype} = \mathcal{P}(\text{Label}) \]
\[ \tau \in \text{Vtype} = \mathcal{P}(\text{Rtype}) \]
\[ \gamma \in \text{Gtype} = \text{Rtype} \to \text{Bool} \]
\[ \sigma \in \text{Signature} = \text{Rtype} \to \text{Vtype} \]

\[ \text{Field, BTag finite,} \]
\[ \text{Field} \cap \text{BTag} = \emptyset, \]
\[ \sigma \neq \emptyset. \]

Figure B.1: S-Net\textsubscript{A} syntax.

### B.2.1 S-Net\textsubscript{A} Syntax

Figure B.1 shows the syntax after the adaptation. All syntactic elements dealing with individual records or label values are removed or simplified. Comparing to the standard syntax, S-Net\textsubscript{A} has the following differences:

- Tags and fields are grouped into one single set Field, because their difference is reflected only when the actual record data exists.

- For structural simplicity, boxes and signed networks are written inline within the topology expressions, instead of defined separately and referred to by names. Unsigned networks are simply replaced by their topology expressions.

- Boxes do not have names because the external functions referred to by the box names, which manipulate the values in the records, are not checked by the type system.

- Filters are constructed as boxes, where the output types may be empty, and all the conditionals are removed.

- Synchrocells do not exist; a synchrocell is simulated by a parallel composition of as many boxes as necessary: the first box takes in the main record type and responds with the main record type and the synchronised record type, and the remaining boxes take in and pass through each auxiliary record type defined in the synchrocell.

- The second form of parallel composition (Bar) is internal. The route inference (see Section B.4) translates all parallel compositions of the first form (Bar\textsubscript{0}) to the second, associating every branch with the routing information, i.e. a set of record types the branch is deemed to attract.

- There are no deterministic operator variants.
The termination pattern $\gamma$ of a serial replication is represented by a partial function, whose domain, $\tau_0 = \text{dom}(\gamma)$ is the set of record types defined in the terminating pattern, and for each $r \in \tau_0$, $\gamma(r)$ (short for $\gamma(r) = \text{true}$) means the record type is associated with a guard, and $\neg \gamma(r)$ means the record type is unconditional.

Despite the list above, even manually translating a standard S-Net program into its abstract interpretation should not inflict any troubles, except the routing information, which the route inference procedure explained in Section B.4 will help. The subtype relation defined in [25] is reprised below using S-Net_A terms: $r_1$ is a subtype of $r_2$, denoted as $r_1 \subseteq r_2$, iff

$$r_2 \subseteq r_1 \land (r_1 \cap \text{BTag}) = (r_2 \cap \text{BTag}).$$

### B.2.2 Type Transformations

Every topology expression in a standard S-Net program can accept certain types of records and produce other types of records in response. In the abstract interpretation, we view it as the topology expression transforming some types into other types. In reality, the declared output types of a box or network only show what may be output: the box or network needs not produce all the declared output types; it needs not even produce a single record to comply with the declaration. However, in the abstract interpretation, we consider all mays as musts, for the type system to capture all possible cases and perform a correct type check.

The following judgment denotes that the topology expression $e$ can transform the input type $r$ to any output types in $\tau$, which may be an empty set:

$$e \vdash r \leadsto \tau.$$  

The judgment holds iff there is a deduction process that concludes it, using the type transformation rules in the remaining parts of this section.

The type transformation rules reflect the actual behaviour of the S-Net runtime.

#### B.2.2.1 Boxes

A box declares an input type and zero or more output types. Naturally, it transforms the declared input type into the declared set of output types.

**Flow Inheritance** A box also accepts all subtypes of the declared input type. For each such type, where there are excess labels, it behaves as if these labels are stripped off from the type before it is fed into the box, and added back to each of the output types. In other words, the excess labels bypass the
box. On adding back the labels, duplicates are naturally eliminated by the nature of simple sets.

\[
\begin{align*}
\frac{r' \sqsubseteq r}{\text{box} \ (r \rightarrow \{v_i\}) \vdash r' \rightarrow \{v_i \cup (r' \setminus r)\}} \quad \text{ttBox}
\end{align*}
\]

In the rule above, \(\{\rightarrow\}\) is a “set-of” notation whose usage is as follows: \(\{v_i\}\) is a shortcut notation of \(\{v_i \mid i = 1..m\}\). When this is pattern-matched with a concrete set, it essentially assigns an index to every element of the set (in arbitrary order) and binds \(m_i\) to a concrete integer. It is invalid to refer to \(m_i\) directly, but later uses of the variable \(i\) are confined within other set-of notations or must be universally quantified, to enforce the same range \(i = 1..m_i\). For example, \(\{v_i \cup (r' \setminus r)\}\) in the rule above is short for \(\{v_i \cup (r' \setminus r) \mid i = 1..m_i\}\), referring to the same upper bound. Note that a set constructed with the set-of notation can bind with an empty set, in which case \(i\) has a nonexistent range.

### B.2.2.2 Serial Compositions

A serial composition \(e_1..e_2\) behaves like a function composition: \(e_1\) transforms an input type into a set of intermediate types, and \(e_2\) transforms each of them to a set of output types. The union of all sets of output types become the result of the transformation.

\[
\begin{align*}
& e_1 \vdash r \leadsto \{v_i\} \\
& \forall i : (e_2 \vdash v_i \leadsto \tau_i) \\
& e_1..e_2 \vdash r \leadsto \bigcup_i \tau_i \quad \text{ttDotdot}
\end{align*}
\]

We also define that the big union operator on a nonexistent range results in an empty set. This definition is used by the rule above as well as other rules with big union operators in this chapter. This means that \(e_1 \vdash r \leadsto \emptyset\) immediately implies \(e_1..e_2 \vdash r \leadsto \emptyset\) without having to look into \(e_2\).

### B.2.2.3 Parallel Compositions

A parallel composition with routing information, \((\tau_1)e_1|(\tau_2)e_2\), lets either \(e_1\) or \(e_2\), whichever better attracts the input type, to perform the transformation. The function below selects the best matching types in \(\tau_1\) or \(\tau_2\) relative to the actual input type \(r\):

\[
BM(r, \tau) \triangleq \text{let } \tau' = \{v_0 \mid v_0 \in \tau \land r \sqsubseteq v_0\} \text{ in } \\
\{v \mid v \in \tau' \land |v| = \max \{|v_0| \mid v_0 \in \tau'\}\}.
\]

The topology expression \(e_1\) better attracts \(r\) compared to the other topology expression \(e_2\), iff the types in the best match set \(BM(r, \tau_1)\) have more labels than those in \(BM(r, \tau_2)\):

\[
\begin{align*}
\exists r_1, r_1 \in BM(r, \tau_1) \land \forall r_2, r_2 \in BM(r, \tau_2) \implies |r_1| > |r_2| \\
(e_1 \vdash r \leadsto \tau) \quad (\tau_1)e_1|(\tau_2)e_2 \vdash r \leadsto \tau \quad \text{ttBar1}
\end{align*}
\]
Note that the function BM guarantees that all types in the resulting set have the same cardinality, i.e. number of labels. The use of the quantifiers in the rule above guarantees that BM(\(r, \tau_1\)) is nonempty, and allows \(e_1\) to be chosen when BM(\(r, \tau_2\)) is empty.

The other route can be covered by equivalence:

\[(\tau_1)e_1|((\tau_2)e_2 \equiv (\tau_2)e_2|((\tau_1)e_1).\]

In case the function has collected types of equal cardinality from both branches, the parallel composition lets both topology expressions transform the input type, and then merges the output types:

\[\exists r_1, r_2. r_1 \in \text{BM}(r, \tau_1) \land r_2 \in \text{BM}(r, \tau_2) \land |r_1| = |r_2|\]

\[e_1 \vdash r \rightsquigarrow \tau_1\]

\[e_2 \vdash r \rightsquigarrow \tau_2\]

\[(\tau_1)e_1|((\tau_2)e_2 \vdash r \rightsquigarrow \tau_1 \cup \tau_2)\]

Careful readers may think that the last rule does not comply with the standard S-Net semantics, where each input record is delivered to only one of the parallel branches nondeterministically. However, the decision process is done per record, and both branches have the probability to be selected. As explained at the beginning of Section B.2.2, all possibilities are considered in the abstract interpretation, and therefore both cases are included in the result.

B.2.2.4 Serial Replications

The abstract interpretation of a serial replication \(e \ast \gamma\) is rather tricky to define. It involves a serial composition chain of as many replicas of \(e\) as necessary to transform all non-terminating intermediate types into terminating output types. We will employ a template approach to find the overall type transformation.

For each serial replication \(e \ast \gamma\), we define a unique special binding tag \(\Lambda_{e \ast \gamma} \in BTag\), which is unused throughout the topology of \(e\) and the entries in \(\gamma\). This is to tag the types that match the terminating pattern and are due to output from the serial chain. Adding such binding tag guarantees that the tagged types will never fit into, and hence be transformed by another replica of \(e\).

Let \(f_{e \ast \gamma} \in TopoExpr\) be a selection filter that tags any terminating types with \(\Lambda_{e \ast \gamma}\) and passes through any other types. If a type is conditionally terminating, then the filter generates both tagged and non-tagged copies:

\[\forall r_0, r_0 \in \text{dom}(\gamma) \implies r \not\subseteq r_0\]

\[f_{e \ast \gamma} \vdash r \rightsquigarrow \{r\}\]

\[\exists r_0, r_0 \in \text{dom}(\gamma) \land r \subseteq r_0 \land \neg \gamma(r_0)\]

\[f_{e \ast \gamma} \vdash r \rightsquigarrow \{r \cup \{\Lambda_{e \ast \gamma}\}\}\]

\[\exists r_0, r_0 \in \text{dom}(\gamma) \land r \subseteq r_0\]

\[\forall r_0, r_0 \in \text{dom}(\gamma) \land r \subseteq r_0 \implies \gamma(r_0)\]

\[f_{e \ast \gamma} \vdash r \rightsquigarrow \{r \cup \{\Lambda_{e \ast \gamma}\}, r\}\]
Note that any type with $\Lambda_{e+\gamma}$ falls into the non-terminating category and is carried over unchanged according to the first rule.

Let $e_{e+\gamma} \in TopoExpr$ be a variation of the topology expression $e$ which, in addition to the type transformations by $e$, can pass through any type with $\Lambda_{e+\gamma}$ unchanged:

$$
\frac{e \vdash r \leadsto \tau}{e_{e+\gamma} \vdash r \leadsto \tau} \quad \text{ttStarEn}
$$

$$
\frac{\Lambda_{e+\gamma} \in \tau}{e_{e+\gamma} \vdash r \leadsto \{r\}} \quad \text{ttStarEp}
$$

Define a new topology expression form $e^{*m}\gamma$, with the following meaning:

$$
e^{*m}\gamma \triangleq f_{e+\gamma} \cdot (e_{e+\gamma} \cdot f_{e+\gamma} \cdot \cdots),
$$

for which we can deduce type transformations using the rule for serial composition. Every element in this serial chain passes through $\Lambda_{e+\gamma}$-tagged types unchanged, which without the tag are the real output types. The following helper function extracts these output types:

$$
\text{Outs}(\tau, \Lambda_{e+\gamma}) \triangleq \{ v \setminus \{\Lambda_{e+\gamma}\} \mid v \in \tau \land \Lambda_{e+\gamma} \in v \}.
$$

Because the number of labels used in the topology is finite, we will reach a fixed point where the set of output types no longer grows by increasing $m$. Then, the set of output types, without the tag, becomes the transformation result of the serial replication. This process is formalised below, where $m_0$ refers to the value of $m$ at the fixed point:

$$
\begin{align*}
\Lambda_{e+\gamma} \notin \tau \\
e^{*m_0}\gamma \vdash r \leadsto \tau_1 \\
\tau &= \text{Outs}(\tau_1, \Lambda_{e+\gamma}) \\
\forall i. \ i > m_0 &\implies \exists \tau_2. \ e^{*i}\gamma \vdash r \leadsto \tau_2 \land \text{Outs}(\tau_2, \Lambda_{e+\gamma}) = \tau \\
\hline
e^{*}\gamma \vdash r \leadsto \tau &\quad \text{ttStar}
\end{align*}
$$

B.2.2.5 Parallel Replications

As far as the type system is concerned, a parallel replication simply demands the label $l$ in the input types. The layout of the parallel replication and the delivery of records are insignificant in the abstract interpretation.

$$
\begin{align*}
\vdash l \in r \\
e \vdash r \leadsto \tau &\quad \text{ttEx}
\end{align*}
$$

B.2.2.6 Signed Networks

A signed network defines a closed environment, only exposing a user-declared interface, i.e. the signature. Provided the signed network passes the type check, we simply use its signature to perform our type
transformations.

Because a network signature may present multiple type transformations, a procedure similar to the best match choice for parallel compositions (see Section B.2.2.3) is applied at the entrance of the network. Also, a higher level of flow inheritance takes place at the borders of a signed network: after the best match decision, excess labels that are in the input but not in the chosen declared input type are flow inherited to the exit of the network.

\[
e \triangleleft \sigma = \{ r \} = BM(r, \text{dom}(\sigma)) \neq \emptyset \quad \text{ttSubnet}
\]

In the rule above, \( e \triangleleft \sigma \) means the topology expression agrees with the signature. Full definition will be covered in Section B.3.

The same rule also describes the abstract interpretation of the execution of the whole program, which itself is a signed network.

### B.3 Type Check

The judgement \( e \triangleleft \sigma \) denotes that the topology expression \( e \) agrees with the signature \( \sigma \). It holds iff there exists a deduction procedure that concludes it, using the type transformations in Section B.2.2 and the type check rule defined below:

\[
\forall r, \tau_0. r \in \text{dom}(\sigma) \land \sigma(r) = \tau_0 \implies \\
\exists \tau : (e \vdash r \rightsquigarrow \tau) \land \tau \sqsubseteq \tau_0 \quad \text{tc}
\]

In the rule above, the subtype relation between two variant types is defined as follows: \( \tau_1 \) is a subtype of \( \tau_2 \), denoted as \( \tau_1 \sqsubseteq \tau_2 \), iff

\[
\forall r_1, r_2. r_1 \in \tau_1 \implies \exists r_2. r_2 \in \tau_2 \land r_1 \sqsubseteq r_2.
\]

The rule requires that all mappings in the signature are backed up by the type transformations, but the reverse is not necessary.

If for a signed network, the topology expression agrees with the declared signature, we say the network passes the type check. An S-Net \( \Lambda \) program is type safe if it passes the type check, meaning that type errors will never occur.

### B.4 Route Inference

The parallel composition type transformation (see Section B.2.2.3) can be performed only with the routing information. It is the task of the route inference to provide such information.
The route inference procedure finds out what types of records are useful for every parallel branch. A type is useful if all labels in the type are potentially used within the branch topology: consumed by a box or a signed network, contributed to a terminating check in a serial replication, or demanded by a parallel replication. A set of these types is then used as the routing information of the branch, for it to attract the appropriate records in competition with the other branch of the same parallel composition.

The route inference procedure consists of two types of computation: collecting the useful types (sig inference), and transforming parallel compositions in the form of $e_1|e_2$ into $(\tau_1)e_1|\tau_2)e_2$ (route inference). They do not occur one after another: by performing route inference on the whole program, the sig inference procedure will be called on demand.

### B.4.1 Foundations

#### B.4.1.1 Required, Guaranteed and Discarded Labels

Consider a box $A$ with the input type $r$ and the output type set $\{v\}$. The box accepts any subtype of $r$, where $r$ being the minimum, as in the number of labels in the set. The labels defined in $r$ are required by the box. Obviously, a type where all labels are required is useful.

To find out the useful types for the whole topology expression as a parallel branch, we need to consider other parts of the branch topology. Suppose the branch is a serial composition of box $A$ and box $B$, the latter requiring $r'$. The labels in $r'$ may originate from two places: guaranteed by box $A$, or provided in the input type to box $A$ and bypassed by flow inheritance (see Section B.2.2.1). The output from box $A$ contains the labels in $v$ plus the excess labels from the input, which means the labels in $v$ are guaranteed even in the case of minimum input.

We need to request the labels not guaranteed by the first operand of the serial composition. To do so, we add the labels in $r'\setminus v$ to the overall required label set and depend on flow inheritance to carry them across box $A$, so that the output from box $A$ is acceptable by box $B$. However, some labels cannot be carried over, namely those in $r\setminus v$, because box $A$ has consumed them. We mark these labels discarded to remind ourselves that they are unavailable to the second operand of the serial composition. If any of the discarded labels are later required, we declare that a type error is found.

In conclusion, to successfully find out the useful types, we need to collect the required, guaranteed, and discarded label sets throughout the branch topology, the latter two being the helpers to infer the correct required label sets.

#### B.4.1.2 Signatures for Route Inference

A box may feature multiple output types. Because the discarded label set is calculated by subtracting the output type from the input type, each of the guaranteed label sets will be accompanied by its own discarded label set. The combination of one required label set and a set of guaranteed-discarded label set pairs is like an extended version of the box signature.

Multiple boxes may be combined in a parallel composition, where each branch has its own extended box signature. We have hence formed a structure of a set of extended box signatures, which looks like an
\[ \Sigma \in Sig = \mathcal{P}(Map) \]
\[ (r \rightarrow \omega) \in Map = Rtype \times Otype \]
\[ \omega \in Otype = \mathcal{P}(Ovar) \]
\[ (v[\delta]) \in Ovar = Rtype \times \mathcal{P}(Field) \]

\[ \forall \Sigma, r, \omega_1, \omega_2. \ (r \rightarrow \omega_1) \in \Sigma \land (r \rightarrow \omega_2) \in \Sigma \implies \omega_1 = \omega_2 \]

\[ \forall \Sigma, r, \omega, v, \delta. \ (r \rightarrow \omega) \in \Sigma \land v[\delta] \in \omega \implies v \cap \delta = \emptyset \land (r \setminus BTag) \subseteq v \cup \delta. \]

Figure B.2: Signature for route inference structure.

extended version of the network signature. This structure suffices for any topology expression in general.

Figure B.2 shows the structure for an extended network signature, abbreviated a \textit{sig}. We also name the extended box signature as a \textit{map}. The difference between a \textit{sig} and a network signature is that the \textit{sig} features the discarded label sets, one per output type per map. A set structure is used instead of a partial function, for easier construction of the \textit{sig}s, while the first constraint in the figure maintains the functional behaviour. We also define the following helper function to retrieve the input types of a \textit{sig}, mirroring the domain function ‘dom’ for partial functions:

\[ \text{Ins}(\Sigma) \triangleq \{ r \mid (r \rightarrow _) \in \Sigma \}. \]

For readability, we use the format \( r \rightarrow \omega \) instead of \( (r, \omega) \) to denote the maps of a \textit{sig}, and \( v[\delta] \) instead of \( (v, \delta) \) to denote the individual variants of the output.

Having the definition of \textit{sig}s, our task to find out the useful types becomes a procedure of \textit{sig} inference. We will now present the \textit{sig} inference formalisation.

### B.4.2 Sig Inference Rules

The judgement \( e : \Sigma \) means the \textit{sig} inference assigns the topology expression \( e \) with the \textit{sig} \( \Sigma \). The \textit{sig} inference rules have a name prefix \textit{si}.

#### B.4.2.1 Boxes

For \textit{boxes}, the inference rule merely extends the \textit{box} signature with the discarded label sets to obtain the \textit{sig}.

\[ \boxed{(r \rightarrow \tau) : \{ r \rightarrow \{ v[(r \setminus BTag) \setminus v] \mid v \in \tau \} \}} \text{Box} \]

#### B.4.2.2 Parallel Compositions

The \textit{sig} inference of a parallel composition uses the routing information to filter the branch \textit{sig}s. The union of the branch \textit{sig}s is the result of the \textit{sig} inference. To maintain the \textit{sig} invariants, a custom union is used.
\[ e_1 : \Sigma_1 \\
\Sigma_2 = \{ (r \rightarrow \omega) \in \Sigma_1 \mid \{ r \} \subseteq \tau \} \quad (i = 1, 2) \]

where

\[
\Sigma_1 \sqcup \Sigma_2 \triangleq \{ r \rightarrow \bigcup_{(r \rightarrow \omega) \in \Sigma_1 \sqcup \Sigma_2} \omega \mid r \in \text{Ins}(\Sigma_1) \cup \text{Ins}(\Sigma_2) \}. 
\]

If the routing information is absent, the rule below is used instead, which uses the whole set of all types as the initial routing information for both branches. This effectively causes the rule siBar not to filter any maps away from the final sig.

\[
(Rtype)e_1|(Rtype)e_2 : \Sigma \\
e_1|e_2 : \Sigma 
\]

Note that only the parallel composition inference rules will increase the number of maps in a sig. In other words, the presence of multiple maps in a sig implies the existence of parallel compositions in the topology.

**B.4.2.3 Serial Compositions**

Unlike the type transformation rules, dealing with sigs for serial compositions is a lot more complex. To make the wording more concise, we assume the serial composition \( e_1..e_2 \) where \( e_1 : \Sigma_1 \) and \( e_2 : \Sigma_2 \). For each map \( (r_1 \rightarrow \omega_1) \in \Sigma_1 \), we need to augment \( r_1 \), the required label set, to make all output types accepted by \( \Sigma_2 \). Four questions need to be answered: 1) what labels to add, 2) whether the augmented input creates a reroute, 3) what behaviour can be observed with \( \Sigma_1 \) and the augmented input, and 4) what are the final outputs. We will answer them one by one, and finally present the route inference rule.

The first question, what labels to add, is a tough one. \( \omega_1 \) may contain multiple output variants. To adapt an output variant \( v[\delta] \in \omega_1 \) to fit into \( \Sigma_2 \), we add into \( v \) zero or more fields, \( \epsilon \subseteq \text{Field} \), which are absent from \( \delta \), so the resulting \( v' = v \cup \epsilon \) is a subtype of some input type \( r_2 \) in \( \Sigma_2 \). We keep the added labels minimum, so every label in \( v' \) is required by \( \Sigma_2 \), or otherwise guaranteed by \( \Sigma_1 \).

However, because \( \Sigma_2 \) may contain multiple maps, each variant in turn may be adapted in multiple ways. Collecting these options of added labels results in a set of label sets per output variant. We name this set the **requirement variants** of the corresponding output variant, and define the function below for calculating it:

\[
\text{RV}(v[\delta], \Sigma_2) \triangleq \left\{ \epsilon \mid \epsilon \subseteq \text{Field} \land \epsilon \cap \delta = \emptyset \land \\
r_2 \in \text{Ins}(\Sigma_2) \land \epsilon = r_2 \setminus v \right\}.
\]

Note that the criteria \( \epsilon \subseteq \text{Field} \) and \( \epsilon = r_2 \setminus v \) together have guaranteed \( (v \cup \epsilon) \subseteq r_2 \).

Now, to make the whole map valid to combine with \( \Sigma_2 \), the augmentation to \( r_1 \) should satisfy at least one requirement variant per output variant. By selecting an arbitrary requirement variant for each output variant and calculating the union of them, we have one sample of augmentation. All possible
augmentations are captured by the function below:

\[ PA\left(\left\{ v_i[\delta_i]\right\}, \Sigma_2\right) \triangleq \left\{ \bigcup_i \epsilon_i \mid \forall \epsilon_i \in RV(v_i[\delta_i], \Sigma_2) \right\} , \]

where the set-of construct \( \left\{ \rightarrow \right\} \) is as defined in Section \[B.2.1\]. Note that

\[ PA(\emptyset, \Sigma_2) = \{\emptyset\} \]

according to the formula, because when there is no range for \( i \), the \( \forall \)-quantified criterion always holds, generating infinite cases to evaluate the pattern, which is a union over an nonexistent range, resulting in \( \emptyset \) by definition. The set of an infinite number of \( \emptyset \) is \( \{\emptyset\} \). The meaning of this result is as follows: for a map with no outputs, there is one possible augmentation, which is to add nothing to the input. The fact that \( e_1 \) produces no outputs does not indicate a type error. Only when one of the output variants cannot match with \( \Sigma_2 \) by any means will the result be \( \emptyset \), which signals a type error.

The second question, whether a reroute is created, originates from the nature of parallel compositions. As mentioned in Section \[B.4.2.2\] multiple maps are a sign of parallel compositions. When we augment the input type, we may have created a better match elsewhere in \( \Sigma_1 \), preventing the selected map \( r_1 \rightarrow \omega_1 \) from being used. These cases should be avoided. As a result, the eligible augmented inputs relative to the map \( (r_1 \rightarrow \omega_1) \in \Sigma_1 \) and the sig \( \Sigma_2 \) are

\[ AI_0(\Sigma_1, (r_1 \rightarrow \omega_1), \Sigma_2) \triangleq \{ r_1 \cup \epsilon \mid \epsilon \in PA(\omega_1, \Sigma_2) \land r_1 \in BM(r_1 \cup \epsilon, Ins(\Sigma_1)) \} , \]

where the function BM has been defined in Section \[B.2.3\]. If the function PA returns \( \emptyset \), \( AI_0 \) also evaluates to \( \emptyset \), effectively disabling the map in question.

We can upscale the function above to the sig level and produce the overall eligible augmented inputs with the function below:

\[ AI(\Sigma_1, \Sigma_2) \triangleq \{ r \mid r \in AI_0(\Sigma_1, (r_1 \rightarrow \omega_1), \Sigma_2) \land (r_1 \rightarrow \omega_1) \in \Sigma_1 \} . \]

The third question, how \( \Sigma_1 \) transforms the augmented inputs, is answered with flow inheritance. Unlike in the type transformations where all possible cases, with or without flow inheritance, are covered, we only have the base cases available in a sig, where the input types are the required label sets. To infer other cases where flow inheritance carries unused fields across, we employ the evaluation judgment \( r \xrightarrow{\Sigma} \omega \) to mean the actual input \( r \) produces the outputs \( \omega \) according to the sig \( \Sigma \), which is governed by the rule below:

\[ \Sigma_0 = \{ r_0 \rightarrow \omega_0 \mid (r_0 \rightarrow \omega_0) \in \Sigma \land r_0 \in BM(r, Ins(\Sigma)) \} \]

\[ \omega = \bigcup_{(r_0 \rightarrow v_i[\delta_i]) \in \Sigma_0} \left\{ (v_i \cup (r_0 \setminus \delta_i))[\delta_i] \right\} \]

The rule above considers the parallel composition behaviour and passes the input through only the eligible routes, which are isolated into a new sig \( \Sigma_0 \). At the output, it also takes into account the discarded label
sets, and adds only those labels in the input type that are neither used nor discarded.

The last question, what the final outputs are, can be answered by applying the rule above in a
nested nature, like what we have done in Section [B.2.2.2] for serial composition type transforma-
tion. The difficulty now is that the intermediate outputs contain the discarded label sets. Appar-
tently they should be merged with the discarded label sets in $\Sigma_2$, but when $\Sigma_2$ reproduces any labels in the intermediate
discarded label sets, they are removed from the resulting discarded set.

We have so far figured out what are the new, augmented input types, and how one type transforms
into outputs according to the sig of the topology expression. Finally, the sig inference rule for serial
compositions is presented as follows:

\[
e_1 : \Sigma_1 \\
e_2 : \Sigma_2 \\
\Sigma = \left\{ r \rightarrow \bigcup_i \omega_i \mid r \in \text{AI}(\Sigma_1, \Sigma_2) \land r \Sigma_1 \rightarrow \{v_i(\delta_i)\} \land \\
\forall i. \; v_i \Sigma_2 \rightarrow \{v_j(\delta_j)\} \land \omega_i = \{v_j(\delta_j) \cup (\delta_i \setminus v_j)\} \right\}
\]

\[
\Sigma \neq \emptyset \\
e_1..e_2 : \Sigma
\]

The last precondition in the rule above assures that the inferred sig is non-empty, indicating that the
serial composition can at least perform some type transformations. An empty inferred sig suggests that
a type error is found.

### B.4.2.4 Serial Replications

Similar to the approach in Section [B.2.2.4] we define the route inference for serial replications using that
for serial compositions. The benefit we have now, comparing to then in Section [B.2.2.4] is that we can
directly manipulate the sig, avoiding the necessity of the special binding tag $\Lambda_{e+\gamma}$.

First of all, we trim the sig of the operand topology expression so it does not contain any maps
requiring inputs that are unconditionally terminated. The resulting ‘core’ sig can be computed with this
function:

\[
C(\Sigma, \gamma) \triangleq \{ r \rightarrow \omega \mid (r \rightarrow \omega) \in \Sigma \land r_0 \in \text{dom}(\gamma) \land r \subseteq r_0 \land \neg \gamma(r_0) \}.
\]

Then, we split the core sig, as well as every intermediate sig that comes later, into two parts, one
containing the terminating cases which include the ‘dead ends’ where there is no output:

\[
T(\Sigma_0, \gamma) \triangleq \{ r \rightarrow \omega' \mid (r \rightarrow \omega) \in \Sigma_0 \land (\omega = \omega' = \emptyset \lor \\
\omega' = \{v(\delta) \in \omega \mid \exists r_0 \in \text{dom}(\gamma) : v \subseteq r_0 \} \neq \emptyset\} \},
\]

and the other with the non-terminating cases:

\[
NT(\Sigma, \gamma) \triangleq \{ r \rightarrow \{v(\delta) \in \omega \mid \forall r_0 \in \text{dom}(\gamma) : r \subseteq r_0 \implies \gamma(r_0)\} \}
\mid (r \rightarrow \omega) \in \Sigma \land \omega \neq \emptyset\}.
\]
The non-terminating part of the core or intermediate sig will be combined in serial composition with another replica, resulting in a new intermediate sig which goes through the splitting again. The process goes on as long as the terminating part of the sig keeps growing. When the growth stops, we end the process by adding into the resulting sig the special cases where the input types terminate immediately:

\[ \text{TI}(\gamma) \triangleq \{ r \rightarrow \{ r[0] \} \mid r \in \text{dom}(\gamma) \}. \]

The complete sig inference rule is as follows:

\[
\begin{align*}
& e : \Sigma \\
& \Sigma_1 = \text{C}(\Sigma, \gamma) \\
& \Sigma'_1 = \text{T}(\Sigma_1, \gamma) \\
\forall i, i > 1 & \implies \Sigma'_i = \text{T}(\Sigma_i, \gamma) \cup \Sigma'_{i-1} \\
\forall i, i \geq 1 & \implies \Sigma''_i = \text{NT}(\Sigma_i, \gamma) \\
\forall i, e_1, e_2, i \geq 1 \land e_1 : \Sigma'_i \land e_2 : \Sigma_1 & \implies e_1 . e_2 : \Sigma_{i+1} \\
m > 1 & \\
\Sigma'_m = \Sigma'_{m-1} & \\
\implies e . \gamma : \Sigma'_m \cup \text{TI}(\gamma) & \text{si-Star}
\end{align*}
\]

where \( \Sigma_1 \cup \Sigma_2 \) is as defined in Section B.4.2.2.

B.4.2.5 Parallel Replications

The sig inference for a parallel replication \( e!l \) requires the label \( l \) from the input. It then reuses the evaluation judgement \( r \xrightarrow{\Sigma} \omega \) defined in Page 165 to obtain the result. When this type system was first published, it was allowed that \( l \in \text{BTag} \). Because this can affect the subtype relation, special care is needed in the rule, hence the term \( \{ l \} \cup r \sqsubseteq r \).

\[
\begin{align*}
& e : \Sigma \\
& \tau = \{ \{ l \} \cup r \mid r \in \text{Ins}(\Sigma) \land \{ l \} \cup r \sqsubseteq r \} \neq \emptyset \\
& e!l : \{ r \rightarrow \omega \mid r \in \tau \land r \xrightarrow{\Sigma} \omega \} & \text{siEx}
\end{align*}
\]

B.4.2.6 Signed Networks

The programmer provides a signature for the program and every signed network in the program. To meet the programmer’s expectation, the route inference for signed networks simply extends the signature with the discarded label sets to form a sig, and leaves its correctness to be checked by the type check procedure.
The rule below reuses the sig inference rules for boxes to construct the sig.

\[
\begin{align*}
\text{dom}(\sigma) &= \{\tau_i\} \\
\forall i. \text{box}(r_i \rightarrow \sigma(r_i)) : \Sigma_i \\
\text{net}(\sigma) = \bigcup_{i} \text{SiSubnet}
\end{align*}
\]

where \( \bigcup \) is the aggregation operator based on \( \Sigma_1 \sqcup \Sigma_2 \) defined in Section B.4.2.2.

### B.4.3 Route Inference Rules

The judgment \( \tau, e \rightarrow_{\text{ri}} e' \) expresses that the topology expression \( e \), provided the set of applicable input types \( \tau \), is transformed to \( e' \) by route inference. In \( e' \), all parallel compositions have the form \( (\tau_1)e_1 | (\tau_2)e_2 \).

The set \( \tau \) does not only contain the types accepted by \( e \); it is in fact the complement of the set of types that the route inference determines inapplicable, and will be used to exclude certain types from the routing information. In some cases where exclusion should be suppressed, \( \tau \) will be the universal set \( \text{Rtype} \), as can be seen from some rules below.

Route inference rules have a name prefix \( ri \).

#### B.4.3.1 Boxes

No transformation is required to perform the route inference for boxes, the base unit of the topology.

\[
\begin{align*}
\exists r_0. r_0 \in \tau_0 \land r_0 \sqsubseteq r \\
\tau_0, \text{box}(r \rightarrow \tau) \rightarrow_{\text{ri}} \text{box}(r \rightarrow \tau)
\end{align*}
\]

#### B.4.3.2 Parallel Compositions

The route inference rules for parallel compositions trigger the route inference for each branch and then collect the inputs of the branch sigs as the routing information. The set of applicable types is used to initiate the route inference, which filters the accepted input types of each branch.

\[
\begin{align*}
\tau_0, (\tau_0)e_1 | (\tau_0)e_2 \rightarrow_{\text{ri}} (\tau_1)e'_1 | (\tau_2)e'_2 \\
\tau_0, e_1 | e_2 \rightarrow_{\text{ri}} (\tau_1)e'_1 | (\tau_2)e'_2
\end{align*}
\]

\[
\tau''_i = \{ r \mid r \in \tau_i \land \exists r_0. r_0 \in \tau_0 \land r_0 \sqsubseteq r \} \quad (i = 1, 2)
\]

\[
\tau'_i, e_i \rightarrow_{\text{ri}} e'_i \quad (i = 1, 2)
\]

\[
\tau'_i : \Sigma_i \quad (i = 1, 2)
\]

\[
\tau'_i = \{ r \mid r \in \text{Ins}(\Sigma_i) \land \exists r_0. r_0 \in \tau_0 \land r_0 \sqsubseteq r \} \quad (i = 1, 2)
\]

\[
\tau, (\tau_1)e_1 | (\tau_2)e_2 \rightarrow_{\text{ri}} (\tau'_1)e'_1 | (\tau'_2)e'_2
\]
B.4.3.3 Serial Compositions

The rule below triggers the route inference for the two operands of the serial composition:

\[
\begin{align*}
\tau_0, e_1 \rightarrow_{\text{a}} e'_1 \\
R_{\text{type}}, e_2 \rightarrow_{\text{a}} e'_2 \\
\tau_0, (e_1 .. e_2) \rightarrow_{\text{a}} e'_1 .. e'_2 & \rightarrow_{\text{riSerial}}
\end{align*}
\]

where we use the universal set \( R_{\text{type}} \) to start the route inference for \( e_2 \), because the benefit to use the set of ‘all types that are subtypes of some output types mentioned in the sig of \( e'_1 \)’ is outweighed by the trouble to collect such set.

B.4.3.4 Serial Replications, Parallel Replications

No special care is needed for the route inference for serial and parallel replications.

\[
\begin{align*}
\tau_0, e \rightarrow_{\text{a}} e' & \rightarrow_{\text{riStar}} \\
\pi_0, e \rightarrow_{\text{a}} e' & \rightarrow_{\text{riEx}}
\end{align*}
\]

B.4.3.5 Signed Networks

A signed network defines the boundary of components. The route inference does not propagate the input type information into the networks, but uses the declared input types to perform the route inference.

\[
\begin{align*}
\text{dom}(\sigma), e \rightarrow_{\text{a}} e' & \rightarrow_{\text{riSubnet}}
\end{align*}
\]

The rule above doubles as the process to prepare an initial \( S\text{-Net}_A \) program with the routing information, before performing the type check.

B.5 Summary

The type system presented in this appendix enables the compiler to perform three tasks: type check, sig inference and route inference. The compiler first performs a route inference to populate an \( S\text{-Net} \) program with the routing information for the runtime to make routing decisions for each record, and then does a type check to guarantee the type safety of the program.

The sig inference procedure is called on demand during the route inference. It attempts to infer a sig for each part of the topology expressions in the program. Its failing to do so indicates that a type error is found. A sig resembles a network signature extended with the discards. The route inference procedure collects the routing information from the sigs of the parallel branches.

This type system has been implemented in the latest \( S\text{-Net} \) compiler published as of the time of
writing this thesis. It is able to handle most common cases correctly, and the routing information it collects from the inferred sigs usually agrees with the programmers' expectations.
Appendix C

Code and Output Samples

This appendix contains the code implementing the type inference algorithm of the new type system, and some sample outputs by the prototype compiler. For more information, see Section 7.2.2.

C.1 Type Inference Algorithm Implementation

```csharp
public static class TypeInference
{
    public static Rep Box(Value v0, IEnumerable<Value> vs0)
    {
        return new Rep
        {
            new Subdomain(v0), vs0.Select(v1 => new OutValue(v0, v0 + v1.Labels, v1))
        };
    }

    private static IEnumerable<Case> Serial3(Subdomain sigma, OutValue omega, Case kappa2)
    {
        var sigmaPrime = sigma.Refine(omega, kappa2.Subdomain);
        if (sigmaPrime.IsEmpty)
        {
            yield break;
        }
        yield return new Case(
            sigmaPrime,
            kappa2.OutValues.Select(omega2 => new OutValue(
                omega.Used + (omega2.Used - omega.Added),
    }

    private static IEnumerable<Case> Serial2(Subdomain sigma, OutValue omega, Rep rho2)
    {
        return rho2.SelectMany(kappa2 => Serial3(sigma, omega, kappa2));
    }

    private static IEnumerable<Case> Serial1(Case kappa, Rep rho2)
    {
        if (kappa.OutValues.Count == 0)
        {
            return new[] { kappa };
        }
        var resultsPerOmega = kappa.OutValues.Select(omega => Serial2(kappa.Subdomain, omega, rho2));
        var potentialKappaPrimes = from kappasInCp in CartesianProduct(resultsPerOmega)
            select kappasInCp.Aggregate((kappa1, kappa2) =>
                kappa1.Concat(kappa2));
```
new Case(kappa1.Subdomain.Intersect(kappa2.Subdomain),
kappa1.OutValues.Union(kappa2.OutValues));
return potentialKappaPrimes.Where(kappaPrime => !kappaPrime.Subdomain.IsEmpty);
}

private static IEnumerable<IEnumerable<Case>> CartesianProduct(
IEnumerable<IEnumerable<Case>> kappass)
{
    IEnumerable<IEnumerable<Case>> seed = new[] { Enumerable.Empty<Case>());
    return kappass.Aggregate(seed, (cp, kappas) =>
        from p in cp
        from kappa in kappas
        select p.Concat(new[] { kappa }));
}

public static Rep Serial(Rep rho1, Rep rho2)
{
    return new Rep(rho1.SelectMany(kappa => Serial1(kappa, rho2)));
}

public static Rep Parallel(Rep rho1, Rep rho2)
{
    return new Rep(ParallelCasesFromBranchJ(rho1, rho2)
        .Union(ParallelCasesFromBranchJ(rho2, rho1))
        .Union(ParallelCasesFromBothBranches(rho1, rho2)));
}

private static IEnumerable<Case> ParallelCasesFromBranchJ(Rep rhoJ, Rep rhoK)
{
    return from kappaJ in rhoJ
    from sigmaPrime in rhoK.Aggregate(
        (IEnumerable<Subdomain>)new[] { kappaJ.Subdomain },
        (sigmas, kappaK) =>
            kappaK.LargestUsedPartSize >= kappaJ.LargestUsedPartSize
                ? Reduce(sigmas, kappaK.Subdomain)
                : sigmas)
    where !sigmaPrime.IsEmpty
    select new Case(sigmaPrime, kappaJ.OutValues);
}

private static IEnumerable<Subdomain> Reduce(
    IEnumerable<Subdomain> sigmaParts, Subdomain toSubtract)
{
    return from sigma in sigmaParts
    from sigmaPrime in sigma.Subtract(toSubtract)
    where !sigmaPrime.IsEmpty
    select sigmaPrime;
}

private static IEnumerable<Case> ParallelCasesFromBothBranches(Rep rho1, Rep rho2)
{
    return from kappa1 in rho1
    from kappa2 in rho2
    where kappa1.Subdomain.Intersects(kappa2.Subdomain)
        && kappa1.LargestUsedPartSize == kappa2.LargestUsedPartSize
    select new Case(kappa1.Subdomain.Intersect(kappa2.Subdomain),
kappa1.OutValues.Union(kappa2.OutValues));
}

public static Rep Star(Rep rhoN, IEnumerable<Value> vs1, IEnumerable<Value> vs2)
{
    Rep rho = Tagger(rhoN, vs1, vs2);
    Rep rhoPrime = Parallel(Serial(rhoN, rho), Passer(vs1, vs2));
    IEnumerable<Subdomain> delta = rho.Domain.ToList();
    List<Tuple<Subdomain, OutValue>> beta = new List<Tuple<Subdomain, OutValue>>(
        from kappa in rho
        from omega in kappa.OutValues
        select Tuple.Create(kappa.Subdomain, omega));
    bool cont = true;
    while (cont)
    {
        cont = false;
    }
}
rho = Serial(rho, rhoPrime);
// Check if domain has shrunk:
IEnumerable<Subdomain> diff = rho.Domain.Aggregate(delta, Reduce);
if (diff.Any())
{
    // After reducing the old domain with the new domain something still remains:
    // domain has shrunk.
    cont = true;
    delta = rho.Domain.ToList();
}
// Check if any subdomain-output pair is new:
foreach (var kappa in rho)
{
    var omegasInBetaCoveringKappasSigma =
        from beta0 in beta
        where beta0.Item1.Precedes(kappa.Subdomain)
        select beta0.Item2;
    var tuplesForMissingOmegas =
        from omega in kappa.OutValues
        where !omegasInBetaCoveringKappasSigma.Contains(omega)
        select Tuple.Create(kappa.Subdomain, omega);
    if (tuplesForMissingOmegas.Any())
    {
        beta.AddRange(tuplesForMissingOmegas);
        cont = true;
    }
}
return Serial(rho, Finalizer(rhoN, vs1, vs2));
}
private static IEnumerable<Value> Root(Rep rho)
{
    var root = new SortedSet<LabelSet>(from sigma in rho.Domain select sigma.Binds);
    return root.Select(ls => new Value(ls, LabelSet.Empty));
}
private static Rep Tagger(Rep rho, IEnumerable<Value> vs1, IEnumerable<Value> vs2)
{
    var boxesForVs1 = from v1 in vs1
        select Box(v1, new[] { v1 + Value.Star });
    var boxesForVs2 = from v2 in vs2
        where !vs1.Any(v1 => v1.Precedes(v2))
        select Box(v2, new[] { v2, v2 + Value.Star });
    var boxesForRho = from v3 in Root(rho)
        where !vs1.Union(vs2).Any(v0 => v0.Precedes(v3))
        select Box(v3, new[] { v3 });
    return boxesForVs1.Union(boxesForVs2).Union(boxesForRho).Aggregate(Parallel);
}
private static Rep Passer(IEnumerable<Value> vs1, IEnumerable<Value> vs2)
{
    var boxes = from v0 in vs1.Union(vs2)
        select Box(v0 + Value.Star, new[] { v0 + Value.Star });
    return boxes.Aggregate(Parallel);
}
private static Rep Finalizer(Rep rho, IEnumerable<Value> vs1, IEnumerable<Value> vs2)
{
    var boxesForTerms = from v0 in vs1.Union(vs2)
        select Box(v0 + Value.Star, new[] { v0 });
    var boxesForInfinite = from v0 in Root(rhs)
        select Box(v0, Enumerable.Empty<Value>());
    return boxesForTerms.Union(boxesForInfinite).Aggregate(Parallel);
}
public static Rep Sync(Value main, IEnumerable<Value> aux)
{
    Rep mainRep = Box(main, new[] { main, aux.Aggregate(main, (v1, v2) => v1 + v2) });
    var auxReps = from bs in aux.Select(v => new Value(v.Binds, LabelSet.Empty))
        select Box(bs, new[] { bs });
    return Parallel(mainRep, auxReps.Aggregate(Parallel));
}
C.2 Sample Outputs

The verbose output from the prototype compiler details the type inference process, printing the rep for each definition and topology subexpression. For a signed network, the output contains the rep inferred for its topology, as well as the rep constructed from its signature.

The compiler generates the following verbose output when given the program `compare` in Section 6.3:

```
Box m1 at compare:5:9:
{x/0 -> (u:x,d:ax,a:a) 
Box m2 at compare:6:9:
{c/0 -> (u:c,d:cd,a:d) 
Bar at compare:7:26:
{c/x -> (u:c,d:cd,a:d),
 cx/0 -> (u:c,d:cd,a:d) | (u:x,d:ax,a:a),
 x/c -> (u:x,d:ax,a:a) 
Net m at compare:7:9:
{c/x -> (u:c,d:cd,a:d),
 cx/0 -> (u:c,d:cd,a:d) | (u:x,d:ax,a:a),
 x/c -> (u:x,d:ax,a:a) 
Box n1 at compare:9:9:
{a/0 -> (u:a,d:ab,a:b) 
Box n2 at compare:10:9:
{e/0 -> (u:e,d:ex,a:x) 
Bar at compare:11:26:
{a/e -> (u:a,d:ab,a:b),
e/a -> (u:e,d:ex,a:x) 
Net n at compare:11:9:
{a/e -> (u:a,d:ab,a:b),
e/a -> (u:e,d:ex,a:x) 
Box p at compare:13:9:
{x/0 -> (u:x,d:xy,a:y) 
Dotdot at compare:15:15:
{ac/ex -> (u:ac,d:abcd,a:bd),
 ace/x -> (u:ac,d:abcd,a:bd) | (u:ce,d:odedx,a:edx),
 acex/0 -> (u:ac,d:abcd,a:bd) | (u:ce,d:odedx,a:edx) | (u:ex,d:aedx,a:edx) | (u:x,d:abx,a:b),
 ace/e -> (u:ce,d:codedx,a:edx),
 ce/ex -> (u:ce,d:codedx,a:edx) | (u:ex,d:acedx,a:edx) | (u:x,d:abx,a:b),
 ex/c -> (u:ex,d:acedx,a:edx) | (u:x,d:abx,a:b),
 x/ce -> (u:x,d:abx,a:b) 
Dotdot at compare:15:20:
{ce/ax -> (u:ce,d:cdexy,a:dy)}
```
It generates the following output for the program in Section 2.2.2:

Box mul at program:3:5:
\{pq/0 \rightarrow (u:pq,d:pq,a:pq)\}

Box dec at program:4:5:
\{q/0 \rightarrow (u:q,d:q,a:q) \mid (u:q,d:q,a:z#0)\}

Box div at program:5:5:
\{pq/0 \rightarrow (u:pq,d:pqr,a:r)\}

Filter at program:8:13:
\{n/0 \rightarrow (u:n,d:npq,a:pq) \mid (u:n,d:npqt,a:pqt)\}

Net split at program:7:5:
\{n/0 \rightarrow (u:n,d:npq,a:pq) \mid (u:n,d:npqt,a:pqt)\}

Filter at program:13:17:
\{t/0 \rightarrow (u:t,d:t,a:t)\}

Dotdot at program:13:28:
\{pqt/0 \rightarrow (u:pqt,d:pqt,a:pqt)\}

Net inner at program:12:9:
\{pqt/0 \rightarrow (u:pqt,d:pqt,a:pqt)\}

Star at program:15:19:
\{pqt/0 \rightarrow (u:pqt,d:pqt,a:pqt) \mid (u:t,d:t,a:t)\}

Net upper at program:10:5; topology:
\{pqt/0 \rightarrow (u:pqt,d:pqt,a:pqt) \mid (u:t,d:t,a:t)\}

signature:
\{pqt/0 \rightarrow (u:pqt,d:pqt,a:pq)\}

Filter at program:18:30:
\{z#0/0 \rightarrow (u:z#0,d:z#0,a:z#0)\}

Bar at program:18:28:
\{pq/0 \rightarrow (u:pq,d:pq,a:pq),
\quad z#0/0 \rightarrow (u:z#0,d:z#0,a:z#0)\}

Dotdot at program:18:19:
\{pq/0 \rightarrow (u:pq,d:pq,a:pq) \mid (u:q,d:q,a:z#0)\}

Star at program:18:48:
\{pq/0 \rightarrow (u:pq,d:pq,a:z#p) \mid (u:q,d:q,a:z#0),
\quad z#0/0 \rightarrow (u:z#0,d:z#0,a:z#0)\}

Filter at program:19:12:
\{z#p/0 \rightarrow (u:z#p,d:z#pq,a:q)\}

Dotdot at program:19:9:
\{pq/0 \rightarrow (u:pq,d:pq,a:q),
\quad z#p/0 \rightarrow (u:z#p,d:z#pq,a:q)\}

Net lower at program:17:5:
\{pq/0 \rightarrow (u:pq,d:pq,a:q),
\quad z#p/0 \rightarrow (u:z#p,d:z#pq,a:q)\}

Bar at program:22:19:
\{pq/t \rightarrow (u:pq,d:pq,a:q),
\quad pqt/0 \rightarrow (u:pqt,d:pqt,a:p),
\quad z#p/0 \rightarrow (u:z#p,d:z#pq,a:q)\}

Net compute at program:21:5:
\{pq/t \rightarrow (u:pq,d:pq,a:q),
\quad pqt/0 \rightarrow (u:pqt,d:pqt,a:p),
\quad z#p/0 \rightarrow (u:z#p,d:z#pq,a:q)\}
Sync at program:25:13:

\{ 0/p \rightarrow (u:0,d:0,a:0),
    p/0 \rightarrow (u:p,d:p,a:p) \mid (u:p,d:pq,a:pq) \} 

Star at program:25:23:

\{ 0/p \rightarrow ,
    p/q \rightarrow (u:p,d:pq,a:pq),
    pq/0 \rightarrow (u:pq,d:pq,a:pq) \} 

Net join at program:24:5:

\{ 0/p \rightarrow ,
    p/q \rightarrow (u:p,d:pq,a:pq),
    pq/0 \rightarrow (u:pq,d:pq,a:pq) \} 

Dotdot at program:27:15:

\{ n/t \rightarrow (u:nq,d:nqt,a:q) \mid (u:nq,d:nqta:p),
    nt/0 \rightarrow (u:nq,d:nqta:p) \mid (u:nt,d:nqta:p) \} 

Dotdot at program:27:26:

\{ n/t \rightarrow (u:nq,d:nqta:p),
    nt/0 \rightarrow (u:nq,d:nqta:p) \mid (u:nt,d:nqta:p) \} 

Dotdot at program:27:34:

\{ n/t \rightarrow (u:nq,d:nqta:r),
    nt/0 \rightarrow (u:nq,d:nqta:r) \mid (u:nt,d:nqta:r) \} 

Net program at program:1:1; topology:

\{ n/t \rightarrow (u:nq,d:nqta:r),
    nt/0 \rightarrow (u:nq,d:nqta:r) \mid (u:nt,d:nqta:r) \} 

signature:

\{ n/0 \rightarrow (u:nq,d:nq,a:r) \}
Bibliography


[34] Stefan Kahrs. About the completeness of type systems. In Maarten de Rijke, editor, Observational Equivalence and Logical Equivalence, page 17. ESSLLI, August 1996.


\( U \), unwanted set extractor, 112
\( X \), extended value to subdomain, 112

infinite execution, 84
initialiser box, 109

label, 56, 79
label locality, 72
label set, 79

main pattern, synchrocell, 34
map, sig, 106
mapping, network signature, 52

match, 106

synchrocell patterns, 34
terminating patterns, 31, 83

network, 29, 57, 88
non-stuck execution, 57, 82
normal execution, 82

operators
\( \parallel \), parallel composition, 57, 88
\( \cdot \), serial composition, 57, 88
\( \ast \), serial replication, 83
.a, added value part extractor, 65, 90
d, deleted value part extractor, 65, 90
.o, output choice set extractor, 65, 90
t, attraction rank extractor, 74, 98
.s, subdomain extractor, 65, 74, 90, 98
.u, used value part extractor, 65, 90
#
, value constructor, 79
( ), rep application, 66, 91
/, extended value constructor, 112
[ ], discarded field set notation, 106
[ ], label set ordered slicing, 115
+, extended value addition, 112
+, value addition, 56, 79
c, value capture, 112
-, value difference, 56, 79
\( \times \), value intersection, 56, 79
\( \leq \), value partial order, 56, 80
\( \mid \mid \), size of value, 79

output choice, rep, 64, 90
output variant, sig, 106

parallel composition, 30, 90
translation, 45
parallel replication, 49
modelling, 49

passer, 86
pattern, synchrocell, 34

rank (attraction), 73
record, 26
record type, 26, 43
translation, 43
rep, 64, 90
replica, 31, 83
representation, 63, 90
root, 80
root alphabet, 86, 98
routing information, 107, 120

self-conflicting, extended value, 112

semantic rules
biBox, 99
biParSun, 99
biPar, 99
biSerSun1, 99
biSerSun2, 99
biSer, 99
biSink, 99
biStarSun, 100
biStar, 100
biStuck, 100
bsBox, 80
bsParSun, 88
bsPar, 88
bsSerSun1, 88