Static Guarantees for Coordinated Components: a Statically Typed Composition Model for Stream-Processing Networks

Frank Penczek

March 2012

A thesis submitted to the
University of Hertfordshire
in partial fulfilment of the requirements of
the degree of

Doctor of Philosophy

in Computer Science

Abstract

Does your program do what it is supposed to be doing? Without running the program providing an answer to this question is much harder if the language does not support static type checking. Of course, even if compile-time checks are in place only certain errors will be detected: compilers can only second-guess the programmer's intention. But, type based techniques go a long way in assisting programmers to detect errors in their computations earlier on. The question if a program behaves correctly is even harder to answer if the program consists of several parts that execute concurrently and need to communicate with each other. Compilers of standard programming languages are typically unable to infer information about how the parts of a concurrent program interact with each other, especially where explicit threading or message passing techniques are used. Hence, correctness guarantees are often conspicuously absent.

Concurrency management in an application is a complex problem. However, it is largely orthogonal to the actual computational functionality that a program realises. Because of this orthogonality, the problem can be considered in isolation. The largest possible separation between concurrency and functionality is achieved if a dedicated language is used for concurrency management, i.e. an additional program manages the concurrent execution and interaction of the computational tasks of the original program. Such an approach does not only help programmers to focus on the core functionality and on the exploitation of concurrency independently, it also allows for a specialised analysis mechanism geared towards concurrency-related properties.

This dissertation shows how an approach that completely decouples coordination from computation is a very supportive substrate for inferring static guarantees of the correctness of concurrent programs. Programs are described as streaming networks connecting independent components that implement the computations of the program, where the network describes the dependencies and interactions between components. A coordination program only requires an abstract notion of computation inside the components and may therefore be used as a generic and reusable design pattern for coordination. A type-based inference and checking mechanism analyses such streaming networks and provides comprehensive guarantees of the consistency and behaviour of coordination programs.

Concrete implementations of components are deliberately left out of the scope of coordination programs: Components may be implemented in an external language, for example C, to provide the desired computational functionality. Based on this separation, a concise semantic framework allows for step-wise interpretation of coordination programs without requiring concrete implementations of their components. The framework also provides clear guidance for the implementation of the language. One such implementation is presented and hands-on examples demonstrate how the language is used in practice.

Contents

1 Introduction

Thesis. *It is possible to provide static correctness guarantees for parallel systems solely on the basis of the coordination aspects, not on that of complete applications.*

Finding ways to solve problems using the arithmetic and logical capabilities of a computer requires an intuition for the machinery as much as an intuition for the problem at hand. It equally importantly requires a degree of creativity to bridge the gap between the two worlds in the process of composing a concrete solution for an abstract problem. Besides creativity and intuition, programming computer systems does also require a systematic and rational approach to problem solving and a profound understanding of algorithmic methods. Reasoning about and proving correctness of programs is not possible without employing a variety of formal methods, introducing abstractions and applying strategies for decomposition and synthesis.

The above characterisation of programming is paraphrasing ideas of Donald E. Knuth's Turing Award Lecture [Knu74], in which he argued that programming often shares more with art than with science. Knuth found, although accepting it might not be completely achievable, that turning programming into a science is a goal worth pursuing; it would lead us to provably correct software and ultimately to fully automatic program derivation. It would help us to advance and deepen our understanding of computer programming.

In the same year as Knuth's lecture, Edsger W. Dijkstra noted in a similar argument that a *separation of concerns* [Dij82] is an essential requirement in thinking about programs and programming. The various aspects of programming, as for example correctness, efficiency or even desirability, are all equally important and require careful attention. However, each of the aspects should be considered in isolation and never at the same time. Approaching these aspects simultaneously is bound to fail for they are to complex; each of them requires a different mindset and a different set of tools to be reasoned about.

Both of the arguments have been made in 1974. Arguably, Knuth's ideas are still unattained today and programming still, at least at times, requires a good amount of artistry. Few will claim that programming is a purely scientific endeavour; in fact, some argue that programming is not even, and most likely never will be, an engineering discipline [Dav11]. What programming is, or may become, is ultimately a secondary matter if a program is doing what it is intended to do. In achieving that, Dijkstra's suggestion of separating concerns has proved successful. Specialised and specifically targeted techniques and tools have been developed that address the various aspects of programming: software requirement specifications to agree on intended behaviour, compilers that use type systems to approve correct code and profilers that help in performance analysis are only a few examples of what we have at our disposal when working with software. However, developing such techniques and tools is a continuous process. Advances in computing technology often require research and development of new programming techniques and tools in order to be useful.

Indeed, research into programming and its related areas has dramatically advanced the way we solve problems using computers. Today, we have a wealth of paradigms and languages to choose from depending on the problem domain (and personal preference); imperative programming in C [Int99] or Fortran [Int10], declarative programming in ML [MTM97] or Prolog [Int00] and object oriented programming in Java [GJSB05] or C# [Int06] are only a few examples of today's diverse field of programming languages. Formal techniques for reasoning about program behaviour and correctness, as for example structural operational semantics [Plo81] and type theory, have become standard tools and are commonly employed today. Type theory in particular has developed from a purely mathematical discipline [Rus08, Chu40] to an integral part of many programming languages. Algorithms that automatically check programs for consistent use of value and variable types [DM82] are indispensable tools for error prevention during program development. Systems that take this idea even further, up to value dependent types and program derivation [CDMM10, MKJ08], or techniques that embed programmable consistency checks into a language [Her10] are only a few examples of recent research efforts that take us closer to automated program verification, provable correctness and maximal efficiency — which is also a step towards Knuth's formulated goals.

Yet, theory and software are not necessarily the main defining factors in determining the way we solve problems using computers. The actual hardware, that is the computer architecture, has an impact on how we formulate problems that we seek to solve as well; the influence, however, is more indirect mainly for two reasons.

Firstly, most programming languages allow us to express computations on a high level of abstraction. The details of the executing hardware are kept well out of sight so that we do not have to deal with the technical details of the underlying machinery. We rely on a compiler to transform a high-level description of the computations that we require into a machine-dependent, executable program. The compiler, however, needs to be aware of the specifics of the targeted machine, as for example the machine's memory hierarchy and how to best exploit it, in order to make efficient use of available computing resources [UM03]. The larger the distance between the high-level description of a program and the targeted hardware is, the more sophisticated transformations have to be employed by a compiler in order to produce efficient code [HPP09, ACW$^+$09]. It is often the case that compilers stand a better chance of bridging the gap if we are aware of architecture specifics and write programs accordingly [BGS94].

The second reason is more fundamental. The way we think of computers and how to program them is almost exclusively based on the von Neumann model [BGvN89, Knu70]. The model comprises a central control and processing unit, a single memory for instructions and data, input and output units and communication busses for interconnecting these components. Leaving the shortcomings of the model aside [Bac78], as for these remedies and alternatives are in place [Dre07, EWC94, Dev11], it is not the technical details that had such a lasting impact on program design; it is the inherently sequential nature of the model.

Until recently the sequential model matched very well the systems available for general purpose computing. Typical systems contained one single processor, and unsurprisingly, programming these computers was dominated by sequential algorithms. Advances in processor and processing technology have increased the number of instructions that are processed per time unit steadily over a long period of time [HP06a]. Program performance, especially for compute bound algorithms, automatically benefited from these increased process rates. However, the pace has slowed down at which the rate of instructions per time unit increases, primarily due to energy constraints and the way processors exploit parallelism internally [BC11]. With single-processor performance hitting a limit, an increase in performance is achieved by placing multiple processors (cores) on a single chip [OH05]. The processing rate of the individual cores only slowly increases over time and as a consequence, existing programs do not automatically benefit from new processor generations as much as they used to do. In order to utilise multiple computing resources from within a single program, special techniques have to be employed. In many cases this involves a re-design of the algorithms that make up a program. Solving a problem in parallel often requires quite different approaches than purely sequential algorithms[Hel78, CP98, Xia10]. Once the algorithm is suitable for concurrent execution, enabling the program to use multiple computing resources additionally requires programmers to add support for this; for most languages this is a manual process that varies widely with the concrete approach that is chosen [Che91, KMZS08, ABD$^+$09, RR10].

Manually adding support for multiple computing resources to a program, i.e. writing a concurrent program, can be a quite challenging task. It typically involves decomposing the program into a collection of tasks, identifying dependencies between the tasks, and then executing the tasks in an order that satisfies these dependencies. The granularity of task decomposition often varies to a great extent; single iterations of a loop may qualify as a separate task just as much as entire sub-systems of a complex application, say, the programs hosted by an operating system. Depending on the chosen approach for implementing concurrent execution, several different ways of tasks interactions are possible, for example, to ensure that tasks wait for each other in case there are dependencies between them or to generally communicate results between each other.

The choice of a concrete programming model for writing concurrent software is influenced by the trade-offs that we are willing to make. The more versatile a model is, i.e. the broader the range of covered granularity is, the more low-level the programming constructs become and the least support of programming tools we can call upon. On the opposite end of the scale, compiler assisted approaches with high-level programming constructs are far less taxing on programmers and give an assurance on the correctness of the program, but they come at the price of narrower applicability.

Using PTHREADS in C is a typical example of a very versatile but low-level approach.

As a programmer we are able to work directly with threads from within the application code, and we are free to decompose and distribute work as we see fit depending on the granularity. But, the responsibility for putting threads to work without causing race conditions, synchronisation issues and deadlocks is also our concern. Hardly anyone will deny that working with code that uses manual thread management is challenging at the best of times, and plainly frustrating at others; most of such codes so far defy attempts of comprehensive static, i.e. compile time, analysis. The accepted way of working is typically to analyse problems as they appear at runtime, using a debugger. Parallel debugging techniques and debuggers are an area of active research [KKKV00, MQB$^+$08, GHKW08, NMT10, DAK$^+$11] and are indispensable tools in the development cycle, but they are curing the symptoms rather than fixing the cause. As the code that is concerned with thread and concurrency management is entwined with the rest of the application, a later separation for dedicated analysis is not possible. One of the causes for the problems is obvious: the principle of the separation of concerns is not followed.

Working with threads as an abstraction from parallel hardware is challenging for compilers and programmers alike, and the problems are well acknowledged [Lee06]. Higher-level approaches, as for example OpenMP [Boa11] or Cilk [FLR98], allow for implicit thread management through annotations. These avert certain programming errors in thread handling and synchronisation, but still require close attention to potential race conditions [LHHW10, BYR$^+$11]. Other approaches abstract even further from the threading model and provide concurrent execution through language constructs and data structures [KV07, CLJ$^+$07, GS06, LP10, GVL10]. Programming with such constructs is convenient as it guarantees the absence of concurrency-related bugs, but, it ties programmers in to using a particular language and design pattern.

A programming paradigm that by design cuts out an entire range of concurrency-related errors is that of message passing based approaches. When using message passing, concurrent tasks cannot access the same data at the same time. Instead, any communication between tasks is established by sending and receiving messages. These approaches are free of the problems that arise from threads working on data in a shared location, but they introduce new challenges as well. The most obvious one is caused by not being able to work on shared data directly: data structures need to be decomposed and tasks need to communicate parts that other tasks depend on explicitly. This complicates data structure design and it also requires developing a communication scheme between tasks in order to make sure that the required parts of a structure are sent and received to and from the correct communication partner. As with thread-based approaches, message passing approaches vary in their level of abstraction and the way correctness of an implementation can be analysed.

For message passing style programming, using MPI [For09] is likely the most adopted approach. MPI provides functions for sending and receiving messages in various ways, and these functions are provided through an API for embedding them natively into application code. Because of the temporal and spatial couplings of tasks that is only fully determined at runtime, opportunities for static analysis of MPI programs are very limited and correctness analysis is therefore inevitably postponed until runtime [KHH$^+$08, Vo11]. A more abstract approach to message passing is taken by data-flow and flow-based programming models [Ack82, JHM04, Mor10]. Approaches of this kind draw upon key ideas from component and coordination models. Component models advocate the specification of computations as independent and self-contained units [McI68], and coordination models provide means to specify the interactions between computations on an abstract level without prescribing to a particular communication scheme [CG89, GC92].

Data-flow based approaches shift the focus from a control-flow centric view to a data centric view in which the communication scheme of a program is implicitly defined by the data dependencies between tasks. A task may execute as soon as its data dependencies are met, and the results produced by a task may then in turn satisfy the dependencies of other tasks. The specifics of how data dependencies are stated range from requesting and placing data to and from a shared space to description languages for complex dependency graphs. An important property of all these approaches is that the implementation of a task is always spatially and temporally decoupled from all other tasks. Tasks become self-contained, independent components. In the context of an application tasks are of course coupled to other tasks; however, the relationship between tasks is always captured externally. This property is important as it leads to the separation of two concerns: the implementation of tasks and the coordination of these tasks.

The general applicability and analysability of the various approaches differ. Approaches that mainly incorporate ideas from component models usually allow components to be implemented in a variety of languages. Interoperability between components is transparently taken care of by mechanisms in the middle-ware that hosts the components. These models provide static checks to ensure compatibility between the interfaces of components that are connected to each other, but they typically do not include behavioural analysis [LW07]. Approaches that put more emphasis on coordination aspects tend to use a single language for the implementation. Linda-inspired approaches provide their API to a host language to deposit and retrieve data from within implementations. Analysis on these approaches is typically based on a mixture of compile time and runtime techniques [FP98, FP99, HS09, YAM$^+$11]. Another category of coordination approaches describes interactions between computational components outside a component's implementation. Such approaches do so by specifying the coordination model in a separate language. For analysing the resulting coordination program, a wide range of techniques from automata theory, modal and temporal logic and process calculi can be employed [BBC10, CPLA11, TSR11, DGS11]; most of these approaches, however, are mainly of theoretical interest and are not fully integrated into programming systems.

The ideas behind component and coordination models prove to be very profitable in supporting our thesis as these models inherently support a separation of concerns and balance programmability, generality and efficiency [WHW$^+$11, MPR06, BCS09b, BCS$^+$09a, dCJL10]. Existing approaches, however, do either not fully abstract from the computational domain and then lack mechanisms for statically checking the coordination model for correctness, or they limit the expressiveness of coordination models to static or semi-static wiring of components.

But, expressiveness and static correctness properties are not mutually exclusive and can be brought together: a complete separation between computation and coordination on the one hand and, on the other, statically checkable yet expressive coordination models with properties that are relevant to programmers of concurrent software is possible.

This dissertation presents an approach that achieves this by using distinct languages for the two tasks of implementing computations and specifying coordination. Computatians are implemented in a conventional programming language such that all analysis tasks are delegated to the tool-chain of the chosen implementation language. The coordination of these computations is expressed in a separate, dedicated language that abstracts completely from actual computations. It is this separation that allows us to comprehensively analyse coordination aspects at compile time, i.e. to statically infer properties about the consistency and the behaviour of a concurrent program.

Centered around this coordination idea, this dissertation deliberates answers to these central questions:

- What are notions of correctness and safety within a coordination program and what are relevant properties for establishing these notions?
- Which mechanisms are needed to guarantee that programs exhibit required properties that establish the above notions?
- How may such a language be implemented and used in practice?

We develop answers to these questions, in the context of S-NET [CPG+10], formally by devising a semantic framework including an inference based type system and by designing an implementation strategy. The third, more pratical question is answered by presenting use-cases including performance evaluation, based on a complete tool-chain for S-NET which has been implemented as well.

An S-NET coordination program approaches concurrency management on a high level of abstraction and it parts with the control-flow centric view that is taken by thread and process based approaches such as PTHREADS and MPI. Instead, the language is built upon data-flow principles: an application is a collection of computational components, also called boxes, that are put in relation to each other only by their data-dependencies. An S-NET coordination program encapsulates computations into opaque components that provide computations without revealing their actual implementations. Only an abstract type signature suffices to provide all required knowledge about a computation. The design of the language does not rely on a specific execution model, but is built on the assumption that all boxes within a coordination program may execute asynchronously as soon as input becomes available. The only requirement on a box implementation is that it does not maintain an internal state: State in an S-NET program may only be expressed using a dedicated language construct to make it fully explicit.

Taking this brief characterisation of S-NET into account we may now concretise the notions of correctness and safety that we are interested in:

Interface Consistency: A box is never presented with an input that it cannot process.

Network Consistency: At no point does a component produce an output for which no further processing is possible,i.e. messages never "get stuck".

Program Consistency: The semantics of the language fully defines the behaviour of a coordination program. A given input and a program definition uniquely define the set of outputs, even where non-deterministic execution order of components is involved.

Liveness: Programs do not deadlock.

This list provides an answer to the first part of the first question that we have raised above and it concretises the overall aim of this work. If we are able to statically prove for a given program that all properties hold, then we will call such a program a *safe* program. With safety being a static property, i.e. a property that is checked at compile time, a safe program will not encounter any runtime errors or deadlocks that stem from the S-NET network specification. Not only does this reassure a programmer that the coordination aspects, i.e. the implemented decomposition and communication scheme, of a parallel program are correct, a static checking mechanism for safety also enables the implementation of provably correct transformations to optimise coordination programs.

The answer to the second question that is, how to check a program for safety, is the major contribution of this dissertation. As several chapters are dedicated to answering it, we limit ourselves here to a brief outline of the involved steps only. In a first step we define a simple, rule based semantics that enable formal interpretation of S-NET programs. The second step is the development of an inference process on top of a constraint based type system. The type inference serves a dual purpose: Used in conjunction with S-NET programs in which types have been annotated by the programmer, the inference process uncovers inconsistencies between the program definition and the programmer's annotations. However, a programmer may also choose to omit type annotations altogether in which case the inference process determines all types automatically from only box signatures. As we will use a traditional type system as the basis for inference, we will be able to show properties such as preservation, progress and generality; properties that are typically expected to be found in computational languages. The third and final part of the answer is the development of a typed version of the semantics. It equips us with the required formal machinery to show properties of the inference process and to abstractly interpret implicitly typed S-NET programs.

For answering the third question as to how to implement and apply the overall approach in practice, we design an implementation strategy for a parallel runtime system. Our two main objectives are: Firstly, the conceptual design of the runtime system should be portable, i.e. the concept should be applicable to a wide range of hardware platforms. Secondly, the runtime system has to be able to utilise as many parallel computing resources as possible to exploit the exposed concurrency of an S-NET application. Achieving these goals is helped by the fact that the level of abstraction of an S-Net program does not imply a concrete implementation strategy or machine model. Although this requires a runtime system implementation to bridge a large gap from the program to the hardware, it at the same time keeps the model portable as it gives us a great degree of freedom for implementations. We only make one central assumption regarding targeted systems, which is the availability of a threading and/or process API. On top of that API we define a collection of independent components that provide specific functionality to map the inherent data flow graph of an S-Net program to computational resources. Individual components for executing box tasks, implementing network combinators and split and merge points of the graph together provide all services that are required to execute S-Net applications. The final part to the answer of our third questions will be a case-study that makes use of a concrete implementation of the runtime system on top of Pthreads and MPI for executing programs on shared-memory machines and clusters of workstations.

We can now close the circle to the discussion in the beginning of this dissertation and summarise: Identifying concurrency management as a separate concern proves to be very useful. If dealt with in isolation we can employ a dedicated set of techniques to reason about concurrency management, which ultimately takes us closer to correct and consistent concurrent software. In achieving this, coordination and component models provide the required, clear dividing line between computational aspects of an application and the concern of concurrency management. The central question is: are properties of correctness and consistency as found in general purpose programming languages also applicable if we exclusively focus on concurrency and its coordination? This dissertation answers the question in the affirmative. For this we identify relevant properties in the context of coordination and interacting components, and we are able to statically infer these properties to guarantee that a coordination program only defines consistent interactions and behaves according to its specification. We complement these more theoretical contemplations by an implementation strategy and a discussion on practical experiences with S-Net by means of concrete use-cases.

1.1 Outline of this Dissertation

The coordination approach that we investigate in this dissertation is based on data-flow principles and models communication as streaming networks. Chapter 2 offers a brief overview of the development of these fields and surveys related work.

Chapter 3 introduces the details of S-Net. The primitives and combinators are discussed. This chapter also outlines the role that types play in the system and provides an example to get to grips with the coordination approach.

Closely related to the introduction of the coordination language is Chapter 4 for it formally defines the semantics of the language by means of a rule based deduction system. The semantics presented in this chapter rely on externally provided type annotations, but they are complete in the sense that they fully specify the language for implementation.

Before we deal with the details of a mechanism for automatic type inference, Chapter 5 discusses the role of type systems in our context. This chapter introduces two foundational works that we will hugely benefit from as we will be able to carry over important properties of their unification and inference algorithms. Essential results of these works are reproduced in this chapter to make this dissertation self-contained.

The mechanism that automatically infers and checks types is developed in Chapter 6. The chapter presents the system of transformations that allows for traditional type inference to be applied to sequential parts of a program. The mechanism that analyses the data-flow graph and identifies safe sub-graphs based on the inference results of earlier transformations is developed. Also part of this mechanism is the computation of required runtime information.

A formal semantics for a variant of the coordination language that makes use and maintains the information inferred by the type inference mechanism is presented in Chapter 7. The chapter contains the relevant proofs to show consistency between the formal semantics and the type information that is provided by the inference mechanism; this establishes similar properties for the system that are traditionally known as preservation and progress.

Chapter 8 develops an implementation of the system using a high-level, functional programming approach. The chapter defines an interface between potential runtime system implementations and a compiler and provides a concrete implementation strategy for a stream-based runtime system.

Selected use cases of the language are presented in Chapter 9, including implementation and runtime evaluation.

Chapter 10 concludes this work with closing remarks and a discussion of potential directions for future developments.

2 Background and Related Work

2.1 Data-Flow as Programming Paradigm

The idea of structuring a program by its data flow rather than its control flow is as intriguingly simple as it was groundbreaking. Many approaches are based on this idea for describing parallel algorithms and concurrent systems. The remainder of this section gives an overview. The surveys [Ack82, WP94, Ste97, BCE$^+$03] are invaluable resources for developing a broad as well as deep impression of the data flow computing landscape.

2.1.1 Foundational Works

Finding alternatives to the standard way of expressing programs as mere sequences of instructions date back to at least 1966. Interestingly, the motivation back then was very similar to the motivation of many research projects today; The ideas presented in [KM66] are driven by the observation that expressing parallel computations and mapping these computations onto hardware are very challenging. Because of the fundamental differences between writing sequential and parallel code and the problems that these entail, i.e. finding appropriate algorithms, scheduling and maintaining sequencing, a new programming model was desired. The proposed approach represents each step of an algorithm as a node in a directed graph. The computation of each node in the graph depends on the inbound edges and is constrained by its outbound edges. If a program is expressed in this way, then several steps (nodes) of the program may be executed at the same time. The authors admit that their approach is limited by the assumption that edges are first-in-first-out queues (or *streams*, a term introduced by [Lan65a, Lan65b]), but that the result of a computation is invariant against unknown and variable execution times of nodes and storage requirements for data elements on edges.

In [Kos73] a set of goals for (operating system) programming are identified. The implementation of the system should be

- parallel but determinate;
- partitionable into small, independent modules;
- understandable in the large and in detail;
- extensible without affecting all parts;
- easy modifiability;

- efficient.

Additionally, it is desirable to be able to measure and predict the performance of the system, and this should even be the case without fully implementing it. In order to achieve these goals a language is proposed in which programs are expressed as directed graphs where the nodes are functions, and the edges are data paths between the functions. Communication between the nodes is synchronised via `presence` and `done` signals. The language has primitives for forks, conditionals and loops. A graphical representation of programs with symbols for the primitives of the language has also been devised. The language supports hierarchical structuring of applications by using sub-graphs as nodes on the next higher level. (The demand for strictly deterministic behaviour was revisited by the same author some years later. In [Kos78] a denotational semantics framework to accurately model non-deterministic behaviour of data flow programs is developed. This is motivated by the observation that in operating system programming the actual behaviour is often influenced by their physical environment and not just by the implementation of the system.)

In his now famous and often cited approach Kahn proposed a mathematical model and a mini programming language [Kah73, Kah74] to describe computations on a network of computing stations. Such stations are connected to each other via communication lines that transmit data in an unknown (but finite) amount of time. When a station is not computing it is waiting for data on one of its lines. Kahn was primarily interested in the provable properties of such systems, e.g. its input-output behaviour and deadlock-freeness (he suggested to think about the system as a set of Turing machines that communicate via one-way tapes with each other). Parallel programs are expressed as parallel program schemata [KM69, Rut64], which are directed graphs with computations being nodes and communication channels being edges between these. Alternatively, programs may also be expressed in a language that allows for the specification of processes that take input and output channels as arguments. The connections between processes are established over named channels that are given to the processes in the main body of a program.

As an approach to structured programming [Den74] presents a data flow language that a graph representation of programs. The computations, called actors, are the nodes of the graph and the edges of the graph transport tokens to and from actors in order to determine which computations are ready to be carried out. Actors fire, i.e. perform their action, when tokens are available on their inbound edges. Actors may have procedures associated with them that are applied to data when an actor fires. Many concurrent activations of procedures are allowed; in order to identify the tokens that belong to a specific invocation of a procedure coloured tokens are introduced. Data that is processed by such a data flow program is kept on a heap that is organised in a evolving graph structure. The tokens of the program graph initially contain pointers to root nodes of the heap graph. If an action of the program requires data it reaches it through the pointer that is associated with an inbound token. When new data is produced, a new node is added to the heap graph and the produced tokens keep pointers to the new node (as a consequence of this, all computations are side-effect free). The work also develops ideas for garbage collection, i.e. removal of inaccessible nodes of the heap graph. An insight of the work is that although most programs may be transformed into their data flow representation, data flow languages often do not satisfactorily capture data parallelism. To address this shortcoming an extension of the language that provides a "for all" construct is proposed as a desirable construct.

In [Bur75] it is argued that defining a program in terms of functions that operate on streams of data is a more natural way to structure a program than conventional programming techniques (i.e. structured programming with loops). The paper develops many of the functions that are taken for granted in (functional) programming languages that support lists today.

Another seminal paper appeared in 1978 [Hoa78]. The motivation of the presented idea is that programs are usually written as a sequence of instructions on a single processor. If more performance is required and multiple processors are to be used, then this is often tried to be hidden from the programmer. Hiding, so it is argued, is the wrong approach and concurrency and parallelism should be explicitly modelled. With "communicating sequential processes" managing concurrency is done by viewing the underlying executing machinery as a collection of sequential processors and a program as a collection of communicating processes on top of these. Commands are proposed that can arrange processes into compound processes in sequence and in parallel. Communication between processes are established through input and output commands that can either retrieve a value from another process and store it locally or place a computed value into a variable of a remote process.

2.1.2 General Purpose Languages

As one of the first data flow programming languages Lucid [AW77] was conceived as a means to express a program and its proof in one and the same language. The primary intention of the authors was not to develop a new language as such but to solve the challenge that one usually faces in proving program correctness. The problem when employing standard proof techniques on programming languages is that statements as $i = i + 1$ or `goto 5` are meaningless in mathematical reasoning. In Lucid, programs are expressed in terms of recursion equations. Variables are sequences of values; the first element of the sequence is the variable's current value which is followed by the history of values that the variable represented in the past. Programs are expressed in terms of operators that, for example, initialise variables, define how the next value is to be computed from the variable's (and others') past, introduce constants and other variables and express conditions.

The ideas of data flow computing, and more specifically the notion of streams, are also useful in other programming models as well as argued in [KL81]. Streams as a way to provide lazy lists (and computations on demand) and feedback loops in programs allow for a more concise specification of programs. In order to trigger computations on more elements on the stream as are immediately required, the concept of anticipation is introduced by means of a special `spar` construct. This operator demands all inputs at once. It is acknowledged that anticipation may lead to over-committing resources and therefore a solution for bounding the demand is also proposed. The problem is tackled by showing how virtual bounded buffers may be modelled with a feedback loop and tokens that are paired with data items that are to be processed. Only data items that are paired with a token create demand for computation.

Functional languages that follow the ideas of data flow languages are, for example, DFL [PBG84], SASL [Tur83] featuring an interactive development system [Ric84], and SISAL [MSA+83]. The former is a language whose syntax resembles that of Pascal [Wir71] and follows ideas of VAL [McG82]. The language features a `forall` construct to express data-parallelism. The latter also has a strong emphasise on data-parallel operations on arrays and takes a streaming approach to expressing such computations. For SISAL, there is also a distributed memory implementation available; the compiler automatically generates MPI code that allows for execution of programs on clusters [GBN+97].

At MIT considerable research has gone into developing languages for the data flow paradigm [NPA86, AN89, NA90]. A key insight of this work was that a purely functional approach to programming incurs performance penalties when it comes to computing with large data structures, especially arrays. I-structures [ANP89], were conceived to address this problem by allowing non-functional arrays of such structures. An I-structure is a write-once cell; read access to an unfilled cell suspends the reader until the write has occurred (repeated write access is illegal) in order to guarantee deterministic semantics. It was also at MIT that hardware platforms for natively supporting data flow computing were investigated [AC86, Pap90, Nik89, AN90]. Related work was also carried out at Manchester University [WG82]. More recently, data-flow optimised architectures have been under investigation again in order to address the challenges of high-end computing. For example, a processor-in-memory architecture based chip design [Ste06] has been chosen as one target for the ParalleX [GSS+07] execution model. ParalleX is a message driven model that provides a global name space across distributed memory machines. The model supports nested threads to exploit parallelism of different granularity and employs mechanisms for automatic latency-hiding.

Data flow programming has also been picked up in other areas of computing, as for example in parallel logic programming. Parlog [CG86, Cla88] computes the solutions to conditions in separate processes. Communication between the processes is establishes through the shared variables which form communication streams.

In database applications streaming has also been successfully employed [BBD+02]. Here, data sets are often too large to fit in memory for processing. Data is delivered on streams and queries are expressed as stream functions. Based on similar ideas is the Aurora/Borealis project [CcC+02, ACc+03, CBB+03].

2.1.3 Synchronous Languages

When programming reactive system one is often concerned with timing constraints. A system has to react to external stimuli within a certain amount of time in order to guarantee correct (which often equates to safe) behaviour. This requires a notion of time in the programming model and usually entails that producer and consumer rates of the components are a-priori known [LM87] to allow for static scheduling. The following section gives a short overview of developments in this area.

Esterel [BC85, BCG88] is an early example of a synchronous language. It is an imperative language but it employs streams to enforce strict order of computations. It is tailored towards the needs of real-time systems.

Lustre [CPHP87, HCRP91] is a synchronous data flow language. It was developed with the goal to provide a simple operational semantics to reason about program reliability in real-time system development. An interesting feature of Lustre is that it allows to call functions that are implemented in an external language, C.

Another language that aims at the same application domain is SIGNAL [GGB87]. This synchronous data flow language for real-time systems defines a calculus that allows for statically checking the correctness of the temporal behaviour of the programmed system.

The design of STREAM [Klo87] is guided by the needs of hardware designers. The developers of STREAM saw the need for a new language as other approaches are often ill-suited for developing and verifying hardware designs. This, so it is argued, is because languages are often conceived with a broader application area in mind and then extended and adapted to fit particular needs. STREAM has been developed solely for one purpose and thus features a lean design that fits its purpose. The language contains only a few constructs, sequential composition, parallel composition and feedback to compose stream processing functions. It allows to treat networks as stream processing functions again.

Also aiming at the hardware specification domain is the language ASTRAL [Ste95]. The language is accompanied by a rich algebraic framework for reasoning about programs written in this language.

Hume [MH00] is a functional language that follows a layered approach. Values and functions are defined in a fully-functional expression language, and interaction between functions is defined in a coordination language. The finite-state machine based coordination language connects any desired amount of inbound and outbound streams to a function to allow for interaction between the components (i.e. the functions) of a program. Hume allows for strict reasoning of space- and time bounds of programs [Ham06].

2.1.4 Streaming Languages

Streaming languages have a strong focus on exploiting data-parallelism. Computations are typically expressed as kernels that operate on one or more input streams and produce data on one or more output streams. A stream contains elements of the same kind. The kernel applies its operations to each element on the stream individually in an SPMD fashion; the operations of a kernel are not dependent on any information from the outside the specification of the kernel other than the data on the input stream.

Score [CDW01] is primarily aimed at media processing. The language allows for several components of an application to communication to each other via streams only. All components are assumed to be kernels that can be mapped onto parallel processing units.

In [TKA02] StreamIt is introduced, a stream based programming language accompanied by an optimising compiler and a runtime system. A distinct feature of StreamIt is the ability to send control messages to components to change their behaviour. These messages use a separate infrastructure from the data items and can therefore be communicated to components independently of the data flow graph of an application.

Streaming architectures developed at Stanford [DHE+03, KRD+03] aim to provide hardware support for stream computations. The goal of the projects are to provide architectures that match the requirements of numerically intense scientific applications better than general purpose hardware. In order to target streaming hardware from a conventional language Brook [Buc03] was developed which comprises extensions to C to support streams and kernel specification.

Wavescript [NCG+08] is primarily aimed at embedded devices and sensor network applications. The language's syntax resembles that of ML but offers imperative programming elements as well. Functions are defined on asynchronous streams but data elements on streams may be arranged into synchronous segments. The compiler of Wavescript analyses the programming and infers the underlying data flow graph. Optimisations are applied to the graph structure and include fusion, placement on computing resources and duplication of state-less operators.

Stream processing has attracted wide-spread industrial attention. Vendor specific implementations that target streaming hardware (such as GPGPUs) have been developed and are heavily marketed [NBGS08, MVM09, Sca08].

2.2 Components, Coordination and Coordinated Components

Component and coordination models date back a fair amount of time. Although they have emerged independently of each other the two approaches now overlap in many parts. Where component models primarily focus on encapsulation and the definition of generic interfaces between computational tasks, coordination models put more focus on providing mechanisms for computations to interact with each other. Increasingly, we see that these two approaches converge into what we may call a model of "coordinated components": Independent, computational components are brought together with a coordination model that defines how the components interact in a concrete setting.

Although not primarily developed for concurrency management, component models provide essential features for supporting programmers in designing concurrent and distributed systems. Component models encapsulate the code of independent tasks into self-contained components that only interact through specifically exposed interfaces. The motivation for the approach is the wish to be able to compose applications from re-usable and mass-produced components. Initially proposed in 1968 [McI68], the component model has been adopted in many areas of software development. The idea of the approach is to describe a complex software system as a collection of simpler components that are "plugged" together to form an application. A clear separation between the definition of components, i.e. computation, on the one hand and the composition of components, i.e. the "plugging", on the other hand is achieved by employing a layered approach as illustrated in Fig. 2.1. Components, i.e. the code

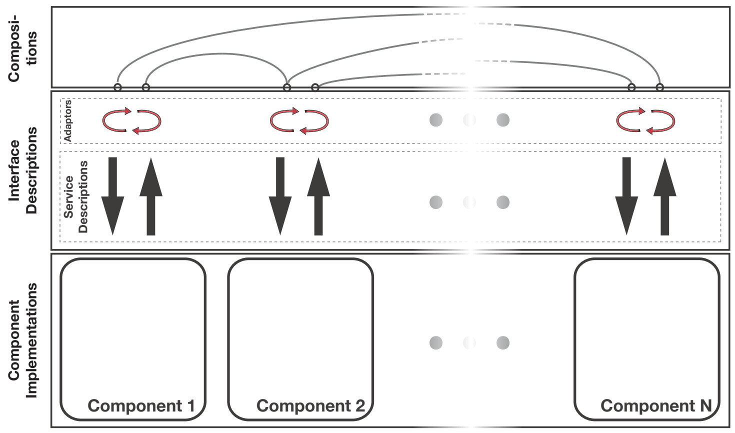


Figure 2.1: A component based model separates concerns into several layers.

that implements the computational aspects of an application, are located in the bottom layer (choosing the bottom layer for this is arbitrary, we could of course equally well reverse the order of layers). The components may be implemented in a range of computational programming languages, typical choices are Java and C++, but this is not an exclusive selection; additional integration of Fortran and Python as done in [AKM$^+$06] and Smalltalk and .Net as is available in [BCS09b] are only a few examples of the broad range of supported languages within component based frameworks.

In order to integrate themselves into a larger application, components provide interfaces to the outside world. The interfaces, also referred to as service descriptions, are exposed to the layer above the component layer in Fig. 2.1. Through this layer, a component declares required services that it needs to operate as well as provided services that the component makes available to others. Depending on the framework, several dedicated interface description languages[1], short IDLs, may be employed; the motivation for using a separate language on this layer is to encode the services that a component provides and requires in a format that is independent of the implementation languages of components; examples of IDLs are CORBA's IDL [Obj08], SIDL as used by [DEKL09] and SLICE as used by [Hen04]. The service descriptions in IDL are often automatically generated from the component implementation. Also, adaptors, i.e. marshalling code, that convert data representations to and from different computational languages are part of the interface layer. Marshalling code is rarely manually

[1]Another term is interface definition language

provided as it is usually part of a framework, e.g. such as Babel [DEKL09]. This layer also houses the middleware that, depending on the concrete model that is used, provides data communication mechanisms, access control to components, component repositories, runtime monitoring facilities and other services.

On top of the interface layer, the composition layer is located. This layer describes how the components are connected to each other to form an application. The composition is often not specified using an additional language but is implicitly defined by components accessing other components directly through the middleware. The composition of components, or "wiring", is checked for consistency to ensure that only compatible services are connected. All component models employ basic checks, e.g. they ensure that the data-types exposed and accessed through services coincide in order to guarantee interoperability. In a parallel or distributed computing environment, the composition layer may also serve as a means to associate components with computing resources. In this case the composition also describes where components are deployed, e.g. a specific node in a cluster, and how communication between components is carried out [MDB04, AKM+06, BCS09b].

Employing a component model in software development almost automatically leads to a separation of concerns, i.e. a separation of computational concerns from those dealing with the interaction between computations. Especially if used as a means to implement parallel software, a component based approach confines management code to the composition layer. The implementation of a component is kept free of such management code; in fact, components are usually completely unaware of their context and the surrounding framework. The strict separation enables component reuse but it also limits the extend of which analysis on the composition level is possible. Analysis is typically focused on interface compatibilities and ignores component behaviour. Dedicated composition languages with semantics that allow for reasoning about the interaction of components within a composition are available [LAN00, AN05], but are hardly used in practice [LW07]. As mentioned above, it is much more common that the composition is implicitly defined by accessing component interfaces from within the application directly.

Quite the opposite is the case in coordination models. As the name suggests, models of this kind strongly focus on coordinating the interactions between various computational processes. One of the earliest examples of a dedicated coordination model dates back to the beginning of the 1980s and the language Linda [Gel85]. Two key ideas motivate Linda: to provide *generic* means for coordination and to *orthogonalise* coordination from computation. Linda has pursued its goals of generality and orthogonality by introducing the notion of a shared tuple space for communication that is visible to all computational processes and by providing dedicated functions as the only means to place and retrieve values to and from the tuple space. The shared space may be transparently implemented on various architectures, and it is suitable for fine-grained and coarse-grained parallelism on shared-memory machines as well as on distributed and heterogeneous platforms. At the same time, due to the set of dedicated functions that may alter the shared tuple space, coordination code is clearly distinguishable from other aspects of an application although it is still written in the same language as the computational code [CG89, GC92].

The orthogonalisation of coordination and computation enforces separation of these concerns. As a consequence, programming breaks down into at least two separate tasks: the implementation of code that focuses on computational aspects on the one hand and code that deals with coordination on the other. Because of this separation, programming tools stand a chance to analyse the two aspects in isolation. Being able to verify the computational code is of course not the major gain here as semantic analyses, e.g. by employing a type system, are widely-used and established techniques. The ability to exclusively focus on coordination aspects, however, opens up new opportunities for analyses, as for example checking for consistency within used communication schemes and reasoning about the interactions between computational processes; for Linda, program development tools allow for inspection of the shared tuple space to trace the interactions between computational tasks. Such tools keep track of where and how access to the tuple space takes place in order to perform consistency checks and optimisations [ACG94]. The consistency checks are rather rudimentary as the tools are generally not able to infer full knowledge about how processes communicate with each other. Other approaches take the idea of orthogonalisation a step further by extending an existing computational language with dedicated syntax for coordination related code, as is for example done in Haskell$_{\#}$ [CLL02]. In approaches like this the computational code is complemented by a separate coordination program that holds the computational parts together. Within the coordination program, computations are only represented by an interface description, i.e. a declaration of required input parameters and provided outputs. Based on this abstract notion of computation, the coordination program defines how computations interact by connecting them to each other, e.g. by connecting provided outputs of a computation to required inputs of another computation via communication streams. Ensuring interoperability of computational processes is offloaded to the host (computational) language and its type system. The compiler for the coordination program may therefore solely focus on the analysis of the coordination model of which it has a holistic view; Haskell$_{\#}$, for example, employs a translation down to Petri Nets [Pet61, Pet77] for this.

Tight coupling of a coordination language and a computational language is a powerful approach as it allows to propagate established properties between the two domains. On the coordination side, type information of the computational code may be used to guarantee that only valid connections are established between processes. Similarly, the coordination model may provide information on how processes are composed to enable the compiler of computational code to perform further optimisations. These benefits, however, also come at a cost: the choice of the coordination language determines the computational language and vice versa.

Coordination languages that do not depend on a particular computation language tend to provide the most freedom to the user while still being able to apply comprehensive analyses to the coordination program. The coordination model becomes a fully generic "recipe" that describes how processes interact with one another but neither exposing nor relying on specific information from the computational domain. Purely theoretical frameworks usually follow this approach, as these aim to capture and analyse coordination properties on a high level of abstraction where the specifics of a concrete computational language are unnecessarily constricting [MK98, LSS08, LO09]. Similarly, systems for exogenous coordination, i.e. systems that connect software components without exposing an interface to the computational language, fall into this category [Ass97, AR03, Lum07, dCJL10]. By nature, none of these approaches give extensive guarantees about data integrity across computational components, for their strength lies in maintaining properties of the coordination model. Depending on the focus area, the list of properties includes guarantees on the number of messages within the system, e.g. the number of messages between components never exceeds or falls below certain thresholds, the detection of deadlocks, timing analysis, schedulability under given constraints, equivalence tests of models for optimisations, and various others.

An overview of recent developments and research issues in the area of component based software models provide, for example, [vdS06, LW07, RRMP08]; an overview over coordination models and related issues may be found in [OV11, PA98]. In the remainder of this section we will briefly summarise the history of component models and coordination approaches and survey a selection of projects in both areas.

2.2.1 Selected Component Model Approaches

Polylith [Pur94] is also a system to decouple the implementation of computational tasks from the organisation of the larger application. The individual components of a system are modelled as modules and using an interconnection language these modules are arranged into a graph structure that defines how the components are connected to each other. The modules may be implemented in various languages and communicate with each other over a common software-bus that resolves data representation issues.

Delivering high performance is the main objective of the Common Component Architecture CCA [AKM^{+}06, AAB^{+}06]. It is primarily aimed at computational scientists with little or no experience in writing concurrent software, and allows to compose an application from components that are transparently distributed across a cluster of hosts. CCA employs Babel [EKK01, DEKL09] to provide interoperability between components that may be written in different languages. Supported languages include C++, Fortran77, Fortran90, Python and Java. The Ccaffeine Framework [AA05] allows for visually supported wiring of components. The consistency checks for the application include interface compatibility but do not include behavioural aspects. From the specified wiring, glue code including required marshalling code is statically generated, resulting in low overhead for the composed application. Reported performance of CCA applications lie within two percent compared to a native implementation that makes no use of the framework.

The Fractal component model [BCS02, BCS09b] couples components with a controller that mediates interactions, i.e. a component communicates with other components through its controller. The controller has an internal interface that is used by the components it controls and an external interface for interacting with other controllers. Controllers may govern their components in different ways and implement various modes of communication. Additionally, controllers also provide means to inspect and intercept in- and outgoing messages. Another notable fact of the Fractal model is the use of an extensible Architecture Description Language [LOQS07]. The purpose of this language is to describe the target architecture and to define the required compilation process down to the architecture. As the language is extensible (it is based on XML), completely user-defined toolchains spanning many stages of compilation may be defined in order to bring a Fractal application down to the hardware.

2.2.2 Selected Coordination Approaches

Although Linda is typically cited as the earliest coordination model Gamma [BM86, BCM88, BLM93] was proposed at around the same time. Gamma's approach to coordination is also based on a shared value space for processes to interact. However, in Gamma there is no explicit access to a shared space from within conventional programs. Instead, programming in Gamma resembles describing chemical reactions based on multi-sets: the elements of the set are worked on by several reaction conditions independently which allows for modelling complex concurrent systems. Although on first glance unusual, the concept is still in use today [BFR08].

The Linda approach has achieved wider adoption, most likely because it does not impose a completely new programming paradigm but integrates into conventional languages. A coordination system that is directly inspired by Linda is the Lime model and middleware [MPR06, PMR99]. It is a coordination framework for use in volatile resource environments. It adopts Linda's idea of a shared tuple space and provides primitives to place and retrieve data elements to and from tuple spaces. Tuple spaces may be transparently distributed across several hosts. Lime supports dynamically growing and shrinking resources, i.e. new hosts may join and hosts may leave the resource pool at any time. The model has a fully specified formal semantics expressed in UNITY [MR98], a temporal logics and state-based semantics framework. UNITY provides proof-logic that allows for reasoning about safety and liveness properties in a system that allows components to dynamically disconnect.

Also following up on Linda's design is NetWorkSpaces [BCS+09a], in which processes interact by placing and retrieving values from shared work spaces. The work spaces are named and centrally managed by a server. All processes that "know" the name of a particular work space may interact with all other processes that share this knowledge. NetWorkSpaces is intended for scripting and quick prototyping of applications that require distributed processing. This is reflected in the set of supported languages which includes MATLAB, Perl, Python and R.

Swift [WHW+11] also follows a scripting approach, but provides different means to communication through shared, single assignment variables (as is done in earlier works of some of the authors [FOT92, Fos99]), but A swift script defines which tasks are applied to data elements. Data elements are either Swift primitive types, such as integers and strings, or data elements are external types, referring to data residing in external files. Processing data of an external type involves invocation of external applications that are made part of a Swift program. This is achieved by defining special tasks that encode the command of the external application, which parameters are required and what the application returns as result. The order in which tasks are applied to data elements is not explicitly specified but implicitly defined by the data flow graph that is inferred from the use of the shared, single-assignment variables. The type system of Swift only covers basic type checks, but the runtime system allows for re-starting an application from the point of failure should execution abort.

An early approach to coordinated components is given in [SPSP87], a specification language called MODEL. A programmer designs a system by breaking it up into several modules and specifies interfaces for each of them to express dependencies. In a second step a data flow network is designed that joins up the individual parts. A compiler carries out consistency checks for the interfaces, infers timing behaviour and attempts to maximise concurrency. A similar approach is followed in CODE2.0 [NB92] that provides a graphical programming environment for developing parallel software (n.b. the year of publication is 1992). The system aims to provide an intuitive way to construct portable, parallel applications from reusable code blocks. A program in CODE is specified as a graph. Nodes of the graph either represent user-defined code blocks, decisions for conditional execution or are sub-graphs of these nodes. Edges in the graph determine data-dependencies; at runtime, a code block can only execute if all its inbound edges contain data elements. Produced results are placed on the nodes outgoing edge. CODE2.0 supports dynamically evolving graphs by instantiated nodes of the statically defined graph multiple times at runtime. Consistency checks of programs are based on graph-theoretical analysis and enforce that all input and output dependencies of nodes are met, and the analysis also determines which nodes may be executed independently [WBS+92]. Quite interestingly, the CODE programming system is used as basis for the more recent project P-COM2 [MDB04]. P-COM2 is an interface description and composition language that allows for the specification of distributed programs from individual components. The compiler generated a data-flow graph from the application description that is then used as input for the CODE tool-chain to generate an executable program.

In Darwin [MKD93, MDK93, RE96] components, which may be implemented in an external language, only expose their interface on the coordination level. The interface specifies which services a components requires in order to operate and which services it offers. Composite components are constructed by instantiating other components and binding them to the provided or requested services. A binding between components establishes a communication link in the form of a queue. Darwin allows for dynamically evolving component networks and dynamic instantiation of components.

Manifold [AHS93] allows for components to interact with each other over streams that are connected to input and output ports of components. The execution of a Manifold application is event driven; components may raise events that trigger a certain configuration to become active. A configuration determines which components are executed and how they are connected. The configuration remains active until a new event is raised. Reo [Arb04] employs similar ideas and extends the concept to components and compounds of components to allow for hierarchical structuring. Connectors are used to join up several components and establish the means of communication between them. An atomic connector, the channel, is provided as basis; from there more complex connectors may be constructed by forming networks of channels. These networks are graph structures where the nodes are start and end points of channels. Nodes automatically replicate all data to their outbound edges. Channels are active entities in Reo and may expose different properties as for example buffer capacity, lossy communication or allowing only synchronous read and writes. Connectors may be reconfigured at runtime, and channels can be dynamically created and discarded. Reo comes with a graphical development environment.

In even broader approach to coordination is taken by Ptolemy [EJL$^+$03] which provides a framework to model heterogeneous systems comprising different models of computation. The goal of Ptolemy is to provide system designers with a tool that allows for modelling of complex, distributed systems for which the behaviour is still predictable. Computations are encapsulated into actors. Actors provide input and output ports for communication between them. Systems are hierarchically organised and a composition of actors may be used as a one-entity composite actor on the next level. On each level the implementation of a self-contained system may be of a different domain, e.g. synchronous data flow, continuous time or discrete events. Each domain provides domain-specific receivers that are attached to ports of actors to implement the mode of communication that is required for the domain. This two-level approach to communication, i.e. actors providing ports and domain-specific receivers implementing the communication, allows for component re-use across domain boundaries.

2.3 A Brief Overview of Parallel Programming

This section provides a brief overview of techniques in hardware and software to exploit parallelism. We will investigate typical pitfalls of programming parallel hardware using explicit, manual thread management and process communication.

2.3.1 Making the Most of Parallel Hardware

Several techniques exist that exploit parallel computing resources.

Instruction-level based approaches attempt to maintain several active instructions per clock cycle. Common techniques include dynamic scheduling, speculative execution and issuing multiple instructions at the same time [RF93]. The vast majority of today's general purpose processors are pipelined super-scalar or VLIW processors that exploit instruction-level parallelism [HP06b]. Exploitation of parallelism on this level is transparent to programmers as it is typically handled by the hardware internally. There are software-assisted approaches as well where compilers employ strategies to transform high-level language constructs into code that makes use of the processor's capabilities automatically [SCD$^+$97, EAH97, ALSU07].

Thread-level parallelism, in order to be exploited within a program, requires a program to expose its internal structure in terms of schedulable code blocks. An increase in performance is achieved if the code blocks are executed in parallel. Truly parallel execution, however, requires multiple computing resources to be available in the executing machine and, equally importantly, does it requires software that makes use of them. This can be achieved in many different ways, and depending on the programming model and its details the notion of a code block is named differently. The list of names includes threads [IG99], processes [Lim95], tasks [Int07], kernels [SGS10], actors [KSA09], vats [Mil06], chares [KRSG94], and many others. For now, although all of these terms may conjure certain preconception of the programming and execution model in our minds, we are not concerned with the specifics of any of them in particular. Ultimately, these are all ways to exploit thread-level parallelism. Hence, for simplicity and consistency we shall pick one and do so by using the term *thread* hereinafter.

As the long list of terms suggests, software that is capable of using multiple processors from within a single program may be enabled to do so by using a wealth of techniques. We should note here that the concept of executing threads in parallel is not a new one. Computers that are capable of parallel processing date back to the late 1950s [HB84, HJ88] and parallel processing machines have been employed in high-performance and super computing for a long time [Cer98]. However, as systems in use in these areas are often custom-built for a particular purpose, the cost of such machines, i.e. the cost of the hardware on the one hand and the cost for developing specially targeted software on the other, has very much confined parallel programming to this specialist's market. Many of the programming systems (a programming language and its tool-chain plus external libraries and tools) in use in the area of parallel programming reflect their origin as specialists' tools. They require programmers to take certain hardware properties into consideration as not all systems are applicable for all scenarios. The most fundamental property is how memory is exposed by the targeted machine as this divides up almost all parallel programming approaches into either of two categories. In shared memory machines all threads of a program are able to see one global memory. Values that are stored in memory by one thread are visible to other threads and the values may be read and written by other threads as well. In distributed memory machines the memory is thread-local; values that are stored in memory are only visible to the thread that created the values. If another thread requires access to another thread's values, then an explicit communication transferring the values between threads has to take place. Programming using either of the two approaches comes with a set of particular, as well as common, challenges.

Programming in a shared memory setting requires attention to potential consistency hazards that arise from threads accessing the same memory. Such hazards cause a program to expose seemingly random behaviour and intermittent faults. The underlying cause is the order in which threads access shared values, i.e. the result of the program is dependent on the scheduling of threads. In a distributed memory setting no concurrent access to values in memory occurs as all data is thread-local. Owing to this, shared data structures are broken down into several parts such that the structure only conceptually exists as a whole. In practice each thread keeps a certain portion of the data and communicates with other threads if a computations depends on non-local parts of the data structure. In distributed memory programming this is where potential problems arise. Designing and implementing distributed data structures and devising a communication scheme accordingly is a non-trivial and error-prone process.

Most erroneous program behaviour stems from only a few typical programming mistakes. Although typical, the causing code is often well-disguised and hard to find; providing sophisticated debugging support is an area of active research [KKKV00, MQB$^+$08, GHKW08, NMT10, DAK$^+$11]. Still, to develop a feel for potential pitfalls in parallel programming we will look at a small example. Using deliberately naive implementation strategies we can produce common problems that occur in practice in far more complex and intricate settings.

2.3.2 Facing the Challenges of Parallel Programming

The more parallel resources we have at our disposal and the more diverse these resources are, the more algorithm-unspecific coordination code we need to supply. As the amount of coordination code that needs to be written grows, the tougher the task of verifying correctness of such code and the application as a whole becomes. As we have seen above, even in simple examples the amount of code that needs to be written to ensure safe access to shared data and code that ensures a certain order of execution of parallel computations is remarkable. The issues that the coordination code addresses are in addition to the computational problem that a program solves. In fact, managing the effects and dependencies between threads is a complex problem in its own right that is independent of the algorithmic parts of a program. Yet, it is common practice to solve the two conceptually independent issues of computation and coordination on the same level of abstraction which results in programs where algorithmic code and coordination code is intricately intertwined. Verification of correctness is not the only aspect that is complicated by this approach; the reusability of code and not least the comprehensibility of a program are other aspects that are often corroded by this style of programming.

The threading-centric, explicitly managed approach to parallel programming very well matches parallel hardware requirements, grants close control over computing resources and consequently promises predictable, high performance. However, programming this way does not only require an intimate understanding of the executing machinery, it also requires a certain degree of creativity to formulate an efficient solution to a problem on this level of abstraction. What is more, in the absence of compile-time analysis for code using e.g. Pthreads or MPI, we are oftentimes left with not much more than intuition about the correctness of the coordination aspects of a program until actually running it. So again we are at a point where the transition from an area of programming as an art towards a purely scientific discipline is held back.

The realisation that the rather low-level approach to parallel programming poses many challenges is not a new one. Substantial efforts are spent on developing alternatives that would ease the burden of programmers who write parallel software.

Semi-automatic approaches hide many technical details of thread management and only rely on programmers to provide guidance. Typically, there is no direct control over threads and the communication between threads is handled automatically by the compiler. Where parallel execution is desired, the programmer explicitly marks code by means of language specific annotations. A prominent example of this is approach is OpenMP [Boa11]. OpenMP automatically introduces fork-join points for threads into an application, guided by code annotations. An OpenMP annotated version of the state transition loop of our example is shown in Fig. 2.2. The `#pragma omp` annotation tells the compiler that a block of code follows for which we request parallelisation. The

```
void gol_step( args_t *A)
{
  ...
  #pragma omp parallel for private(x)
  for( y=A->start_y; y<ysize; y++) {
    for( x=A->start_x; x<xsize; x++) {
      n = 0;
      n += grid_curr[y][x+1]; // right
      ...
    }
  }
}
```

Figure 2.2: Implementation of the state transition using OpenMP

keyword `for` specifics this request to be a parallel loop. The compiler automatically splits up the iteration space of the loop and distributes it among several threads. The number of threads is configurable at runtime and defaults to the number of CPUs of the host machine. We still have to pay attention to shared variables though. The loop index of the parallelised loop is automatically protected by the compiler. However, the counter of the inner loop would be shared by default which would lead to wrong results as all threads would increment *the same* loop counter `x`. In order to avoid this problem we annotate `private(x)` at the parallelised loop. This instructs the compiler to ensures that each thread is accessing its own, un-shared copy of `x` only.

The most drastic approach is taken by languages that provide automatic parallelisation where no means of thread coordination are provided. Languages, and more specifically their compilers, that implement such a fully transparent approach manage threads and communication between them autonomously and hide the specifics from a programmer completely. Programmers can focus on the purely algorithmic aspects of an application without being concerned with parallelisation. However, as a consequence, programmers depend on the compiler's ability to identify code that can be executed in parallel. Because of the complex code analysis that is required to achieve this automatic parallelisation is known to be a hard problem [BBH$^+$08, PBCV07, BVZ$^+$08] and it is usually applied to specific code patterns only. Still, as an example for an automatic approach let's rewrite the state transition loop of our example in SAC [Sch03, GS06]. SAC is a functional array processing language that provides a data-parallel construct, the `with`-loop, that the compiler is able to parallelise automatically [Gre01]. Fig. 2.3 shows the example implemented in SAC. The `with`-loops are automatically compiled into code that makes use of several threads, all executing on a different part of the array. In added benefit of of the functional semantics is that we do not have to worry about pointers and swapping the grids.

```
int[.,.] gol_step( int[.,.] grid)
{
  grid = with { ( [1,1] <= iv < [X-1,Y-1] ) {

    /* count neighbours */
    n = with { (iv-[1,1] <= idx <= iv+[1,1]) : grid[idx];
        } : fold( +, 0); /* folds with +, 0 as neutral element*/

    if( grid[iv] == 0) { /* cell is dead */
      next_val = (n == 3) ? 1 : 0;
    }
    else { /* cell is alive;
              n-1 as cell was counted in fold */
      next_val = ((n-1) == 2 || (n-1) == 3) ? 1 : 0;
    }
    } : next_val; /* block above determines value of next_val */
  } : modarray(grid); /* modarray computes new modified array */
  return( grid);
}
```

Figure 2.3: Implementation of the state transition in SAC

2.3.3 An Excursion: Programming With Explicitly Managed Threads

As an illustrative example let's consider an implementation of a simplified version of Conway's *Game of Life* [Gar70]. The Game of Life simulates the birth, survival and death of cells in discrete time steps. The cells exist in an infinite, 2-dimensional space where each cell is in one of the two states *alive* or *dead*. The amount of living neighbours, of which a cell can have a maximum of eight (Fig. 2.4), determines the future of a cell. In each time step a cell inspects its four neighbours. Depending on

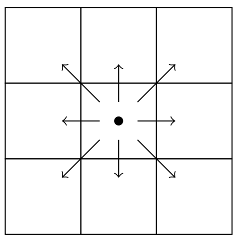


Figure 2.4: A cell and its eight neighbours.

its own state and the number of alive cells around it, the cell determines its new state:

current state	number of alive neighbours	new state
dead	3	alive
	otherwise	dead
alive	2, 3	alive
	otherwise	dead

In our simplified implementation using C shown in Fig. 2.5 (in some examples not all code is not shown, this is indicated by ...), we model the cell grid as a finite, two-dimensional array and assume to have only perpetually dead cells outside the boundaries of the array. The values for `X, Y, STEP` are assumed to be provided from the outside; these define the total size of the first and the second dimension of the cell grid and the number of time steps to be simulated. We use two grids to implement the simulation. One grid holds the current state of all cells. The cells after the state transition, i.e. the next state, are stored on a separate grid. The function `gol_step` simulates one time step and transitions all cells into a new state. The function `gol` calls the state transition function for a given number of time steps. After each step it swaps the two grids. The `main` function initialises the grids, sets up the required parameters, triggers the simulation, and prints the result in the end. By setting the start index of the loop of the state transition to 1 instead of 0 and the stop index to $X - 1$ resp. $Y - 1$ we mimic the boundary of dead cells around the grid. Fig. 2.6 shows the initial state of the grid and Fig. 2.7 shows the grid after 1, 10, 100, and 1000 steps for `X=50`, `Y=50`.

Before we attempt to parallelise the code we need to identify where parallelism can be exploited. The iteration over the time steps would be a candidate, as well as the loop that implements the state transition for the cells on the grid. On closer inspection we have to rule out the time step based approach as time step $t + 1$ depends on the previous time step t; the iterations of the loop have to be executed in sequence. The loop that iterates over the cell grid, however, is parallelisable. We may spread the index space that is covered by the loop over several threads. Each thread works on a separate part of the cell grid independently. In our first attempt we use two threads and divide the grid up along the Y axis. The first thread processes the top half and the second thread processes the bottom half of the grid. In order to communicate the index space limits to the threads, we create a two argument containers `arg_1` and `arg_2`, one for each thread. The upper limit for the Y axis is changed to `Y/2` for thread one and the lower limit the axis is changed to `Y/2` for thread two. For thread management we use a simplified version of the Pthreads API [IG99]; the function `thread_create` spawns a new thread and the function `thread_join` waits for a thread to finish its execution. By spawning two threads, both executing the same function `gol` but with different arguments `arg_1` and `arg_2`, a first parallel version of the simulation is almost ready. Almost, because we have to remember that two threads are both executing the function `gol`; as the grid pointers are swapped in `gol` we have to make sure that only one thread swaps the pointers. We use the thread id that is stored in the argument container to achieve this. The changed code implementing this approach is shown in

```
typedef struct { int start_x; int start_y;
                 int end_x; int end_y;
                 int t_id; // for later use
        } args_t;

void gol_step( args_t *A)
{
  int xsize = A->end_x;
  int ysize = A->end_y;
  int x,y,n;

  for( y=A->start_y; y<ysize; y++) {
    for( x=A->start_x; x<xsize; x++) {
      n = 0;
      // neighbours right, left and up and down
      n += grid_curr[y][x+1]; n += grid_curr[y][x-1];
      n += grid_curr[y+1][x]; n += grid_curr[y-1][x];
      // diagonal neighbours
      n += grid_curr[y+1][x+1]; n += grid_curr[y-1][x+1];
      n += grid_curr[y+1][x-1]; n += grid_curr[y-1][x-1];
      switch( grid_curr[y][x]) {
        case 0: // cell dead
          grid_next[y][x] = (n == 3) ? 1 : 0;
          break;
        default: // cell alive
          grid_next[y][x] = ((n == 2) || (n == 3)) ? 1 : 0;
      }
    }
  }
}

void gol( args_t *A)
{
  int s;

  for( s=0; s<STEPS; s++) {
    gol_step( arg);
    grid_tmp = grid_curr; // swap grids
    grid_curr = grid_next;
    grid_next = grid_tmp;
  }
}

int main( int argc, char **argv)
{

  args_t *arg;
  ...
  initialise_grids( X, Y); // not shown

  arg_1->start_x = 1; arg_1->end_x = X-1;
  arg_1->start_y = 1; arg_1->end_y = Y-1;

  gol( arg);

  print_result( X, Y); // not shown

  return( 0);
}
```

Figure 2.5: A sequential implementation of the *Game of Life* in C.

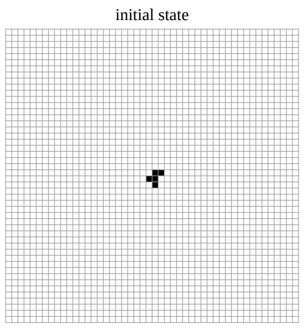


Figure 2.6: The initial state of the cell grid.

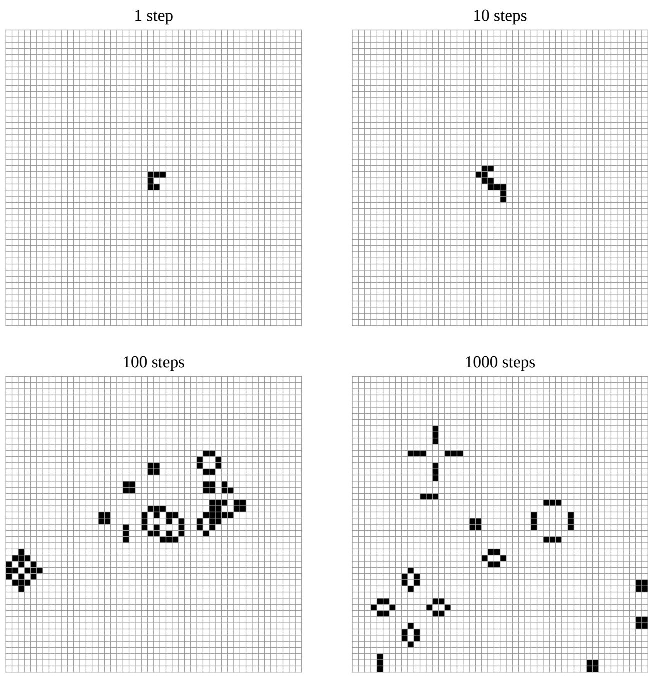


Figure 2.7: The state of the grid after several generations (time steps).

Fig. 2.8. Fig. 2.9 shows the resulting grids after several time steps. Compared to the

```
void gol( args_t *A)
{
  int s;

  for( s=0; s<STEPS; s++) {
    gol_step( arg);
    if( A->t_id == 1) {
      grid_tmp = grid_curr; // swap grids
      grid_curr = grid_next;
      grid_next = grid_tmp;
    }
  }
}
...
int main( int argc, char **argv)
{

  args_t *arg_1, *arg_2;
  thread_t *t_1, *t_2;
  ...
  arg_1->start_x = 1; arg_1->end_x = X-1;
  arg_1->start_y = 1; arg_1->end_y = Y/2;
  arg_1->t_id = 1;

  arg_2->start_x = 1;   arg_2->end_x = X-1;
  arg_2->start_y = Y/2; arg_2->end_y = Y-1;
  arg_2->t_id = 2;

  t_1 = thread_create( gol, arg_1);
  t_2 = thread_create( gol, arg_2);
  thread_join( t_1);
  thread_join( t_2);
  ...
}
```

Figure 2.8: A first implementation attempt of a parallel simulation of *Game of Life*.

results of the sequential implementation the grids look rather different. Why is this? How did the parallelisation effect the result? What we need to observe here is that the two threads are not executing synchronously. After the threads are spawned, they both execute the state transition loop for all time steps. However, the threads do not operate in lockstep. If one thread progresses faster through the loop than the other, the threads end up simulating different time steps. What is worse, because of the way we swap the pointers it is possible that the pointers are swapped while the other thread is still computing. For short program runtimes, i.e. for only a few simulation steps, this problem doesn't seem to hit us, but the longer the runtimes the more likely it is that the threads run out of sync, for example because one thread is de-scheduled by

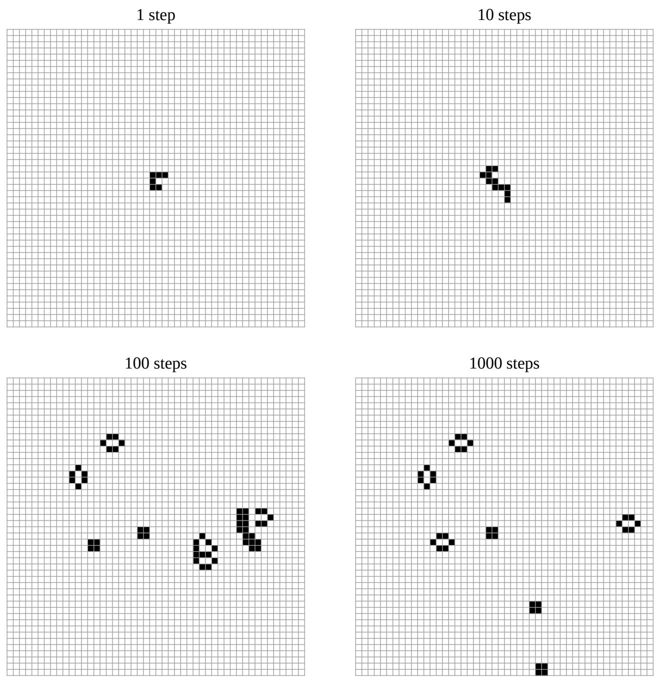


Figure 2.9: The results of a parallel run of the simulation.

the operating system while the other thread keeps running.

In order to ensure that both threads always execute the same time step and to ensure that the pointers are swapped at the right time we need to synchronise the threads. Only the thread that finishes last may swap the pointers. All other threads have to wait for this to happen. Only then may all threads continue simulating the next time step. We achieve this by introducing a counter with an initial value of 0 that is shared between the threads. Each thread increases the counter by one after finishing the state transition of one time step and checks the counter value against the number of threads (`NUM_T`, two in our example). If the counter value does not equal the number of threads, the thread waits for the counter to be reset to 0. Otherwise, the thread swaps the pointers and resets the counter to 0. The code for this approach is shown in Fig. 2.10. This new approach seems to behave even more randomly. Sometimes, the program produces correct results, other times the results are wrong or the program just hangs and no result is produced. What is going on now? One problem is that the increasing of the counter and the check of the value are not atomic, i.e. after increasing but before checking the value another thread might have changed the value as well:

```
int counter = 0;

void gol( args_t *A)
{
  int s;

  for( s=0; s<STEPS; s++) {
    gol_step( arg);

    counter += 1;
    if( counter < NUM_T) {
      while( counter != 0); // wait
    }
    else {
      grid_tmp = grid_curr; // swap grids
      grid_curr = grid_next;
      grid_next = grid_tmp;
      counter = 0;
    }
  }
}
```

Figure 2.10: A synchronised, parallel simulation of *Game of Life*.

thread 1	thread 2	value of counter
		0
`counter += 1`		1
	`counter += 1`	2
`if(counter<NUM_T)`		2
	`if(counter<NUM_T)`	2
swap pointers and reset		0
	swap pointers and reset	0

In this example both threads swap the pointers causing wrong results. Another problem is that we cannot ensure that a waiting thread checks the counter value in time to not miss its "wake-up call":

thread 1	**thread 2**	**value of counter**
		0
`counter += 1`		1
`if(counter<NUM_T)`		1
wait		
	`counter += 1`	2
	`if(counter<NUM_T)`	2
	swap pointers and reset	0
	`gol_step`	0
	`counter += 1`	1
	`if(counter<NUM_T)`	1
	wait	

The result in this case is that both threads are waiting forever. The program will not produce any results and "hang" indefinitely.

In order to avoid the first issue we have to introduce a *critical section*, i.e. a piece of code that is never executed by more than one thread at a time. If we establish such a region around the increment of the counter and the check of its value the situation where multiple threads see a counter value of `NUM_T` is avoided. Implementing critical section is typically achieved by using mutual exclusion [Dij65]. Entry into a critical section is guarded by a *mutex lock*. If a thread tries to lock an unlocked mutex, the mutex is locked and the thread may continue its execution as normal. If, however, the lock is already locked, the thread has to wait until the lock has been unlocked again. Our simple threading API (very much like PTHREADS) provides `mutex_lock` and `mutex_unlock` for this.

The second issue where all threads end up waiting may be solved by using a reliable way of signalling a change of the counter to waiting threads. A way of implementing this is by means of *condition variables*. Such variables usually support at least two operations, *waiting* and *signalling*. A thread that waits on a condition variable will do so until another thread broadcasts a signals on the same variable. Multiple threads may wait on the same variable and they all resume execution after receiving a broadcast signal. Waiting on a condition variable also involves a mutex; when a thread is waiting for a condition, the mutex is unlocked. When the thread receives a signal, the mutex is locked before the thread is allowed to continue execution. We assume our threading API to provide `thread_mutex_lock`, `thread_mutex_unlock`, `thread_cond_var_wait` and `thread_cond_var_broadcast`. The code implementing an approach using a mutex and a condition variable is shown in Fig. 2.11; this program produces correct results and finalises our parallel implementation using shared memory.

Implementing a parallel program for distributed memory poses a different set of challenges. As threads cannot access the same values in memory there are no interference issues as those we have encountered in the previous example. However, for the same reason, implementing a distributed algorithm requires us to come up with a solution that is solely based on thread-local data structures and yet implements the

```
mutex_t mutex;
cond_var_t cvar;

void gol( args_t *A)
{
  int s;

  for( s=0; s<STEPS; s++) {
    gol_step( arg);

    thread_mutex_lock( mutex);
    counter += 1;
    if( counter < NUM_T) {
      // wait on condition variable, while waiting 'mutex'
      // is unlocked, when resuming it is locked again
      thread_cond_var_wait( cvar, mutex);
    }
    else {
      grid_tmp = grid_curr; // swap grids
      grid_curr = grid_next;
      grid_next = grid_tmp;
      counter = 0;
      // send signal to waiting threads
      thread_cond_broadcast( cvar);
    }
    thread_mutex_unlock( mutex);
  }
}
```

Figure 2.11: A synchronised, parallel simulation of *Game of Life*.

same behaviour as the simulation on a single, globally accessible grid. For this example, let's assume we are using a simple message passing API (resembling MPI [For09]) that provides the following functions:

Function	**Description**
`my_id()`	returns an integer, uniquely identifying the calling thread
`msg_send(`*id, loc, num*`)`	sends *num* elements to thread *id* starting at local memory location *loc*
`msg_recv(`*id, loc, num*`)`	receives *num* elements from thread *id* storing them into local memory starting at location *loc*

For simplicity we assume that messages can only contain integers. Also, a call to any of the communication functions only returns once the communication has succeeded. Our API does not provide explicit thread management. Instead, all threads execute the same program from the beginning. Having threads executing different parts of the program the thread ids have to be used for distinction, e.g.

```
if( my_id() == 1) {
  // do something
}
else {
  // do something else
}
```

We start our implementation from the sequential code in Fig. 2.5 and use the same parallelisation strategy as before, i.e. we split the grid evenly along the Y axis. Instead of allocating the entire grid, each thread only allocates a grid the size of the part that it is responsible for. Here we have a first challenge. The cells that are located at an artificial boundary, i.e. a boundary that only exists because the grid has been split up, cannot see all neighbours. Some of the neighbouring cells are located on the grid of other threads. In order to still be able to correctly compute the state transition for all cells, those remote neighbouring cells have to be communicated between threads. A common technique to implement this in a way that is transparent for the actual algorithm is to introduce *halo regions*. A halo region is populated with values from remote threads. For an example where we use two threads a setup for a distributed grid including appropriate halo regions is shown in Fig. 2.12.

In order to update the halo regions with the required values we need to implement communication between those threads that share a boundary. The communication has to be triggered before each time step. Fig. 2.13 shows the code of a first attempt to implements this. This first implementation unfortunately deadlocks. As the threads start sending at the same time, no communication succeeds and the calls to `msg_send` cannot return. We use the thread ids to fix this problem by making sure that adjacent threads call send and receive in inverse order. Also, we have to pay attention to the fact that some threads do not have neighbours on both sides. In our distribution scheme these are the threads of id `1` and id `NUM_T`. Keeping all this in mind, we write the code

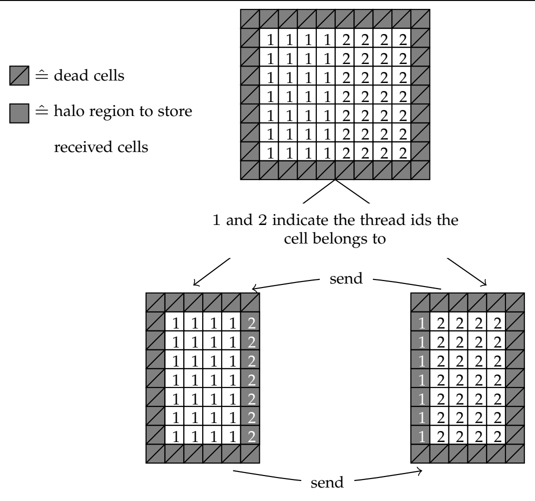

≙ dead cells
≙ halo region to store received cells
1 and 2 indicate the thread ids the cell belongs to
send
send

Figure 2.12: A distributed grid including halo regions.

```
void update_halos( args_t *A)
{
  // the halo cells are located at row 0 and row ysize+1
  // local cells are consequently located between 1 and ysize

  // exchange lower boundary
  msg_send( my_id()-1, grid_curr[1], xsize);
  msg_recv( my_id()-1, grid_curr[0], xsize);

  // exchange lower boundary
  msg_send( my_id()+1, grid_curr[ysize], xsize);
  msg_recv( my_id()+1, grid_curr[ysize+1], xsize);
}

void gol( args_t *A)
{
  int s;

  for( s=0; s<STEPS; s++) {
    update_halos( A);
    gol_step( arg);
    grid_tmp = grid_curr; // swap (local!) grids
    grid_curr = grid_next;
    grid_next = grid_tmp;
  }
}
```

Figure 2.13: An implementation that updates halo regions before each time step.

shown in Fig. 2.14. As a final challenge we implement a function that prints out the

```
void halo_transfer( args_t *A)
{
  // the halo cells are located at row 0 and row ysize+1
  // local cells are consequently located between 1 and ysize
  if( my_id()%2 == 0) {
    // exchange lower boundary
    if( my_id() != 1) {
      msg_send( my_id()-1, grid_curr[1], xsize);
      msg_recv( my_id()-1, grid_curr[0], xsize);
    }
    // exchange upper boundary
    if( my_id() != NUM_T) {
      msg_send( my_id()+1, grid_curr[ysize], xsize);
      msg_recv( my_id()+1, grid_curr[ysize+1], xsize);
    }
  } else {
    // exchange upper boundary
    if( my_id() != NUM_T) {
      msg_recv( my_id()+1, grid_curr[ysize+1], xsize);
      msg_send( my_id()+1, grid_curr[ysize], xsize);
    }
    // exchange lower boundary
    if( my_id() != 1) {
      msg_recv( my_id()-1, grid_curr[0], xsize);
      msg_send( my_id()-1, grid_curr[1], xsize);
    }
  }
}
```

Figure 2.14: Implementation of a communication scheme.

cell grid. What we need to achieve is to print the local grids of the threads in the right order, one after the other. This can be done by using a token that is handed from one thread to the next by means of sending a message. Only one token exists and only the thread holding the token is allowed to print. The thread with id 1 generates the token, prints out its local grind and then hands over the token to its neighbouring thread. The process repeats until the token reaches the last thread. After the thread with id `NUM_T` finishes printing, it discards the token. The implementation of this technique is shown in Fig. 2.15.

We could continue the examples for a while. An obvious next step is to combine the two approaches into a hybrid implementation where the grid is first distributed over several threads and each thread again uses a parallelised state transition loop on its local grid. In further steps we may also want to make use of GPGPUs through CUDA [NBGS08] or OpenCL [SGS10] and model the state transition function as a data-parallel operation for the entire of just parts of the grid. With each step we ideally enable the application to make use of more diverse, heterogeneous hardware to

```
void print_result( args_t *A)
{
  int token = 0;

  // first thread does not wait for token
  if( my_id() > 1) {
    msg_recv( my_id()-1, &token, 1);
  }
  print_local_result( A); // not shown
  // last thread does not send token
  if( rank < NUM_T) {
    msg_send( my_id()+1, &token, 1);
  }
}
```

Figure 2.15: Implementation of distributed printing.

perform its task faster and more efficient. On the downside of these advantages lies the increased complexity that every such step brings with it as the parallelisation, data distribution and communication techniques become more intricate. Additionally, alongside the algorithmic complexity that inevitably ensues, the code complexity also grows with each layer. The more APIs are involved and the more parallelisation strategies are employed, the more code for coordination is required to bring all strategies together in one application.

3 S-Net: A Language for Stream-Based Coordination

This chapter introduces S-NET, a stream-based coordination language that we will base our further investigations on. The language provides essential constructs and a methodology to describe streaming networks of computational components.

We focus on the features and core components of the S-NET programming system relevant to this dissertation. A full account on S-NET is given in [CPG$^+$10].

3.1 Overview

S-NET is a coordination language. The focus of the language lies on providing means to express the interaction between computing components rather than the computations that are carried out by these components. As such, S-NET does not provide the typical operations that one expects to find in programming languages. The computational aspects of an application, i.e. the implementation of algorithms that operate on data, are left to external languages. Likewise, the standard data types, e.g. int, double, char, and operations on these data elements are not provided. Instead, computations are only abstractly represented by a name and a function signature, very much similar to function prototypes in a C header file.

Using only an abstract notion of computation an S-NET program instead describes how these computations interact rather than what exactly they compute. An S-NET program encodes how the computations are arranged and joined up to form an application. The arrangement is expressed through the use of combinators that join up computations into compounds. Channels are established between the constituents of such a compound for communication.

Combinators provide means to define the interconnection network between components in terms of algebraic formulae. Applying a combinator to a components forms a compound component; within this compound the combinator defines the dependencies between its constituents. The dependencies are captured by a data-flow graph that is inherent to the combinator. S-NET provides several combinators, each with its own, generic data-flow graph that is instantiated for the operands given to the combinator. More specifically, the operands form the nodes of the graph; the edges between them are implicitly defined by the combinator. The edges establish the communication links between components. These links provide directed first-in-first-out (FIFO) channels over which messages between components are exchanged.

In the remainder of this chapter we will first deal with the representation of messages and data within the system. Computations as atomic building blocks of the data-flow network that an S-Net program represents are introduced and we will see how a computation is represented in terms of observable effects on messages in Sect. 3.3.1 and Sect. 3.3.2 deals with the aspects of synchronisation. In Sect. 3.5 combinators and their inherent network constructions are presented.

3.2 Messages

Communication in S-Net is based on uni-directional first-in first-out channels which we will also refer to as streams. Any two components that wish to communicate will only be able to do so if a combinator has placed a stream between these two components. In order to ensure that a connection is only established between components that send and receive compatible messages we require a component to declare what type of messages it is able to process and what type of messages it will produce. We shall deal with the details on how messages are structured in Sect. 3.2.1 and deal with properties and relations of messages in Sect. 3.4.1.

3.2.1 Records

Messages, which we will refer to as records, are sets of zero or more data elements. The data elements are associated with a unique name, i.e. a record is a collection of label-value pairs. Label-value pairs are typed, but unlike most other programming languages we limit the set of possible types to *fields*, *tags* and *binding tags*. Fields refer to data that has been produced by a computational box. Since boxes are implemented in some external language the nature of the data is unknown; as far as S-Net is concerned a field label represents an opaque data object that cannot be inspected. The data of these elements is only meaningful to computational boxes. Tags, on the other hand, carry values that are visible on the S-Net layer as well. As such, tag values are restricted to integers only. The value of tags may be read and written by the implementation of a computational box and other S-Net entities alike. The same holds true for binding tags, i.e. the third category of labels. Binding tags also carry plain integer values, just like tags, but have a different effect on sub-typing; we shall revisit this point in Sect. 3.4.1.

The syntactical convention for representing records is given in Syntax 1.

Syntax 1. *Field names, tag names and binding tag names may be strings over an arbitrary (finite) alphabet.*

RecordType	$\Rightarrow$	{ *[RecordEntry [* , *RecordEntry]*]* }
RecordEntry	$\Rightarrow$	*Field* | *Tag*
Field	$\Rightarrow$	*FieldName*
Tag	$\Rightarrow$	*SimpleTag* | *BindingTag*
SimpleTag	$\Rightarrow$	< *SimpleTagName* >
BindingTag	$\Rightarrow$	< # *BindingTagName* >

Let's assume we are dealing with messages in an image processing application. A record that contains image data and the image's dimensions may contain a field for the image data and tags to encode the size; a record that contains all these information may be encoded as `{img, <x_res>, <y_res>}`.

In this representation of records the values of the record's constituents are not visible. The representation abstracts from the actual data that the labels refer to, i.e. this record description may be used in place for all records that contain the same labels regardless of their values. Consequently, we should more accurately call this description a *record type* (Sect. 3.4. We will allow ourselves to be occasionally inconsistent in the use of the terms record and record type and use them interchangeably. If we refer to a specific record we may include the values in the record description as in `{<x_res=256>, <y_res=256>}`; this, because of the nature of field data, usually only makes sense for tags and binding tags. Also, this notation is not legal S-NET syntax and we only informally use this to make a clearer distinction between record types and concrete instances, i.e. records, of that type.

3.3 Atomic Boxes

3.3.1 Computational Boxes

Computational boxes, or just boxes for short, are the only entities within a program that may implement computations on field data. The computations that are carried out within a box are expressed in an external language, commonly referred to as the box language. Hence, boxes link the code of a computational language as for example C or SAC [Sch03, GS06] to the coordination domain of S-NET.

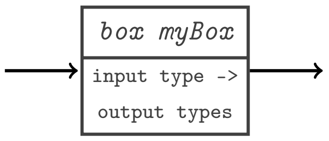
Regardless of its potentially complex inner workings all that a box declares to S-NET is its type signature. A type signature defines what type of records the box is able to process and what results it may produce. A box declares exactly one record type as its input. The output declaration is less restrictive. A box may declare several record types as output. The input record type and the output record types make up the type signature of the box. The signature is the contract between the external language and S-NET. For S-NET it states that all records that are given to the box will be of the type as declared in the input type. For the external language this contract states two things. Firstly, it requires that the box implementation will only ever produce results that are of the types that declared as output types. Secondly, boxes do not retain any state information across multiple computations. After the box finishes computing on one input no information about previous computations is stored.

The result that a box produces may be made up from zero, one or many records; a box produces a stream of records as response to an input. On this output stream any (finite) number of records of any of the declared output types may appear in any order.

The syntax for boxes in S-NET is given in the following.

Syntax 2. *The definition of a computational entity in* S-NET *commences with the keyword* `box` *followed by an identifier for the box's name and its type signature. The graphical representation of a computational box is given below as well.*

BoxDef	⇒	**box** *BoxName* **(** *BoxSignature* **)** **;**
BoxSignature	⇒	*RecordType* **->** *RecordType [* **|** *RecordType]**



A box that implements an image filter and accepts as input a record that carries image data named `img` and produces as output a filtered image called `flt_img` may be declared as

```
box flt({img} -> {flt_img});
```

If we also have a box that applies multiple filters to an input image and which produces multiple results, say an edge detection filter that enhances horizontal and vertical edges, then we might declare such a box as

```
box flt_sobel({img} ->   {img_x_edges}
                       | {img_y_edges});
```

This box may produce, on input of one record, a series of outputs of either type arbitrarily interleaved, exactly one record of one of the two types or none at all.

3.3.2 Synchronisation Boxes

All boxes, as well as all other constructs in S-NET as we will see later, are connected to one single input stream and one single output stream. It is possible, however, to communicate different types of records over these streams. Boxes with multiple right-hand sides are an example for this. Choice combinators that are introduced in Sect. 3.5 are another.

Joining multiple records of different types into a single result, i.e. synchronising records, with only state-less boxes and single input stream components cannot be achieved. Still, the ability to synchronise data and computations is essential in many application scenarios. The *synchro cell* in S-NET provides this feature.

A synchro cell is also a single-input single-output stream component but this atomic box is parametric in two record types. These record types, also referred to as *pattern*, establish the left-hand side of the cell's type signature and the set of labels in each of the types need to be disjoint. The synchro cell accepts records of either type as input. A record that arrives at the cell is tested against the two types using the sub-type relationship of Def. 1. If the input record's type is a sub-type of the first pattern, then the record is stored by the cell. The cell also marks the first type as being matched; no output is produced in this case. If the first has already been marked as being matched, then the record is not put into storage but forwarded to the output stream. The behaviour for a record matching the second pattern is analogous to matching the first.

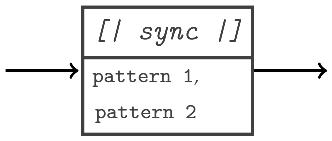
The situation is different, however, if a record matches an unmatched pattern when the other pattern has also been matched before. In this case the cell merges the record from storage and the input record into one result. The result then is the input record plus all the labels that the other pattern stated for the record being in storage. This is consistent with flow-inheritance; the merge operation is adding the labels of the record in storage to the record that has just arrived. Where the record in storage did not trigger any output the arrival of the second record did, and consequently keeps all labels that are not mentioned in the pattern that it matched due to flow inheritance.

After both pattern have been matched and the output has been written to the output stream the cell disappears. Any record that arrives after this point will be forwarded untouched. No further matching and merging is taking place. Continuous synchronisation may be achieved in conjunction with the star combinator which we will deal with in Sect. 3.5.

Syntax 3 shows the syntactical convention for defining synchro cells.

Syntax 3. *A synchro cell is parametric in two record types. The record types are also called* patterns *as they are used to match inbound records against them. The graphical representation of a synchro cell is shown below the grammar rules.*

Sync	$\Rightarrow$	**[|** *Pattern* **,** *Pattern* **|]**
Pattern	$\Rightarrow$	*RecordType*



3.4 Types

Record types and type related issues in general will be dealt with in great detail in Chapter 6. For now we keep the details to a minimum and work with an intuitive understanding of record types as sets of labels, define a sub-type relation on such types and introduce a notion of inheritance.

3.4.1 Record Types

In a view where record types are simple sets of labels the order of labels within a record type is irrelevant. Consequently, the two record types

$$\texttt{\{<x\_res>, <y\_res>\}} = \texttt{\{<y\_res>, <x\_res>\}}$$

describe the same set of records.

Interpreting record types as sets of labels allows for a definition of *sub-typing* based on set inclusion.

Definition 1. *A record type τ' is a* sub-type *of record type τ if all field labels and tag labels of τ also appear in τ' and the set of binding tags of τ and τ' are identical. Let $BT(\alpha)$ denote the set of binding tags of a record type α. Then, record type τ' is a sub-type of τ if*

$$\tau \subseteq \tau' \ \wedge \ BT(\tau) = BT(\tau')$$

We will write $\tau' \sqsubseteq \tau$ in this case. We may also say that τ is a super-type *of τ'.*

For example, the record type of a record that contains the resolution of an image in addition to the image data is a subtype of record type that only contains one label for the image data, `{img, <x_res>, <y_res>}` $\sqsubseteq$ `{img}`. If we use binding tags, say to encode if the image is a colour or a grey-scale image, than the presence of a binding tag label restricts any sub-type and super-type relation to those record types that carry the same binding tag label. This means that `{img, #<colour>}` $\not\sqsubseteq$ `{img}` but `{img, #<colour>}` $\sqsubseteq$ `{#<colour>}`. Def. 1 also states that for any record type τ it holds that $\tau \sqsubseteq \tau$.

3.4.2 Flow Inheritance

Box declarations are asymmetric as they allow multiple right-hand sides but only exactly one input type. The concept of *flow inheritance* implicitly loosens the restriction on the input type declaration. Flow inheritance widens the set of acceptable record types of a box by all sub-types of the box's declared input type. For example, the filter box

```
  box flt({img} -> {flt_img});
```

is not only compatible with records that carry exactly one label of name `img` but with all records that carry *at least* a label of that name. Let's assume we enrich an appropriate record by information about the resolution of the image as we have done before. Such a record, {img, <x_res>, <y_res>}, is still compatible with box flt. The box implementation, however, will only be presented with data that is associated with labels that are explicitly mentioned in a box's input type. Flow inheritance is a mechanism of S-NET. All labels that are in excess to an input type are removed from a record before it is passed to the box implementation. However, these labels are not lost; they are inserted into the records that the box produces in response to the input record that the labels have been stripped off from. For example, if the box flt is presented with record {img, <x_res>, <y_res>} then the two tags of this record are retained by S-NET. The box produces a record {flt_img} in response. By flow inheritance the two tags are now inserted into the result such that the final result of the box is {flt_img, <x_res>, <y_res>}. For a box that produces more than one output, any retained label is carried over to all produced results. The box flt_sobel, for example, may produce two records {img_x_edges} and {img_y_edges} in response to one input record. If the input record was {img, <x_res>, <y_res>} then the two output records are automatically extended to {img_x_edges, <x_res>, <y_res>} and {img_y_edges, <x_res>, <y_res>}.

A formal account on flow inheritance is given in the following Definition 2.

Definition 2. *Let $v^{[i]} \to \tau^{[i]}$, $i \in [1, \ldots, n]$, be the type signature of a box X. Furthermore, let each output type $\tau^{[i]}$ have m_i variants $\tau^{[i]} = \{w_1^{[i]}, \ldots, w_{m_i}^{[i]}\}$. Then for any $k \leq n$ and any field or non-binding tag $\phi \notin v^{[k]}$ such that*

$$(\forall i \neq k) BT(v^{[k]}) \neq BT(v^{[i]}) \ \vee \ v^{[k]} \cup \{\phi\} \not\subseteq v^{[i]} \, ,$$

the box X can be sub-typed by flow inheritance *to the type $X' : V^{[i]} \to T^{[i]}$, where*

$$V^{[i]} = \begin{cases} v^{[i]} & \text{if } i \neq k, \\ v^{[k]} \cup \{\phi\} & otherwise; \end{cases}$$

and

$$T^{[i]} = \begin{cases} \tau^{[i]} & \text{if } i \neq k, \\ \tau_* & otherwise. \end{cases}$$

Here $\tau_* = \{V_1, \ldots, V_{m_k}\}$ and each $V_i = w_i^{[k]} \cup \{\phi\}$.

We should note here that this definition of flow inheritance also covers type signatures that contain multiple left-hand sides. This broader definition is put into place to cover the cases concerning signatures for entire networks rather than only boxes. We will deal with more complex signatures later; for now it suffices to assume that i does not range over multiple values but is fixed to 0.

3.5 Constructing Networks

The atomic boxes introduced in Sect. 3.3 are the basic building blocks of networks. The construction of a network in S-NET is achieved by applying combinators to building blocks. Each combinator provides a specific network layout into which the combinator embeds its operands. A combinator turns one or two building blocks into a new building block, i.e. network construction is an inductive process. The combinator turns its operands into a new single-input, single-output component based on its intrinsic network layout. The newly formed component may now be used by combinators, just as atomic boxes, to construct more complex networks.

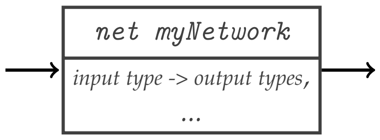
Networks may be nested. This enables the use of structured programming practices as an outer network may refer to one of its inner networks by its (non-optionally given) name. Similarly, networks that are not nested but defined on the same level may also refer to each other by their given names in their respective topology expressions.

Syntax 4 gives a general overview of the constructs for defining networks in S-Net.

Syntax 4. *Networks are always given a name for identification and a network signature. The signature defines the record types that the network accepts as input and the types of records that it produces as output.*

```
NetDef          ⇒   net NetName ( TypeSignature ) NetBody
TypeSignature   ⇒   BoxSignature [ , TypeSignature ]*
NetBody         ⇒   [ { [Definition ]* } ] connect TopoExpr ;
TopoExpr        ⇒   BoxName
                |   NetName
                |   Sync
                |   Combination
                |   ( TopoExpr )
Combination     ⇒   Serial | Star | Choice | Split
```

The graphical representation of networks resembles that of atomic boxes. The main difference is that a network may have multiple type mappings.



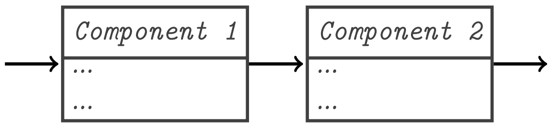
The available combinators and resulting combinations are introduced in the following Sections 3.5.1-3.5.4.

3.5.1 Serial Combinator

A serial combinator arranges two components in sequence. The combinator takes two components as operands and establishes a link between the two in such a way that the output of the first operand becomes the input of the second. All output of the first operand is made available to the second operand in the same order as it is produced by the first, i.e. the combinator instantiates a two-stage computational pipeline where each operand is used as one stage. The representation of serial combinations is given in Syntax 5.

Syntax 5. *The serial combinator is represented by two dots; in its graphical representation it is a sequence of two (compound) components with a stream in between them as shown below.*

Serial	⇒	*TopoExpr SerialComb TopoExpr*
SerialComb	⇒	. .



3.5.2 Choice Combinator

The choice combinator's intrinsic network layout arranges two component operands in parallel. More specifically, the combinator introduces a split point, two branches and a merge point into a network. The two component operands of the combinator are placed each on one branch, the split point connects the two branches to the preceding component and the merge point collects both branches to provide a single output connection. In addition to the two component operands the combinator also requires two type operands to be provided. These type operands each provide lists of one or more record types.

At the split point an inbound record is forwarded to exactly one of the branches. The decision as to which branch is chosen is type based. If the type of the inbound record is a sub-type of the input type of the first branch, then this branch is chosen. If the record's type is a sub-type of the second branch, than the second branch is chosen. In case both branches are possible destinations one of them is chosen non-deterministically[1]

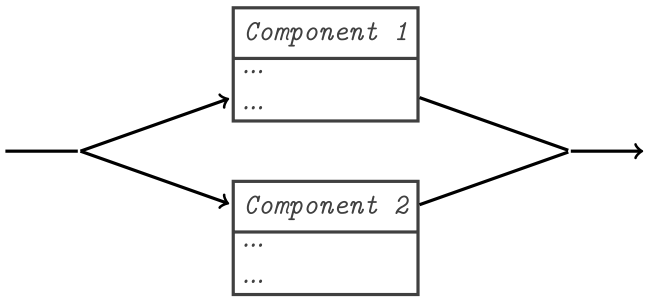
For the decision at the split point to take place the input types of both branches need to be available to the combinator. The input types of the branches are given to the combinator as operands alongside with the two component operands.

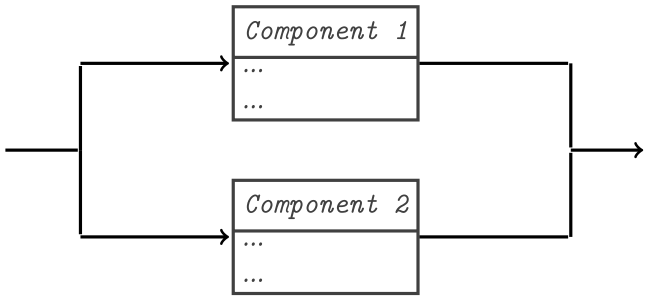
The merge point outputs the result records of the branches in the order in which the input appeared at the split point. Any output produced from one record is fully output before output of a later record appears on the outbound stream, i.e. the result streams are not interleaved.

Syntax 6. *The textual representation of the choice combinator is two vertical bars. Two graphical representations are given below. If the branch types are not immediately obvious in the graphical representation, e.g. if each branch contains compound components, we annotate the input types on top and below the respective streams after the split point.*

[1]One may also wish to employ a best-match strategy here; the branch that offers the larger overlap of the label sets with the input record is chosen. However, for simplicity, we work without a best-match strategy here.

Choice	⇒	*ChoiceOp ChoiceComb ChoiceOp*
ChoiceOp	⇒	*TopoExpr [* , { *[RecordEntry]** } *]+*
ChoiceComb	⇒	||





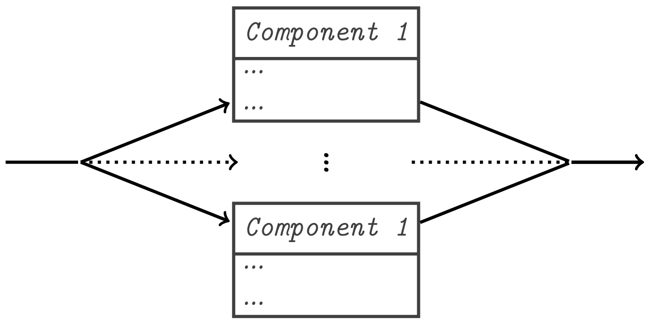
3.5.3 Split Combinator

The split combinator is a binary combinator that dynamically introduces branches into a network. Similar to the choice combinator of Sect. 3.5.2 the split combinator introduces a split and a merge point into the network. Unlike the choice combinator, however, the split combinator places *the same* component on all branches. Because of this, the combinator does not require the input type of the component as a separate operand. Instead of a list of record types, the second operand is a simple tag name. The value of the tag is what steers the dynamic behaviour of the combinator. A record that arrives at the combinator's split point is expected to carry a tag of the specified name. The value of the tag determines the branch that the combinator forwards the record to. Hence, the combinator maintains a conceptionally unbounded number of branches where each branch contains one instance of the operand such that records that carry the same tag value are forwarded to the same instance.

The merge point outputs the results of the branches in the order in which the split point forwarded the inbound records, i.e. the result streams of the individual operands are not interleaved.

Syntax 7. *The textual representation of the split combinator is two exclamation marks. The graphical representations is given below; if it is not obvious from the context the name of the tag operand may be given above the split point.*

Split	⇒	*TopoExpr SplitComb SimpleTag*
SplitComb	⇒	`!!`



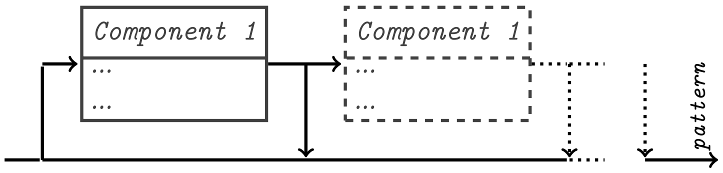
3.5.4 Star Combinator

The star combinator is a dynamic combinator as well. Where the split combinator arranges its component operand in parallel, the star combinator arranges instances of a component in sequence. The combinator maintains a conceptually unbounded pipeline of operand instances. This involves a split point that is placed in front of each operand instance. The split point manages two outbound streams. The first outbound stream is connected to the operand instance. The second outbound stream, and this is the case for all split points, is connected to one, single merge point. The merge point constitutes the global output stream of the combinator.

In order to determine the length of the sequential chain,i.e. the pipeline of operands, the star combinator requires a type operand in addition to a component operand. The given type, sometime referred to as *exit pattern*, is checked against all records that are about to enter a stage of the computational pipeline. If the type of the record is not a sub-type of the type given as operand, then the record enters the stage. However, if the type of the record is a sub-type of the given type, then the record is not processed by the operand instance of the stage. Instead, the record leaves the sequential chain of operands, i.e. the record is made available on the output stream of the combinator. Just like with the other combinators, output records are made available on the output stream in the same order as their respective input arrived at the combinator.

Syntax 8. *The textual representation of the star combinator is two asterisks. The graphical representations is given below. The exit pattern of the combinator is given above the global output stream.*

Star	⇒	*TopoExpr StarComb RecordType*
StarComb	⇒	`**`



3.5.5 Non-Deterministic Variants of Combinators

All combinators that we have introduced so far maintain a causal order between the input and the output that is produced. Let's assume we feed two records r_1 and r_2 to a combinator, and let's further assume that the operands of the combinator produce $r'_{1_1} \ldots, r'_{1_m}$ in response to r_1 and $r'_{2_1} \ldots, r'_{2_n}$ in response to r_2. If the order of records at the input is r_1 first and r_2 second, then the order of the output is such that $r'_{1_i}, i \in \{1, \ldots, m\}$ are seen first and only then $r'_{2_j}, j \in \{1, \ldots, n\}$ appear at the output.

In S-Net it is possible to drop the order-preserving property by using *non-deterministic* variants of the combinators. Non-deterministic variants exist of all but the sequential combinator. With these combinators the result streams of several input records may be interleaved. For example, for input r_1, r_2 as above a result stream of

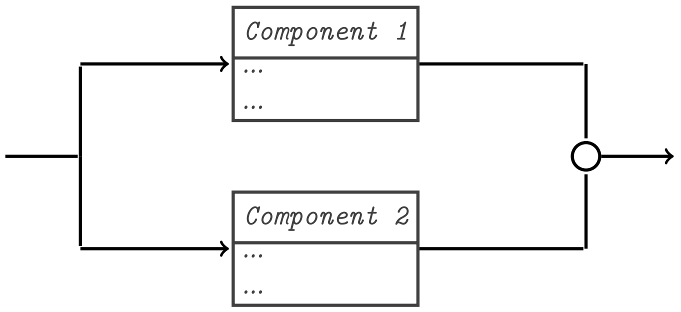
$$r'_{1_1}, r'_{2_1}, r'_{1_2}, r'_{2_2}, \ldots, r'_{1_i}, r'_{2_j}$$

is a legal output if a non-deterministic combinator is used.

Syntactically the combinators resemble their deterministic counterparts with the second symbol dropped, as is shown in Syntax 9.

Syntax 9. *We extend the syntax of combinators by three non-deterministic variants of the available combinators. The graphical representation uses an unfilled circle as merging points when the merge point belongs to a non-deterministic combinator. As an example a non-deterministic choice combinator is shown below.*

StarComb	⇒	`**` | *StarNdComb*
ChoiceComb	⇒	`||` | *ChoiceNdComb*
SplitComb	⇒	`!!` | *SplitNdComb*
StarNdComb	⇒	`*`
ChoiceNdComb	⇒	`|`
SplitNdComb	⇒	`!`



3.6 An Example

In order for us to develop a more intuitive understanding of the S-NET concepts and combinators we develop a small example application. The example we are going to use is a small image processing application that applies a few filters to an input image. The details of the implementation of the image processing stages we leave aside here as the goal is to provide insight into how to model such an application in S-NET.

In a first step we deal with the actual filtering stages that images will go through. We assume to have two filters that work independently of each other; this could be two edge detection filters, one filter that detects vertical edges and one filter that detects horizontal edges in an image. For independent tasks like these the choice combinator gives us a way to combine these computations in parallel. Fig. 3.1 shows this first step. The network accepts any record as input that carries either a field `img_x` or a field `img_y`. Detecting the horizontal and vertical edges of an image we may now send in two records where the same image is once named `img_x` and then names `img_y`. The choice combinator forwards one record to its first branch to detect horizontal edges and the other record to the second branch. The decision is of which branch to take is based on the name of the field that carries the image.

Having to send in the same image twice is inconvenient. Because of this we may decide to add a pre-processing stage that takes one image as input and then sends out two copies of the image automatically, named `img_x` and `espectively. We may` ealise this with a box that declares two different outputs. This box we place in front of the parallel combination of the two filter stages as shown in Fig. 3.2 With this in place only a single input record has to be given to the network carrying a single field `img`. The pre-processing box `preProc` outputs the two records that are processed by the two branches of the parallel filter combination.

As a further addition to this image processing application we may envision a post-processing stage as well. In this stage the individual filter results, i.e. the detected horizontal and vertical edges, are combined into one, single result image. Such a box may be of the form

```
box postProc({edges_x, edges_y} -> {all_edges});
```

```
net img_filter({img_x} -> {edges_x},
               {img_y} -> {edges_y})
{
  box sobel_x({img_x} -> {edges_x});
  box sobel_y({img_y} -> {edges_y});
}
connect sobel_x,{img_x} || sobel_y,{img_y};
```

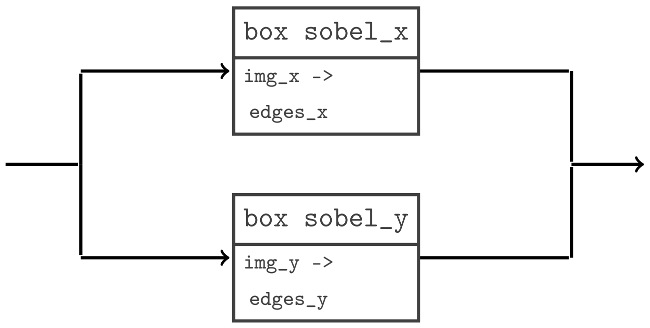


Figure 3.1: Independent tasks may be arranged in parallel using the choice combinator.

A simple serial composition of this box with the rest of the network is not giving the intended result. As boxes are not allowed to retain any internal state across computations the post-processing box cannot store one image until the second image has arrived as well. For situations like this the synchro cell provides us with the required functionality. As the two patterns of the synchro cell we specify `edges_x` and `edges_y`, i.e. the outputs of the branches of the filtering stage:

```
[| {edges_x}, {edges_y} |]
```

If placed in front of `pstProc` the cell combines the two individual result records of the filters into one record such that the merged record is of a type that the post-processing box can accept as input. This arrangement, however, would work exactly once because of the single-shot semantics of the synchro cell. After the first pair if records has been combined the synchro cell acts as simple identity forwarding all input unaltered. In order to achieve continuous synchronisation of record pairs a star combinator may be wrapped around the synchro cell. As the exit pattern of the star combinator we specify the type of the merged result of the synchro cell.

```
([| {edges_x}, {edges_y} |]) ** {edges_x, edges_y}
```

The inbound records to this combination do not match the exit pattern as they only carry either `edges_x` or `edges_y`. Thus, these records are given to the star's operand where they are stored by the synchro cell. Once the first instance of the operand, i.e. the first synchro cell, has received both records it produces a result that matches the

```
net img_filter({img} -> {edges_x},
               {img} -> {edges_y})
{
  box preProc({img} -> {img_x} | {img_y};
  box sobel_x({img_x} -> {edges_x});
  box sobel_y({img_y} -> {edges_y});
}
connect preProc
        .. (sobel_x,{img_x} || sobel_y,{img_y});
```

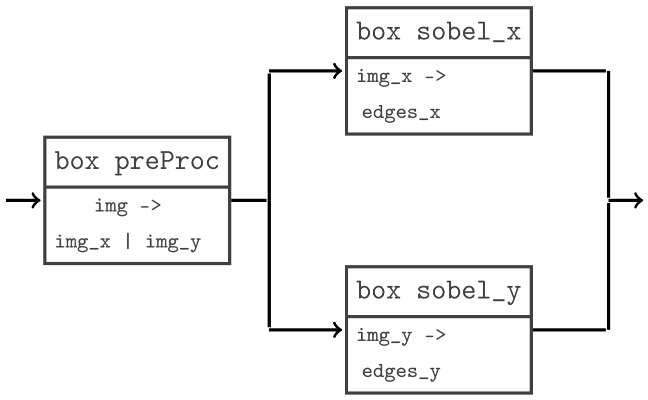


Figure 3.2: A pre-processing stage may send out records of different types.

exit pattern of the star. The record leaves the combinator and the synchro cell turns into an identity. The next records that arrive at the star combinator will also enter the operand of the star. As the first instance of the operand is an identity synchro cell the records pass through this instance unaltered, and thus, do not match the exit pattern of the combinator. Consequently, the records enter the next stage of the sequential chain of operands where a new synchro cell will store and eventually merge the two input records. As the process repeats it continuously merges the outputs of the filter branches and outputs records that the post-processing box accepts as inputs.

The complete network is shown in Fig. 3.3. It takes a single record `{img}` as input. The first box `preProc` produces two copies of the input and gives different names to the copies such that the branches of the parallel filter combination processes one record each. The filtered results are merged into one single record by the synchro cell-star combination after which the box `postProc` combines the two filtered images into one result record `{all_edges}`.

```
net img_filter( {img} -> {all_edges})
{
  box preProc({img} -> {img_x} | {img_y});
  box sobel_x({img_x} -> {edges_x});
  box sobel_y({img_y} -> {edges_y});
  box postProc({edges_x, edges_y} -> {all_edges});
}
connect
  preProc
  .. (sobel_x,{img_x} || sobel_y,{img_y})
  .. ([| {edges_x}, {edges_y} |]**{edges_x, edges_y})
  .. postProc;
```

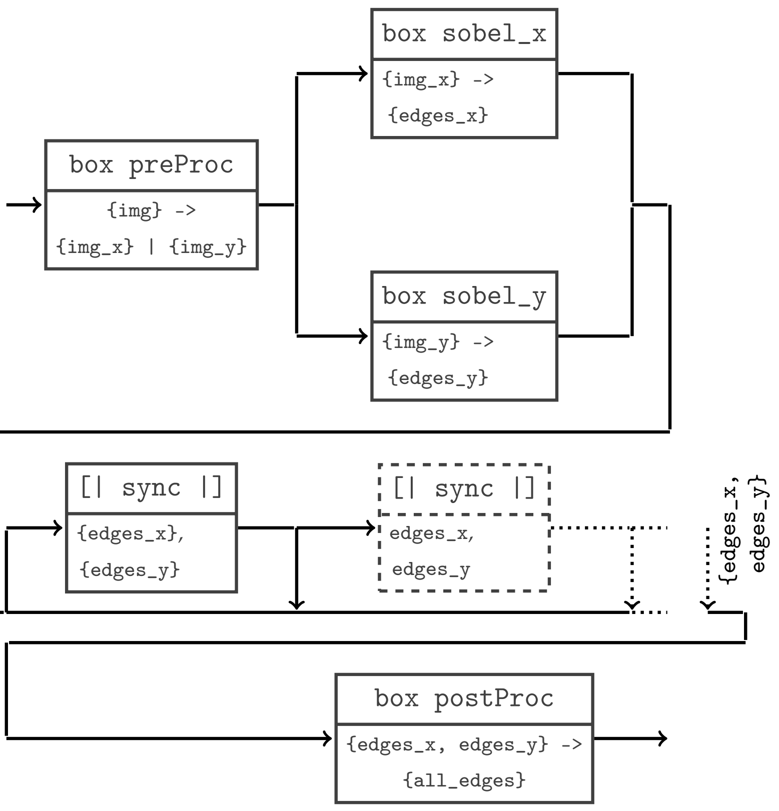


Figure 3.3: The network takes a single input record, processes two copies of the image in parallel branches and then produces a single result record.

4 Semantics of Explicitly Typed S-NET

In the previous chapter we have developed a high-level view on S-NET. By now we have an intuition of how the various constructs of S-NET behave and how these may be used to express desired behaviour. In this chapter we develop a framework that allows us to back up our intuition about language constructs, and in fact entire programs. This is an essential step towards the development of type checking and type inference mechanisms for S-NET: While it is the case that we can check a program's syntactical correctness by testing it against the syntax definition, we so far have no equivalent test for the semantics. We have no formal way of describing what the *meaning* of a program is, and thus, the results of type checking and inference would also be rather meaningless.

In addition to its integral role in typing, providing a formal specification of the semantics of S-NET serves in fact two further purposes. On the one hand it is for us and users in general to be able to know what a program means, what the result for a given input will be and how each construct of the language influences this result. On the other hand it also provides strict guidance for an implementation of the language. A concrete implementation of S-NET can always be tested for correctness by comparing the implementation against the formal specification of the semantics.

Ideally, from a user's perspective, a formal specification is complete and leaves no room for interpretation. In practice, however, this may result in a complex description of the semantics that is impractical to work with. This is even more so the case if a language is intended for use across a wide range of computing platforms. Assumptions that are made in the specification of the semantics may not hold, or incur performance penalties, on some architectures that are targeted.

One approach to evade this problem is to leave certain aspects of the programming system undefined. A famous advocate of this approach is the C programming language. Its standard [Int99] contains cases where the behaviour is intentionally left undefined. This allows for implementations to choose the most appropriate interpretation of the standard, for example depending on architecture specifics.

In a system as S-NET the problem that is solved by such an approach seems not to exist. As we do not express computations on data but merely coordinate such computations dealing with these issues is offloaded to the box implementations and their programming systems. However, we face a very similar situation, if on a different level, that mainly stems from the streaming aspect of S-NET. The computations that are carried out on a stream of data produce a stream of results. In many applications it is not strictly necessary that the order in which the results appear in the output is the same order in which the data was taken from the input stream. One class of applications that expose this property are those that employ fork-join patterns. In a fork-join approach one problem is decomposed into several sub-problems, each of which may be solved independently. The results of all sub-problems are joined up into a final result in the end. For the merging process it is important that all sub-results are available at some point, but the order in which these become available is usually insignificant.

In cases where the order of results is not important we can make use of S-NET's non-deterministic combinators. By applying these combinators instead of their deterministic counterparts we explicitly expose the order-insensitive parts of an application. For these parts we have more freedom in choosing how to evaluate the program in this part as we may do this without employing mechanisms that retain or restore order. Which benefits we can exploit by this, e.g. using more resources in parallel for a task or reducing overheads imposed by order maintaining mechanisms, is determined and specific to a concrete implementation and will also vary with each implementation. With this observation we close the circle to the discussion on undefined behaviour that can for example be found in the specification of C.

Leaving the order in which results are produced by non-deterministic combinators completely unspecified, however, is not an option. We need to put certain restrictions in place to maintain order between results that correspond to the same input record. Non-deterministic combinators may interleave results from different records arbitrarily but reordering of results for the same record is illegal. We will enforce these restrictions in our semantics definition by fully specifying the reordering mechanism but making a concrete decisions dependent on values from an auxiliary decision stream. This decision stream, or oracle stream, provides Boolean values $v \in \{\top, \bot\}$. Every non-deterministic decision that we need to make consumes a value from the oracle stream such that a subsequent decision is based on a new value. In an implementation centric view we may see this oracle stream as our connection to the underlying implementation that provides us with values to influence decisions based on implementation specific characteristics. For purely theoretical considerations we may equally well assume this oracle stream to be the result of an oracle machine on input of an S-NET program and its input data. With this approach we can fully specify the behaviour of all combinators and at the same time we still retain some flexibility on how to steer decisions.

With these preliminary considerations in place we are ready to dive into the details of a semantics framework for S-NET. In order to give a meaning to programs we employ the idea of a derivation tree that evaluates a given program as a series of transitions, each of these turning input data into (intermediate) results. The transitions that we allow are defined in terms of logical implications expressed as rules. For each language construct in S-NET we define such rules to describe what the effects of the construct are. Each rule provides a specific transition which enables us to deduce step-by-step what the output of a program for a given input will be. In more theoretical terms we may also interpret this process as proof construction; given a program, its input and an output we can prove this output to be correct if and only if there exists a composition of transition rules that yield this output for the program and input.

In the remainder of this chapter we first introduce means for stream- and record manipulations and we introduce the necessary formalism to deal with transitions and transition rules. In Section 4.2 we define the transition rules for basic components, i.e. boxes and synchro cells. Combinator rules for deterministic combinators are dealt with in Section 4.3, and Section 4.4 introduces the necessary means to model non-deterministic combinator behaviour.

4.1 Basic Definitions

As a streaming language, S-NET connects the parts of a program via streams. Between individual program parts data is delivered via an inbound stream and results are handed on over an outbound stream. Inside each part various components perform their specific tasks, and they communicate with one another via streams again. The communication that takes place via streams exchanges data elements, i.e. S-NET records. The computations that components carry out are observable as transformations of these records. Combinators may even base their behaviour on the specific structure of records. In order to express all this we introduce formal representations of streams and stream operations as well as means to express the effect that components have if connected to these streams.

4.1.1 Streams and Stream Operations

We represent streams as a finite, ordered list of elements. The following definitions will allow us to create and modify streams. Streams are introduced in Def. 3, creation and modification operations are given in Defs. 3-7.

Definition 3. *A* stream $\vec{s}$ *is either the empty stream* ϵ *(empty sequence) or a sequence of* $n \in \mathbb{N}$ *elements* $s_0, \ldots, s_{n-1}$*. The elements on a stream are indexed by an integer value such that the index of an element determines the position of the element on the stream. For the indices of two adjacent elements* s_i, s_j *on a stream it holds that* $i = j + 1 \ \vee \ j = i + 1$*.*

We will write

$$s_i \in \vec{s}$$

to assert that s_i *is an element of stream* $\vec{s}$*.*

Definition 4. *The construction of a stream is defined by successive concatenation of elements to an existing stream using the* concatenation operator $\triangleright$*. A stream of* n *elements* $s_0, \ldots, s_{n-1}$ *may be written as*

$$\vec{s} = \epsilon \triangleright s_0 \triangleright s_1 \triangleright \ldots \triangleright s_{n-2} \triangleright s_{n-1}$$

The concatenation operator is left-associative, i.e., $s \triangleright e_1 \triangleright e_2 = ((s \triangleright e_1) \triangleright e_2)$*.*

Definition 5. *The elements of multiple streams are arranged into a single stream using the* append operator $+$*. The elements of two streams*

$$\begin{array}{rcl} \vec{s} & = & \epsilon \triangleright s_0 \triangleright \ldots \triangleright s_{n-1} \\ \vec{t} & = & \epsilon \triangleright t_0 \triangleright \ldots \triangleright t_{m-1} \end{array}$$

may be concatenated into a single stream $\vec{u}$:

$$\vec{u} = \vec{s} + \vec{t} = \epsilon \triangleright s_0 \triangleright \ldots \triangleright s_{n-1} \triangleright t_0 \triangleright \ldots \triangleright t_{m-1}$$

As a shorthand notation for the concatenation of the elements of k streams $\vec{s_0}, \ldots, \vec{s_{k-1}}$ into one stream $\vec{t}$ we may also write

$$\vec{t} = \overset{k-1}{\underset{i=0}{+}} \vec{s_i}$$

Definition 6. *Multiple streams are arranged into a stream of streams by successive concatenation using the* stream collection operator $++$ *A stream of streams is denoted by two arrows; given n streams $\vec{s_0}, \ldots, \vec{s_{n-1}}$ the stream of n streams is*

$$\vec{\vec{s}} = \vec{\vec{\epsilon}} ++ \vec{s_0} ++ \vec{s_1} ++ \ldots ++ \vec{s_{n-1}}$$

As a shorthand notation we may use

$$\vec{\vec{s}} = \overset{n-1}{\underset{i=0}{++}} \vec{s_i}$$

Definition 7. *A stream of streams may be* flattened *to a stream of elements by concatenating the elements of the sub-streams. Let*

$$\vec{\vec{s}} = \overset{n-1}{\underset{i=0}{++}} \vec{s_i} = \vec{\vec{\epsilon}} ++ \epsilon \triangleright s_{1_1} \triangleright \ldots \triangleright s_{1_k} ++ \ldots ++ \epsilon \triangleright s_{n-1_1} \triangleright \ldots \triangleright s_{n-1_j}$$

The flattened stream $\vec{s}$ of $\vec{\vec{s}}$ is defined as

$$\vec{s} = flat(\vec{\vec{s}}) = \epsilon \triangleright s_{1_1} \triangleright \ldots \triangleright s_{1_k} \triangleright \ldots \triangleright s_{n_1} \triangleright \ldots \triangleright s_{n_j}$$

4.1.2 Records and Record Transformations

From an S-NET perspective records are sets of label-value pairs. The labels within a record are symbolic handles, i.e. names, for arbitrary data objects. By purely syntactical means we are able to discriminate labels into the three categories of fields, tags and binding tags. Data that is referenced by a field label is an opaque object for all S-NET components, i.e. the actual contents of a data object are only visible to box implementations. Tag and binding tag labels, in contrast, reference plain integer values. These values are readable and writable by all S-NET components and from within box implementations alike.

For the discussion of operational semantics a purely syntactical distinction between the three kinds of labels is sufficient. We do not require a more sophisticated type system at this point; we interpret a record as a set of distinct labels and express operations on these using standard set notation. This approach blurs the line between values and types. We will use the same notation for the representation of a record as a collection of data items and for the representation of the type of a record. Hence, the notation

```
{A, B, <C>}
```

fulfils a dual purpose. On the one hand this may be a specific record that references data by its two field labels `A` and `B` and a concrete integer value by its tag label `C`. On the other hand the notation may also refer to an entire class of records, i.e. it may refer to all records that contain the same labels `A`, `B` and `<C>`. A typical example for the latter interpretation is the specification of patterns for combinators whereas the former interpretation will usually be used when a concrete record is transformed by a box. Where a distinction between these two purposes of the notation is important but not clear from the context we will refer to it as a *record type* when it is used as a representative for a class and a *data record* otherwise.

With this distinction between data records and record types in place we now define operations on records. In order to characterise an application of a box to a record in an abstract way we introduce record transformations in Def. 8. We introduce a subtype relation on record types and an interpretation of pattern matching in Def. 10 and Def. 11.

Definition 8. *Let r, α and β be record types. A* record transformation $RT_\alpha^\beta(r)$ *turns record type r into record type r' with respect to α and β; for distinction we shall use Greek letters to denote the intrinsic source and target types of the transformation and a Latin letter to denote the argument of the transformation.*

A record transformation $r' = RT_\alpha^\beta(r)$ is defined as

$$r' = \beta \cup (r \setminus \alpha)$$

That is, a transformation removes all labels from r that appear in α and adds all labels that appear in β. All labels of r that are not mentioned in α are not removed but kept to model flow-inheritance, i.e. the set of flow-inherited labels is represented by $r \setminus \alpha$.

Definition 9. *Let r be a record type. The set of binding tags within r is denoted $BT(r)$.*

Definition 10. *Let r and r' be record types.*

The record type r is a subtype *of r' if both record types share the same set of binding tags and the set of labels in r' is a subset of the set of labels of r. We use the symbol $\sqsubseteq$ to denote this in the following way:*

$$r \sqsubseteq r' \iff r' \subseteq r \ \wedge \ BT(r) = BT(r')$$

Definition 11. *Let p be a record type and let r be a data record of type τ_r. We define that record r is a* pattern match *for p if τ_r is a subtype of p; to quantify the overlap between a record and a pattern we define*

$$match(r, p) = \begin{cases} |\tau_r \cap p| + 1 & \text{if } \tau_r \sqsubseteq p \\ 0 & \text{otherwise} \end{cases}$$

In order to distinguish a match between a record and an empty pattern from a non-match we add 1 to the number of shared labels.

For all data records r and patterns (record types) p for which $match(r, p) = 0$ we say that r does not match p. For two record types p_1 and p_2 we say that p_1 is a better match for r than p_2 if $match(r, p_1) > match(r, p_2)$.

4.1.3 Transitions

We describe the behaviour of a program P in terms of all transitions that are to be carried out for P if executed on a specific input. In order to do this we need to define what the possible steps are and how each of them may be used to work on input data. Here we introduce the general framework and its constructs that we will be using to reason about possible steps and the relation between inputs and outputs. Definition 12 introduces the general format of the transition relation that we will be working with. In Def. 13 we introduce the notation for a deduction system that we use to define what legal transitions are. The notation that we adopt here to express rules and their conditions is that of natural deduction systems [Gen34].

Definition 12. *The* transition relation $\rightarrow$ *relates an oracle stream, an input stream and an* S-NET *program to a new oracle stream, a new program and an output.*

Let R be the set of all records, S be the set of all streams over R, P be the set of all S-NET *programs, and E the set of all oracle streams. Then, the transition relation defines a set*

$$T \subseteq \{A, B \mid A \in E \times R \times P \ \wedge \ B \in E \times P \times S\}$$

of input and output pairs $(A, B) \in (E \times R \times P) \times (E \times P \times S)$ for which there exists a series of transitions that produces B on input A.

We will write $A \rightarrow B$ if $(A, B) \in T$.

Definition 13. *Let $P_1, \ldots, P_n$ and Q be predicates. A* deduction rule *is a rule of the form*

$$\frac{P_1 \quad P_2 \quad \ldots \quad P_n}{Q}$$

where on top of the horizontal line predicates are listed (preconditions) that, if they hold, imply that the predicate below the line holds as well, i.e.

$$P_1 \wedge P_2 \wedge \ldots \wedge P_n \Rightarrow Q$$

In this case we say that we can deduce or infer Q from $P_1, \ldots, P_n$. Axioms are introduced by defining a rule that has an empty set of preconditions.

A deduction system *defines a set of base deduction rules that allow for the derivation of further rules through the composition of base rules.*

A deduction system is a tool for us to prove that a statement is true. We construct a proof by breaking up a statement into smaller parts. For each part we do the same until we reach a representation for which the deduction system offers a matching rule. From there we derive the desired property for the next bigger compound statement and repeat this process until we have constructed a derivation for the original statement. As an illustration let's assume we are working with the following deduction system to make statements about properties of the length of a stream. The property that we are investigating is the divisibility by two, i.e. we like to know if the number of elements on a stream is odd (`O`) or even (`E`).

$$\frac{}{\epsilon \ \textbf{is}\ \texttt{E}} \quad \frac{}{\epsilon \triangleright r \ \textbf{is}\ \texttt{O}}$$

$$\frac{n \textbf{ is } \mathtt{E} \quad m \textbf{ is } \mathtt{E}}{n+m \textbf{ is } \mathtt{E}} \quad \frac{n \textbf{ is } \mathtt{O} \quad m \textbf{ is } \mathtt{E}}{n+m \textbf{ is } \mathtt{O}} \quad \frac{n \textbf{ is } \mathtt{O} \quad m \textbf{ is } \mathtt{O}}{n+m \textbf{ is } \mathtt{E}}$$

Equipped with the five base rules of the system above we may now set out to show that the stream

$$\vec{s} = \epsilon \triangleright r_0 \triangleright r_1 \triangleright r_2 \triangleright r_3 \triangleright r_4$$

has an odd number of elements, i.e. $\vec{s}$ **is** $\mathtt{O}$. We need to compose the base rules in such a way that they form a derivation for our statement $\vec{s}$ **is** $\mathtt{O}$.

$$\frac{\dfrac{\dfrac{\dfrac{\epsilon \triangleright r_0 \textbf{ is } \mathtt{O} \quad \epsilon \triangleright r_1 \textbf{ is } \mathtt{O}}{\epsilon \triangleright r_0 + \epsilon \triangleright r_1 \textbf{ is } \mathtt{E}}}{\epsilon \triangleright r_0 \triangleright r_1 \textbf{ is } \mathtt{E}} \qquad \dfrac{\dfrac{\epsilon \triangleright r_2 \textbf{ is } \mathtt{O} \quad \dfrac{\epsilon \triangleright r_3 \textbf{ is } \mathtt{O} \quad \epsilon \triangleright r_4 \textbf{ is } \mathtt{O}}{\epsilon \triangleright r_3 + \epsilon \triangleright r_4 \textbf{ is } \mathtt{E}}}{\epsilon \triangleright r_2 + \epsilon \triangleright r_3 \triangleright r_4 \textbf{ is } \mathtt{O}}}{\epsilon \triangleright r_2 \triangleright r_3 \triangleright r_4 \textbf{ is } \mathtt{O}}}{\epsilon \triangleright r_0 \triangleright r_1 \quad + \quad \epsilon \triangleright r_2 \triangleright r_3 \triangleright r_4 \textbf{ is } \mathtt{O}}}{\vec{s} = \epsilon \triangleright r_0 \triangleright r_1 \triangleright r_2 \triangleright r_3 \triangleright r_4 \textbf{ is } \mathtt{O}}$$

This derivation tree is our proof that the length of $\vec{s}$ is odd. In addition to giving an idea on how to work with deduction rules, what the example also shows is that a deduction system provides us with rules from which we may construct proofs, but it does not provide us with an application strategy for these rules. Generally there are many derivation trees that represent a proof for the same property. In the above example we could have chosen a different way of splitting up the streams; this would have yielded a very different derivation tree, but it would have proven the same property.

4.2 Definition of Components

In this section we devise formal definitions for the components of the S-NET system.

4.2.1 The Box

Any S-NET program that is able to carry out computations on fields contains at least one box. As such, boxes are essential ingredients in any real-world application. Yet, from an S-NET perspective we know very little about what a box is doing between the point where it received data to compute on and the point where the box produces its result. In order to capture its operational behaviour we have to make a decision on how to model this apparent gap in our framework. There are approaches that embed foreign language semantics into encompassing frameworks [MF07]. However, if we were to follow the same approach we have to rely on the box language semantics to be available and formalised in a way that allows for sensible embedding. As S-NET intentionally leaves the choice of box language open, this approach may limit the applicability of our framework. We instead develop a semantic model of boxes that captures the observable behaviour in a record-centric view and abstracts from the actual computation on data that is carried out within the box.

The observable behaviour of a box comprises the consumption of one record from the input stream and the production of a variable amount of records that are emitted on the output stream. Of course, the amount of records that is produced is generally influenced by the data that was received by the box; if we are dealing with a box of a multi-variant output type, i.e. a box with multiple right-hand sides, then the type of records is also dependent on the input data, and so is the order in which the records of different types appear on the output stream. The variety of possible outputs requires the semantic description of a box to be flexible enough to cater for these cases. At the same time we need to put restrictions into place to define a framework that allows for precise reasoning. We achieve this by leaving the concrete form of produced output unspecified in terms of amount and order of records that it contains. We give, however, guarantees on the individual elements of the result. This enables instantiating a generic box description for a specific result as long as the constraints on the produced output hold. We formalise this approach in Transition Rule 1.

Transition Rule 1. *The output of a box may be instantiated to an arbitrary but finite-length result stream $\vec{s}$ as long as all elements of the result stream adhere to the right-hand side of the box signature.*

$$\text{Box} \quad : \quad \frac{r \sqsubseteq \tau \qquad \forall s_i \in \vec{s}\ \exists \gamma \in T : s_i = RT_\tau^\gamma(r)}{(E,\ r,\ \texttt{box}\ BoxId\texttt{(}\tau\ \texttt{->}\ T\texttt{)}) \rightarrow (E,\ \texttt{box}\ BoxId\texttt{(}\tau\ \texttt{->}\ T\texttt{)},\ \vec{s})}$$

The box name replaces $BoxId$ in a concrete application of this rule.

4.2.2 The Synchro Cell

The synchro cell provides a means to merge two records of different, non-overlapping types. In order to specify which types of records are to be merged by the cell two pattern are specified. These two patterns directly influence the behaviour of the cell. We may think of a synchro cell as a component that conceptually has two input streams, one for each pattern. The records that the first stream carries match the first pattern, the records on the second stream match the second pattern. These two streams are mapped onto three output streams. Two of the output streams resemble exactly the two input streams. The third output stream carries the result of the merge process that the synchro cell performs. The first record on the first input stream and the first record on the second input stream are merged. It is these two records that make up the result that is put on the third output stream. All later records on both input streams are forwarded to their respective output streams.

In order to implement the switch in behaviour after the first records have been read the synchro cell keeps an internal state. The state allows us to model complex behaviour in one single component; this feature is unique to synchro cells in S-Net. Owing to this, the formal definition of this component is more elaborate than for all other (stateless) components. As a starting point we model the behaviour of a synchro cell as a deterministic finite-state automaton (DFA). In Fig. 4.1 two of such

DFAs are shown. At the top of the figure we use a Mealy automaton [Mea55]. The automaton models the behaviour of a synchro cell in a compact representation with readability in mind. At the bottom of that figure the behaviour is modelled as a Moore automaton [Moo56]. This representation uses precisely one output for each state which allows for a direct translation of the automaton into a set of transition rules. Both models capture the behaviour of a synchro cell over pattern p_1 and p_2 on receiving record r. The state machine uses the subtype relation $\sqsubseteq$ between the input record r and the two patterns p_1, p_2 as input for the state transition. In terms of streams we may say that if $r \sqsubseteq p_1$ the record was received from the first input stream, if $r \sqsubseteq p_2$ it was received from the second stream. However, the analogy to multiple input streams is only a conceptual one as a record may be a subtype of both patterns at the same time. The output that is produced in response to an input is shown in the lower half of the circle of the destination state within the Moore automaton and as second component on state transition within the Mealy model. States that produce no output because a record was put into storage show ϵ as their output. The starting state of both automata is q_0; the final state is id. In state id all records are forwarded to the output without modification. For perspicuity of the graphical representation of the automata we do not show the cases of inputs that neither match p_1 nor p_2. The automata handles these case by using a second final state Err that does not produce any output. Transitions from all states to Err for the case $\neg(r \sqsubseteq p_1 \vee r \sqsubseteq p_2)$ are required for this as well as a transition from Err to Err for any input.

As mentioned above, the Moore DFA model assists us in devising a formal description of synchro cells within the operational semantics framework. The states and transitions may be almost directly mapped into the rules which are shown in Transition Rule 2. In order to indicate the state of the synchro cell we use an index on the right to the syntactical description to indicate which pattern has been matched before. The index p_1 (first pattern was matched) corresponds to the states q_1, q_1', q_1'' in Fig.4.1(top) and state q_1 in Fig. 4.1(bottom); the same holds for p_2 respectively.

Transition Rule 2. *The synchro cell requires more than one transition rule to describe the full behaviour due to the internal state of the component that needs to be modelled.*

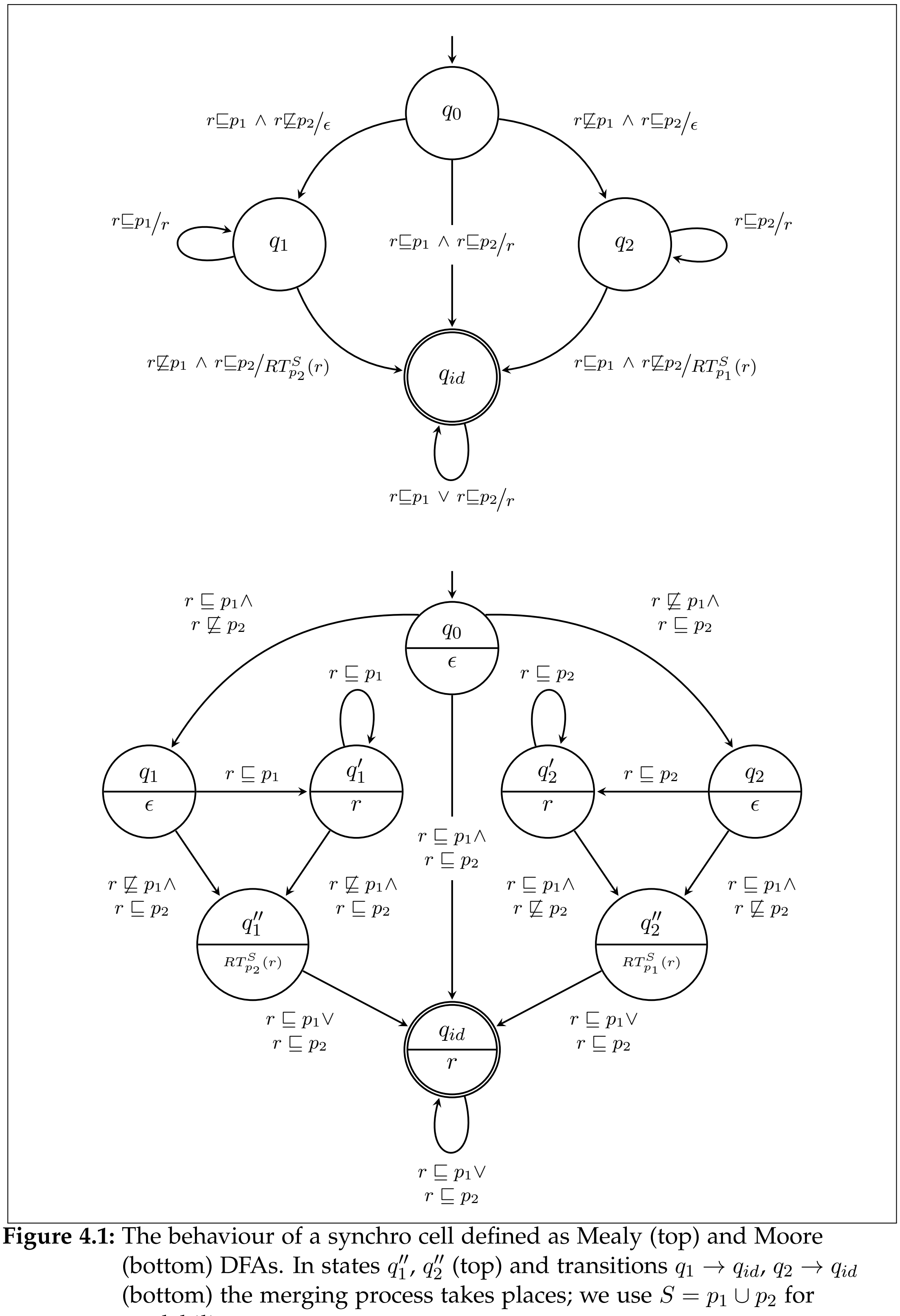

q_0
$r \sqsubseteq p_1 \wedge r \not\sqsubseteq p_2 / \epsilon$
$r \not\sqsubseteq p_1 \wedge r \sqsubseteq p_2 / \epsilon$
$r \sqsubseteq p_1 / r$
q_1
$r \sqsubseteq p_1 \wedge r \sqsubseteq p_2 / r$
q_2
$r \sqsubseteq p_2 / r$
$r \not\sqsubseteq p_1 \wedge r \sqsubseteq p_2 / RT^S_{p_2}(r)$
q_{id}
$r \sqsubseteq p_1 \wedge r \not\sqsubseteq p_2 / RT^S_{p_1}(r)$
$r \sqsubseteq p_1 \vee r \sqsubseteq p_2 / r$
$r \sqsubseteq p_1 \wedge$ $r \not\sqsubseteq p_2$
q_0
ϵ
$r \not\sqsubseteq p_1 \wedge$ $r \sqsubseteq p_2$
$r \sqsubseteq p_1$
$r \sqsubseteq p_2$
q_1
ϵ
$r \sqsubseteq p_1$
q_1'
r
q_2'
r
$r \sqsubseteq p_2$
q_2
ϵ
$r \sqsubseteq p_1 \wedge$ $r \sqsubseteq p_2$
$r \not\sqsubseteq p_1 \wedge$ $r \sqsubseteq p_2$
$r \not\sqsubseteq p_1 \wedge$ $r \sqsubseteq p_2$
$r \sqsubseteq p_1 \wedge$ $r \not\sqsubseteq p_2$
$r \sqsubseteq p_1 \wedge$ $r \not\sqsubseteq p_2$
q_1''
$RT^S_{p_2}(r)$
q_2''
$RT^S_{p_1}(r)$
$r \sqsubseteq p_1 \vee$ $r \sqsubseteq p_2$
q_{id}
r
$r \sqsubseteq p_1 \vee$ $r \sqsubseteq p_2$
$r \sqsubseteq p_1 \vee$ $r \sqsubseteq p_2$

Figure 4.1: The behaviour of a synchro cell defined as Mealy (top) and Moore (bottom) DFAs. In states q_1'', q_2'' (top) and transitions $q_1 \rightarrow q_{id}$, $q_2 \rightarrow q_{id}$ (bottom) the merging process takes places; we use $S = p_1 \cup p_2$ for readability.

$$\text{SY}\textsc{match}\text{L} \quad : \quad \frac{r \sqsubseteq p_1 \wedge r \not\sqsubseteq p_2}{(E,\ r,\ \llbracket p_1, p_2 \rrbracket) \to (E,\ \llbracket p_1, p_2 \rrbracket_{p_1},\ \epsilon)}$$

$$\text{SY}\textsc{match}\text{R} \quad : \quad \frac{r \not\sqsubseteq p_1 \wedge r \sqsubseteq p_2}{(E,\ r,\ \llbracket p_1, p_2 \rrbracket) \to (E,\ \llbracket p_1, p_2 \rrbracket_{p_2},\ \epsilon)}$$

$$\text{SY}\textsc{ofl}\text{L} \quad : \quad \frac{r \sqsubseteq p_1}{(E,\ r,\ \llbracket p_1, p_2 \rrbracket_{p_1}) \to (E,\ \llbracket p_1, p_2 \rrbracket_{p_1},\ \epsilon \triangleright r)}$$

$$\text{SY}\textsc{ofl}\text{R} \quad : \quad \frac{r \sqsubseteq p_2}{(E,\ r,\ \llbracket p_1, p_2 \rrbracket_{p_2}) \to (E,\ \llbracket p_1, p_2 \rrbracket_{p_2},\ \epsilon \triangleright r)}$$

$$\text{SY}\textsc{merge}\text{L} \quad : \quad \frac{r \not\sqsubseteq p_1 \wedge r \sqsubseteq p_2}{(E,\ r,\ \llbracket p_1, p_2 \rrbracket_{p_1}) \to (E,\ Id,\ \epsilon \triangleright RT_{p_2}^{p_1 \cup p_2}(r))}$$

$$\text{SY}\textsc{merge}\text{R} \quad : \quad \frac{r \sqsubseteq p_1 \wedge r \not\sqsubseteq p_2}{(E,\ r,\ \llbracket p_1, p_2 \rrbracket_{p_2}) \to (E,\ Id,\ \epsilon \triangleright RT_{p_1}^{p_1 \cup p_2}(r))}$$

$$\text{SY}\textsc{match}\text{LR} \quad : \quad \frac{r \sqsubseteq p_1 \wedge r \sqsubseteq p_2}{(E,\ r,\ \llbracket p_1, p_2 \rrbracket) \to (E,\ Id,\ \epsilon \triangleright r)}$$

$$\text{SY}\textsc{id} \quad : \quad \frac{r \sqsubseteq p_1 \vee r \sqsubseteq p_2}{(E,\ r,\ Id) \to (E,\ Id,\ \epsilon \triangleright r)}$$

The Id component of rule SY<span>ID</span> *is an immutable component that forwards any input to the output without alteration. No further state changes within the synchro cell occur.*

4.2.3 Input Streams

The rules that we have introduced to model boxes and synchro cells turn a single element stream, i.e. one input record, into a stream of output records. This asymmetry seems unnatural as we intuitively expect any component to turn a stream of inputs into a stream of outputs. This mismatch is rectified by the MAP rule in Transition Rule 3. This rule allows us to work with a stream of inputs by applying component rules to each of the records on the input stream and then concatenating the resulting output streams to a final result.

Transition Rule 3. *The* MAP *rule introduces a $\to^\star$ relation that defines the application of component rules to streams by repeated application to single elements.* MAP *is defined as a generic rule that may be used in conjunction with any component. As synchro cells may change their internal state depending on the input and their current state, these potential state changes of components and compounds are captured by associating an index with each program under consideration. This index serves as a unique identifier for this program and it is incremented with every record that is transformed. A program that did not have a counter associated with it previously is assumed to have a counter value of* 0.

The MAP *rule constructs streams of streams; in order to demote a stream of streams to a stream of elements we may use the $flat$ function of Def. 7.*

$$\text{MAP} \quad : \quad \frac{\vec{r} = \epsilon \triangleright r_0 \triangleright \ldots \triangleright r_{n-1} \qquad \forall_{j=0}^{n-1} : (E_{i+j},\ r_j,\ C_{i+j}) \to (E_{i+j+1},\ C_{i+j+1},\ \vec{s}_j) \qquad \vec{s} = flat(+\!+_{j=0}^{n-1}\ \vec{s}_j)}{(E_i,\ \vec{r},\ C_i) \to^{\star} (E_{i+n},\ C_{i+n},\ \vec{s})}$$

For completeness we shall also note that an empty stream may always be processed without causing any effect. This is captured by Transition Rule 4.

Transition Rule 4. *The* EMPTY *rule allows for a effect-free transition on empty streams.*

$$\text{EMPTY} \quad : \quad \frac{}{(E,\ \epsilon,\ C_i) \to (E,\ C_i,\ \epsilon)}$$

4.3 Definition of Combinators

In this section we devise formal definitions for the combinators of the S-NET system.

4.3.1 Sequential Composition

A sequential composition in S-NET expresses a dependency between two computations. The relationship between the two operands of a sequential combinator is such that the first operand produces the input for the second operand. Because of this dependency the computation that this compound network represents has to be broken down into two stages and hence we may think of a sequential composition as a computational pipeline. Formally, the two stages of computation are represented by the two pre-conditions of the rule for this combinator, as shown in Transition Rule 5.

Transition Rule 5. *The result of a sequential combination is determined by the composition of computations that the left operand L and the right operand R of the combinator represent.*

$$\text{SEQ} \quad : \quad \frac{(E,\ r,\ L_i) \to (E'\ ,L_{j\geq i},\ \vec{s}\ ) \qquad (E',\ \vec{s},\ R_k) \to^{\star} (E'',\ R_{l\geq k},\ \vec{t}\ )}{(E,\ r,\ L_i\ ..\ R_k) \to (E'',\ L_j\ ..\ R_l,\ \vec{t}\ )}$$

4.3.2 Choice Composition

With the choice combinator we have a means to apply different computations to a stream of records. This introduces two alternative strands of computations each of which is implemented by one of the combinator's operands. In order to determine which operand is used to perform a computation on a record from the input stream the combinator queries the two patterns that are associated with it. If the first pattern is matched by a record, then the left operand is used for the computation, and consequently, if the right pattern is matched, the right operand is applied.

However, if the decision is made based by pattern matching alone it has a severe shortcoming for cases in which both patterns are matched by a record at the same time. In this case it is always the left operand that wins the decision. Although this is a possible strategy, a more sophisticated behaviour is usually desirable. As a first refinement a best-match strategy is a solution for a sub-set of records that fall in the category of records matching both patterns. With this in place the operand whose pattern has the greater overlap of labels with an inbound record is selected as destination. However, this mechanism remains inconclusive for records that overlap with the two pattern by the same amount of labels.

Possible strategies for dispatching records that match both operands equally well may range from statically picking the first or second branch to elaborate scheduling mechanisms. In order to not predetermine a strategy and to leave a degree of freedom of choice to a concrete implementation of S-NET we externalise the decision mechanism. This is achieved by parametrising the specification of the operational behaviour of the choice combinator over an oracle stream and a decision function that bases its result on the oracle stream. This is reflected by incorporating a decision function f, see Def. 14, into the transition rules. The motivation behind this approach is that an implementation may choose to take information of the executing environment into account to steer a dispatch decision. Although the concrete form of these information are intentionally left outside the scope of this framework we may envision resource aware implementations that take the underlying, executing machinery into account. This may for example include knowledge about the utilisation of available computing cores, the amount of available memory or the current saturation of network connections.

Before defining the transition rules for the combinator, we first characterise decision functions in Def. 14 and introduce a decision in Def. 15 that takes such a decision functions into account when necessary. The combinator rules are presented in Transition Rule 6.

Definition 14. *A* decision function f *takes an oracle stream E as argument and defines a result pair (x, E') where $x \in \{L, R\}$ and E' is a new oracle stream. This is achieved by using the first element of the oracle stream as a decision for the value of x and the remaining oracle stream as new oracle stream.*

$$f((\epsilon \triangleright e) + E) = \begin{cases} (L, E) & \textit{if } e = \top \\ (R, E) & \textit{otherwise} \end{cases}$$

Definition 15. *The decision* `pick` *determines which of the two operands of a choice combination is to be used for processing an input record. The result of the decision is either L if the first operand is to be taken or R otherwise.*

Let r be a record, E an oracle stream, f a decision function and p_1, p_2 the two patterns of a choice combinator. The decision is defined by the function `pick` *as follows:*

$$\texttt{pick}(r, E, f, p_1, p_2) = \begin{cases} (L, E) & \textit{if } match(r, p_1) > match(r, p_2) \\ (R, E) & \textit{if } match(r, p_1) < match(r, p_2) \\ f(E) & \textit{otherwise} \end{cases}$$

Transition Rule 6. *The definition of the choice combinator is split up into two parts; the first rule covers the case for a record that is processed by the first operand, the second rule covers the symmetric case for processing by the second operand.*

$$\textsc{ChoiceL}_f \quad : \quad \frac{\begin{array}{c}\texttt{pick}(r, E, f, p_1, p_2) = (L, E') \\ (E',\ r,\ M_i) \rightarrow (E'',\ M_{j \geq i},\ \vec{s})\end{array}}{(E,\ r,\ M_i\ \texttt{||}\ N_k) \rightarrow (E'',\ M_j\ \texttt{||}\ N_k,\ \vec{s})}$$

$$\textsc{ChoiceR}_f \quad : \quad \frac{\begin{array}{c}\texttt{pick}(r, E, f, p_1, p_2) = (R, E') \\ (E',\ r,\ N_k) \rightarrow (E'',\ N_{l \geq k},\ \vec{s})\end{array}}{(E,\ r,\ M_i\ \texttt{||}\ N_k) \rightarrow (E'',\ M_i\ \texttt{||}\ N_l,\ \vec{s})}$$

4.3.3 Parallel Replication

The parallel replication combinator in S-NET may be regarded as an infinite fork-join construct over the combinator's operand. The combinator establishes a split and a merge point between which the forked instances are connected in parallel. For identification purposes the forked instances of the operand are uniquely tagged with an integer value.

In addition to the operand for computation the combinator also requires a tag name as a second operand. This tag name establishes a link between records on the input stream and the parallel instances. All records on the input stream are assumed to contain a tag of the name that was given to the combinator. The combinator examines the value of this tag in a record and uses this value to select the matching instance of its replicated operand. The match is established via the tag value and the unique integer value that is associated with each operand instance.

The infinite amount of operand instances is not reflected in the syntax of this combinator and hence we deviate from the original syntax of the language in Transition Rule 7 that defines the combinator's behaviour. We make the operands explicit by extending `M!!<k>` to

$$\{\texttt{M}^i \mid i \in \mathbb{Z}\}\texttt{!!<k>}$$

where i represents the identification value of each instance of operand `M`.

Transition Rule 7. *We use $val(r,$ `<k>`$)$ to refer to the integer value of tag* `<k>` *in data record r.*

$$\text{Split} \quad : \quad \frac{\begin{array}{c} v = val(r,\, \texttt{<k>}) \\ M^v_{j_v} \in \left\{ M^i_{j_i} \mid i \in \mathbb{Z} \right\} \\ (E,\ r,\ M^v_{j_v}) \to (E', M^v_{j'_v \geq j_v}, \vec{s}) \end{array}}{\begin{array}{l} (E,\ r,\ \left\{ M^i_{j_i} \mid i \in \mathbb{Z} \right\} \texttt{!!<k>}) \\ \quad \to (E',\ ((\left\{ M^i_{j_i} \mid i \in \mathbb{Z} \right\} \setminus \{M^v_{j_v}\}) \cup \{M^v_{j'_v}\})\texttt{!!<k>},\ \vec{s}) \end{array}}$$

Every record that is processed by this combinator is strictly confined to one instance of the operand. The effect that this has on the overall set of instance is therefore also confined to one element of the set. This is modelled in Transition Rule 7 by

$$(\{M^i_{j_i} \mid i \in \mathbb{Z}\} \setminus \{M^v_{j_v}\}) \cup \{M^v_{j'_v}\}$$

which replaces the previous version of the operand that was picked for processing by a new instance. The new version is the original instance plus the potential state changes that have occurred during processing.

4.3.4 Sequential Replication

The sequential replication combinator introduces a means to express repeated application of an operand. Similar to a `while` loop in imperative languages the combinator repeatedly applies its operand until an exit predicate is true. The exit predicate of the combinator is the subtype relation $\sqsubseteq$ between a result and the annotated pattern.

In order to realise repeated application of its operand in a feed-forward fashion the combinator employs an unbounded sequential chain of operand instances. After an operand instance produced a result, the result is sent to the next instance down the sequential chain. This process repeats until a result matches the annotated pattern of the combinator. In this case the result is taken out of the chain as processing for this record has finished.

The feed-forward semantics of the combinator is made prominently visible in the definition of the combinator's behaviour in Transition Rule 8 which defines he behaviour of the combinator terms of a recursive application of sequential and parallel combinators.

Transition Rule 8. *The behaviour of the combinator is recursively defined to establish an unbounded sequential chain of operands. Records that match the annotated pattern p will exit, records that to not match stay within the chain.*

$$\text{Star} \quad : \quad \frac{(E,\ r,\ (M\ \texttt{..}\ M\texttt{**}p)\ \texttt{,\{\}||\ Id,}p\ ) \to^{\star} (E',\ N,\ \vec{s})}{(E,\ r,\ M\texttt{**}p) \to (E',\ N,\ \vec{s})}$$

The sequential combination between the operand and the recursive application of the serial replication combinator establishes an unbounded sequential chain of operand instances. The choice combinator on the outside forces records that match the right branch to exit the sequential chain. We achieve this behaviour by carefully choosing the two patterns of the choice combinator. We assign an empty pattern to the recursively defined processing chain on the left of the combinator. As the right operand of the choice we use the identity as done in the SYid rule of the synchro cell definition. As a pattern for the branch we use the pattern p of the combinator. This way any record that is a subtype of p will be picked by the choice combinator for the right (exit) branch as the other pattern is empty; this ensures that the branch for further applications of the operand does never have a more specific match (see Transition Rule 6) than the exit branch.

4.4 Non-Deterministic Combinators

All combinators that we have introduced so far give strong guarantees on the order of results. As combinators are defined on single records, the result of a combinator application always results in an output stream that corresponds directly to the single element input. An application of a combinator to a stream of inputs results in a concatenation of these individual result. The order of the individual results again directly corresponds to the order of the records in the input. This is ensured by the definition of the Map rule in Transition Rule 3.

The non-deterministic variants of the combinators give us a means to weaken these guarantees on order. Where the standard combinators define a one-to-one mapping from a given input and a network to one output, the non-deterministic combinators introduce a one-to-many relationship. One specific input to a network may now correspond to an entire set of possible outputs. The specification of the formal semantics of the non-deterministic combinators gives us a characterisation of the set of possible outputs. A concrete implementation of S-Net has to ensure that the result that is produced by a network containing non-deterministic combinators is an element of the set of possible outputs.

The set of possible outputs for a non-deterministic combinator is defined by the result of its deterministic counterpart plus restricted permutations of this result. We define the kind of allowed permutations in Def. 16.

Definition 16. *Let $\vec{\vec{s}}$ be a finite stream of streams and let $\vec{s_0}, \ldots, \vec{s_{n-1}}$ for $n \in \mathbb{N}$ be streams such that*

$$\vec{\vec{s}} = \mathop{++}_{i=0}^{n-1} \vec{s_i}.$$

Then, for all $i \in \{0, \ldots, n-1\}$ a stream $\vec{s_i}$ is either ϵ or it holds that

$$\forall i \in \{0, \ldots, n-1\} \exists k \in \mathbb{N}_0 : \vec{s_i} = \epsilon \triangleright s_{i_0} \triangleright \ldots \triangleright s_{i_k}$$

where we assume that $i_0 < \ldots < i_k$. A permutation σ applied to $\vec{\vec{s}} = ++_{i=0}^{n-1} \vec{s_i}$ is a sub-stream order preserving permutation *if for the result $\vec{\vec{q}} = ++_{i=0}^{n-1} q_i = \sigma(\vec{\vec{s}})$ it holds that* *for* $\vec{q} = flat(\vec{\vec{q}})$

$$\begin{array}{rrcl} \forall i \in \{0,\ldots,n-1\} : \forall s_{i_k}, s_{i_l} \in \vec{s_i} : \exists q_r, q_s \in \vec{q} : & s_{i_k} & = & q_r \\ \wedge & s_{i_l} & = & q_s \\ \wedge & i_k & < & i_l \\ & & \Rightarrow & r < s \end{array}$$

The above definition allows arbitrary interleaving of elements from different sub-streams but it does not allow the order of elements within a sub-stream to be altered. As an illustration let's consider a stream of integer values

$$\vec{s} = \epsilon \triangleright 1 \triangleright 2 \triangleright 3 \triangleright 4 \triangleright 5 \triangleright 6 \triangleright 7 \triangleright 8 \triangleright 9$$

which may be divided up into three sub-streams

$$\begin{array}{ccccc} \vec{s_0} & = & \epsilon \triangleright s_{0_0} \triangleright s_{0_1} \triangleright s_{0_2} & = & \epsilon \triangleright 1 \triangleright 2 \triangleright 3 \\ \vec{s_1} & = & \epsilon \triangleright s_{1_0} \triangleright s_{1_1} \triangleright s_{1_2} & = & \epsilon \triangleright 4 \triangleright 5 \triangleright 6 \\ \vec{s_2} & = & \epsilon \triangleright s_{2_0} \triangleright s_{2_1} \triangleright s_{2_2} & = & \epsilon \triangleright 7 \triangleright 8 \triangleright 9 \end{array}$$

such that $\vec{\vec{s}} = \vec{s_0} +\!+ \vec{s_1} +\!+ \vec{s_2}$. Sub-stream order preserving permutations of $\vec{\vec{s}}$ are for example

$$flat(\sigma(\vec{\vec{s}})) = \epsilon \triangleright 7 \triangleright 8 \triangleright 9 \triangleright 4 \triangleright 5 \triangleright 6 \triangleright 1 \triangleright 2 \triangleright 3$$

$$flat(\sigma'(\vec{\vec{s}})) = \epsilon \triangleright 1 \triangleright 4 \triangleright 7 \triangleright 2 \triangleright 5 \triangleright 8 \triangleright 3 \triangleright 6 \triangleright 9$$

as here the elements of each sub-stream appear in the same order as in the original stream. If we remove the elements of two of the three sub-streams then we are left with the remaining sub-stream in its original form. For

$$flat(\sigma''(\vec{\vec{s}})) = \epsilon \triangleright 1 \triangleright 3 \triangleright 2 \triangleright 4 \triangleright 5 \triangleright 6 \triangleright 7 \triangleright 8 \triangleright 9$$

this is not the case and hence σ'' is not a sub-stream order preserving permutation of $\vec{\vec{s}}$.

With the definition of sub-stream order preserving permutations in place we are now in a position to define non-deterministic combinators in terms of such permutations and a refined MAP rule. We are able to confine the effects of non-strict ordering to the map rule as long as we are only considering a single record as input to non-deterministic combinators. In this case the resulting stream contains exactly one sub-stream that directly corresponds to one input, and hence, this one sub-stream cannot be reordered. As such the definition of non-deterministic combinators are mapped onto their deterministic counterparts as shown in Transition Rule 9.

Transition Rule 9. *The point-wise definitions of non-deterministic combinators are based on the definitions of the deterministic variants of the combinators.*

$$\textsc{NSplit} \quad : \quad \frac{(E,\ r,\ \left\{M^i_{j_i} \mid i \in \mathbb{Z}\right\}\ \texttt{!!<k>}) \to (E',\ M',\ \vec{s})}{(E,\ r,\ \left\{M^i_{j_i} \mid i \in \mathbb{Z}\right\}\ \texttt{!<k>}) \to (E',\ M',\ \vec{s})}$$

$$\textsc{NStar} \quad : \quad \frac{(E,\ r,\ M\texttt{**p}) \to (E',\ M',\ \vec{s})}{(E,\ r,\ M\texttt{*p}) \to (E',\ M',\ \vec{s})}$$

$$\textsc{NChoice} \quad : \quad \frac{(E,\ r,\ M\texttt{||}N) \to (E',\ M',\ \vec{s})}{(E,\ r,\ M\texttt{|}N) \to (E',\ M',\ \vec{s})}$$

As discussed above, reordering only has an effect if we are considering a stream of inputs by means of the MAP rule. The stream of inputs is turned into a stream of outputs which is made up from one (potentially empty) sub-stream of results for each input. To allow for reordering when non-deterministic combinators are involved we define an additional map rule in Transition Rule 10. This new rule makes use of a function Σ in order to apply a permutation to the result. Similar to the approach that we have taken for the choice combinator in Sect. 4.3.2 we parametrise the rule over this function. We assume that the function Σ uses the oracle stream and a stream of sub-streams as input and produces a stream of sub-streams of reordered elements that is a sub-stream order preserving permutation of its input. Transition Rule 10 defines the non-deterministic map rule.

Transition Rule 10. *The* NMAP *rule is parametrised over a function Σ that applies a sub-stream order preserving permutation to a stream of sub-streams.*

$$\textsc{NMap}_{\Sigma} \quad : \quad \frac{\begin{array}{c} M \in \{P\texttt{|}Q,\ P\texttt{*p},\ \left\{P^i_{j_i} \mid i \in \mathbb{Z}\right\}\ \texttt{!<k>}\} \\ (E,\ \vec{r}, M) \to^{\star} (E', M', flat(++_{i=0}^{n-1}(\vec{s_i}))) \\ (E'', ++_{i=0}^{n-1}\ \vec{q_i}) = \Sigma(E', ++_{i=0}^{n-1}(\vec{s_i})) \\ \vec{\sigma} = flat(++_{i=0}^{n-1}(\vec{q_i})) \end{array}}{(E,\ \vec{r},\ M) \to^{N\star} (E'', M', \vec{\sigma})}$$

We may accept arbitrary choices of Σ as long as the function respects the demanded properties. For $\Sigma = Id$, i.e. the identity function, the non-deterministic combinators are identical to their deterministic counterparts. For completeness we propose a possible choice of Σ in Def. 17.

Definition 17. *Let $\vec{\vec{s}} = ++_{i=0}^{n-1}\ s_i$ be a stream of sub-streams and let*

$$\vec{f} = \epsilon \triangleright f_0 \triangleright \ldots \triangleright f_{l-1} = flat(\vec{\vec{s}})$$

Furthermore, let E be an oracle stream and let $\vec{q_0} = \ldots = \vec{q_{n-1}} = \epsilon$.

The algorithm to compute a sub-stream order preserving permutation of $\vec{\vec{s}}$ performs the following steps:

1. *read* n *values* $e_0, \ldots, e_{n-1}$ *from* $E = \epsilon \triangleright e_0 \triangleright \ldots \triangleright e_{n-1} + E'$
2. *let* $m = |\{e_i \mid i \in \{0, \ldots, n-1\} \wedge e_i = \top\}|$
3. *read the first element of* $\vec{f} = \epsilon \triangleright f + \vec{f'}$
4. *append* f *to* q_m *such that* $q'_m = q_m \triangleright f$
5. *repeat all steps with* $E = E', \vec{f} = \vec{f'}, \vec{q_m} = \vec{q'_m}$ *for the remaining elements on* $\vec{f}$

The result $\Sigma(E, \vec{\vec{s}}) = +\!+_{i=0}^{n-1} q_i$ *is a sub-stream order preserving permutation of* $\vec{\vec{s}}$

5 Foundations of the Type System

Type systems for programming languages are commonly put into place as a means to prove properties of a program. Such properties are usually not concerned with the actual computations and algorithms that are expressed by a program but more with the question if a program is composed in a meaningful way. The term "meaningful", or type safe, may range from giving guarantees about the absence of runtime errors and non-termination to more modest properties such as consistency between annotated types and actual values. Whatever the goals of type systems in particular scenarios are, they all establish knowledge about how values within a program are used. Ideally the type system is able to guarantee that all values are only used in contexts in which they make sense. For example, in many languages a nonsensical use of values within a computation is the addition of an integer value to string as in `3 + "four"`. A system that employs type checking to rule out unsafe programs would not attempt to carry out a computation like the one above. In the absence of type safety a program that contains nonsensical computations may abort its execution with a runtime error in the best case or it may produce random or wrong results in the worst case.

Type checking approaches coarsely divide up into two groups. The two groups are static type checking approaches on the one hand and dynamic approaches on the other hand. Static type checking attempts to establish guarantees at compile time whereas dynamic approaches delay much of the type checking until program runtime. The advantage of static type checking is that a program is proven to be type safe at compile time. Computations on incompatible values are detected before runtime and cannot cause errors at runtime. The disadvantage of static type checking is that it may be too conservative; if a compiler fails to find a proof of safety for all expressions within a program, it will reject the program as unsafe. However, such a program may still be executed correctly and only produce correct results if it is used in a certain way. If the assumptions that are required to come to this conclusion are not available to the compiler, the compiler rejects a useful program. To overcome this problem, dynamic type checking delays the checking process until runtime. This allows a compiler to be more liberal in its acceptance of programs, however, if at runtime a computation is attempted on incompatible data, the execution of the program may abort. Some languages may attempt to convert data on-the-fly in order to avoid such runtime errors. This however might not always be in the programmers interest as computations are carried out that weren't intended. If such conversion is not sensibly possible or not even attempted and a runtime error occurs, this is ideally accompanied by a meaningful error message that informs the user where the problem occurred and provides guidance as to how to avoid the issue in the future.

In statically typed languages, and to a lesser extend in dynamically typed languages, the question arises where the compiler knowledge about types for data objects in a program comes from. For constants this is usually a matter of a simple syntactical analysis; variables and functions require more sophisticated analysis. Again, two major approaches can be identified. One approach requires the programmer to declare the types of variables and functions by annotations. These annotations are used by a compiler as assertions that variables hold values of their declared types and further checks are based on these assumptions; the process is also called *type checking*. In the second approach a compiler attempts to extend its knowledge itself from a few built-in assumptions. In this approach a programmer does not have to annotate any type information. Working from the available assumptions the required types of variables and functions are automatically inferred, in a process called *type inference*. Languages that take the former approach are referred to as being explicitly typed whereas languages using the latter approach are known as implicitly typed languages.

As language designers we have the freedom to pick and mix from the set of properties, and the above summary is not exhaustive, to put a type checking mechanism into place that best suits our requirements. But what are our requirements in S-Net?

When writing an S-Net application we usually have a certain scenario in mind in which the application will be used, and accordingly, we make assumptions on what kind of data the application needs to be able to process. We also have expectations as to what kind of results the application will produce in response to the anticipated input. We express this knowledge as network signatures to document what kind of input-output behaviour the application exposes. But in reality it would be more precise to say that we document our *expectations* as to how the application behaves. Manually working out what acceptable inputs and potentially produced outputs are may be possible for small applications, but it quickly becomes a tedious and error prone task for larger networks.

If a network signature does not reflect the real behaviour of a network it defeats the purpose of having these as a means of documentation. On the input side of the signature it is not a major issue if we miss a few inputs that would be acceptable. In the worst case this restricts the applicability of the application unnecessarily, but usually we manage to cover the inputs that we had in mind when designing the application. On the output side of the network the situation is different. If we miss some cases from the output declaration then the application may produce results that we don't expect. For stand-alone or interactive applications this is not necessarily a problem as we might just go back to the implementation and change it accordingly. If the network is deployed as part of a larger system, however, these unexpected outputs may be used as input to another system that is not able to process them. The consequences may range from silent errors over unpredictable behaviour and wrong results to catastrophic system failures.

So clearly, if we develop a type checker then the purpose of it must be to provide an automatic validation mechanism for annotated signatures. The compiler has to become a tool that confirms or corrects our intuitions about the network behaviour by matching the annotations to a compiler inferred signature. The implication of having such a mechanism in place is that S-Net can be used as an implicitly typed language;

as the compiler is able to infer signatures automatically explicit signature annotations on networks become obsolete. For documentation and reassurance, however, we can still choose to put signatures where we deem this beneficial and use the compiler to check the annotations for consistency. Of course, we expect the type checking mechanism to produce the broadest possible signature for a given network. If it is too restrictive and not all possible legal routes through a network are found, user-annotated network signatures may be falsely rejected as incorrect or the applicability of a network is limited unnecessarily.

A major contribution of this work is the development of type checking- and inference mechanisms for data-flow based languages. The key insight that we will heavily exploit is that data-flow languages and functional languages share many characteristics, and thus requirements, in their ways of computing and utilising type information. Because of the close relationship to functional languages, we will let our further elaborations be guided by type checking approaches that are employed by this category of languages. We shall be in the mindset of using functions rather than procedures and think of computations as evaluation of expressions and not as a sequence of potentially state-changing statements.

Regardless of these similarities, one of the apparent differences between S-Net and most other programming languages is that the real computations on data are carried out behind the scenes, hidden inside boxes. This is of course where most of the functionality of an application stems from. But if we liberate ourselves from the view that these are the important computations we can shift our focus onto what is observable on the S-Net level and what consequently should receive the most attention. This is the major enabling factor in establishing a close link between data-flow and functional languages. We realise this shift of focus by viewing boxes as record transformers that have the sole purpose of computing new record structures from input records. The view abstracts from the internal workings of a box. Regardless of the computations that boxes are carrying out internally, the only observable effect from a type checking point of view is how they turn input records into output records. This is the essence of computation that we have to deal with in a type system. We can also ignore the presence of multiplicities for now, i.e. the number of outputs produced, as it doesn't matter if a box produces one or any other number of results. As long as we know what kind of records will be produced, it is important to check if such a record can be dealt with by the rest of the system. If this is the case for one, it is the case for many as well. Based on these observations boxes may be viewed as S-Net's counterpart of ordinary functions in other languages. The fact that a box is transforming records on a stream is irrelevant as long as we do not take multiplicities into account. In this case the transformation of records on streams can be dealt with as a point-wise operation which in turn allows us to understand this process as mere function application that we also have in other languages. The benefit of this insight is that it very naturally extends to other parts of the language. The serial combinator in S-Net that connects the output and the input stream of two boxes to each other can now be understood as applying one function to the result of another, i.e. serial composition is just an incarnation of the basic concept of function composition.

The view that we have developed ignores a few key aspects, however. First of all there is flow-inheritance that we have to take into account. Here the view of boxes as entities that implement record transformations is key to realise that type systems for extensible records [Oho95, RWH90, HS94, HP91, CM90, Bet00, BAC06] have tackled and incorporated this feature in theoretical (and some practical) frameworks before. This allows us to benefit from the work that has been done in this area, and in fact we will base our system on what has been achieved before to model flow inheritance (to be specific, we will base our approach on qualified types [Jon95a] and a system of extensible records based on qualified types [GJ96, Gas98]).

A second aspect of boxes that complicate matters are variant output types. As a box is allowed to transform an input record into different kinds of output records, it is ostensibly not just an ordinary function. But again, this is a superficial difference from other approaches. Sum types or algebraic data types (see for example [Pie02]) provide means to capture heterogeneous collections of values by a single type. Although we will not straight-forwardly adopt these approaches, it is important to acknowledge the work that has been done on the underlying problem as it helps understanding the requirements and challenges for a system like S-Net.

Closely related to boxes that have multiple right-hand sides in their S-Net description are choice combination of boxes. The combinator introduces split and merge points in a network that allow for data items to take alternative routes between these points. A signature that describes the behaviour of such a combination will in general have more than one rule and thus introduce multiple left-hand sides for a single combinator. On first glance, the choice combinator appears to provide what other programming languages offer via conditional constructs. Conditional execution of code as in

```
result = if <predicate> then <expr1> else <expr2>
```

may be expressed in S-Net by using a box that evaluates the predicate in sequence with a choice combination over the cases. What makes this comparison slightly inaccurate is the fact that the decision on which branch to take is based on the structure of the record that is to be processed, i.e. the branch is determined by the record's type. Because of the mutual influence of types on routing and the resulting influence of routing on types we should see the choice combinator as a vehicle for introducing ad-hoc polymorphism. This way a choice combination becomes a function that holds several implementations. Depending on the type of the argument, i.e. the type of the input record, an implementation is chosen appropriately from the set of branches. Unlike most other languages S-Net allows for multiple implementations for the same argument types that do not even have to agree in their result type. As we are working on record types it is not unreasonable to think of the dispatch mechanism as a pattern matching process that operates on types. Where some approaches allow this via dynamic typing constructs [LM91, Hen94, ACPR95] we will implement a fully static approach. Experiments with a prototypical implementation of the proposed type checker suggest that the resulting complexity, although prohibitive in the general case, is not a major concern for many practical applications.

The lack of any constraints on what may be combined is what makes the choice combinator extremely versatile. However, the implications of this on type checking are literally far-reaching as choice combinations cannot be treated in isolation. An example, if admittedly somewhat contrived, illustrates the challenge. Let's assume we develop a simple image processing application that can apply two different kinds of filter operations to an image as shown in Fig. 5.1.

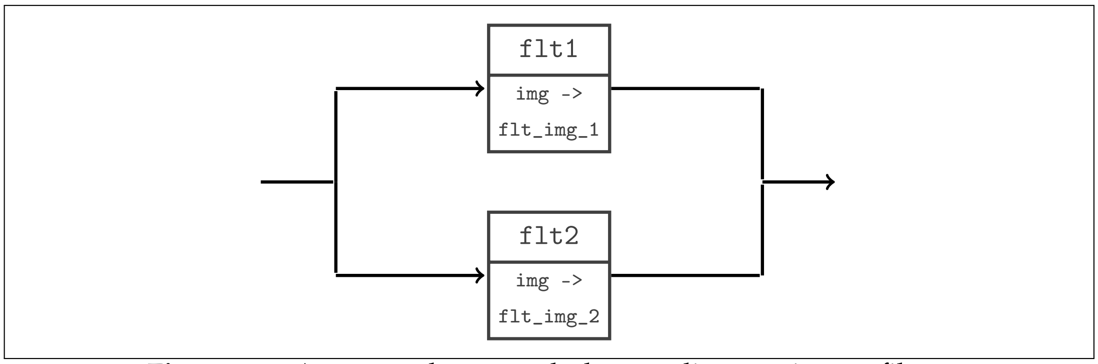


Figure 5.1: An example network that applies two image filters.

As type signatures of each of the branches in this example we can just use the box signatures. A combined signature for the entire network, assuming that we have a type checking mechanism in place, we would expect to reflect this by stating that acceptable input to this network is anything that brings an `img`. The output of this network is either `flt_img_1` or `flt_img_2` depending on which of the two branches was taken.

The network signatures of choice combinator operands play an important role during runtime as these branch signatures provide a basis for routing decisions. The choice combinator introduces a split point. At this point the function overloading (or ad-hoc polymorphism) has to be resolved and an appropriate branch of the combination has to be picked depending on the type of the incoming data. Reassurance by a type checking mechanism that confirms that our annotation is indeed correct is certainly welcome.

Now we extend our image processing application by adding a post-processing stage implemented as a third box. The post processing box, however, can only process certain kinds of images, as shown in Fig. 5.2.

We made a mistake. We should have combined the post processing box with filter box `flt1` only but instead placed it outside the choice combination. This creates a erroneous combination. By adding the `postProc` box in this way the legal routing decisions, i.e. decisions for which we can prove that records cannot get stuck, have changed. However the branch signatures on their own have not changed. If an input record contains just `img` then the second branch containing `flt2` must not be taken as there is nowhere for its result to go. Of course, one option is to declare this program illegal as it is obviously an oversight on the programmers part. However, if we think

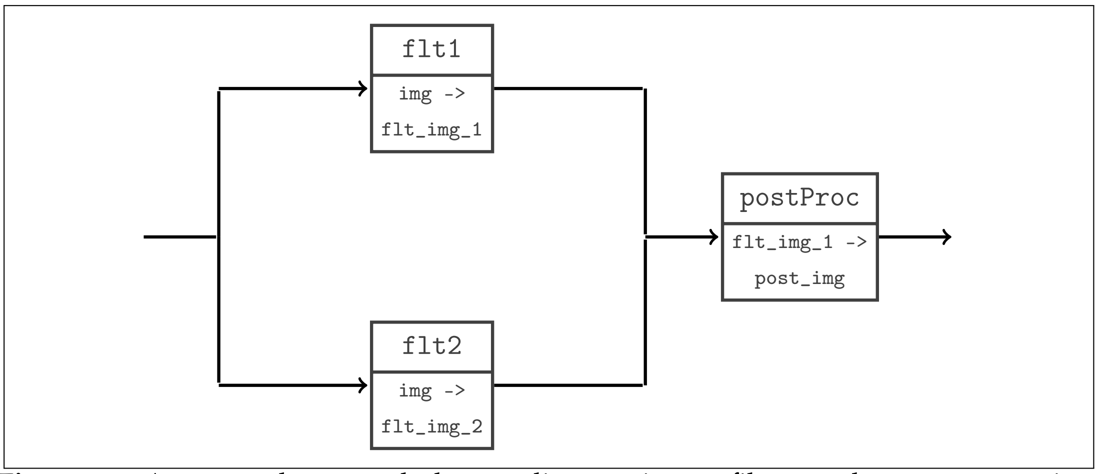


Figure 5.2: An example network that applies two image filters and a post-processing step.

about what a type checking mechanism has to do remembering that the aim is to find solutions whenever they exist, this is not the best possible option. The analysis of the entire network should ideally tell us that by taking flow-inheritance into account the input of `img, flt_img_1` is acceptable and will produce `flt_img_1`, `flt_img_2` and that an input of `img` will produce output `flt_img_1` as before.

What this simple example illustrates is that a type checking mechanism cannot easily be implemented by a bottom-up approach. The signatures of choice combinator operands will generally be too permissive with respect to the dispatch decisions if the combination is surrounded by further networks, i.e. if the combinations is embedded into a non-empty context. A choice combination that is safe in one context may very well be unsafe in other contexts, although the signatures of the operands are identical. Rephrased in terms of our previous analogy to ad-hoc polymorphism these findings show that a resolution for the dispatch onto the set of overloaded functions cannot be computed locally.

This shall close the setting of the stage on which we develop a type checking mechanism for S-Net. The design goals of the approach are:

- programs are implicitly typed from only box signatures;
- no restrictions are imposed on network construction by type checking requirements;
- the approach is fully static such that all properties are computed at compile time, i.e. there is no need for dynamic type checking;
- the type that is computed for individual paths of the network is most general;
- inferred signatures for choice combinations are the most permissive possible for a given context;

- all sub-types of legal inputs with respect to an inferred signature are also legal inputs.

The remainder of this and the next chapter develop a type checking mechanism that meets all the set out design objectives. The presented approach is an exploration of the challenges in statically type checking S-NET programs without having any constraints on network construction. It is a proposal for constructive mechanisms that may be employed to solve the type checking problem and gives guidance for a practical implementation.

5.1 Foundational Works

We have been using records in previous chapters and assumed a common understanding of what records are. And indeed, records in S-NET are not fundamentally different from records in other languages as for example C and ML. At first glance records are more versatile constructs in these established languages. `structs` in C and records in ML may be arbitrarily nested, for example, whereas records in S-NET are only allowed to be flat structures. It is a similar situation with what kind of elements are allowed within records. Where usually elements may be of any type (including records), record elements in S-NET are limited to the three types field, tag and binding tag.

From this perspective it would appear that the same techniques that are used to integrate records into the type systems of other languages can be straight-forwardly reused to accommodate records in a type system for S-NET.

What complicates the matter, however, is that S-NET allows us to work with incomplete record specifications. Many other languages, as for example C and ML, require the set of labels of a record to be statically known. But it is exactly the loosening of this restriction that renders S-NET's powerful concept of flow-inheritance possible. Without flow inheritance we would be required to explicitly list all possible record configurations that a box shall process. For example, assume we provide a box that inserts label `b` into a record `{a}`, and for completeness assume that `b` is an integer of value 0.

```
box AddLabelB( (a) -> (a, b));
```

Obviously, we expect this box to be able to process a record `{a}` and add label `b` to it. Likewise, we expect this box to turn record `{a,x}` into `{a,x,b}` and record `{a,x,y}` into `{a,x,y,b}`. In S-NET flow inheritance allows us to achieve exactly that with this one box only, and what is more, box `AddLabelB` will process *any* record that carries an `a` and add label `b` to it. In languages that demand all record labels to be statically known or inferable from the context, the specification of the same behaviour is more cumbersome. Consider ML again, to achieve the same behaviour we are required to specify those three cases explicitly:

```
fun AddLabelB1 {a=val_a} =
                  {a=val_a, b=0};
```

```
fun AddLabelB2 {a=val_a, x=val_x} =
                 {a=val_a, x=val_x, b=0};
fun AddLabelB3 {a=val_a, x=val_x, y=val_y} =
                 {a=val_a, x=val_x, y=val_y, b=0};
```

The three functions cover our three example cases, but unfortunately only those. Any other record will require us to extend the collection of functions to cater for it as an application of any of the functions to, say, record `{a=1, y=2}` results in a type mismatch error.

This is not a new problem and it has been tackled in the past by various approaches in different settings, as was mentioned in the introduction to this chapter already. We will adopt the approach suggested by Gaster in his thesis [Gas98] which uses the term *extensible records*, and we will adopt this term as well. Extensible records, as opposed to non-extensible records, may be modified without giving a full specification of their structure. One way to achieve this, is the introduction of the notion of a remainder r. In addition to the explicitly given record elements the remainder may be instantiated to a set of additional labels in order to extend the assumed structure of a record on demand. It is also useful to think of the remainder as something that implicitly represents all those parts of a record that are not relevant when a specific operation is to be applied to it. We adopt the idea of a reminder for our work as well.

For conciseness and convenience the syntax of S-Net does not require (or even allow) us to explicitly mention remainders in box signatures. As a programmer we may remember that the remainder is implicitly there; however, as a type system designer we need to have some sort of manifestation of the remainder in order to work with it. For this reason we need another, richer representation of types. To make a clear distinction between signatures and types we will use special brackets $(\!|\,|\!)$ for types. We will also make the remainder visible after a delimiting symbol | as the last element in a record type. This example illustrates the type representation of signatures using box `AddLabelB` of above:

$$\texttt{AddLabelB((a) -> (a,b))} :: (\!|\texttt{a}\ \mid r|\!) \rightarrow (\!|\texttt{a,b}\ \mid r|\!)$$

The type representation makes it more apparent that we can reformulate the description of `AddLabelB`'s behaviour to this: Take a record that contains an element `a` and some remainder r and add label `b` to this such that the result is a record with labels `a` and `b` and remainder r.

By allowing the remainder r to be instantiated to everything that is not explicitly mentioned we do not have to enumerate all possible records. This, in informal terms, describes the idea of *generalised* record manipulations.

As was mentioned above, a number of systems have been developed that offer this flexibility for the specification of record manipulations. The work that we use as foundation [Gas98] includes the development of an efficient type inference algorithm and is also available as a prototypical extension to Haskell [Pey03, Jon97]. Gaster's work is an extension of a type system for *qualified types* which has been developed by Jones in [Jon95a]. Qualified types provide the underlying theory that allows us to use all-quantification and constraints within a type which we have been using in our introductory examples of record manipulation already. Gaster extended this system to cater for records and variants, but we limit our exploration into this work to records only.

The remainder of this section will introduce these two systems and will also formally develop a notion of record types to complete the picture that we started with the examples above. We cover these topics by reproducing the relevant parts of these foundational works.

Please note, what follows in Sections 5.1.1-5.1.4 is a reproduction of Jones' and Gaster's original works [Jon95a, Gas98]. It is not a contribution of this thesis. Many of the definitions and theorems have been adopted with only minor modifications to fit the context of this work.

5.1.1 Qualified Types

The concept of qualified types was proposed by Jones in [Jon95a] and has been further developed in [Jon92, Jon95b]. Qualified types are constrained types that make a typing dependent on the satisfiability of a set of conditions. This places qualified types somewhere in the middle between most other systems: "The use of qualified types may be thought of in two ways: Either as a restricted form of polymorphism, or as an extension of the use of monotypes, commonly described as overloading, in which a function may have different interpretations according to the types of its arguments." [Jon95a].

Conditions are expressed as predicates on types and are an integral part of the type language. In order to develop an intuition on how to work with predicates, assume that $\pi(t)$ is a predicate on types $t \in T$ and that function f takes two arguments and produces one result, all of the same type. The qualified type of f may be of the form

$$f :: \forall t : \pi(t) \Rightarrow t \times t \rightarrow t.$$

This qualified type is to be read as follows. For all t for which the predicate $\pi(t)$ holds, f has the simple type $t \rightarrow t \rightarrow t$. As there may be several t for which the predicate holds, the qualified type defines a set of simple types from which we can draw elements to instantiate a simple type for f.

As a more concrete example, assume a set of basic types

$$T = \{\mathit{Int}, \mathit{Float}, \mathit{Char}\}$$

and a partition of $T = \mathit{Num} \cup \mathit{NonNum}$ with

$$\mathit{Num} = \{\mathit{Int}, \mathit{Float}\} \text{ and } \mathit{NonNum} = \{\mathit{Char}\}.$$

Furthermore, assume that we define a function add

$$add\ a\ b = a + b$$

with the common interpretation of $+$. We allow predicates to assert set membership, e.g. $t \in T$, again with the common interpretation of $\in$. If our intention is to limit the application of add to arguments that have a base type within Num, the following qualified type expresses this intention:

$$add :: \forall t : t \in Num \Rightarrow t \times t \rightarrow t$$

The predicate within the qualified type allows instantiation of $t \times t \rightarrow t$ to elements of Num, which means that add may be assigned any of the two simple types

$$\begin{array}{lll} add & :: & Int \times Int \rightarrow Int \\ add & :: & Float \times Float \rightarrow Float \end{array}$$

Term and Type Language

In most programming systems there is a strict distinction between the language in which computations are expressed, i.e. the term language, and the language in which properties about these terms are expressed, i.e. the type language. We follow this approach and define two distinct languages for these two purposes.

See Syntax 10 for a definition of the term language and Syntax 11 for the syntax of the type language that we will be using.

Syntax 10. *The term language resembles simple untyped λ-calculus extended by a let construct.*

Expr	⇒	*Var* | *App* | *Abs* | *Def*
Var	⇒	*id*
App	⇒	*Expr Expr*
Abs	⇒	λ *Var* **.** *Expr*
Def	⇒	**let** *Var* **=** *Expr* **in** *Expr*

In this definition id ranges over some given set of term variables. We write $FV(E)$ for the set of all free (term) variables of term E. The substitution $[E/x]F$ replaces all free occurrences of x in F by E which may involve renaming of bound variables to avoid capture problems (parasitic bindings).

Syntax 11. *In the following syntax definition* tvar *denotes a type variable drawn from a countably infinite set of type variables. The exact syntax for* pred *is left unspecified here as it usually changes with each application of the system. We will see one such application and its specific syntax later. For now we only note that predicates, regardless of their specifics, are part of the type language.*

Scheme	$\Rightarrow$	$\forall$ *TVars* **:** *QType*
QType	$\Rightarrow$	*Preds* $\Rightarrow$ *SType*
SType	$\Rightarrow$	*tvar*
	$\mid$	*SType* $\rightarrow$ *SType*
TVars	$\Rightarrow$	*tvar [, tvar]**
Preds	$\Rightarrow$	*pred [, pred]**

Predicates

The predicates of a qualified type determine the constraints under which the type may be used, but yet the system of qualified types does not prescribe any specific choice of predicates. The set of types over which predicates may range and the choice of relations within the predicates is flexible. This allows the predicate language to be tailor-made for a particular application. There is, however, one restriction on the choice of predicates. For any predicate system that we choose to use within the framework of qualified types, an entailment relation has to be definable. The properties that have to be met for this are laid out in Def. 18.

Definition 18. *For a set of predicates P and a predicate π an* entailment $P \Vdash \{\pi\}$ *asserts that the predicate π can be inferred from the predicates in P.*

Let P, Q, R be sets of predicates and let S be a substitution of type variables. Furthermore let SP denote the application of substitution S to the set of predicates P. A relation $\Vdash$ defined between two (arbitrary) finite sets of predicates is an entailment relation if

$$P \Vdash Q \iff \forall \pi \in Q : P \Vdash \{\pi\} \tag{5.1}$$

$$P' \subseteq P \Rightarrow P \Vdash P' \tag{5.2}$$

$$P \Vdash Q \wedge Q \Vdash R \Rightarrow P \Vdash R \tag{5.3}$$

$$P \Vdash Q \Rightarrow SP \Vdash SQ \tag{5.4}$$

Typing Rules

In order to make statements in the type language about statements in the term language, a set of typing rules defines how these two entities relate to each other. The set of typing rules is based on the standard rules of the ML type system [DM82] which have been complemented to deal with qualified types and their predicates.

Before presenting the typing rules we define the syntactic form of constructs that are used within the rules in Syntax 12. The meaning of these constructs are given in the following definitions Def. 19-Def. 21. The actual typing rules for qualified types are then presented in Def. 22.

Definition 19. *A* typing statement $x :: \sigma$ *associates term variable x with type σ.*

Definition 20. *A* type assignment *is a set of typing statements. A type assignment must not contain more than one typing statement for any term variable.*

Definition 21. *A* typing $P|A \vdash E :: \sigma$ *is to be interpreted as: if the predicates in P are satisfied and assuming the types of free variables in E are as defined by typing statements in A, then the term E has type σ.*

Syntax 12. *This syntax definition specifies the structure of* type assignments, typing statements *and* typings.

TAssgn	$\Rightarrow$	**{** *[TStmt [***,***TStmt]*]* **}**
TStmt	$\Rightarrow$	*termvar* **::** *Scheme*
Typing	$\Rightarrow$	**{** *Preds* **}** **|** *TAssgn* $\vdash$ *Term* **::** *Scheme*
Subst	$\Rightarrow$	**[** *SType* **/** *tvar* **]**

Definition 22. *The typing rules define the basic type system of qualified types.*

We impose restrictions on the application of certain rules by the choice of symbols and define that:

τ *is limited to simple types (SType in Syntax 11).*

ρ *is limited to non-schemes (QType in Syntax 11).*

σ *has no restrictions imposed (Scheme in Syntax 11).*

The notation $TV(A)$ resp. $TV(P)$ is used to denote the set of free type variables of statements in A resp. predicates in P. A substitution $[\tau/\alpha]\sigma$ replaces all free occurrences of type variable α in σ by τ.

$$\textsc{var} \quad : \quad \frac{(x :: \sigma) \in A}{P|A \vdash x :: \sigma}$$

$$\rightarrow E \quad : \quad \frac{P|A \vdash E :: \tau' \rightarrow \tau \quad P|A \vdash F :: \tau'}{P|A \vdash EF :: \tau}$$

$$\rightarrow I \quad : \quad \frac{P|(A = (A' \cup \{x :: \tau'\})) \vdash E :: \tau}{P|A \vdash \lambda x : E :: \tau' \rightarrow \tau}$$

$$\Rightarrow E \quad : \quad \frac{P|A \vdash E :: \pi \Rightarrow \rho \quad P \Vdash \pi}{P|A \vdash E :: \rho}$$

$$\Rightarrow I \quad : \quad \frac{P \cup \{\pi\}|A \vdash E :: \rho}{P|A \vdash E :: \pi \Rightarrow \rho}$$

$$\forall E \quad : \quad \frac{P|A \vdash E :: \forall \alpha : \sigma}{P|A \vdash E :: [\tau/\alpha]\sigma}$$

$$\forall I \quad : \quad \frac{P|A \vdash E :: \sigma \quad \alpha \notin TV(A) \cup TV(P)}{P|A \vdash E :: \forall \alpha : \sigma}$$

$$\textsc{let} \quad : \quad \frac{P|A \vdash E :: \sigma \quad Q|(A = A' \cup \{x :: \sigma\}) \vdash F :: \tau}{P, Q|A \vdash (\mathit{let}\ x = E\ \mathit{in}\ F) :: \tau}$$

The rule names indicate whether the rule allows for the introduction (I) or the elimination (E) of $\rightarrow$, $\Rightarrow$ or $\forall$ to resp. from a type.

Ordering of Types

We develop a notion of *generality* in this section in order for us to make statements of the form "type τ is more general than τ'". What we mean by such a statement is that whenever an object of type τ is required it is also legal to use an object of type τ'. In the context of qualified types, such statements have to include associated predicate sets as well.

Definition 23. *Let $P|A \vdash E : \sigma$ be a typing that assigns type scheme σ to term E. The instantiation of σ is constrained to environments in which the predicates in P hold. As we will work with pairs of constrains and type schemes in the following, we shall introduce the notion of a* constrained type scheme *for such pairs and use the notation $(P|\sigma)$ for these.*

As stated above our goal is to define a way to make statements about the generality of types. We will do so by introducing a preorder $\leq$ on constraint type schemes. In preparation of this we need to have a means of quantifying the applicability of constraint type schemes.

Definition 24. *A qualified type $R \Rightarrow \mu$ is a* generic instance *of the constrained type scheme $(P|\forall\alpha_i : Q \Rightarrow \tau)$ if there exist types τ_i such that*

$$R \Vdash P \cup [\tau_i/\alpha_i]Q \text{ and } \mu = [\tau_i/\alpha_i]\tau$$

By using Def. 24 as a measure for the applicability of constraint type schemes we can now introduce an ordering of these.

Definition 25. *The constrained type scheme $(Q|\eta)$ is* more general *than a constrained type scheme $(P|\sigma)$ if every generic instance of $(P|\sigma)$ is a generic instance of $(Q|\eta)$. We will write $(P|\sigma) \leq (Q|\eta)$ in this case.*

With the previous definition in place we are now in a position to define the most general type that is derivable with the rules of Def. 22 with respect to the $\leq$ ordering.

Definition 26. *A* principal type scheme *for a term E under a type assignment A is a constrained type scheme $(P|\sigma)$ such that $P|A \vdash E :: \sigma$ and $(P'|\sigma') \leq (P|\sigma)$ whenever $P'|A \vdash E :: \sigma'$.*

5.1.2 Extensible Records

In this section we put the theory of qualified types to use by developing the idea of extensible records. The system of extensible records will substantiate the so-far abstract concept of predicates in types. In this section we will also present a type inference algorithm that automatically derives a typing for a given term using the typing rules of the previous section.

We start our endeavour by formally introducing record types. This is followed by a presentation of required constraints and a suitable entailment relation for these. The type inference algorithm is presented afterwards. The last part of this section introduces basic operations for record manipulation.

Rows

Gaster's work adopts the notion of *rows* [Wan87, Wan88] to encode record types. A row contains the same labels as the record whose type it describes and additionally attaches a type to each of its labels. In general these *label types t* may range over arbitrary (finite) sets. In the setting of S-Net, however, we may limit the set of label types to a single element T, i.e. $t \in \{T\}$, as we shall distinguish between fields, tags and binding tags by purely syntactical means. It is noteworthy, still, that the system of extensible records allows for more flexibility here. We may choose to extend the type system of S-Net to support more sophisticated label types, for example, to expose type information from foreign languages. With the underlying theory of extensible records, such extensions are possible with minimal effort.

Rows[1] are recursively constructed by extension starting from an empty row. If read from left to right we may say that a row contains a first element and a rest, where the

[1]Readers may notice that rows resemble lists, and as such, the operations on rows are equivalent to those on lists. For consistency we keep the terminology of the original work.

rest is again a row until we end up with a rest that is the empty row. An empty record has as its type the empty row.

Syntax 13 defines the syntax for rows. Definition 27 introduces a membership relation on elements and rows and Def. 28 defines equality on rows.

Syntax 13.

$$\begin{array}{lll} \mathit{Row} & \Rightarrow & (\!|\,|\!) \\ & | & (\!|\ \mathit{RowElem}\ \mid\ \mathit{Row}\ |\!) \\ \mathit{RowElem} & \Rightarrow & \mathit{LabelId} : \mathit{TypeId} \\ \mathit{LabelId} & \Rightarrow & \mathit{String} \\ \mathit{TypeId} & \Rightarrow & \mathbf{F} \mid \mathbf{T} \mid \mathbf{B} \end{array}$$

Definition 27. *The membership of an element $(l : \tau)$ in a row is determined by*

$$\epsilon(\mathcal{E}) = \begin{cases} \dfrac{\mathcal{E} = ((l : t), (\!|l : t|r|\!))}{\mathcal{E} \to \top} \\[2ex] \dfrac{\mathcal{E} = ((l : t), (\!|l' : t'|r|\!)) \qquad l \neq l'}{\mathcal{E} \to \epsilon((l : t), r)} \\[2ex] \dfrac{\mathcal{E} = ((l : t), (\!|\,|\!))}{\mathcal{E} \to \bot} \end{cases}$$

As alternative (shorthand) notation for the membership of element e in row R we will be using $e \in R$ if $\epsilon(e, R) = \top$ and $e \notin R$ if $\epsilon(e, R) = \bot$.

Definition 28. *Two rows r_1, r_2 are equal if they contain the same label type pairs:*

$$\begin{array}{rcl} r_1 = r_2 & \iff & \forall l \in r_1 : l \in r_2 \\ & \wedge & \forall l \in r_2 : l \in r_1 \end{array}$$

We will use several ways to denote the type of a record. Let record

$$r = \{< seqNo >, data, < \# fix >\}.$$

Then we may use the notations

$$r \quad :: \quad (\!|seqNo : T|(\!|data : F|(\!|fix : B|(\!|\,|\!)|\!)|\!)|\!)$$

to indicate r's type. Alternative notations that we may use interchangeably are $\tau(r)$ and τ_r.

The amount of brackets may sometimes be distracting. Thus, as shorthand notation for nested rows we will usually drop the divider symbol and inner pairs of parenthesis and use commas instead. The notation of the above type becomes

$$r :: (\!|seqNo : T,\ data : F,\ fix : B|\!)$$

in this case, which leaves more focus on the labels of the type and less on the nesting structure of rows.

In order to remove an element from a row Def. 29 introduces a restriction operation.

Definition 29. *Given a row and a label, the* restriction operator *removes the corresponding label type pair from the row if it exists.*

$$\begin{aligned} (\!| l : t | r |\!) - l &= r \\ (\!| l' : t | r |\!) - l &= (\!| l' : t | r - l |\!) \text{ for } l \neq l' \\ (\!||\!) - l &= (\!||\!) \end{aligned}$$

Predicates

The construction rule for rows does not prevent us from constructing rows that contain the same label name multiple times. Although there are systems that explicitly allow this [Lei05], the system that we employ here requires all rows to have pairwise disjoint labels. As a consequence, record transformations that add new labels to a record have to be designed in a way that does not introduce duplicate labels. The design choices range from checking for the presence of a label and prevent adding a new one of the same name to implicitly removing the old label before adding a new one of the same name. The presented system adopts the former approach. Transformations that cause label clashes are prevented.

Qualified types, and more specifically the predicates within a qualified type, give us the means to formulate these construction restrictions as part of the type language. The only mechanism that we will employ in order to achieve this is a relation that asserts the absence of a label within the constraint set of a qualified type. We shall adequately call this the "lacks" predicate.

Syntax 14. *Lacks* $\Rightarrow$ *Row* $\setminus$ *LabelId*

The *lacks* predicate introduced in Syntax 14 is true if the row on the left-hand side does not contain the label given on the right-hand side. For example,

$$(\!| seqNo : T | (\!| data : F | (\!| fix : B |\!) |\!) |\!) \setminus first$$

is true whereas

$$(\!| seqNo : T | (\!| data : F | (\!| fix : B |\!) |\!) |\!) \setminus data$$

is not.

The rules of Def. 30 introduce a way to construct a proof that the predicate is true, i.e. a composition of these rules may be used to show that row r and label l are in relation $r \setminus l$.

Definition 30. *The following rules define the* lacks *relation.*

$$\textsc{Id} \quad : \quad \frac{}{P \cup \{\pi\} \Vdash \pi}$$

$$\textsc{SubRow} \quad : \quad \frac{P \Vdash r \setminus l \qquad l \neq l'}{P \Vdash (\!| l' : \tau | r |\!) \setminus l}$$

$$\textsc{ERow} \quad : \quad \frac{}{P \Vdash (\!||\!) \setminus l}$$

Theorem 1. *The lacks predicate (relation) $\setminus$ with the rules defined in Def. 30 satisfies the requirements of Def. 18. Thus, the lacks relation is a predicate entailment relation.*

Proof. See original work [Gas98] for details. □

5.1.3 Type Inference for Extensible Records

The typing rules for qualified types of Def. 22 define the legal steps that may be taken in order to construct a typing for a given term. However, these rules do not predetermine an order in which they may be applied nor do they guarantee that a result is the most general typing that is achievable.

In order to automate the process of applying the typing rules we employ a type inference algorithm. The algorithm works on a given term and an initial type assignment and applies the typing rules in a specific order to construct a set of predicates and a type for the term.

The algorithm that we will employ computes substitutions to equalise types during the inference process. These substitutions are referred to as *unifiers*, and the process of computing these is known as *unification* [Rob65].

In the presence of rows, standard unification is not powerful enough. For two rows $(\!| l : \tau | r |\!), (\!| l' : \tau' | r |\!)$ we need a way to construct a substitution S such that

$$S(\!| l : \tau | r |\!) = (\!| l : S\tau | Sr |\!) = (\!| l' : S\tau' | Sr' |\!) = S(\!| l' : \tau' | r |\!)$$

Motivated by the fact that such substitutions will generally have to insert missing fields on both sides, these substitutions are referred to as *inserters*. The process of computing and applying these is referred to as *insertion*.

The rules presented in Def. 31 define a way to compute unifiers and inserters.

Definition 31. *Let T, T' be two qualified types. A substitution S for which $ST = ST'$ is called a* unifier *of T and T'. We will write $T \overset{S}{\sim} T'$ if substitution S is a unifier of T and T'.*

Let r be a row. A substitution I that inserts $(l : \tau)$ into r is called an inserter. *We write $(l : \tau) \overset{I}{\in} r$ if $(l : I\tau) \in Ir$.*

The following rules define an algorithm to calculate such substitutions.

$$\text{U-ID} \quad : \quad \frac{}{C \overset{id}{\sim} C}$$

$$\text{U-BINDL} \quad : \quad \frac{}{\alpha \overset{[C/\alpha]}{\sim} C \quad \alpha \notin TV(C)}$$

$$\text{U-BINDR} \quad : \quad \frac{}{C \overset{[C/\alpha]}{\sim} \alpha \quad \alpha \notin TV(C)}$$

$$\text{U-APPLY} \quad : \quad \frac{C \overset{U}{\sim} C' \quad UD \overset{U'}{\sim} UD'}{CD \overset{U'U}{\sim} C'D'}$$

$$\text{U-ROW} \quad : \quad \frac{(l:\tau) \overset{I}{\in} r' \quad Ir \overset{U}{\sim} (Ir' - l)}{(\!| l:\tau \mid r |\!) \overset{UI}{\sim} r'}$$

$$\text{I-INVAR} \quad : \quad \frac{}{(l:\tau) \overset{[(\!| l:\tau \mid r' |\!)/r]}{\in} r, \quad r \notin TV(\tau), r' \; new}$$

$$\text{I-INTAIL} \quad : \quad \frac{(l:\tau) \overset{I}{\in} r \quad l \neq l'}{(l:\tau) \overset{I}{\in} (\!| l':\tau \mid r |\!)}$$

$$\text{I-INHEAD} \quad : \quad \frac{\tau \overset{U}{\sim} \tau'}{(l:\tau) \overset{U}{\in} (\!| l:\tau' \mid r |\!)}$$

Definition 32. *Let T, T' be two qualified types. A substitution S for which $ST = ST'$ is called a* most general unifier *of T and T' if every unifier of T and T' can be written in the form RS for some substitution R.*

Definition 33. *Let r be a row. An inserter I of $(l:\tau)$ into r is a* most general inserter *into r if every such inserter can be written as RI for some substitution R.*

Theorem 2. *The algorithm defined in Def. 31 calculates most general unifiers and most general inserters whenever they exit. The algorithm fails precisely when no unifier or inserter exists.*

Proof. See original work [Gas98] for details. □

With all the above definitions in place we have all the required ingredients to formulate a type inference algorithm (Def. 34) for the system of extensible records. The algorithm has been developed in the context of the original work of qualified types [Jon95a] and has been extended by the computation of inserters during the unification process.

Definition 34. *The following rules define a type inference algorithm that calculates a typing judgement $P|TA \vdash E :: \tau$ for a given term E and assignment A if it exists, i.e. the predicate assignment P, type τ and substitution T are synthesized by this algorithm.*

$$\text{VAR} \quad : \quad \frac{(x : \forall\alpha_i.P \Rightarrow \tau) \in A \quad \beta_i \; new}{[\beta_i/\alpha_i]P|A \vdash x : [\beta_i/\alpha_i]\tau}$$

$$\rightarrow E \quad : \quad \frac{P|TA \vdash E : \tau \quad Q|T'TA \vdash F : \tau' \quad T'\tau \overset{U}{\sim} \tau' \rightarrow \alpha \qquad \alpha \; new}{U(T'P \cup Q)|UT'TA \vdash EF : U\alpha}$$

$$\rightarrow I \quad : \quad \frac{P|T(A_x, x : \alpha) \vdash E : \tau \quad \alpha \; new}{P|TA \vdash \lambda x.E : T\alpha \rightarrow \tau}$$

$$\text{LET} \quad : \quad \frac{P|TA \vdash E : \tau \quad \sigma = Gen(TA, P \Rightarrow \tau) \quad P'|T'(TA_x, x : \sigma) \vdash F : \tau'}{P'|T'TA \vdash (let \; x = E \; in \; F) : \tau'}$$

We use $Gen(A, \rho)$ to denote the generalisation of a qualified type ρ and assignment A, i.e.

$$Gen(A, \rho) = \forall\alpha_i : \rho \; where \; \{\alpha_i\} = TV(\rho) \setminus TV(A)$$

Theorem 3. *The algorithm of Def. 34 calculates a principal type (Def. 26) for a given term E under assumptions A. The algorithm fails precisely when there is no typing for E under A.*

Proof. See original work [Jon95a] for details. □

5.1.4 Basic Record Operations

With the basic theory in place, we are now in a position to formally define the basic record operations that we introduced in the beginning of this section.

Restriction

The *restriction* operation removes a label from a record regardless of the label type. We assume one restriction operation per unique label l.

$$\texttt{rem}_l :: \forall\alpha, r : (r \setminus l) \Rightarrow (\!| l : \alpha | r |\!) \rightarrow r$$

Extension

The *extension* operation adds a label to a record, regardless of the label type. We assume one extension operation per unique label l.

$$\texttt{add}_l :: \forall\alpha, r : (r \setminus l) \Rightarrow \alpha \rightarrow r \rightarrow (\!| l : \alpha | r |\!)$$

Selection

The *selection* operation extracts a label from a record, regardless of the label type. We assume one selection operation per unique label l.

$$\texttt{sel}_l :: \forall\alpha, r : (r \setminus l) \Rightarrow (\!| l : \alpha | r |\!) \rightarrow \alpha$$

Syntactical Convention

We will use these basic record operations throughout the remainder of this chapter. For convenience we introduce shorthands for their types:

$$\begin{array}{llll}
\texttt{rem}_l & :: & \tau_{rem_l} & = & \forall\alpha, r : (r \setminus l) \Rightarrow (\!|l : \alpha|r|\!) \rightarrow r \\
\texttt{add}_l & :: & \tau_{add_l} & = & \forall\alpha, r : (r \setminus l) \Rightarrow \alpha \rightarrow r \rightarrow (\!|l : \alpha|r|\!) \\
\texttt{sel}_l & :: & \tau_{sel_l} & = & \forall\alpha, r : (r \setminus l) \Rightarrow (\!|l : \alpha|r|\!) \rightarrow \alpha
\end{array}$$

6 Type Checking and Type Inference

In this chapter we will develop a set of algorithms that, given an S-Net program, computes network signatures for this program automatically. This type inference process only requires boxes to be annotated with their signatures and computes all remaining types for a network described by a connect expression automatically. A connect expression describes complex relationships between boxes and other networks on a rather high level of abstraction, which unnecessarily complicates the definition of inference algorithm if done on the same level. In order to bridge the gap between an actual S-Net program and what is a reasonable representation of such program for type inference purposes, a set of transformation systematically breaks down S-Net expression into smaller sub-expressions that encode only one possible route through the original network. At the end of the process we will be able to describe such a route, and more specifically, the transformations that are applied to a record on its way along this route, in terms of basic record transformations.

The first objective in the process of putting together a type inference system for S-Net is to develop a set of transformations that decompose and translate S-Net connect expressions into a form that allows us to apply the inference algorithm for extensible records. If this is achieved, inferring a type would only be a matter of applying this algorithm to the transformed expression. It will not be quite as straight forward as this, but we will develop mechanisms that transform individual routes into suitable input for extensible record type inference. The result, consequently, is the type of one path through the original network. What remains to be done is the reversal of the transformations to reconstitute the original S-Net expression again. Several consistency checks ensure that a recombination of routes and the combination of their inferred types still yield legal results. We develop mechanisms that automatically amend re-combinations in cases where illegal networks are formed.

The formalisation of the transformation process exploits the inherent tree structure of expressions. Transformation steps will generally be defined in terms of rewriting rules that operate on nodes of such trees. In each step a node may be enriched with more information or replaced by a new node including new child nodes altogether. The standard language syntax already exposes the tree structure of programs, but as it is primarily designed for readability it encodes various information only implicitly to keep it concise. As most transformation will lower the level of abstraction and introduce more explicit representation of information, the standard syntax would quickly be rendered unwieldy by these extensions. Therefore, we employ a tuple representation of expressions that will accommodate auxiliary information more naturally.

The transformation process is broken down into several stages. A first transformation translates an S-Net program into a tuple representation that we refer to as

S-NET$_{Tup}$. A second transformation rewrites the program in a way that still exposes the same observable input and output behaviour but only contains basic boxes and the network combinators, i.e. the program is void of variant boxes and synchro cells. This representation is called S-NET$_{\mathcal{T}}$. In a third step the choice combinations within a network are dissolved into a collection of separate routes which means that form this point onwards we are dealing with a collection of sub-programs that are defined in terms of sequential composition only. The result is a program in S $\cdot\cdot$ NET representation. These sub-programs are then translated into compositions of basic record transformations. The result of this is a collection of expressions that are all suitable for processing by the type inference algorithm for extensible records.

At this stage we have reached a turning point in the process. Starting from the record transformation expressions of each route and their now associated types, network signatures are constructed. From the single-rule signatures of each route we recombine the routes into a full network signature step-by-step. The decomposition transformations introduce auxiliary routes which the recombination process needs to identify. At each recombination step it is made sure that the addition of a route does not allow input to be accepted that would not have been acceptable by the original program. After the recombination process finishes, we extend the original network by the inferred signatures.

The following sections will introduce all transformations in the sequence in which they are applied to a program. First, however, we briefly introduce a few required formalisms.

6.1 Basic Definitions

6.1.1 Tuples

The predominant structures that we will be using throughout the next sections are n-tuples. As a convenient way to represent a variable number of elements of various kinds in an ordered way, tuples allow for a concise encoding of program fragments including any number of additional information that we might desire. A complete program, including its tree-like structure, is represented by a nesting of these tuples.

In addition to standard tuple notation we will need a mechanism that disallows nonsensical compositions, as tuples without any restrictions allow for arbitrary combination and nesting of elements. A syntax-directed approach enables us to define the intended structure of tuples, which implicitly formulates restrictions in terms of a simple, context-free grammar. In order to discriminate different kinds of tuples, we will, by convention, always use the first element of a tuple as classifier.

In general, tuples follow the syntactic rules (presented in EBNF [ISO96]) as defined in Def. 35. In some situations it is desirable to access elements of a tuple without giving its full structure. For this we introduce a generic projection function for tuples (Def. 36).

Definition 35. *A* ***syntax restricted*** n***-tuple*** *is a sequence of an identifier followed by* $n \in \mathbb{N}$ *elements of some universe* U.

Tuple	$\Rightarrow$	**(** *Id* **,** *Elements* **)**
Elements	$\Rightarrow$	*elem[* **,** *Elements]**
	$\mid$	*Tuple[* **,** *Elements]**
Id	$\Rightarrow$	*String*

An empty tuple contains no further elements in addition to an identifier; in this case the enclosing brackets may be omitted.

The set of all $\mathbb{N}_0 \ni n$*-tuples is denoted by* $\mathbb{U}^n$ *and* $\bigcup\limits_{k \in \mathbb{N}_0} U^k = \mathbb{U}^\star$.

Definition 36. *A* ***projection function***

$$\pi : \mathbb{N}_0 \times \mathbb{U}^\star \to \mathbb{U}^\star$$

selects an element from a given tuple. The selection of the $\mathbb{N}_0 \ni i$*'th element of tuple* t *is denoted by* $\pi(i,t)$ *or* $\pi_i(t)$. *For* $i = 0$ *the identifier is selected. The result of this function shall be undefined if* t *contains fewer than* i *elements. Projection functions may be nested to select elements from nested tuples. As a syntactical convenience we define for* $i, j, \ldots, k \in \mathbb{N}_0$

$$\pi_{i,j,\ldots,k}(t) = \pi_i(\pi_j(\ldots(\pi_k(t))))$$

If a projection function is applied to a set P *of tuples then we interpret this as*

$$\pi_i(P) = \{\pi_i(p) \mid p \in P\}$$

6.1.2 Presentation of Transformations

As in previous chapters, algorithms and transformation will be presented in the form of deduction rules. To emphasise the algorithmic character of transformations we will not name individual rules but entire groups of rules. A function identifier will be given to such a group which allows referencing by name a transformation within rules. Additionally, to mimic pattern matching within rule definitions the function identifier of a transformation takes an argument that appears in the precondition of rules in order to restrict application of rules to specific inputs.

6.1.3 Sub-Typing

The only sub-typing that we will be using is sub-typing on record types.

Definition 37. *A record type* τ_r *is a subtype of record type* $\tau_{r'}$, *written* $\tau_r \sqsubseteq \tau_{r'}$ *if the row representing* τ_r *contains all labels that are present in* $\tau_{r'}$ *and possibly more.*

$$\tau_r \sqsubseteq \tau_{r'} \iff \forall (l : \alpha) \in \tau_{r'} : (l : \alpha) \in \tau_r$$

Let R be the set of all rows over an arbitrary but fixed set of labels L and label types A such that $\forall \tau_r \in R : \forall (l : a) \in \tau_r : l \in L \wedge a \in A$. It is easily established that $\sqsubseteq$ defines a partial order on R as

$$\begin{aligned}
\forall \tau_r \in R &: \tau_r \sqsubseteq \tau_r \\
\forall \tau_r, \tau_{r'} \in R &: \tau_r \sqsubseteq \tau_{r'} \wedge \tau_{r'} \sqsubseteq \tau_r \Rightarrow \tau_r = \tau_{r'} \\
\forall \tau_r, \tau_{r'}, \tau_{r''} \in R &: \tau_r \sqsubseteq \tau_{r'} \wedge \tau_{r'} \sqsubseteq \tau_{r''} \Rightarrow \tau_r \sqsubseteq \tau_{r''}
\end{aligned}$$

6.2 Translation of S-NET to S-NET$_{Tup}$

The goal of this transformation is the translation of standard S-NET programs into a tuple representation which will contribute to concise definitions of further transformations. For simplification we assume that boxes have at most two variants in their output. All transformations naturally extend to boxes with more than two outputs but dealing with these requires more complexity in early transformations and does not provide any further insights. Additionally, we also assume that three trivial transformations have been applied already. Firstly, we expect box definitions to be inlined into the `connect` expression of a program. Secondly, as the type inference engine that we develop infers network signatures, we are not considering networks that are referenced from within connect expressions. Nested networks are dealt with by first inferring inner network's signatures recursively and then work with the result of the inference process as a set of plain box signatures. We postpone a detailed discussion of this approach until Sect. 6.10.3. Thirdly, and lastly, we assume that star combinators are always applied to networks. This allows us to separate out type inference for this combinator until a later stage in Sect. 6.9.1 and we consequently do not include the star combinator in the process at this point.

Assuming all this, standard S-NET expressions may be mapped into nested tuples straight-forwardly. Most components may be translated into 2-tuples by changing the infix notation that is used for standard expressions into a prefix notation where the operator poses as tuple identifier, see Fig. 6.1 for an example. Pattern matching on these tuples allows for a purely declarative specification of algorithms on programs in this representation and it also allows us to steer complex program transformations by purely syntactical means.

The syntax for this tuple representation is given in Syntax 15, transformation rules are given in Transformation 1.

Syntax 15. *The subscript indices on tuple identifiers syntactically restrict the use of tuples to certain transformations.*

$$\begin{array}{lcl}
\textit{Expr1} & \Rightarrow & (\ \mathbf{Box_1},\ \textit{Id}\ ,\ \textit{Rec}\ ,\ \textit{Reclist}\ ) \\
 & | & (\ \mathbf{Sync_1},\ \textit{Id}\ ,\ \textit{Rec}\ ,\ \textit{Rec}\ ) \\
 & | & (\ \mathbf{Seq_1},\ \textit{Expr1}\ ,\ \textit{Expr1}\ ) \\
 & | & (\ \mathbf{Choice_1},\ \textit{Expr1}\ ,\ \textit{Expr1}\ ) \\
 & | & (\ \mathbf{Split_1},\ \textit{Expr1}\ ,\ \textit{Label}\ )
\end{array}$$

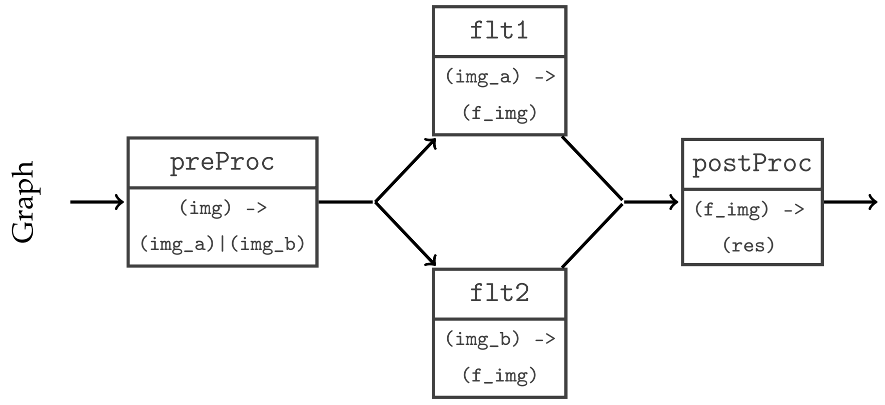
Transformation 1. *This transformation turns a program into the tuple representation that is used in all later stages of the inference process. The type inference is unaffected by operational differences between deterministic and non-deterministic combinators; deterministic combinators are mapped onto the same syntactical construct as the non-deterministic combinators.*

$$T^{Tup}_{SNet}(\mathcal{E}) = \begin{cases} \mathcal{E} = \ \texttt{net}\ N\ \texttt{connect}\ expr; \\ \dfrac{expr \to e'}{\mathcal{E} \to e'} \\ \\ \dfrac{}{\texttt{box Id(l -> r)} \to (\mathbf{Box_1}, Id, l, r)} \\ \\ \dfrac{}{\texttt{[|l, r|]} \to (\mathbf{Sync_1}, l, r)} \\ \\ \dfrac{l \to p \qquad r \to q}{l\ \texttt{..}\ r \to (\mathbf{Seq_1}, p, q)} \\ \\ \dfrac{l \to p \qquad r \to q}{r\ \texttt{|}\ l \to (\mathbf{Choice_1}, p, q)} \\ \\ \dfrac{e \to q}{e\ \texttt{!}\ \texttt{<p>} \to (\mathbf{Split_1}, q, p)} \\ \\ \dfrac{l \to p \qquad r \to q}{r\ \texttt{||}\ l \to (\mathbf{Choice_1}, p, q)} \\ \\ \dfrac{e \to q}{e\ \texttt{!!}\ \texttt{<p>} \to (\mathbf{Split_1}, q, p)} \end{cases}$$

6.3 Transformation of S-NET$_{Tup}$ to S-NET$_{\mathcal{T}}$

One of the key challenges in type checking S-NET is dealing with alternative routes within a network. It is not so much a problem of identifying these alternatives but deciding which combinations of alternatives lead to a legal composition of routes. Alternatives are most prominently introduced by choice combinators as these allow for a record to be sent down one of two paths. However, alternatives are also introduced less prominently by boxes with more than one right-hand side. Depending on which of the results is produced in response to an input the set of potential routes through the remaining network generally changes. The same holds for synchro cells as their output changes with the state they are in and the input that is received.

In order to avoid dealing with different kinds of alternatives in later stages of the



S-Net program

```
net imgProc {
  box preProc((img) -> (img_a) | (img_b));
  box flt1((img_a) -> (f_img));
  box flt2((img_b) -> (f_img));
  box postProc((f_img) -> (res));
}
connect preProc .. (flt1 | flt2) .. postProc;
```

Inlined boxes

```
  net imgProc connect
    box preProc((img) -> (img_a) | (img_b))
    .. (   box flt1((img_a) -> (f_img))
         | box flt2((img_b) -> (f_img)))
       )
    .. box postProc((f_img) -> (res));
```

Tuple representation

$(\mathbf{Seq_1},$
$\quad(\mathbf{Seq_1},$
$\quad\quad(\mathbf{Box_1}, \texttt{preProc}, \{img\}, \{\{img_a\}, \{img_b\}\})$
$\quad\quad(\mathbf{Choice_1},$
$\quad\quad\quad(\mathbf{Box_1}, \texttt{flt1}, \{img_a\}, \{\{f_img\}\})$
$\quad\quad\quad(\mathbf{Box_1}, \texttt{flt2}, \{img_b\}, \{\{f_img\}\})),$
$\quad(\mathbf{Box_1}, \texttt{postProc}, \{f_img\}, \{\{res\}\}))$
$);$

Figure 6.1: A S-Net program with all its boxes inlined and in tuple representation

checking process we homogenise the representations of alternatives. The transformation that we develop in this section achieves this by developing mechanisms that substitute variant boxes and synchro cells by choice combinations over simple boxes only.

6.3.1 Variant Boxes in S-NET$_{\mathcal{T}}$

A box in S-NET is defined on one single record type. The co-domain of the box, however, may contain several record types. This is the case for boxes with multiple right-hand sides, for example:

```
box preProc((img) -> (img_a) | (img_b));
```

Instead of tackling the issue on this level by defining a more expressive base language that encodes these variants, boxes as the one shown above are split up into several boxes each with a single left-hand side. These replacement boxes are joined by choice combinators which results in a new sub-program that contains as many branches as the original box had alternatives on the right-hand side. See Fig. 6.2 for an example. This transformed program now obviously departs from the operational semantics of the original program, but it still exhibits the same observable behaviour in terms of record transformations. Where previously the original box would produce any combination of output records as defined by the variants on its right-hand side, the replacement expression offers multiple routes for a record. Each of the routes maps the record into one of the potential outputs of the box. The introduction of these auxiliary routes has artificially shifted the decision of which of the variants is produced from a box to the coordination layer. The fact that the replacement expression only produces exactly one of the variants per record, whereas the box could potentially produce all of them, is irrelevant if the inference system can prove that there is a continuation for each of the potential outputs of the choice combination. This is an important aspect that has to be checked when the overall network signature is assembled from the inferred types for the routes. We will deal with this in detail in Section 6.7. For now we note that it is important for auxiliary routes to stay distinguishable from original routes. Because of this, the current transformation introduces markers for the operands inside a choice combinator tuple. The markers that are used within regular choice combinators differ from the markers in newly introduced combinators and thus enable straightforward identification.

A further important aspect is the loosened restriction on inherited labels that the decomposition of a box into multiple branches of a choice combinator entails. To avoid label overwriting in the transformed program it suffices to rule out only those labels as input that appear on the right-hand side of the substituting boxes. However, as the original program does not contain parallel branches and no choice can be made depending on inherited labels, all labels that appear on the right-hand side of the original box have to be protected from being overwritten. One way of achieving this is to collect the constraints that the inference algorithm computes for each of the branches and then to analyse and propagate the accumulated constraints to each of the individual branches. This approach requires extra machinery to be carried out as it has to access several routes, relate them to one another and then compute one common and consistent constraint set for all of them. However, there is a surprisingly simple approach that achieves the same result without the need for post-processing steps. In each of the generated branches, we use two boxes in sequence rather than just one. A first box maps the input to the union of all right-hand sides to introduce all necessary constraints. A following box consumes all labels except those that belong to the variant that the branch represents. The result of a sequential composition of these two boxes achieves our desired goal of producing the same result as before but additionally producing all possible labels as an intermediate result to put clash-avoiding constraints into place.

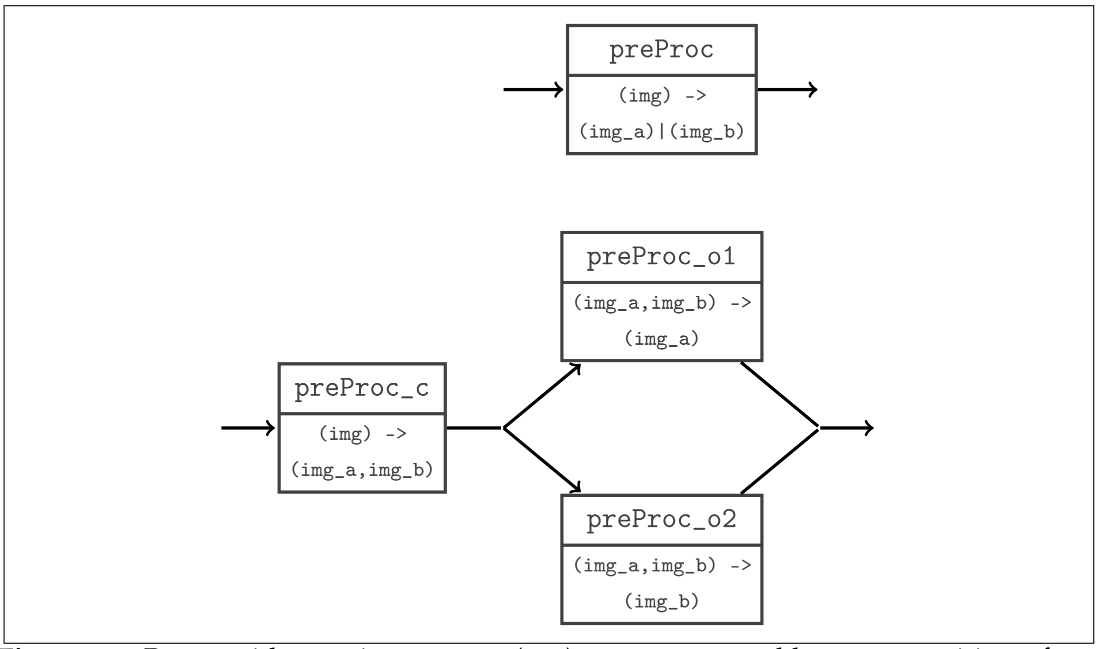


Figure 6.2: Boxes with a variant output (top) are represented by a composition of three non-variant boxes in S-Net$_\mathcal{T}$ (bottom)

6.3.2 Synchro cells in S-Net$_\mathcal{T}$

The synchro cell accepts two different kinds of input. These are records that either match the first pattern or the second pattern over which the cell is parametrised. It also accepts records that match both patterns at the same time, but this is merely a special case of the two base cases. Additionally, the output of a synchro cell is determined by the current state the cell is in and the input that is provided. On the output side and from a type perspective, this is identical to a box with multiple right-hand sides. Although we have solved this problem above, the existence of multiple left-hand sides requires additional attention.

The goal, again, is to map a synchro cell to basic constructs to avoid special rules and mechanisms during the remaining treatment of this component. To achieve this we replace synchro cells by a choice composition of simple boxes. Each alternative of this composition represents the behaviour of a synchro cell in a specific state and a specific input as defined by its operational semantics in Sect. 4.2.2. Let's assume our original program contains a synchro cell `[|{img1}, {img2}|]`. The representation in S-NET$_{\mathcal{T}}$ where the synchro cell is broken down into its states is shown in Fig. 6.3.

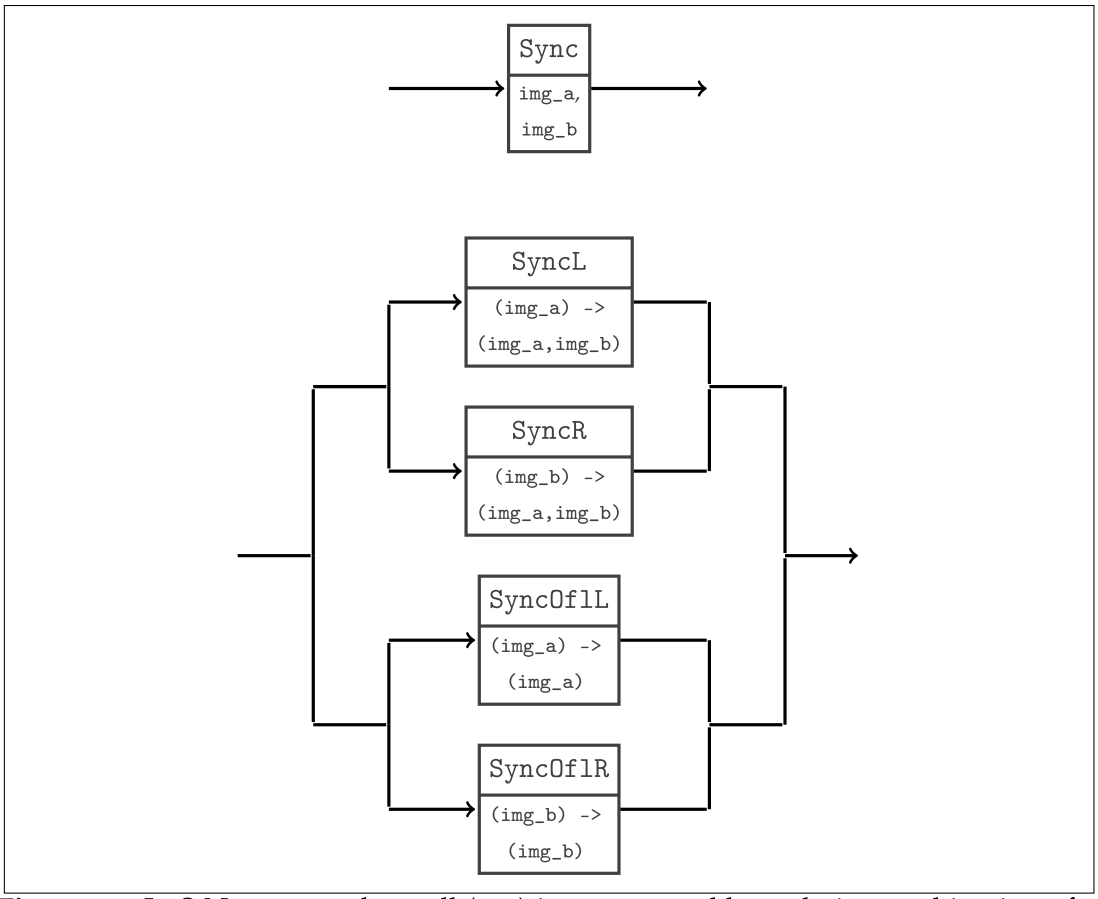


Figure 6.3: In S-NET$_{\mathcal{T}}$ a synchro cell (top) is represented by a choice combination of simple boxes. Each box represents the behaviour of one state of the original synchro cell.

This construction introduces choice points that have no counterpart in the original program. To allow for later identification of these false choice points, we use markers that allow identification of these new auxiliary routes. As before, this will be important when constructing the overall type from the now independent branches. A detailed discussion of this process takes place in Section 6.7.

6.3.3 Index splits in S-Net$_\tau$

The index split combinator `!` does not require rewriting into several routes. A simple box that takes the tag of the combinator as input and outputs the same again is sufficient to model all typing requirements.

6.3.4 Transformation Rules

Transformation 2 introduces a new set of identifiers for tuples and extends the choice combinator representation by entries for markers, see Syntax 16. The markers distinguish the choice combinator origins, as the transformation of variant boxes and synchro cells both introduce new choice combinators.

Syntax 16. *The markers are used to tag branches of choice combinations; each marker identifies the origin of a certain branch.*

$$
\begin{array}{lcl}
\textit{Expr2} & \Rightarrow & (\ \mathbf{Box_2},\ \textit{Id}\ ,\ \textit{Rec}\ ,\ \textit{Rec}\ ) \\
 & | & (\ \mathbf{Seq_2},\ \textit{Expr2}\ ,\ \textit{Expr2}\ ) \\
 & | & (\ \mathbf{Choice_2},\ \textit{Expr2}\ ,\textit{Marker}\ ,\ \textit{Expr2}\ ,\ \textit{Marker}\ ) \\
\textit{Marker} & \Rightarrow & \textit{VarBoxM}\ |\ \textit{SyncM}\ |\ \textit{ChoiceM}\ |\ \textit{None} \\
\textit{VarBoxM} & \Rightarrow & \mathbf{V}_1\ |\ \mathbf{V}_2 \\
\textit{SyncM} & \Rightarrow & \mathbf{S}_{1o}\ |\ \mathbf{S}_{1s}\ |\ \mathbf{S}_{2o}\ |\ \mathbf{S}_{2s} \\
\textit{ChoiceM} & \Rightarrow & \mathbf{C}_1\ |\ \mathbf{C}_2 \\
\textit{None} & \Rightarrow & \epsilon
\end{array}
$$

Transformation 2. *The transformation function $T^{\tau}_{Tup}(\mathcal{E})$ drives the transformation process from a program $\mathcal{E}$ in* S-Net$_{Tup}$ *representation to a presentation in* S-Net$_\tau$*. The function comprises several cases, one for each possible syntactic construct in* S-Net$_{Tup}$*.*

$$T^{\mathcal{T}}_{Tup}(\mathcal{E}) = \begin{cases}
\dfrac{\mathcal{E} = (\mathbf{Sync_1}, p_1, p_2) \qquad id = \texttt{newId}()}{\begin{array}{lll} \mathcal{E} \rightarrow (\mathbf{Choice_2}, & & \\ & (\mathbf{Choice_2}, & (\mathbf{Box_2}, id \wedge \text{"S1O"}, p_l, p_l), S_{1o}, \\ & & (\mathbf{Box_2}, id \wedge \text{"S2O"}, p_r, p_r,), S_{2o}) \\ & (\mathbf{Choice_2}, & (\mathbf{Box_2}, id \wedge \text{"S1S"}, p_l, p_l \cup p_r), S_{1s}, \\ & & (\mathbf{Box_2}, id \wedge \text{"S2S"}, p_r, p_l \cup p_r), S_{2s}), \epsilon) \end{array}} \\[2ex]
\dfrac{\mathcal{E} = (\mathbf{Box_1}, id, r_{in}, \{r_{o1}, r_{o2}\}) \quad vid = \texttt{newid}()}{\begin{array}{lll} \mathcal{E} \rightarrow (\mathbf{Seq_2}, & & \\ & (\mathbf{Box_2}, id \wedge vid \wedge \text{"c"}, r_{o1} \cup r_{o2}), & \\ & (\mathbf{Choice_2}, & (\mathbf{Box_2}, id \wedge vid \wedge \text{"V1"}, r_{o1} \cup r_{o2}, r_{o1}), V_1, \\ & & (\mathbf{Box_2}, id \wedge vid \wedge \text{"V2"}, r_{o1} \cup r_{o2}, r_{o2}), V_2)) \end{array}} \\[2ex]
\dfrac{\mathcal{E} = (\mathbf{Split_1}, e, l) \qquad vid = \texttt{newid}()}{\mathcal{E} \rightarrow (\mathbf{Seq_2}, (\mathbf{Box_2}, id \wedge \text{"split"}, l, l), e)} \\[2ex]
\dfrac{\mathcal{E} = (\mathbf{Choice_1}, l, r) \quad l \rightarrow l_2 \quad r \rightarrow r_2}{\mathcal{E} \rightarrow (\mathbf{Choice_2}, l_2, C_1, r_2, C_2)} \\[2ex]
\dfrac{\mathcal{E} = (\mathbf{Seq_1}, l, r) \quad l \rightarrow l_2 \quad r \rightarrow r_2}{\mathcal{E} \rightarrow (\mathbf{Seq_2}, l_2, r_2)}
\end{cases}$$

6.4 Transformation of S-Net$_{\mathcal{T}}$ to S ·· Net

In the previous transformation that produced an S-Net$_{\mathcal{T}}$ program from an S-Net program, we were able to replace all components that did not have a one-to-one mapping of input types to output types by simpler alternatives. On the type level, the resulting program still exposes the same observable record transformations as the original S-Net program, although it only contains sequential and parallel composition of simple boxes. In this transformation it is the choice combinators we are after.

At runtime, the choice combinators in a program mark those points at which program continuation alternatives have to be considered. For a data item that reaches such a point there are, in general, two potential routes to choose from. The decision that is taken here may limit the degree of freedom for later choices, but as long as there is a continuation for all of the choices that are possible at this point, no restrictions have to be put in place. Unfortunately, this is not always the case.

The decision that is made at a choice point may have a legal, immediate continuation. But it is possible that, after a choice has been made and a few further processing steps have occurred, a result is produced for which no route through the rest of the network exists. The problem is that there is no way of knowing this in advance without analysing the remaining program. A decision at a choice point cannot be made locally without taking the surrounding context into consideration, as we have discussed in Chapter 5; we will see more concrete examples of this later in Sect. 6.7. Identifying and ruling out decisions that would send data items down a route that eventually turns into a dead end has to be realised on a level where context information is available. Consequently, a pre-selection of only legal decisions is computed during compile time, i.e. at a time when context information is readily available. At runtime only safe alternatives are presented to the choice combinators to choose from.

The computation of the set of safe alternatives is carried out as the last step of the inference process (see Sect. 6.7) as only then all context information is available to identify those hazardous routes which we must prevent from being chosen by choice combinators.

As a step towards this solution, we expand the representation of a program such that it exposes its possible routes on the outer-most level. In order to describe this process it is instructive to think in terms of Boolean conjunctions, $\wedge$ and disjunctions, $\vee$. A choice composition `A|B` within a program represents two possible strands of computation, i.e. either `A` *or* `B`. Sequential composition on the other hand, as in `A..B`, entails that both `A` *and* `B` will be applied. Now consider the program `A..(B|C)`. Following the above two rules we rewrite this to

$$\mathtt{A} \wedge (\mathtt{B} \vee \mathtt{C})$$

Now, in order to decompose the program into its individual routes we transform the expression into its disjunctive normal form

$$(\mathtt{A} \wedge \mathtt{B}) \vee (\mathtt{A} \wedge \mathtt{C})$$

which encodes the two possible paths `(A..B)` and `(A..C)` of the program. In this representation a program is a disjunction of conjunctions, i.e. the *or* combinations have been driven as far to the outside as possible. Consequently, a program that comprises n routes $r_1, \ldots, r_n$ can be written as

$$\bigvee_{i \in \{1,\ldots,n\}} r_i$$

where each r_i is of the form $r_{i_1} \wedge \ldots \wedge r_{i_{n_i}}$ for some $n_i \in \mathbb{N}$. This transformation is of course nothing but repeated application of the expansion rules of Boolean distributivity. Interpreted in terms of standard S-Net expressions this transformation yields a program that has choice combinations only on the outer levels followed by only sequential compositions on the inner levels.

The process of decomposing a program into its individual routes is defined in Transformation 3 where $T_\tau^{Seq'}(\mathcal{E})$ applies the expansion rules until a fixed point is reached and no further expansions are possible. To avoid a further change in representation the expansion rules are defined on standard expressions rather than Boolean expressions in the same way as discussed above.

As a further simplification step the outer choice compositions are folded into just one large set of sequential routes. For this we introduce a new node type **Subst₃** that collects the individual routes together with a marker string for identification. The string is constructed from the markers at choice nodes that we have introduced during Transformation 2.

For demonstration purposes of how these transformations work, reconsider the example of Fig. 6.1 and refer to it as $\mathcal{E}$. After the application of Transformation 2 to the program $\mathcal{E}$ the situation is as shown in Fig. 6.4(top). In the network illustration the markers that are kept at choice nodes in tuple representation are shown above and below the arrows pointing away from the split point of its respective choice tuple. The tuple representation of the program at this stage is as shown in Fig. 6.4(bottom). In this representation the tuple expression still contains inner choice combinators. As the tuple representation is for larger examples unwieldy to work with we shall replace the box tuples by only their box identifiers. This will allow us to more easily focus on the structure of expressions which in this case is what we are interested in. An application of $T_\tau^{Seq'}(\mathcal{E})$, which is a driver for $T_{Expd}\left(\ldots T_{Expd}(\mathcal{E})\right)$, moves these inner choices to the outer level where a fix-point is reached at the presentation that is shown in Fig. 6.5. In terms of Boolean expressions the equivalent transformation would turn

```
preProc_c ∧ (preProc_o1 ∨ preProc_o2)
          ∧ (flt1 ∨ flt) ∧ postProc
```

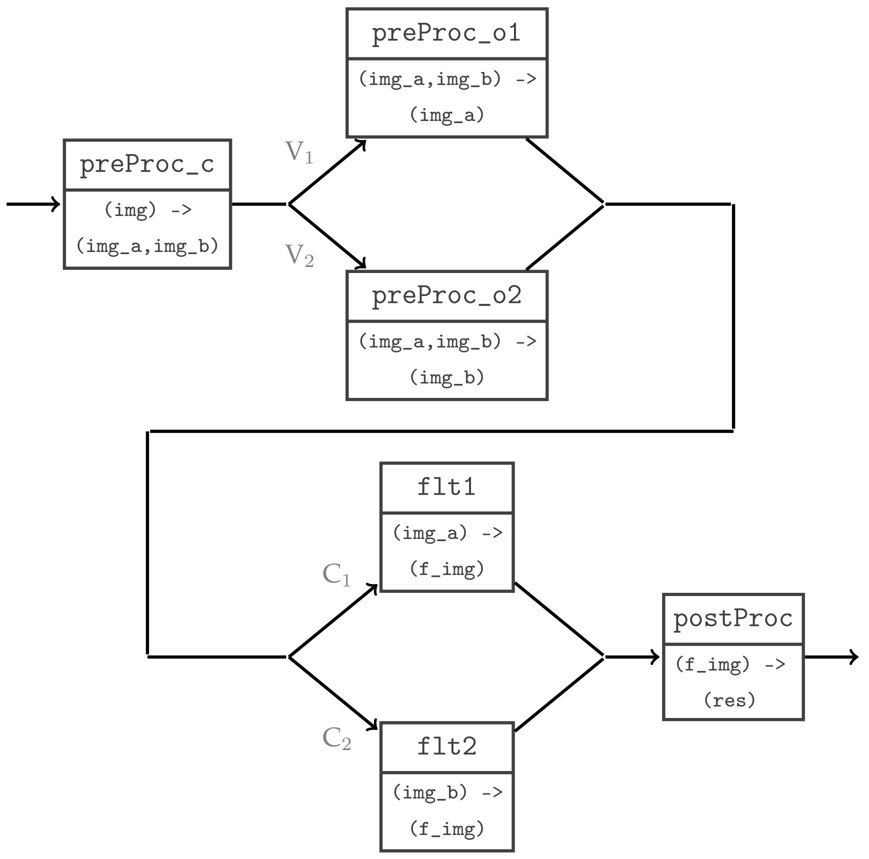
into its disjunctive normal form which is

```
  preProc_c ∧ preProc_o1 ∧ flt1 ∧ postProc
∨ preProc_c ∧ preProc_o2 ∧ flt1 ∧ postProc
∨ preProc_c ∧ preProc_o1 ∧ flt2 ∧ postProc
∨ preProc_c ∧ preProc_o2 ∧ flt2 ∧ postProc
```

The last phase of this transformation lifts all separate routes which are currently connected by the outer choice tuples to the top level. $T_{Seq'}^{Seq}\left(T_\tau^{Seq'}(\mathcal{E})\right)$ also concatenates the markers of choice nodes during the lifting process. This is done in such a way that the resulting string of markers precisely describes the route through the network by stating each decision that would have to be taken at runtime to construct the route one after the other. The separate routes as they are kept in the top-level node after this transformation are shown in Fig. 6.5(top) with their newly constructed marker strings on the left.

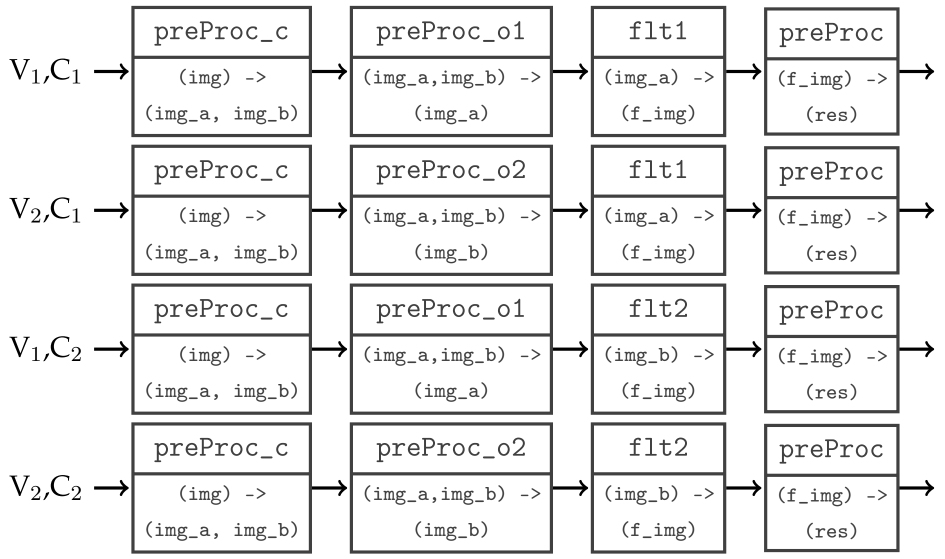
6.4.1 Transformation Rules

The goal of the following transformations is to produce a program that does not contain any choice combinators anymore and adheres to the syntax shown in Syntax 17. Instead of choice combinators a top-level substnode is introduced which keeps the alternative routes and associates a marker string with each of them.



```
(Seq₂,
 (Box₂, "preProc_c", {img}, {img_a,img_b}),
 (Seq₂,
  ((Seq₂,
    (Choice₂,
     (Box₂, "preProc_o1", {img_a,img_b},{img_a}),
     V₁,
     (Box₂, "preProc_o2", {img_a,img_b},{img_b}),
     V₂)),
    (Choice₂,
     (Box₂, "flt1", {img_a},{f_img}),
     C₁,
     (Box₂, "flt2", {img_b},{f_img}),
     C₂)),
 (Box₂, "postProc", {f_img},{res}))
)
```

Figure 6.4: The example application of Fig. 6.1 after Transformation 2 has been applied. The introduced markers are shown on the edges in the graphical representation (top) and inside the Choice tuples (bottom).



```
(Choice2,
 (Choice2,
  (Seq2, "preProc_c",
   (Seq2, (Seq2, "preProc_o1", "flt1"), "postProc")),
  C1,
  (Seq2, "preProc_c",
   (Seq2, (Seq2, "preProc_o1", "flt2"), "postProc")),
  C2),
 V1,
 (Choice2,
  (Seq2, "preProc_c",
   (Seq2, (Seq2, "preProc_o2", "flt1"), "postProc")),
  C1,
  (Seq2, "preProc_c",
   (Seq2, (Seq2, "preProc_o2", "flt2"), "postProc")),
  C2),
 V2)
```

Figure 6.5: The example of Fig. 6.4 after a transformation has moved all Choice combinations to the outer levels. The marker strings as they are constructed from the individual markers of Choice tuples (bottom) are shown on the left to their corresponding path (top).

Syntax 17. *In this representation multiple routes are collected in a dedicated tuple.*

$$\begin{array}{lcl}
\textit{Expr3} & \Rightarrow & (\ \textbf{Box}_\mathbf{3},\ \textit{Id}\ ,\ \textit{Rec}\ ,\ \textit{Rec}\ ) \\
 & | & (\ \textbf{Seq}_\mathbf{3},\ \textit{Expr3}\ ,\ \textit{Expr3}\ ) \\
 & | & (\ \textbf{Subst}_\mathbf{3},\ \{\ [\ \textit{EMPair}[,\ \textit{EMPair}\ ]^*\ ]\ \}\ \ ) \\
\textit{EMPair} & \Rightarrow & (\ \textit{Expr3},\ \textit{MarkerStr}\ ) \\
\textit{MarkerStr} & \Rightarrow & [\textit{Marker}\ ]+
\end{array}$$

Transformation 3. *The transformation function* $T_{Expd}(\mathcal{E})$ *implements a one-level expansion of sequential composition inside a choice composition. To apply the expansion rules until a fixed-point is reached* $T_{\mathcal{T}}^{Seq'}(\mathcal{E})$ *drives the transformation using the rules of* $T_{Expd}(\mathcal{E})$*. The third transformation must only be applied to a fully expanded representation as in* $T_{Seq'}^{Seq}\left(T_{\mathcal{T}}^{Seq'}(\mathcal{E})\right)$*. The result is a top-level* **Subst₃** *node that contains a collection of sequential compositions. Each of the compositions has a marker string associated with it that is built up during the transformation by concatenating individual markers of the sub-expressions.*

$$T_{Expd}(\mathcal{E}) = \begin{cases}
\dfrac{\mathcal{E} = (\mathbf{Box_2}, id, r_{in}, r_{out})}{\mathcal{E} \rightarrow (\mathbf{Box_2}, id, r_{in}, r_{out})} \\[2ex]
\dfrac{\begin{array}{c}\mathcal{E} = (\mathbf{Choice_2}, l, m, r, m') \\ l \rightarrow l' \qquad r \rightarrow r'\end{array}}{\mathcal{E} \rightarrow (\mathbf{Choice_2}, l', m, r', m')} \\[2ex]
\dfrac{\mathcal{E} = (\mathbf{Seq_2}, (\mathbf{Choice_2}, sl, M_1, sr, M_2), r)}{\mathcal{E} \rightarrow (\mathbf{Choice_2}, (\mathbf{Seq_2}, sl, r), M_1, (\mathbf{Seq_2}, sr, r), M_2))} \\[2ex]
\dfrac{\mathcal{E} = (\mathbf{Seq_2}, l, (\mathbf{Choice_2}, sl, M_1, sr, M_2))}{\mathcal{E} \rightarrow (\mathbf{Choice_2}, (\mathbf{Seq_2}, l, sl), M_1, (\mathbf{Seq_2}, l, sr), M_2)} \\[2ex]
\dfrac{\begin{array}{c}\mathcal{E} = (\mathbf{Seq_2}, l, r) \\ l \neq (\mathbf{Choice_2}, sl, m_1, sr, m_2) \wedge r \neq (\mathbf{Choice_2}, sl', m_1', sr', m_2') \\ l \rightarrow l' \qquad r \rightarrow r'\end{array}}{\mathcal{E} \rightarrow (\mathbf{Seq_2}, l', r',)}
\end{cases}$$

$$
T^{Seq'}_{\tau}(\mathcal{E}) = \begin{cases} \dfrac{\begin{array}{c} E = T_{Expd}(\mathcal{E}) \\ E \neq \mathcal{E} \end{array}}{\mathcal{E} \to T^{Seq'}_{\tau}(E)} \\[2em] \dfrac{T_{Expd}(\mathcal{E}) = \mathcal{E}}{\mathcal{E} \to \mathcal{E}} \end{cases}
$$

$$
T^{Seq}_{Seq'}(\mathcal{E}) = \begin{cases} \dfrac{\mathcal{E} = (\mathbf{Box_2}, id, r_{in}, r_{out})}{\mathcal{E} \to (\mathbf{Subst_3}, \{((\mathbf{Box_3}, id, r_{in}, r_{out}), \epsilon)\})} \\[2em] \dfrac{\mathcal{E} = (\mathbf{Seq_2}, l, r)}{\mathcal{E} \to (\mathbf{Subst_3}, \{((\mathbf{Seq_3}, l, r), \epsilon)\})} \\[2em] \dfrac{\begin{array}{c} \mathcal{E} = (\mathbf{Choice_2}, l, m, r, m') \\ l \to (\mathbf{Subst_3}, \{(e_{l_i}, m_{l_i})\}_{i=1}^{n}) \\ r \to (\mathbf{Subst_3}, \{(e_{r_j}, m_{r_j})\}_{j=1}^{m}) \end{array}}{\mathcal{E} \to (\mathbf{Subst_3}, \{(e_{l_i}, m \wedge m_{l_i})\}_{i=1}^{n} \cup \{(e_{r_j}, m' \wedge m_{r_j})\}_{j=1}^{m})} \end{cases}
$$

6.5 Transformation from S ·· Net to S-Net$_\lambda$

After the previous transformations have been applied a program is a collection of sequential compositions of simple boxes. This transformation is now targeted exclusively at boxes. The objective is to replace the occurrence of a box by something that is closer to a representation that the inference algorithm for extensible records can work on. The basic record operations introduced in Sect. 5.1.4 will turn out to be the essential ingredients in this transformation. Basic record operations, and more specifically, a composition of these, can be used to mimic the alteration of record structures in the same way as an application of a box would.

Because of flow inheritance it is helpful to think of a box not as something that consumes, but rather that as something that removes certain labels from a record and then adds new labels to the remainder of the record. This view very naturally captures flow-inheritance where everything that a box does not consume is carried over and made part of produced results.

This view is consistent with the model of extensible records and is reflected in the definition of the basic record operations. Only two of these operations, or more specifically compositions of two operations, are sufficient to describe the effect that any box might have on a record.

Extension	$\mathtt{add}_l :: \forall \alpha, r : (r \setminus l) \Rightarrow \alpha \to r \to (\!| l : \alpha | r |\!)$
Restriction	$\mathtt{rem}_l :: \forall \alpha, r : (r \setminus l) \Rightarrow (\!| l : \alpha | r |\!) \to r$

To model a box in terms of these two operations, we use one restriction operation to remove it from a record for each label that appears on the left-hand side of a box. The labels on the right-hand side of a box are added one at a time by extension operations. For example, the boxes

```
box preProc((img) -> (img_a, img_b))
box flt1((img_a) -> (f_img))
```

may be modelled by the two compositions of basic record operations

$$\begin{aligned} \lambda r &\quad . \quad (\mathtt{add}_{\mathtt{img\_a}}\ (\mathtt{add}_{\mathtt{img\_b}}\ (\mathtt{rem}_{\mathtt{img}}\ r))) \\ \lambda r &\quad . \quad (\mathtt{add}_{\mathtt{f\_img}}\ (\mathtt{rem}_{\mathtt{img\_a}}\ r)) \end{aligned}$$

A sequential composition of the two boxes as in `preProc .. flt1` boils down to a simple composition of the two functions:

$$\begin{aligned} & (\lambda r.(\mathtt{add}_{\mathtt{f\_img}}\ (\mathtt{rem}_{\mathtt{img\_a}}\ r))(\lambda r.(\mathtt{add}_{\mathtt{img\_a}}\ (\mathtt{add}_{\mathtt{img\_b}}\ (\mathtt{rem}_{\mathtt{img}}\ r))))) \\ = \ & \lambda r.(\mathtt{add}_{\mathtt{f\_img}}\ (\mathtt{rem}_{\mathtt{img\_a}}\ (\mathtt{add}_{\mathtt{img\_a}}\ (\mathtt{add}_{\mathtt{img\_b}}\ (\mathtt{rem}_{\mathtt{img}}\ r)))))) \end{aligned}$$

6.5.1 Transformation Rules

The formal definition of this transformation that turns boxes into compositions of basic row operations is given in Transformation 4, the syntax definition for the new representation is given in Syntax 18.

Syntax 18. *In this representation boxes are replaced by expressions of basic record operations and their types.*

Expr4	⇒	**(Box$_4$,** *Id* **,** *ErecExpr* **,** *TAssgn* **)**
	|	**(Seq$_4$,** *Expr4* **,** *Expr4* **)**
	|	**(Subst$_4$,** { *[EMPair[, EMPair]*]* } **)**

Transformation 4. *An* S ·· Net *program $\mathcal{E}$ is transformed by $T^{Core}_{Seq}(\mathcal{E})$ into a program representation in* S-Net$_\lambda$*. The function comprises several cases, one for each possible syntactic construct in* S ·· Net*.*

$$T^{Core}_{Seq}(\mathcal{E}) = \begin{cases} \dfrac{\mathcal{E} = (\mathbf{Box_3}, id, \{in_k\}_{k=1}^{n}, \{out_l\}_{l=1}^{m})}{\begin{array}{l}\mathcal{E} \to (\mathbf{Box_4},\ \ id, \\ \qquad\qquad \lambda r.(\mathtt{add}_{out_1}(\ldots(\mathtt{add}_{out_m} \\ \qquad\qquad\quad (\mathtt{rem}_{in_1}(\ldots(\mathtt{rem}_{in_n}\ \ r)\ldots), \\ \qquad\qquad \{(\mathtt{rem}_{in_k} :: \tau_{\mathtt{rem}_{in_k}})\}_{k=1}^{n} \\ \qquad\qquad \cup\{(\mathtt{add}_{out_l} :: \tau_{\mathtt{add}_{in_k}})\}_{l=1}^{m})\end{array}} \\ \\ \dfrac{\mathcal{E} = (\mathbf{Seq_3}, l, r) \quad l \to l_4 \quad r \to r_4}{\mathcal{E} \to (\mathbf{Seq_4}, l_4, r_4)} \\ \\ \dfrac{\mathcal{E} = (\mathbf{Subst_3}, \{(e_i, m_i)\}_{i=1}^{n}) \quad \forall i \in \{1, \ldots, n\} : e_i \to e_{i_4}}{\mathcal{E} \to (\mathbf{Subst_4}, \{(e_{i_4}, m_i)\}_{i=1}^{n})} \end{cases}$$

6.6 Transformation from S-NET$_\lambda$ to S-NET$_{TY}$

This transformation is the last link in the chain from user-annotated code to a fully typed S-NET program. Finally, we will use the type inference algorithm W of Sect. 5.1.3 to infer types for individual routes. As of the last transformation, all routes are encoded as compositions of basic record operations. This transformation operates bottom-up, i.e. box types are inferred first. Based on the box types the transformation then infers types for entire routes, i.e. serial compositions of boxes, using the same inference algorithm.

6.6.1 Type Inference for Boxes and Sequential Combinations

Type inference for boxes is a straight-forward process as boxes have been translated into expressions comprising basic record operations by the previous transformation 4. The box's record transformation, together with their initial type assignments, are passed directly as input to type inference algorithm W. For example,

```
box flt1((img_a) -> (f_img));
```

has up to this transformation been translated into

$$(\mathbf{Box_4},\ \mathtt{flt1},\ \underbrace{\lambda r.(\mathtt{add}_{\mathtt{f\_img}}\ (\mathtt{rem}_{\mathtt{img\_a}}\ r))}_{E_{flt1}},\ \underbrace{\{\mathtt{rem}_{\mathtt{img\_a}} :: \tau_{\mathtt{rem}_{\mathtt{img\_a}}}, \mathtt{add}_{\mathtt{f\_img}} :: \tau_{\mathtt{add}_{\mathtt{f\_img}}}\}}_{A_{flt1}}).$$

From this representation we can extract everything that is necessary to infer a type for this box using algorithm W. The algorithm computes a set of predicates P and a type

T from a given type assignment and a given expression. The result of this is

$$\begin{array}{rcl} P|A_{flt1} & \vdash & E_{flt1} :: T \\ \textbf{where} & & \\ P & = & \{(r' \setminus \texttt{f\_img})\} \\ T & = & (\!|\texttt{img\_a} : F \mid r'|\!) \to (\!|\texttt{f\_img} : F \mid r'|\!) \end{array}$$

where r' is a new type variables that is introduced during the process.

Sequential compositions are dealt with in an equally straight-forward way. What we need to do is to apply the right operand of a sequential composition to the result of the left operand, i.e. we are dealing with function composition. Let's assume we are dealing with a sequential composition of box

```
box preProc( (img) -> (img_a, img_b));
```

and box `flt1` from above as in `preProc .. flt1`. The tuple representation of box `preProc` at this stage is

$$(\textbf{Box}_\textbf{4},\quad \texttt{preProc},\quad \underbrace{\lambda r.(\texttt{add}_{\texttt{img\_a}}\ (\texttt{add}_{\texttt{img\_b}}\ (\texttt{rem}_{\texttt{img}}\ r)))}_{E_{preProc}},$$
$$\underbrace{\{\texttt{rem}_{\texttt{img}} :: \tau_{\texttt{rem}_{\texttt{img}}}, \texttt{add}_{\texttt{img\_a}} :: \tau_{\texttt{add}_{\texttt{img\_a}}}, \texttt{add}_{\texttt{img\_b}} :: \tau_{\texttt{add}_{\texttt{img\_b}}}\}}_{A_{preProc}}).$$

and the tuple representation of `flt1` is as above. In the language of extensible records we may implement a sequential composition of the two boxes as

```
let preProc = $E_{preProc}$
in
  let flt1 = $E_{flt\_1}$
  in
    $\lambda r.$(flt1 (preProc $r$))
```

If we refer to this expression as $E_{preProc..flt\_1}$ then inferring a type for this sequential composition comes down to an application of algorithm W again, this time using the joined type assignment $A_{preProc..flt\_1} = A_{flt1} \cup A_{preProc}$ of the sub-expressions E_{flt1} and $E_{preProc}$:

$$\begin{array}{rcl} P|A_{preProc..flt\_1} & \vdash & E_{preProc..flt\_1} :: T \\ \textbf{where} & & \\ P & = & \{((\!|\texttt{img\_b} : \texttt{F} \mid \texttt{r}'|\!) \setminus \texttt{f\_img}), \\ & & ((\!|\texttt{img\_a} : \texttt{F} \mid \texttt{r}'|\!) \setminus \texttt{img\_b}), \\ & & (r' \setminus \texttt{img\_a})\} \\ T & = & (\!|\texttt{img} : F \mid r'|\!) \to (\!|\texttt{f\_img} : F,\ \texttt{img\_b} : F \mid r'|\!) \end{array}$$

We may simplify the constraints of P by removing the statically provable restrictions and use a shorthand notation for constraints for better readability. We also generalise

(Def. 34) the inferred qualified type to a type scheme by $Gen(A_{preproc..flt\_1}, P \Rightarrow T)$. We have reached a major result. We have computed a type for a compound S-NET expression based on algorithm W for extensible records.

$$\begin{aligned}\texttt{preProc..flt1} :: \forall r.(r \setminus \texttt{img\_a}, \texttt{img\_b}, \texttt{f\_img}) \Rightarrow \\ (\!|\texttt{img} : F \mid r|\!) \rightarrow (\!|\texttt{f\_img} : F,\ \texttt{img\_b} : F \mid r|\!)\end{aligned}$$

By recursively applying this construction mechanism we generate one expression for an entire branch. Inferring a type for each branch is achieved by an application of W to the expression that we have constructed in this way.

6.6.2 Transformation Rules

Syntax 19 shows the extended tuple syntax for the program representation that stores the inferred types of the extensible record algorithm. **Expr** are only allowed in the intermediate representation.

Transformation 5 comprises two functions. $T_{Core}^{TY}(\mathcal{E})$ assigns to each element of the top-level **Subst₅**node a type scheme of the form $\forall r.P \Rightarrow T$. The constraint set P may contain predicates that are statically disprovable. This is in particular the case if a predicate is of the form $(\!|l \mid r|\!) \setminus l$ or in general if for predicate $(\!|l' \mid r|\!) \setminus l$ it holds that $l \neq l'$ but $l \in r$. In order to filter out all those type schemes that contain such predicates in their predicate sets we use the predicate entailment $\Vdash$ of Def. 18. We attempt to construct a proof for all predicates within a set P under the assumption that the row variable does not contain the label l of the predicate in question. For example, for

$$\begin{aligned} P \quad = \quad & \{((\!|\texttt{img\_b} : \texttt{F} \mid \texttt{r}'|\!) \setminus \texttt{f\_img}), \\ & ((\!|\texttt{img\_a} : \texttt{F} \mid \texttt{r}'|\!) \setminus \texttt{img\_b}), \\ & (r' \setminus \texttt{img\_a})\} \end{aligned}$$

we are able to positively infer that

$$\begin{aligned} \{(r' \setminus \texttt{f\_img})\} & \Vdash ((\!|\texttt{img\_b} : \texttt{F} \mid \texttt{r}'|\!) \setminus \texttt{f\_img}) \\ \{(r' \setminus \texttt{img\_b})\} & \Vdash ((\!|\texttt{img\_a} : \texttt{F} \mid \texttt{r}'|\!) \setminus \texttt{img\_b}) \\ \{(r' \setminus \texttt{img\_a})\} & \Vdash (r' \setminus \texttt{img\_a}) \end{aligned}$$

whereas for a predicate set

$$P \quad = \quad \{((\!|\texttt{img\_b} : \texttt{F} \mid \texttt{r}'|\!) \setminus \texttt{img\_b})\}$$

this is not the case as

$$\{(r' \setminus \texttt{img\_b})\} \Vdash ((\!|\texttt{img\_b} : \texttt{F} \mid \texttt{r}'|\!) \setminus \texttt{img\_b})$$

is not derivable.

These checks are carried out by the second transformation $T_{TY}^{TYC}(\mathcal{E})$. Only the elements of the **Subst₅** node that contain predicate sets for which proofs are constructible in the way described above are kept in the collection of routes.

Syntax 19. *Sequential compositions are represented by nested expressions for which types are inferred and stored in dedicated tuples.*

Expr5	$\Rightarrow$	*ExprNode*
	$\mid$	**(** **Subst$_5$,** **{** *[TypNode[, TypNode]*]* **}** **)**
ExprNode	$\Rightarrow$	**(** **Expr,** *Id* **,** *ErecExpr* **,** *TAssgn* **)**
TypNode	$\Rightarrow$	**(** **Typ,** *Id* **,** *Scheme* **,** *MarkerStr* **)**

Transformation 5. *The transformation $T^{TY}_{Core}(\mathcal{E})$ translates a program $\mathcal{E}$into* S-NET$_{TY}$*. The resulting set of types may be checked for the presence of unsatisfiable constraints by $T^{TYC}_{TY}\left(T^{TY}_{Core}(\mathcal{E})\right)$.*

$$T^{TY}_{Core}(\mathcal{E}) = \begin{cases} \dfrac{\mathcal{E} = (\mathbf{Box_4}, id, E, A)}{\mathcal{E} \to (\mathbf{Expr}, id, E, A)} \\[2em] \dfrac{\begin{array}{c}\mathcal{E} = (\mathbf{Seq_4}, l, r) \\ l \to (\mathbf{Expr}, id_l, E_l, A_l) \qquad r \to (\mathbf{Expr}, id_r, E_r, A_r)\end{array}}{\begin{array}{ll}\mathcal{E} \to & (\mathbf{Expr}, id_l \,\hat{}\, id_r, \\ & \texttt{let}\ id_l = E_l \\ & \texttt{in let}\ id_r = E_r \\ & \quad \texttt{in}\ \lambda r.(id_r\ (id_l\ r)), \\ & A_l \cup A_r)\end{array}} \\[2em] \dfrac{\begin{array}{c}\mathcal{E} = (\mathbf{Subst_4}, \{(e_i, m_i)\}^n_{i=1}) \\ e_i \to (\mathbf{Expr}, id_i, E_i, A_i) \\ P_i | A_i \vdash E_i :: T_i \\ Q_i = Gen(A_i, P_i \Rightarrow T_i)\end{array}}{\mathcal{E} \to (\mathbf{Subst_5}, \{(\mathbf{Typ}, id_i, Q_i, m_i)\}^n_{i=1})} \end{cases}$$

$$T^{TYC}_{TY}(\mathcal{E}) = \begin{cases} \dfrac{\begin{array}{c}\mathcal{E} = (\mathbf{Subst_5}, S) \\ Q = \{P \mid (\mathbf{Typ}, id, \forall r.P \Rightarrow T, m) \in S\} \\ C = \{P \mid P \in Q \ \wedge\ \forall (p \setminus l) \in P : \{r \setminus l\} \Vdash (p \setminus l)\} \\ S' = \{q \mid q = (\mathbf{Typ}, id, \forall r.P \Rightarrow T, m) \wedge q \in Q \wedge P \in C\}\end{array}}{\mathcal{E} \to (\mathbf{Subst_5}, S')} \end{cases}$$

Theorem 4. *The presented algorithm infers the most general type for each route.*

Proof. Direct consequence of Theorem 3. □

6.7 Route Analysis and Adjustment

At this stage the type inference for the routes within the transformed program is complete. Each of the individual routes has been fully processed and a function type, if it exists, has been annotated. From this collection of routes and their types we have to construct the network type of the original program. Thus, whereas in the earlier transformation, we dissolved components into independent paths we here develop a mechanism that re-associates the individual paths with each other. This is not a straight-forward process as we have to check various constraints when combining types of individual routes to sets of route types which ultimately represent a complete network signature of the original network.

The constraints arise from assuming freedom of choice where it didn't exist. The transformations that we used to model synchro cells and variant boxes as choice combinations introduced auxiliary routes. A box with multiple right-hand sides may produce any of its annotated outputs as response to an input. In the transformed program, however, the various outputs are modelled as different routes. This is potentially hazardous: The type inference process may have been unable to prove that all constraints hold. This entails a removal of the route from the set of routes which in turn means that a certain output variant of a box has no representation anymore. In the original program, however, these choices do not exist and the box that this route was a part of may very well decide to produce a record of this type nonetheless.

Let's reconsider the program of Fig. 6.1. Assume that we have a new version of one of the filter boxes. This new version, however, requires a different input format, which is why it does not accept `img_a` or `img_b` but `img_c` instead:

```
box flt3( (img_c) -> (res));
```

Because of some oversight we replace `flt2` by this new version without making the necessary changes to box `preProc`. The new network that we have produced now is shown in Fig. 6.6(top). In this case the `img_b` output of box `preProc` has nowhere to go. Box `flt1` is limited to handling records that carry an `img_a` label and the new filter box requires records to bring an `img_c`. After all transformations have been applied, however, we are effectively dealing with four programs R_1, R_2, R_3, R_4 (see Fig. 6.6(bottom)) as we treat the variant output of box `preProc` as separate routes and we model the alternatives between the filters again as two routes. The type inference algorithm that is applied during Transformation 5 will succeed for R_1, R_3, R_4 but not for R_2. We indicate this in Fig. 6.6 by check marks for legal and crosses for illegal routes. The inferred types for the individual routes are:

$$\begin{aligned}
R_1 \;&::\; \forall r : r \setminus \texttt{img}, \texttt{img\_a}, \texttt{img\_b}, \texttt{f\_img}, \texttt{res} \Rightarrow (\!|\texttt{img} : F \;|r|\!) \rightarrow (\!|\texttt{res} : F \;|r|\!) \\
R_3 \;&::\; \forall r : r \setminus \texttt{img}, \texttt{img\_a}, \texttt{img\_b}, \texttt{img\_c}, \texttt{f\_img}, \texttt{res} \Rightarrow \\
&\qquad\qquad (\!|\texttt{img} : F,\ \texttt{img\_c} : F \;|r|\!) \rightarrow (\!|\texttt{res} : F,\ \texttt{img\_a} : F \;|r|\!) \\
R_4 \;&::\; \forall r : r \setminus \texttt{img}, \texttt{img\_a}, \texttt{img\_b}, \texttt{img\_c}, \texttt{f\_img}, \texttt{res} \Rightarrow \\
&\qquad\qquad (\!|\texttt{img} : F,\ \texttt{img\_c} : F \;|r|\!) \rightarrow (\!|\texttt{res} : F,\ \texttt{img\_b} : F \;|r|\!)
\end{aligned}$$

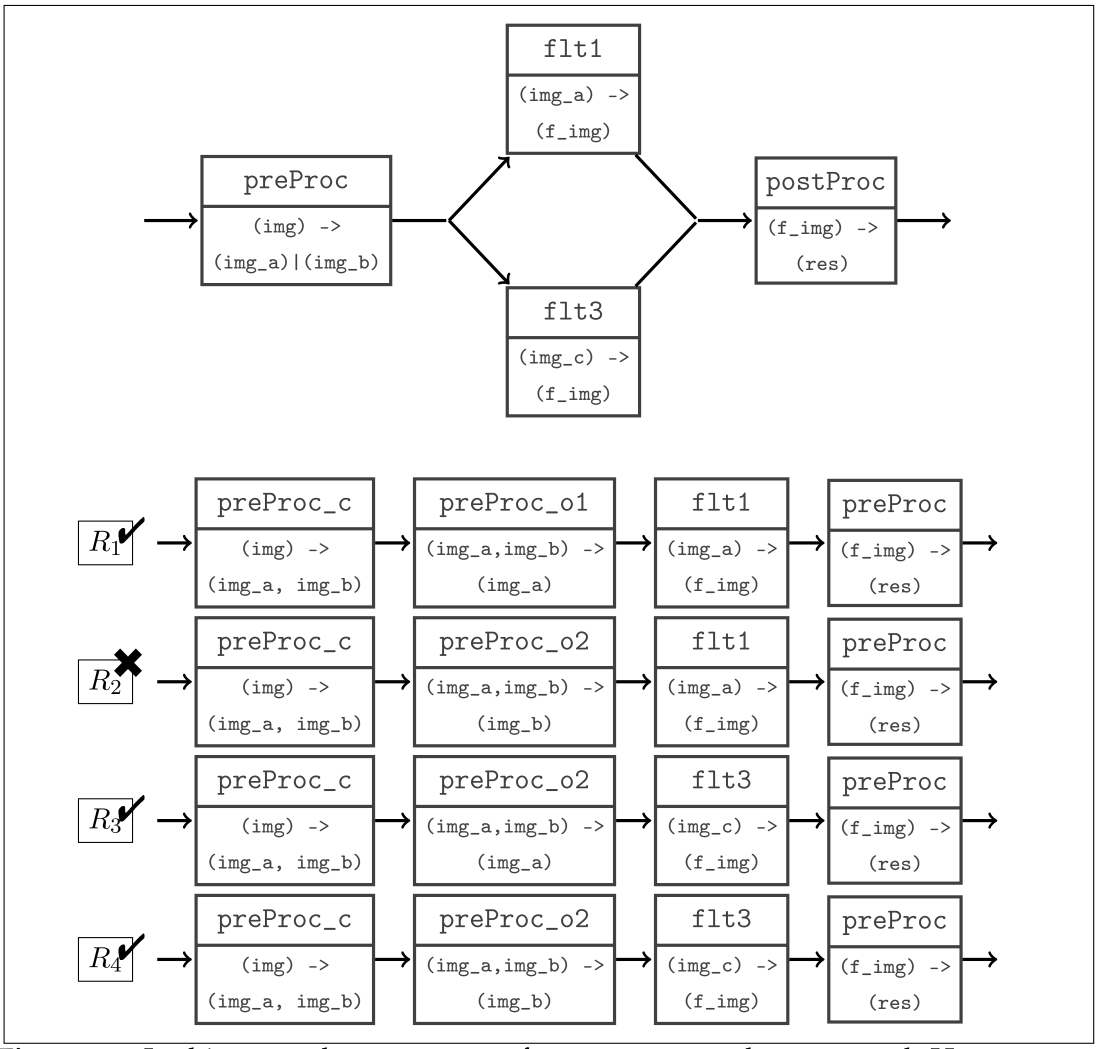


Figure 6.6: In this example one output of `preProc` cannot be processed. However, after decomposition (bottom) there exist several legal routes and one illegal route.

The result is of course not a proper reflection of what the program can actually process. It illegally suggests that a record {img} can be accepted as input. If box preProc is given such a record and if it then produces img_b as output, then the output record is left without any possible route to go.

This is where the different encodings of marker elements, introduced in Sect. 6.4, come into play. Routes that have been introduced to replace boxes with multiple right-hand sides are identifiable by V_1 and V_2 markers at choice nodes first and also in the constructed marker string of a route. These elements do not only identify the origin of a route as being one box, but they also identify the exact variant that a route represents. This information allows us to check if all variants of the original box are represented by at least one route for a particular input type. We note here that it is not sufficient to check if all variants are covered by just *any* route as it is possible that one variant is covered by some input type α and another variant is covered by a route of input type $\beta \neq \alpha$, which still leaves us with the same undesired situation. If this isn't the case, i.e. not all box variants are covered, all routes belonging to this one box should be marked as illegal and removed from the final result.

For an example like the previous one, a simple approach that detects uncovered variants per input type and removes the remaining routes would ensure that no input is accepted that potentially leads to a record for which no continuation exists. However, such aggressive measures restrict the input type of a network unnecessarily. Consider the example of Fig. 6.7 in which one of the filter boxes requires an additional parameter mask that defines the filter mask to be applied. For this network we can infer types for R_1 and R_4:

$$\begin{aligned} R_1 \;&::\; \forall r : r \setminus \texttt{img}, \texttt{img\_a}, \texttt{img\_b}, \texttt{mask}, \texttt{f\_img}, \texttt{res} \Rightarrow \\ &\qquad (\!| \texttt{img} : F,\ \texttt{mask} : F \mid r |\!) \rightarrow (\!| \texttt{res} : F \mid r |\!) \\ R_4 \;&::\; \forall r : r \setminus \texttt{img}, \texttt{img\_a}, \texttt{img\_b}, \texttt{f\_img}, \texttt{res} \Rightarrow \\ &\qquad (\!| \texttt{img} : F,\ \mid r |\!) \rightarrow (\!| \texttt{res} : F \mid r |\!) \end{aligned}$$

Neither of the two input types covers both box variants. Applying the same approach of removing the routes is unnecessarily restrictive. A simple down-coercion of R_4 onto R_1's input type, i.e. using

$$\begin{aligned} R_4 \;&::\; \forall r : r \setminus \texttt{img}, \texttt{img\_a}, \texttt{img\_b}, \texttt{mask}, \texttt{f\_img}, \texttt{res} \Rightarrow \\ &\qquad (\!| \texttt{img} : F,\ \texttt{mask} : F \mid r |\!) \rightarrow (\!| \texttt{res},\ mask : F : F \mid r |\!) \end{aligned}$$

resolves the issue. If done so, R_1 and R_4 both have the same input type. This input type now covers both variants of box preProc which in turn means that no routes would have to be removed.

Finding a coercion that covers all variants was uncomplicated in the previous example. Although only one of the filters required an additional parameter by ensuring that *every* record carries the extra field all decisions of the box were made safe. Thus, for the two routes where one input type was a subtype of the other, a down-coercion onto the more specific one was all that was required.

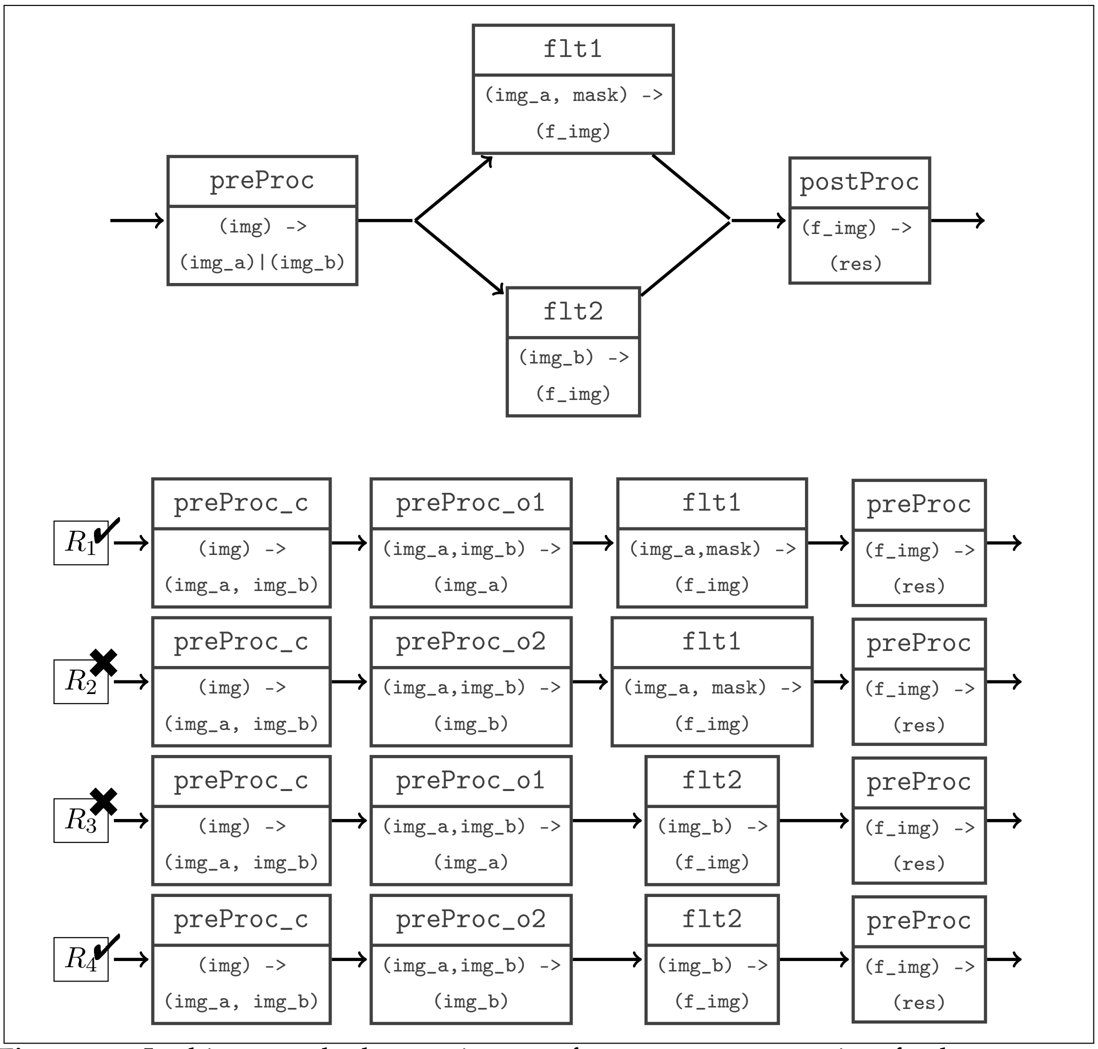


Figure 6.7: In this example the requirement for a parameter requires further attention. After decomposition (bottom) there exist two legal routes and two illegal routes.

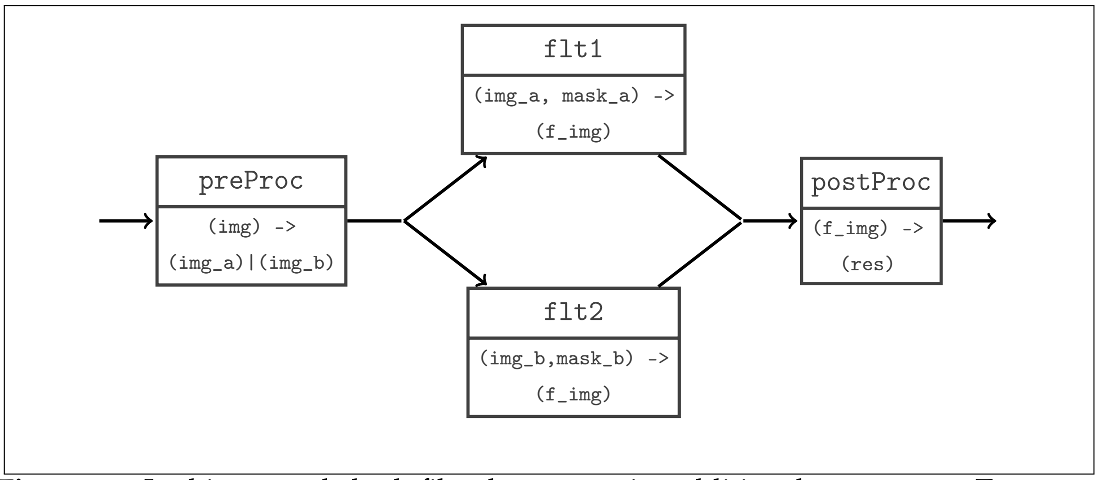


Figure 6.8: In this example both filter boxes require additional parameters. Type inference succeeds for only two of the four possible routes.

However, there are cases where this simple approach, again, is not sufficient. Consider the network of Fig. 6.8 (as the decomposition of the network is very similar to that of Fig. 6.7 we do not repeat it here). The example uses two filter boxes that both require an extra parameter, filter `flt_1` requires `mask_a` and filter `flt_2` requires `mask_b`. The only two routes that we can infer a type for are R_1 and R_4:

$$\begin{aligned} R_1 \; &:: \; \forall r : r \setminus \mathtt{img}, \mathtt{img\_a}, \mathtt{img\_b}, \mathtt{mask\_a}, \mathtt{f\_img}, \mathtt{res} \Rightarrow \\ &\qquad (\!|\mathtt{img} : F,\ \mathtt{mask\_a} : F \;|r|\!) \rightarrow (\!|\mathtt{res} : F \;|r|\!) \\ R_4 \; &:: \; \forall r : r \setminus \mathtt{img}, \mathtt{img\_a}, \mathtt{img\_b}, \mathtt{mask\_b}, \mathtt{f\_img}, \mathtt{res} \Rightarrow \\ &\qquad (\!|\mathtt{img} : F,\ \mathtt{mask\_b} : F \;|r|\!) \rightarrow (\!|\mathtt{res} : F \;|r|\!) \end{aligned}$$

The two input types of R_1 and R_2 are unrelated, neither is a subtype of the other and consequently the coercion approach of the previous example cannot be reused as it is. What is required here is a coercion onto a common subtype of the two input types of the routes:

$$\begin{aligned} R_1 \; &:: \; \forall r : r \setminus \mathtt{img}, \mathtt{img\_a}, \mathtt{img\_b}, \mathtt{mask\_a}, \mathtt{mask\_b}, \mathtt{f\_img}, \mathtt{res} \Rightarrow \\ &\qquad (\!|\mathtt{img} : F,\ \mathtt{mask\_a} : F,\ \mathtt{mask\_b} : F \;|r|\!) \rightarrow (\!|\mathtt{res} : F,\ \mathtt{mask\_b} : F \;|r|\!) \\ R_4 \; &:: \; \forall r : r \setminus \mathtt{img}, \mathtt{img\_a}, \mathtt{img\_b}, \mathtt{mask\_a}, \mathtt{mask\_b}, \mathtt{f\_img}, \mathtt{res} \Rightarrow \\ &\qquad (\!|\mathtt{img} : F,\ \mathtt{mask\_a} : F,\ \mathtt{mask\_b} : F \;|r|\!) \rightarrow (\!|\mathtt{res} : F,\ \mathtt{mask\_a} : F \;|r|\!) \end{aligned}$$

This simultaneous coercion transforms the program that previously contained unsafe box decisions for all its routes into a safe program. The challenge in cases like this, although it was straight-forward in this example, is to find the smallest common subtype that exposes this property.

Clearly, a selective mechanism is called for that automates the identification process of unsafe routes. Also, such a mechanism should not blindly remove sets of unsafe inputs but it should first try to coerce these into safe sub-types wherever possible.

We will develop an algorithm that achieves this in several stages. Before covering the algorithm step-by-step let us first identify the required mechanisms and develop a general idea of how to achieve our goal. From what we have seen in the previous examples, we need to arrange routes into groups depending on their input types. Within each group, we may then identify potentially unsafe routes for treatment. This treatment should involve coercion of routes into other groups if this turns previous unsafe routes into safe ones. If no such group exists, it should also include creating new groups for common sub-types of already existing groups. Only if coercion then still leaves unresolved safety hazards, offending routes should be discarded.

6.7.1 Algorithms for Route Analysis and Adjustment

As we will see, an infrastructure that permits easy navigation on subtype hierarchies is a key ingredient to a concise algorithm that identifies and amends unsafe routes. We need a structure that expresses subtype relations between input types of routes that gives us easy access from a given route r to all routes whose input types are sub-types of the input type of r. A graph whose nodes represent input types with directed edges between nodes that stand in a subtype relation fulfills our requirements:

Definition 38. *A* subtype graph *is a directed graph $G = (N, E)$ for which elements of N are tuples (τ, C) where τ is a record type and C is a (potentially empty) set of* **Typ** *tuples such that the type scheme as contained in the* **Typ** *tuple in C is of the form $\forall r.P \Rightarrow \tau \rightarrow \beta$. Furthermore, E contains an edge between two nodes*

$$((\tau_1, C_1), (\tau_2, C_2)) \in E \iff \tau_2 \sqsubseteq \tau_1 \wedge \tag{6.1}$$

$$\nexists(\tau, C) \in N : \tau \neq \tau_2 \Rightarrow \tau_2 \sqsubseteq \tau \sqsubseteq \tau_1 \tag{6.2}$$

Theorem 5. *For a subtype-graph structure $G = (N, E)$ as defined in Def. 38 the following properties hold:*

1. *The graph is acyclic.*
2. *The graph is complete with respect to $\sqsubseteq$, i.e. for any two elements $(\tau_1, C_1), (\tau_2, C_2) \in N$ for which $\tau_2 \sqsubseteq \tau_1$, there exists a path from (τ_1, C_1) to (τ_2, C_2).*
3. *The number of edges in E is minimal, i.e. if an edge is removed from E then property 2. is violated.*

Proof. We will show that the properties hold for each of these separately:
1. direct consequence of the properties of $\sqsubseteq$ (order relation) and construction rule Def. 38(6.1)
2. by definition 38(6.1)
3. assume the opposite, then there exist $(m = (m_\tau, m_C), n = (n_\tau, n_C)) \in E$ for which there is at least one alternative path between m and n of length $k > 1$. Let this path be $m, m^1, \ldots, m^{k-1}, n$. By definition $n_\tau \sqsubseteq m_\tau^{k-1} \sqsubseteq \ldots \sqsubseteq m_\tau^1 \sqsubseteq m_\tau$, which contradicts Def. 38(6.2). □

After we have inferred types for individual routes of a program we can express the relationship between the input types of all routes by arranging the resulting **Typ** tuples into a subtype-graph as defined above. Such a subtype-graph contains a node for each unique input type for a given set of routes. There may be several entry points into the structure, which is why we demand that the set of input types under consideration *always* contains the empty record type $\{\!\mid\mid\!\}$. If for a set of routes this is not the case, we shall add $(\{\!\mid\mid\!\}, \{\})$ to the set of nodes. This will give us one single entry point to the graph structure as of course all other input types are sub-types of $\{\!\mid\mid\!\}$. There will still be several sinks in the graph, i.e. nodes that only have inbound edges but no outbound edges. In preparation for later stages of the algorithm where coercion into common sub-types may become necessary, we extend the subtype-graph to a *type lattice* as described in Def. 41.

Definition 39. *A* type lattice *is a subtype-graph $G = (N, E)$ for which the following, additional properties hold:*

$$(\{\!\mid\mid\!\}, C) \in N \textit{for some (potentially empty) } C \tag{6.3}$$

$$\forall(\alpha, C_\alpha), (\beta, C_\beta) \in N \;\; \exists(\gamma, C_\gamma) \in N : \alpha \sqsubseteq \gamma \wedge \beta \sqsubseteq \gamma \tag{6.4}$$

Definition 40. *Let α, β be two record types and let S be the substitution computed by the algorithm of Def. 31, i.e. $\alpha \overset{S}{\sim} \beta$. Then $\gamma = S\alpha = S\beta$ is called the* least common subtype *of α and β with $\gamma \sqsubseteq \alpha$ and $\gamma \sqsubseteq \beta$. We will use $scs(\alpha, \beta)$ as shorthand notation for the smallest common subtype of α and β.*

Definition 41. *Let P be a set of a set of* **Typ** *tuples for of a given program. Furthermore, let N_0 be a set of tuples $\{(\tau_i, C_I)\}_{i=1}^{n}$ as defined in Def. 38 covering all elements of P and let $T_0 = \{\alpha \mid (\alpha, C_\alpha) \in N_0\}$. For $\mathbb{N} \ni i \geq 1$ we define*

$$T_i = \{scs(\alpha, \beta) \mid \alpha \in T_{i-1} \wedge \beta \in T_{i-1}\} \cup T_{i-1} \tag{6.5}$$

$$N_i = \{(\tau, \emptyset) \mid \tau \in T_i\} \cup N_{i-1} \tag{6.6}$$

Choose $k \in \mathbb{N}$ such that $N_k = N_{k+1}$ and a set of edges $E \subset N_k \times N_k$ that satisfies the properties of Def. 38. Then we call $G = (N_k, E)$ the corresponding type lattice *of P, denoted $G^P_\sqsubseteq$.*

In order the get a better feel for working with type lattices let's consider the network shown in Fig. 6.9. The network uses all of the filters that we have been using in previous examples at the same time. After decomposition there are eight routes in total as box `preProc` offers two different outputs and the choice over the filter boxes comprises four cases. Type inference applied to each of the cases succeeds for four routes. This time we will also include strings that are composed of the markers denoting routing decisions (constructed by Transformation 3) with each route, as we will need this information later. The markers for each routing and box decision are shown in Fig. 6.9 on the respective branches and with the corresponding routes. We use the markers for identification by concatenating these to marker strings. Read from left to right each element represents one choice decision. The inferred types for the typeable routes are

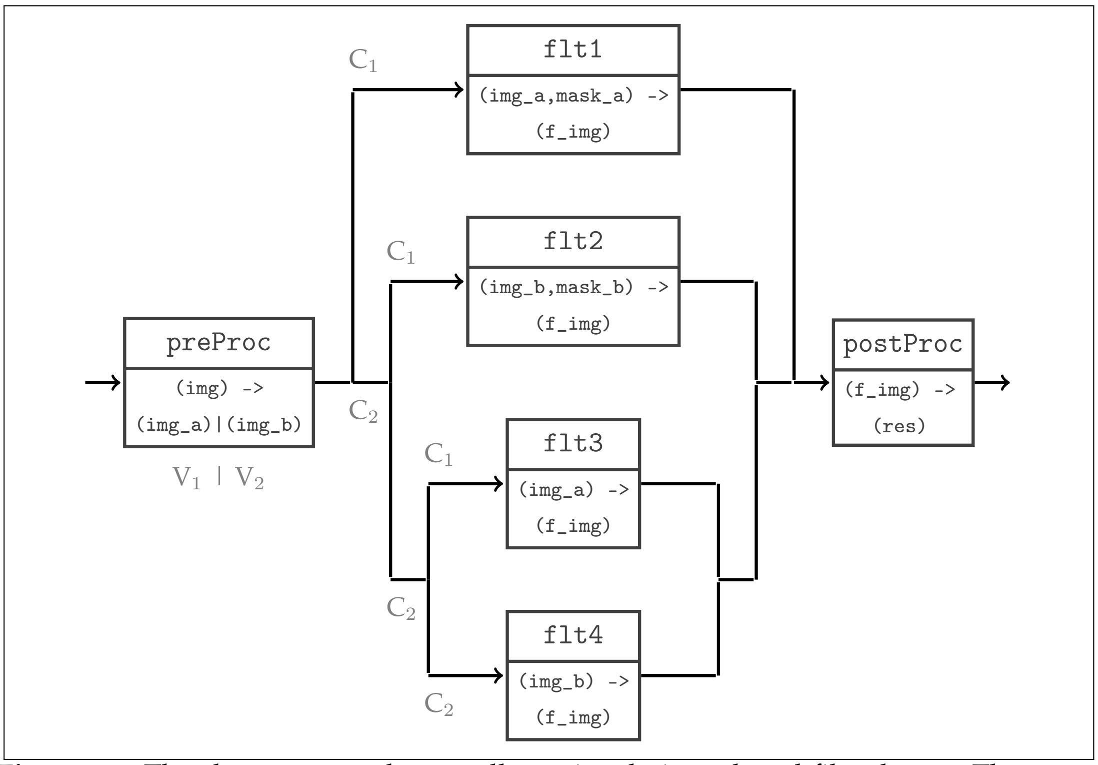


Figure 6.9: The shown network uses all previously introduced filter boxes. The markers that are associated with the choice combinations are shown on top and below their respective branches.

as follows; routes are shown with marker strings that correspond to the markers in Fig. 6.9.

$$\begin{aligned}
\langle \mathrm{V}_1,\mathrm{C}_1\rangle \quad &:: \quad \tau_1 = \forall r : r \setminus \mathtt{img}, \mathtt{img\_a}, \mathtt{img\_b}, \mathtt{mask\_a}, \mathtt{f\_img}, \mathtt{res} \Rightarrow \\
&\qquad (\!|\mathtt{img}: F,\ \mathtt{mask\_a}: F,\ \mid r|\!) \to (\!|\mathtt{res}: F \mid r|\!) \\
\langle \mathrm{V}_1,\mathrm{C}_2,\mathrm{C}_2,\mathrm{C}_1\rangle \quad &:: \quad \tau_3 = \forall r : r \setminus \mathtt{img}, \mathtt{img\_a}, \mathtt{img\_b}, \mathtt{f\_img}, \mathtt{res} \Rightarrow \\
&\qquad (\!|\mathtt{img}: F,\ \mid r|\!) \to (\!|\mathtt{res}: F \mid r|\!) \\
\langle \mathrm{V}_2,\mathrm{C}_2,\mathrm{C}_1\rangle \quad &:: \quad \tau_6 = \forall r : r \setminus \mathtt{img}, \mathtt{img\_a}, \mathtt{img\_b}, \mathtt{mask\_b}, \mathtt{f\_img}, \mathtt{res} \Rightarrow \\
&\qquad (\!|\mathtt{img}: F,\ \mathtt{mask\_b}: F \mid r|\!) \to (\!|\mathtt{res}: F,\ \mid r|\!) \\
\langle \mathrm{V}_2,\mathrm{C}_2,\mathrm{C}_2,\mathrm{C}_2\rangle \quad &:: \quad \tau_8 = \forall r : r \setminus \mathtt{img}, \mathtt{img\_a}, \mathtt{img\_b}, \mathtt{f\_img}, \mathtt{res} \Rightarrow \\
&\qquad (\!|\mathtt{img}: F,\ \mid r|\!) \to (\!|\mathtt{res}: F \mid r|\!)
\end{aligned}$$

The subtype graph for this program contains three nodes, one for each possible input type $(\!|\mathtt{img}: F|\!)$, $(\!|\mathtt{img}: F,\ \mathtt{mask\_a}: F|\!)$, $(\!|\mathtt{img}: F,\ \mathtt{mask\_b}: F|\!)$. This collection is then completed into a subtype lattice by adding all common sub-types, which in this case is only $(\!|\mathtt{img}: F,\ \mathtt{mask\_a}: F,\ \mathtt{mask\_b}: F|\!)$ and the empty record type $(\!||\!)$. The resulting subtype lattice is shown in a graphical representation in Fig. 6.10. The routes for each type are displayed in a box which is connected to the associated type in the lattice. Added nodes that complete the graph to a lattice have no corresponding routes and consequently only show $\emptyset$ in their boxes.

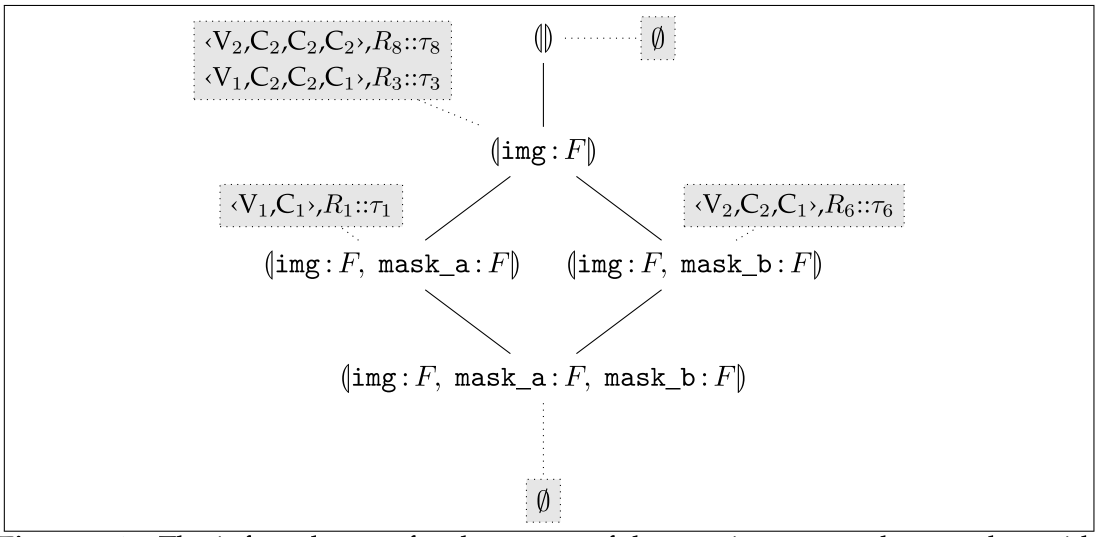


Figure 6.10: The inferred types for the routes of the previous examples together with added common sub-types form a type lattice as shown. The input type of a route is used to associate the route with a node in the lattice.

This structure allows easy access to routes of the same input type and to their common sub- and super-types. After having found a way to order routes depending on their input types, we are now developing a mechanism to identify and amend routes that include unsafe decisions. In order to achieve this, we again employ a structure that arranges routes into a structure that allows for easy access to the properties of interest, which in this case are the possible routing-influencing decisions at every stage. Routing decisions that are involved for any particular route have been encoded by previous transformations by means of markers that are kept together with the inferred type of the route in **Typ** tuples. We will use the string of markers to define a (directed) graph structure in which the marker elements are used for navigation. Each node in this marker graph represents one routing decision in such a way that routes that share a common prefix of their marker string will also share a common path in the graph. Where a path represents the complete marker string of a route, this route is kept at the node the last edge of the path is pointing to (Def. 43).

For illustration purposes assume that we are working with this (contrived) set of routes $R_1, \ldots, R_4$ and their marker strings:

R_1	‹C$_1$, C$_1$›
R_2	‹C$_1$, C$_2$, V$_1$›
R_3	‹C$_1$, C$_2$, V$_2$›
R_4	‹C$_2$, C$_1$, V$_1$›

The structure encoding the set as described above is shown in Fig. 6.11. On top

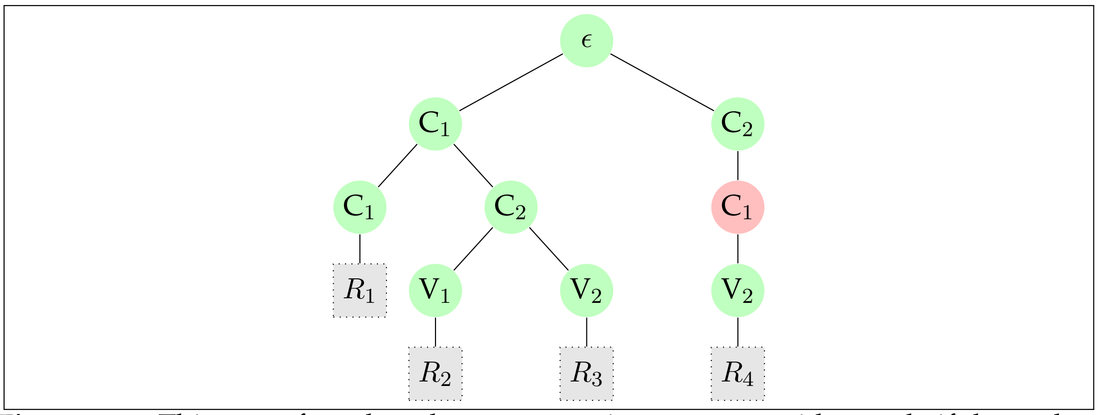


Figure 6.11: This tree of marker elements associates a route with a node if the path to the node resembles a route's marker string. If a node in the tree does not have all required box markers as successors it is coloured red and green otherwise.

of this structure, checking if a route contains unsafe decisions can be reduced to the problem of checking for the presence or absence of edges: If a node has either all or no edges for box-based decisions pointing away from it, i.e. edges that represent V$_1$ resp. V$_2$ as next symbols in a marker string, the node may be considered safe. The same property needs to hold for synchro cell decisions where the check requires all or none of the edges of relevant markers need (see Syntax 17) to be present. Consequently, a route may be considered safe if there exists a path that contains only safe nodes in the graph structure that corresponds to the route's marker string. Revisiting the example of Fig. 6.11 where safe nodes are coloured green and unsafe nodes are coloured red, we may say that routes $R_1, \ldots, R_3$ are safe and route R_4 is not.

Definition 42 introduces string operations and symbols for special alphabets as we will need these in the remainder of this section.

Definition 42. *We introduce the following notation for the alphabet of marker elements.*

$$\Sigma_{RV} = \{C_1, C_2, V_1, V_2, S_{1o}, S_{2o}, S_{1s}, S_{2s}, \epsilon\}$$

The set of strings of length $k \in \mathbb{N}$ over Σ_{RV} is denoted by

$$\Sigma_{RV}^k \text{ and } \bigcup_{k \in \mathbb{N}} \Sigma_{RV}^k = \Sigma_{RV}^\star$$

The length of string s is denoted by $|s|$ such that $|s_1 \ldots s_n| = n \in \mathbb{N}$ and $|\epsilon|$=0.
The set of prefixes of a string $s = s_1 \ldots s_n$ is denoted by

$$Pref(s) = \{\epsilon, s_1, s_1 s_2, \ldots, s_1 s_2 \ldots s_{n-1}\}$$

Definition 43. *A* marker graph *is a directed graph $G = (N, E)$ for which elements of N are tuples (s, C) where s is a string over $\Sigma_{RV}^\star$ and C is a (potentially empty) set of* **Typ** *tuples such that $\forall c \in C : \pi_3(c) = s$. Furthermore, the set of edges E contains an edge between two nodes*

$$((s_1, C_1), (s_2, C_2)) \in E \iff \exists x \in \Sigma_{RV} : s_2 = s_1 {}^\wedge x \tag{6.7}$$

Definition 44. *For a given set P of* **Typ** *tuples the* corresponding marker graph *G is a marker graph (Def. 43) for which N contains one node for each unique marker string that is present in the elements of P. In addition to those nodes, N also contains a node for each prefix of marker strings that are present in P: Let $S = \{\pi_3(p) \mid p \in P\}$ be the set of all unique marker strings and let*

$$N_P = \{(s, C) \mid s \in S \wedge C = \{p \mid p \in P \wedge s = \pi_3(p)\}\} \tag{6.8}$$

be the set of all nodes capturing the elements of P. Furthermore, let

$$N_x = \left\{ (q, \emptyset) \mid q \in \bigcup_{s \in S} Pref(s) \right\} \tag{6.9}$$

be the set of all nodes capturing possible prefixes of marker string. Then, $G = (N_P \cup N_x, E)$ with E as defined in Def. 43 is a corresponding marker graph.

Theorem 6. *A corresponding marker graph $G = (N, E)$ (Def. 44) is a directed tree with root node (ϵ, C).*

Proof. G is connected: From Def. 44(6.9) it follows that N contains a node for each possible prefix of marker strings. As direct consequence of Def. 43(6.7) it holds that $\forall n \in N \setminus \{(\epsilon, C)\} \exists q \in N : (q, n) \in E$. An edge (p, q) connects two nodes if the string of p is a prefix of q; as ϵ is by definition a prefix of every string a node (ϵ, C) cannot be the destination of an edge E (root node).

G contains no cycles, even if E is interpreted as set of undirected edges: Assume the opposite, then there is a pair of nodes $(p, C), (q, C')$ which are connected to each other by two distinct paths $p_1 \neq p_2$. As the nodes are connected one of the two strings p, q is a sub-string of the other, and thus, w.l.o.g. $q = p \,^\wedge\, s$ for $s \in \Sigma_{RV}^n$. From Def. 43(6.7) if follows that both paths are of the same length $n = |s|$ and from the definition of the set of nodes Defs. 44(6.8), 44(6.9) it follows that both paths must cover the same nodes, and consequently $p_1 = p_2$. Contradiction. $\square$

In order to perform the checks for the presence or the absence of edges pointing away from nodes in a corresponding marker graph, we introduce a red-green labelling of nodes in a marker graph. Nodes that represent unsafe decisions will be marked red, those nodes that represent safe decisions will be marked green. Colours lend themselves nicely to illustration purposes, but of course, red and green are nothing but attribute names which may be arbitrarily exchanged.

Definition 45. *Given a marker graph $G = (N, E)$ a* red-green labelling *of G's nodes is achieved by categorising the edges in E. Let the predicate*

$$P_x(s) \quad = \quad \exists((s, C_1), (s', C_2)) \in E : s' - s = x$$

be defined for all $x \in \Sigma_{RV}$. Furthermore, for $(s, C) \in N$ and $X_1 = \{S_{1s}, S_{1o}\}$, $X_2 = \{S_{2s}, S_{2o}\}$ and $X = X_1 \cup X_2$ let the following sets

$$B_1 = \{(s, C) \mid P_{V_1}(s) \wedge P_{V_2}(s)\} \tag{6.10}$$

$$B_2 = \{(s, C) \mid \neg(P_{V_1}(s) \vee P_{V_2}(s))\} \tag{6.11}$$

$$Y_1' = \left\{(s, C) \mid \bigwedge_{x \in X_1} P_x(s)\right\}$$

$$Y_1'' = \left\{(s, C) \mid \bigwedge_{x \in X_2} P_x(s)\right\}$$

$$Y_1 = Y_1' \cup Y_1'' \tag{6.12}$$

$$Y_2 = \left\{(s, C) \mid \neg\left(\bigvee_{x \in X} P_x(s)\right)\right\} \tag{6.13}$$

describe subsets of N. Set B_1 contains nodes that cover both outputs of variant boxes; B_2 contains nodes that do not contain any outputs of variant boxes; Y_1' contains nodes that cover the first pattern of a synchro cell, Y_1'' covers the second pattern; Y_1 contains the nodes of fully covered synchro cells, and finally, Y_2 contains nodes that do not contain any synchro cell marker.

Based on these sets we define the set of green nodes and red nodes as

$$\begin{aligned} Green &= (B_1 \cup B_2) \cap (Y_1 \cup Y_2) \\ Red &= N \setminus Green \end{aligned}$$

such that $G = (N, E) = (Green \cup Red, E)$.

Based on a red-green labelled graph we can formulate the following theorem that allows us to formally identify safe and unsafe routes.

Theorem 7. *Let* $G = (Green \cup Red, E)$ *be a red-green labelled corresponding marker graph. All elements* $c \in C$ *of a set of* **Typ** *tuples that are present at a node* $(s, C) \in N$ *of* G *represent routes that only involve safe decisions if there is a path* $P \subset E$ *from the root of* G *to* (s, C) *for which* $\forall (m, n) \in P : m \in Green \wedge n \in Green$.

Proof. As a red-green labelled corresponding marker graph is a tree (Theorem 6) any set of **Typ** tuples that is kept at a given node is reachable by exactly one unique path from the root node. The sets defined in Def. 45 represent all nodes that have associated with them either all or none of the edges for specific cases, i.e. B_1 contains nodes that have two out-going edges, one for each possible box output, B_2 contains nodes that have no box decisions associated with them, Y_1 contains nodes that contain edges for all of the four possible outputs of a synchro cell and Y_2 contains those nodes that have no involvement with synchro cell decisions. As $Green = (B_1 \cup B_2) \cap (Y_1 \cup Y_2)$ the set contains only those nodes that are safe with respect to box decisions and synchro cell decisions at the same time. Remaining proof via induction over the length of a path. □

With Theorem 7 a given set of **Typ** tuples can be partitioned into a set of safe and a set of unsafe routes. We will be using this in conjunction with a type lattice (Def. 41) as a means to identify candidates within such a graph for down-coercion where possible and complete removal where necessary. This process is described by the stabilisation algorithm in Transformation 6. The algorithm makes use of tree traversals and coercions which we specify in the following definitions first.

Definition 46. *A* breadth first traversal *of a graph* $G = (N, E)$ *with a dedicated starting node* $n \in N$, $BFS_n(G)$ *visits the nearest neighbours of* n *first. For each of these neighbours it visits those nodes' nearest neighbours and so on, until all nodes have been visited. Every node is visited exactly once.*

A breadth first traversal of a graph generally not uniquely defined. For example, for $N = \{1, 2, 3, 4, 5, 6, 7\}$, $n = 1$ and $E = \{(1, 3), (3, 5), (1, 2), (2, 4), (2, 6)\}$

$$\begin{tikzcd} & 1 \arrow[dl, dash] \arrow[dr, dash] & & \\ 3 \arrow[d, dash] & & 2 \arrow[dl, dash] \arrow[dr, dash] & \\ 5 & 4 & & 6 \end{tikzcd}$$

a possible result of $BFS_{(N,E)}(1)$ is $1, 3, 2, 5, 4, 6$, but so are $1, 2, 3, 6, 4, 5$ and $1, 2, 3, 4, 6, 5$ and so on. However, the list of nodes that is produced by a breadth first traversal contains the nodes in such an order that the distance from the starting node is monotonic increasing.

Definition 47. *Let $G = (N, E)$ be a graph and let $n \in N$. The* level *of a node $n' \in N$ with respect to n is the smallest number of edges that form a path between n and n'. We may also call this the* shortest distance *between n and n', denoted $\mathtt{d}(n, n')$. The distance of a node to itself is zero, i.e. $\forall n \in N : \mathtt{d}(n, n) = 0$.*

Theorem 8. *Let $G = (N, E)$ be a graph, $n_1 \in N$ and $BFS_{n_1}(G) = n_1, n_2, \ldots, n_k$. Then,*

$$BFS_{n_1}(G) = n_1, n_2, \ldots, n_k \Rightarrow \mathtt{d}(n_1, n_1) \le \mathtt{d}(n_1, n_2) \le \ldots \le \mathtt{d}(n_1, n_k)$$

Proof. Dijkstra's Algorithm [Dij59] (assume all edges have a cost of 1). □

Definition 48. *Let $G = (N, E)$ be a graph, $n_1 \in N$ and $BFS_{n_1}(G)$ be an arbitrary but fixed breadth first traversal of G. For $i \in \mathbb{N}$ we define $BFS_G(n_1; i)$ to be the i'th node that was visited during a breadth first traversal.*

Definition 49. *Let σ be a type scheme of the form $\forall r.P \Rightarrow \alpha \to \beta$ and let τ be a record type such that $\tau \sqsubseteq \alpha$. A* route coercion *of σ onto τ extends the input type α to τ and extends the set of predicates and the output type β accordingly.*

Let τ' be the type that is obtained by substituting $(\!|\,|\!)$ in τ by a type variable $v \notin TV(\sigma)$. Let S be a substitution such that $\alpha \overset{S}{\sim} \tau'$. Then we call

$$\sigma' = \forall v.SP \Rightarrow S\alpha \to S\beta$$

the result of the route coercion of σ onto τ.

We say σ' is a valid coercion if the set of predicates SP does not contain constraints that are statically known to be unsatisfiable. This may be determined by the same mechanism as was used in $T_{TY}^{TYC}(\mathcal{E})$ of Transformation 5.

We define

$$rcor(\sigma, \tau) = \begin{cases} \sigma' & \text{if } \sigma' \text{ is a valid coercion} \\ \text{undefined} & \text{otherwise} \end{cases}$$

with σ, σ', τ as above.

Transformation 6. *The* route stabilisation algorithm *identifies unsafe routes within a corresponding type lattice and selectively coerces the identified routes down the established subtype hierarchy level by level. If no further coercions are possible an identified unsafe route is removed entirely.*

Let $\overline{\wedge}$ denote the node for $(\!|\,|\!)$, i.e. the root node, of the corresponding type lattice. We also assume the breadth first traversal to be fixed.

$$T_{TYC}^{STAB}(\mathcal{E}) = \begin{cases} \dfrac{\begin{array}{c}\mathcal{E} = (\textbf{Set}, E, n, R) \\ (N, K) = G_{\sqsubseteq}^{E} \\ n \leq |N|\end{array}}{\mathcal{E} \to T_{TYC}^{STAB}((\textbf{Graph}, (N, K), n, R))} \\ \\ \dfrac{\begin{array}{c}\mathcal{E} = (\textbf{Graph}, (N, K), n, R) \\ v = BFS_{\overline{\wedge}}((N, K); n) \\ (Red \cup Green, K') = G_{\Sigma_{RV}}^{\pi_2(v)} \\ S = \{s \mid (v, s) \in K\} \\ C = \{rcor(\pi_2(r), \pi_2(s)) \mid r \in Red \wedge s \in S\}\end{array}}{\mathcal{E} \to T_{TYC}^{STAB}((\textbf{Set}, C \cup \pi_2(Red \cup Green), n + 1, R \cup Red))} \\ \\ \dfrac{\begin{array}{c}\mathcal{E} = (\textbf{Set}, E, n, R) \\ (N, K) = G_{\sqsubseteq}^{E} \\ n > |N|\end{array}}{\mathcal{E} \to E \setminus R} \end{cases}$$

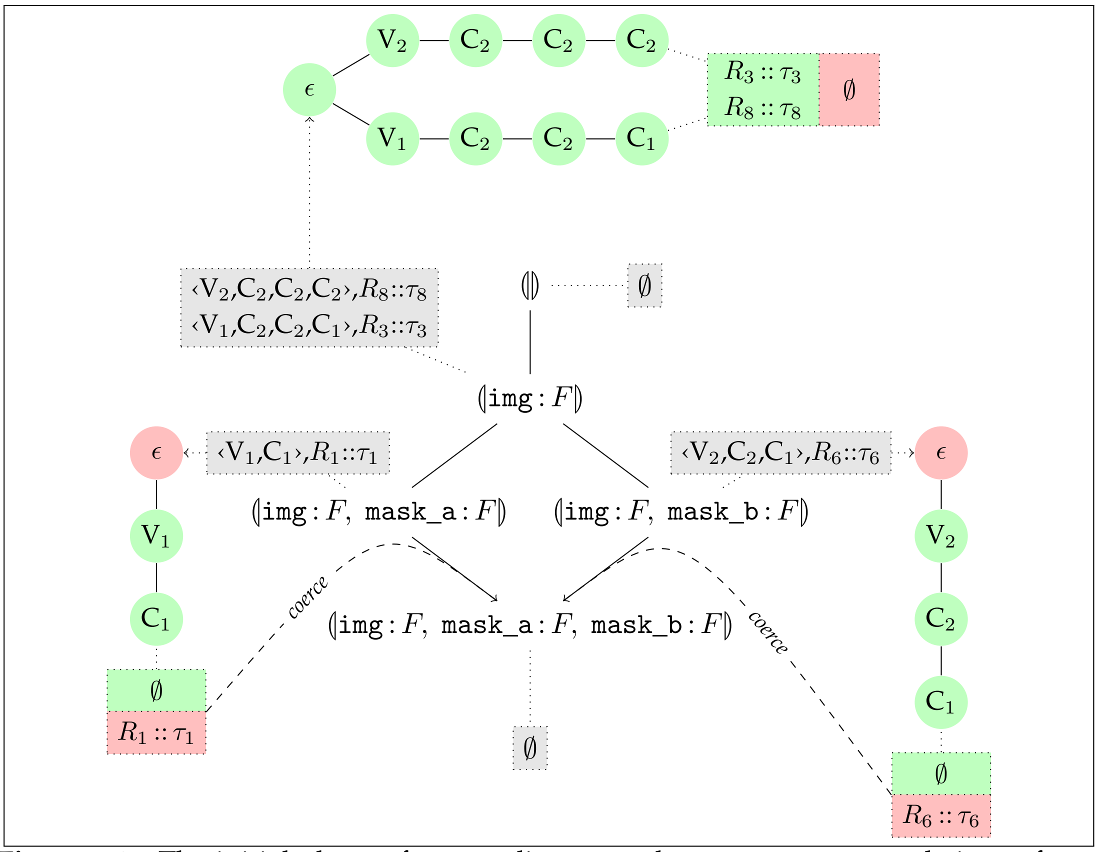
We illustrate the process using the previous example shown in Fig. 6.9. After the types for the individual routes have been inferred, the results are arranged into a type lattice as shown in Fig. 6.10. Starting from the root node of the structure, the associated sets of routes are checked one after another. The set located at node $(\!|\texttt{img}: F|\!)$ contains R_8 and R_3. For this set the red-green labelling of the corresponding marker graph reveals no problems. All nodes are safe and so are the two routes in question. No further action is required here (see top of Fig. 6.12). The next set that we examine is located at node $(\!|\texttt{img}: F,\ \texttt{mask\_a}: F|\!)$. For this set it turns out that the only route R_1 stored at this node is unsafe as no corresponding route for box decision V_2 can be found. The red set that is computed by the labelling algorithm is coerced down to all nearest neighbours of the current node. In this example the only neighbour is $(\!|\texttt{img}: F,\ \texttt{mask\_a}: F,\ \texttt{mask\_b}: F|\!)$. Coercion into its input type transforms τ_1 to τ_1' as

$$R_1 \quad :: \quad \tau_1 = \forall r : r \setminus \texttt{img}, \texttt{img\_a}, \texttt{img\_b}, \texttt{mask\_a}, \texttt{f\_img}, \texttt{res} \Rightarrow$$
$$(\!|\texttt{img}: F,\ \texttt{mask\_a}: F,\ |r|\!) \to (\!|\texttt{res}: F \mid r|\!)$$

is down-coerced to

$$R_1 \quad :: \quad \tau_1' = \forall r : r \setminus \texttt{img}, \texttt{img\_a}, \texttt{img\_b}, \texttt{mask\_a}, \texttt{mask\_b}, \texttt{f\_img}, \texttt{res} \Rightarrow$$
$$(\!|\texttt{img}: F,\ \texttt{mask\_a}: F,\ \texttt{mask\_b}: F,\ |r|\!) \to (\!|\texttt{res}: F,\ \texttt{mask\_b}: F \mid r|\!)$$

and R_1 with its newly coerced type is relocated to the appropriate node. We encounter a very similar situation at node $(\!|\texttt{img}: F,\ \texttt{mask\_b}: F|\!)$ where route R_6 has to be coerced



ε
V$_2$ C$_2$ C$_2$ C$_2$
V$_1$ C$_2$ C$_2$ C$_1$
$R_3 :: \tau_3$
$R_8 :: \tau_8$
∅
‹V$_2$,C$_2$,C$_2$,C$_2$›,R_8::τ_8
‹V$_1$,C$_2$,C$_2$,C$_1$›,R_3::τ_3
⦇⦈
∅
⦇img : F⦈
‹V$_1$,C$_1$›,R_1::τ_1
‹V$_2$,C$_2$,C$_1$›,R_6::τ_6
⦇img : F, mask_a : F⦈
⦇img : F, mask_b : F⦈
⦇img : F, mask_a : F, mask_b : F⦈
coerce
coerce
ε
V$_1$
C$_1$
∅
$R_1 :: \tau_1$
∅
ε
V$_2$
C$_2$
C$_1$
∅
$R_6 :: \tau_6$

Figure 6.12: The initial phase of route adjustment detects two routes as being safe and two routes as being unsafe. The coercions that occur during the next step are indicated by dashed lines.

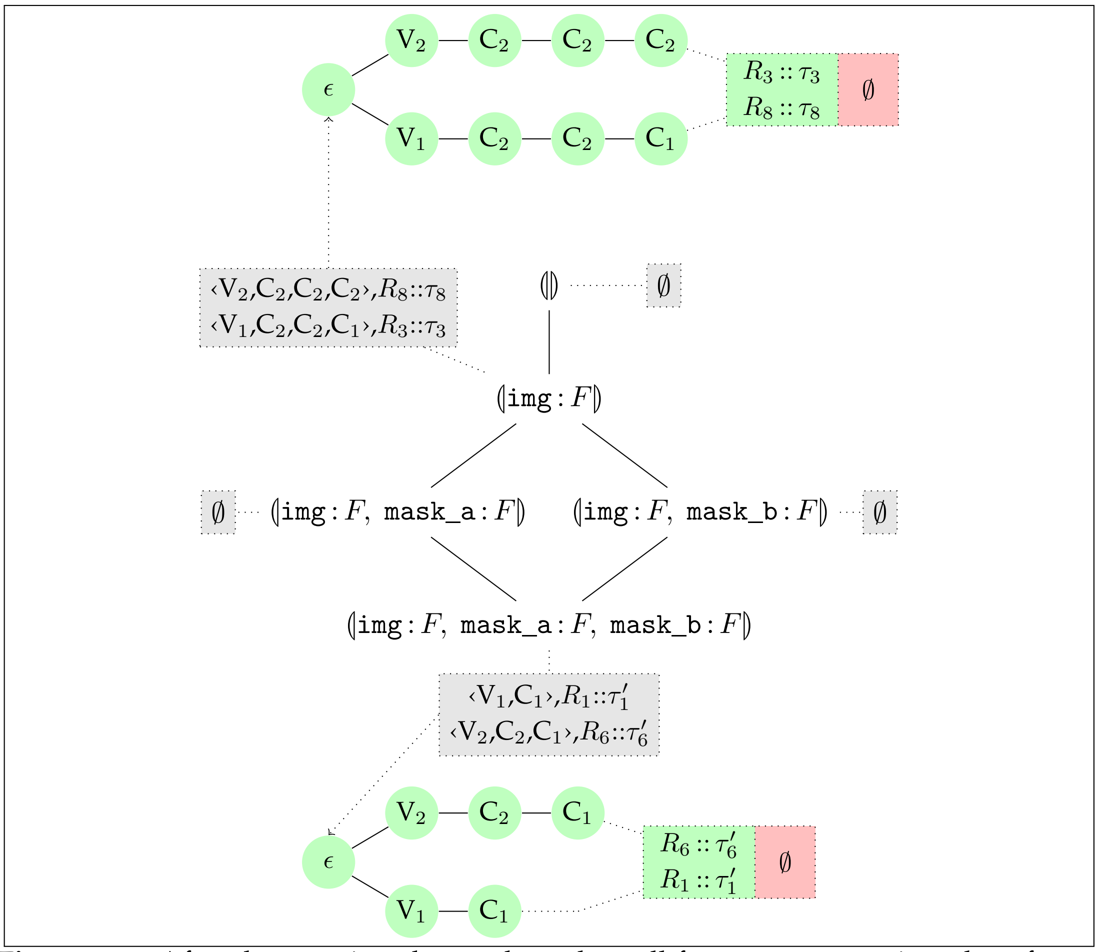


Figure 6.13: After the coercions have taken place all four routes contain only safe decisions. No further coercions are necessary.

down. The neighbour of this node is $(\!|\texttt{img}: F,\ \texttt{mask\_a}: F,\ \texttt{mask\_b}: F|\!)$ again and consequently the coercion results in

$$\begin{aligned} R_6 \quad :: \quad & \tau_6' = \forall r : r \setminus \texttt{img}, \texttt{img\_a}, \texttt{img\_b}, \texttt{mask\_a}, \texttt{mask\_b}, \texttt{f\_img}, \texttt{res} \Rightarrow \\ & (\!|\texttt{img}: F,\ \texttt{mask\_a}: F,\ \texttt{mask\_b}: F,\ |r|\!) \to (\!|\texttt{res}: F,\ \texttt{mask\_a}: F \mid r|\!) \end{aligned}$$

See Fig. 6.12(bottom) for an illustration.

That last remaining node to be checked is $(\!|\texttt{img}: F,\ \texttt{mask\_a}: F,\ \texttt{mask\_b}: F|\!)$ where the previously down-coerced routes R_1 and R_6 are now located, see Fig. 6.13. The red-green labelling of the marker graph now asserts that all routes located at this node are safe. As a result of this we do not have to coerce any of the routes further down the structure, which in this case would have led to the disposal of routes as the node does not have any further neighbours nodes.

After this last step all routes contained within the lattice have been confirmed to be safe. The resulting types for the network shown in Fig. 6.9 are:

$$\begin{aligned} R_1 \quad :: \quad & \tau_1' = \forall r : r \setminus \texttt{img}, \texttt{img\_a}, \texttt{img\_b}, \texttt{mask\_a}, \texttt{mask\_b}, \texttt{f\_img}, \texttt{res} \Rightarrow \\ & (\!|\texttt{img}: F,\ \texttt{mask\_a}: F,\ \texttt{mask\_b}: F,\ |r|\!) \to (\!|\texttt{res},\ \texttt{mask\_b}: F : F\ |r|\!) \\ R_3 \quad :: \quad & \tau_3 = \forall r : r \setminus \texttt{img}, \texttt{img\_a}, \texttt{img\_b}, \texttt{f\_img}, \texttt{res} \Rightarrow \\ & \qquad (\!|\texttt{img}: F,\ |r|\!) \to (\!|\texttt{res}: F\ |r|\!) \\ R_6 \quad :: \quad & \tau_6' = \forall r : r \setminus \texttt{img}, \texttt{img\_a}, \texttt{img\_b}, \texttt{mask\_a}, \texttt{mask\_b}, \texttt{f\_img}, \texttt{res} \Rightarrow \\ & (\!|\texttt{img}: F,\ \texttt{mask\_a}: F,\ \texttt{mask\_b}: F,\ |r|\!) \to (\!|\texttt{res},\ \texttt{mask\_a}: F : F\ |r|\!) \\ R_8 \quad :: \quad & \tau_8 = \forall r : r \setminus \texttt{img}, \texttt{img\_a}, \texttt{img\_b}, \texttt{f\_img}, \texttt{res} \Rightarrow \\ & \qquad (\!|\texttt{img}: F,\ |r|\!) \to (\!|\texttt{res}: F\ |r|\!) \end{aligned}$$

We close this section with the following theorem:

Theorem 9. *The algorithm presented in Transformation 6 computes the maximal set of safe routes from a given set E of* **Typ** *tuples on input* $\left(\mathbf{Graph}, G^E_{\sqsubseteq}, 1, \emptyset\right)$*, as*

1. *the algorithm always terminates;*
2. *the algorithm identifies and removes all unsafe routes within a given set of* **Typ** *tuples;*
2. *an unsafe route will be down-coerced to the smallest possible subtype of its input type with respect to the established subtype hierarchy;*
3. *only where no further coercion is possible the route is discarded.*

Proof. Starting from 1 the value of n strictly increments by 1 before any of the cases in $T^{STAB}_{TYC}(\mathcal{E})$ can be applied a second time. By design, the first two cases alternate until n is greater than the number of nodes in the subtype graph under consideration. Coercion is only done onto input types that already exist in the graph, and hence none of the cases increase the number of nodes. This proves (1.).

The structure of the subtype graph stays unchanged, and so every node of the graph is visited by $BFS_{G_{\sqsubseteq}=(V,E)}(\overline{\wedge};)i$ as i covers the full range of $1, \ldots, n = |V|$. At node i

a red-green labelling of the associated **Typ** tuples is carried out, which identifies the unsafe routes at this node (Theorem 7). Potentially, new **Typ** tuples are created by coercion, but by constructions these new elements will be associated with a node that is only visited by $BFS_G(\overline{\wedge}; j)$ for $j > i$ (Def. 48). The **Typ** tuples of unsafe routes are collected at each node and removed from the initial set in case three where the algorithm terminates. This proves (2.).

Coercion of unsafe routes at node v is only carried out onto the associated input-type of directly neighbouring nodes in $G_{\sqsubseteq}$ which represents the smallest possible sub-types of the current input type (Theorem 5 and Theorem 2). This proves (3.). As this process is repeated for all neighbouring nodes in $G_{\sqsubseteq}$ coerced **Typ** tuples are potentially coerced again further down the established hierarchy when the node is visited by $BFS_G(\overline{\wedge}; j)$. Only if no further sub-types exist, no coercion takes place as in case $S = \emptyset \Rightarrow C = \emptyset$.This proves (4.). □

6.8 The Type Inference Process at a Glance

We have achieved an important goal. Starting out from a S-Net program (without star combinators) we have a system in place that infers network signatures automatically. We have effectively turned S-Net into an implicitly typed language that allows us to drop all manual type annotations except for the signatures of boxes. Particularly for defining choice combinations this is a major simplification as branches do not need to be annotated with their branch types anymore. The required information is automatically inferred and inserted without any manual annotations provided by the programmer.

An overview of all the steps that were required in the process are shown in Fig. 6.14. In later section we will refer to this process as algorithm S.

6.9 Completing the Picture

In this section we deal with the remaining issues of the type inference process to complete it for all aspects of S-Net. This first of all demands a treatment of star combinators as the inference system so far does not support these. We introduce an iterative approach that, based on the inference system so far, infers signatures for star combinators in Sect. 6.9.1. This is followed by a transformation that allows us to re-introduce binding tags into the system in Sect. 6.9.2.

6.9.1 Type Inference for the star combinator

Operationally, the star combinator unfolds its operand network into a conceptually infinitely long sequential chain of replicas. The combinator is annotated with a pattern which resembles a record type. This pattern is used in conjunction with inbound records. If the type of an inbound record matches the annotated pattern of the com-

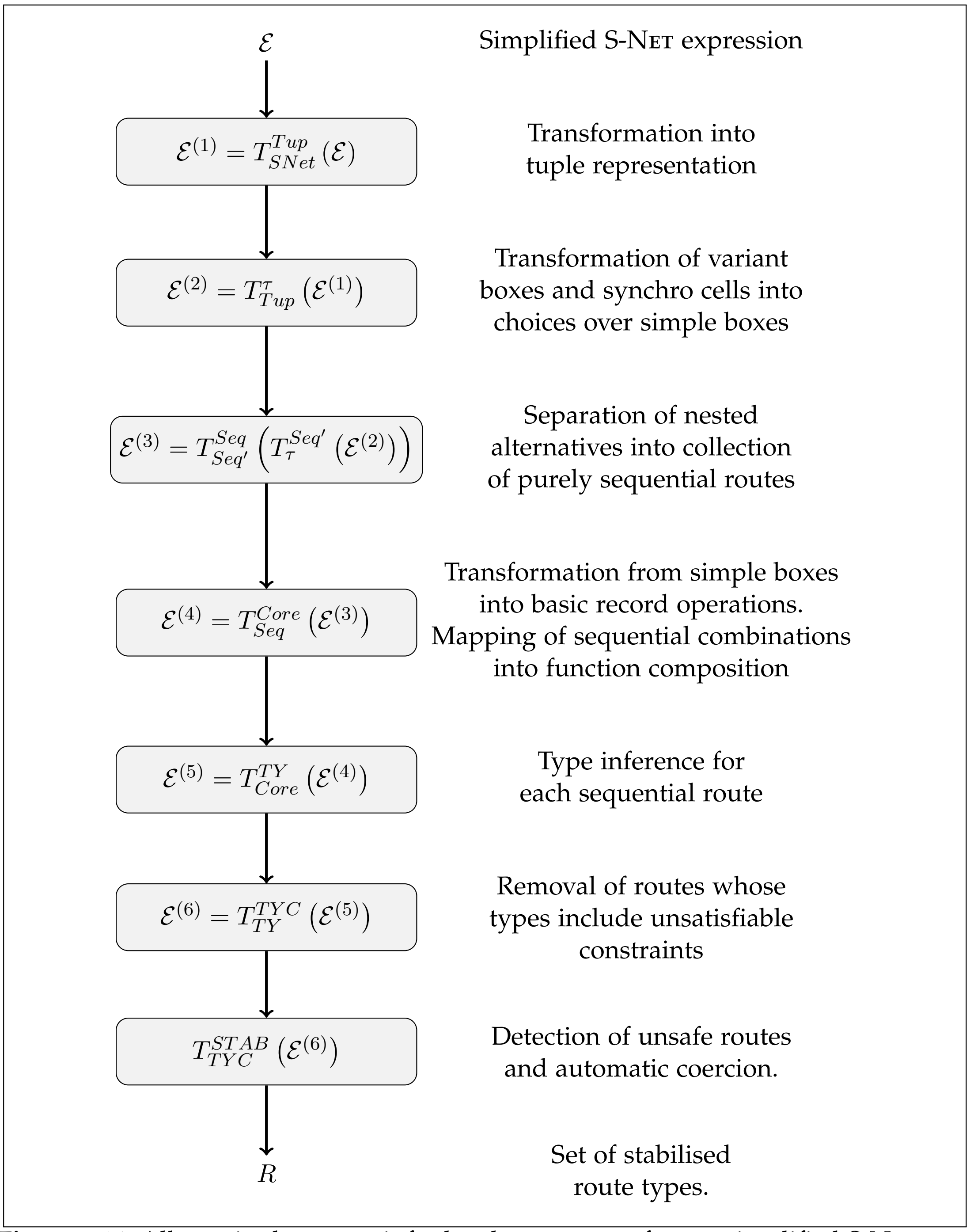

$\mathcal{E}$
Simplified S-Net expression
$\mathcal{E}^{(1)} = T^{Tup}_{SNet}\left(\mathcal{E}\right)$
Transformation into tuple representation
$\mathcal{E}^{(2)} = T^{\tau}_{Tup}\left(\mathcal{E}^{(1)}\right)$
Transformation of variant boxes and synchro cells into choices over simple boxes
$\mathcal{E}^{(3)} = T^{Seq}_{Seq'}\left(T^{Seq'}_{\tau}\left(\mathcal{E}^{(2)}\right)\right)$
Separation of nested alternatives into collection of purely sequential routes
$\mathcal{E}^{(4)} = T^{Core}_{Seq}\left(\mathcal{E}^{(3)}\right)$
Transformation from simple boxes into basic record operations. Mapping of sequential combinations into function composition
$\mathcal{E}^{(5)} = T^{TY}_{Core}\left(\mathcal{E}^{(4)}\right)$
Type inference for each sequential route
$\mathcal{E}^{(6)} = T^{TYC}_{TY}\left(\mathcal{E}^{(5)}\right)$
Removal of routes whose types include unsatisfiable constraints
$T^{STAB}_{TYC}\left(\mathcal{E}^{(6)}\right)$
Detection of unsafe routes and automatic coercion.
R
Set of stabilised route types.

Figure 6.14: All required steps to infer legal route types from a simplified S-Net program.

binator, i.e. if it is a subtype of it, the record is taken out of the sequential chain of operand instances and sent out of the star combination.

If the type of the record does not match the pattern, it is forwarded to the first operand instance in the chain. The results produced by the first operand are treated in the same way. If a result matches the pattern, it is taken out of the chain. If it doesn't, it is forwarded to the second instance of the operand within the chain. This is a recursive process and potentially continues indefinitely.

We may design the type inference for a star combinator following the ideas of the operational behaviour. In a first step, the type of the operand is inferred using algorithm S. In a second step the set of inferred schemes is partitioned into two sub-sets. The first set contains all schemes whose output types match the pattern. The second set contains those that do not match the pattern.

We repeat the process, but we use a sequential combination of the operand with itself as the new operand network; this procedure corresponds to the unfolding of the operand network into a chain of operands in the operational behaviour. This means that, during the first iteration the chain only contains the operand itself. In the second iteration we infer the type of the sequential composition of the operand with itself, in the third iteration we increases the number of operands in the sequential chain to three and so on.

The initial two sets of the matching and non-matching type schemes are carried over to the second iteration. After inferring the types for the first sequential composition we partition the set of results as before and add the two parts to their respective sets of the initial step. At each iteration we do exactly the same and add the partitioned result to their respective sets of the previous iteration such that both sets either grow or stay unchanged in each step. We never remove any elements from either of the sets.

At each iteration we can derive all legal input types and all possible output types from the set of matching schemes up to the current length of the sequential chain. The set of non-matching schemes defines all inputs and outputs that will stay within the sequential chain up to this level of unrolling of the operand. Thus, as long as an iteration adds to one of the sets, a further step will potentially yield more types for the matching set. Consequently, the inference process finishes after an iteration in which neither of the two sets grew, i.e. when the process has reached a fixed point.

Transformation Rules

We assume that the operand of the star combinator is of the form

```
  net N connect expr;
```

where $expr$ is adequate as input for type inference algorithm S.

Syntax 20. *We extend the syntax for $Expr1$ of Syntax 15 to allow for star combinators. The tuple contains, after the identifier, the original operand network of the combinator as its first element. The second element carries the growing, sequential combination of operand networks during the inference process. The pattern for checking record types is the third element, which is followed by the set for non-matching and for matching routes.*

Three dots are a placeholder for the remainder of the original syntax which we do not repeat here.

$$\begin{array}{lcl} \textit{Expr1} & \Rightarrow & \textbf{(}\ \textbf{Star}_\mathbf{1}\textbf{,}\ \textit{Expr1}\ \textbf{,}\ \textit{Expr1}\ \textbf{,}\ \textit{Rec}\ \textbf{,}\ \textit{RSet}\ \textbf{,}\ \textit{RSet}\ \textbf{)} \\ & \vdots & \\ \textit{RSet} & \Rightarrow & \{\ [\textit{Scheme}[,\ \textit{Scheme}\ ]^{*}\ ]\ \} \end{array}$$

Transformation 7. *The inference process for a star combinator uses algorithm S to infer types for the operand and then computes a fixed point at which neither the set N of input types for the next iteration nor the output set O grow by further applications of the algorithm.*

We extend Transformation 1 by a rule to translate expressions of star combinations. Three dots represent the remaining rules of the original transformation which we do not reproduce here.

$$T^{Tup}_{SNet}(\mathcal{E}) = \begin{cases} \dfrac{\begin{array}{c}\mathcal{E} = expr\ \texttt{*}\ p \\ Se = (\textbf{Choice}_\mathbf{1},\ expr,\ (\textbf{Box}_\mathbf{1},\ \texttt{StarExit}\,p,\ \{p\}))\end{array}}{\mathcal{E} \to (\textbf{Star}_\mathbf{1}, Se, Se, p, \emptyset, \emptyset)} \\ \\ \vdots \end{cases}$$

$$T^{Star}(\mathcal{E}) = \begin{cases} \dfrac{\begin{array}{c}\mathcal{E} = (\textbf{Star}_\mathbf{1}, e', e, p, N, O) \\ \Sigma = S(e') \\ N' = \{\overbrace{\forall r.P \Rightarrow \alpha \to \beta}^{\sigma} \mid \sigma \in \Sigma\ \wedge\ \beta \not\sqsubseteq p\} \\ N \neq (N \cup N')\ \vee\ O \neq (O \cup (\Sigma \setminus N'))\end{array}}{\mathcal{E} \to T^{Star}((\textbf{Star}_\mathbf{1}, (\textbf{Seq}_\mathbf{1}, e', e), e, p, N \cup N', O \cup (\Sigma \setminus N')))} \\ \\ \dfrac{\begin{array}{c}\mathcal{E} = (\textbf{Star}_\mathbf{1}, e, e', p, N, O) \\ \Sigma = S(e) \\ N' = \{\overbrace{\forall r.P \Rightarrow \alpha \to \beta}^{\sigma} \mid \sigma \in \Sigma\ \wedge\ \beta \not\sqsubseteq p\} \\ N = (N \cup N')\ \wedge\ O = (O \cup (\Sigma \setminus N'))\end{array}}{\mathcal{E} \to O} \end{cases}$$

Theorem 10. *The fixed point for the type inference of that star combinator always exists. Furthermore, the algorithm of Transformation 7 is guaranteed to reach this fixed point.*

Proof. The types that are inferred by S are constructed over fixed, finite set of labels. As records only contain pairwise distinct labels, the set of inferable types is finite. In neither of its two cases does the algorithm remove elements from the set of matching or non-matching routes. In each iteration at least one of the sets increases in size, i.e. a new type has been inferred, or the algorithm terminates. □

6.9.2 Dealing with Binding Tags

Binding tags seem to pose a challenge for type checking purposes because of their special behaviour in terms of sub-typing and flow-inheritance. The definition of sub-typing on record types is complicated by the demand that the set of binding tags is identical and not just a subset. This severely complicates sub-type based algorithms as for example the coercion mechanisms developed in Sect. 6.7. The construction of sub-type graphs has to be repeated for every possible set of binding tags as each binding tag induces its own subtype hierarchy.

The restriction on flow-inheritance offers an elegant way out of these complications. As binding tags cannot be inherited, sequential composition is only legal if the set of produced binding tags of the left operand exactly matches the set of required binding tags of the right operand. The non-inheritance restriction also implies that co-ercions between hierarchies are not possible. We exploit these properties and extend the definition of Transformation 3 by the checking mechanism that is introduced in Transformation 8.

We apply these checks to a set of routes as produced by $T_{Seq'}^{Seq}(\mathcal{E})$ and remove all routes from the set for which the check does not succeed. This discards all routes for which the sets of binding tags of both operands in a sequential composition do not match. After the routes have been removed we may also remove the binding tags from all box signatures in the remaining routes. We have ensured that the binding tags exactly match and no further type checking is required for them.

Transformation 8. *As before, we use $BT(r)$ to denote the set of binding tags of r. The first two transformations extract the set of binding tags for the left-hand (right-hand) side of a box; if dealing with a sequential compositions, the left-hand (right-hand) side of the left-most (right-most) box in that composition is extracted.*

The third transformation is the actual checking mechanism. It determines if the set of binding tags match for all members of a sequential composition.

$$BT_{in}(\mathcal{E}) = \begin{cases} \dfrac{\mathcal{E} = (\mathbf{Box_3}, id, r_{in}, r_{out})}{\mathcal{E} \to BT(r_{in})} \\[2ex] \dfrac{\mathcal{E} = (\mathbf{Seq_3}, l, r)}{\mathcal{E} \to BT_{in}(l)} \end{cases}$$

$$BT_{out}(\mathcal{E}) = \begin{cases} \dfrac{\mathcal{E} = (\mathbf{Box_3}, id, r_{in}, r_{out})}{\mathcal{E} \rightarrow BT(r_{out})} \\ \\ \dfrac{\mathcal{E} = (\mathbf{Seq_3}, l, r)}{\mathcal{E} \rightarrow BT_{out}(r)} \end{cases}$$

$$T^{BTChk}_{SEQ}(\mathcal{E}) = \begin{cases} \dfrac{\mathcal{E} = (\mathbf{Box_3}, id, in, out)}{\mathcal{E} \rightarrow \top} \\ \\ \dfrac{\mathcal{E} = (\mathbf{Seq_3}, l, r) \quad l \rightarrow \top \wedge r \rightarrow \top}{\mathcal{E} \rightarrow (BT_{in}(r) = BT_{out}(l))} \end{cases}$$

6.10 Discussion

In this section we sketch out a few further topics for which a detailed treatment is beyond the scope of this dissertation.

6.10.1 Complexity Considerations

The flexibility of the inference system comes at a price. In the general case, the route analysis algorithm is of exponential complexity, which from a theoretical standpoint, is a clearly unacceptable solution. In practice, however, the impact is manageable by

Using binding tags to contain flow-inheritance to cases where it is explicitly requested in order to reduce the number of considered routes: Flow-inheritance allows for flexible composition of components but usually leads to a large number of possible but unintended routes through a network. As binding tags cannot be inherited their presence can outrule certain combinations of routes from being considered. For example

```
net N {
  box A((a) -> (t));
  box B((b) -> (u));
  box C((t) -> (x));
  box D((u) -> (y));
} connect (A | B) .. (C | D);
```

contains four legal routes, i.e. the networks accepts `{a}`, `{b}`, `{a,u}`, `{b,t}` as inputs. Oftentimes the latter two alternatives are not required and are considered unecessarily during type inference. Using binding tags, as in

```
net N {
  box A((a) -> (t,<#bind_AC>));
  box B((b) -> (u,<#bind_BD>));
  box C((t,<#bind_AC>) -> (x));
  box D((u,<#bind_BD>) -> (y));
} connect (A | B) .. (C | D);
```

the unintended routes will not be considered. We may consider introducing binding fields in addition to binding tags. This would increase programming convenience as no artificial bind tags need to be introduced, in the above example we could replace `(t,<#bind_AC>)` by `(#t)` and `(u,<#bind_BD>)` by `(#u)`. A more radical approach would be to reverse the current behaviour and make all labels binding by default. Flow inheritance is then only considered when explicitly requested.

specialising networks by providing network signatures that only expose the minimal set of required routes by manually adding signatures to networks instead of having them fully infered. For this to be practical the right-hand side of signatures may be left open such that the input types are fixed by the signature but the possible output types are still automatically inferred. By restricting the input of a network to those types that are explicitly mentioned in the signature routes that require different input types are ruled out and will not propagate throught the remaining inference process. If external, precompiled networks are used for which the signature cannot be changed, such external networks may be wrapped inside an outer network that narrows the signature. This is particularly useful for networks that provide a broade range of functionality as these typically expose large signatures. In a concrete application scenario often only a few provided features of a network are required. The signature may be cut down by wrapping a network with a user-provided signature around it that limits input to those types that are relevant to the specific application.

introducing auxiliary networks to break down large connect expressions in order to rule out untypable routes as early as possible: As the type inference process discards untypable routes at network boundaries the introduction of auxiliary network may lead to an earlier removal of illegal combinations and consequently a reduction of the amount of routes that are checked in total. By limiting connect expressions to contain a single combination we carry out type inference on the smallest possible sub-network structures. Although rather artificial, this example illustrated the approach:

```
net N {
  box A((a) -> (b));    box B((a) -> (x));
  box C((b) -> (x));    box D((a) -> (y));
  box E((x) -> (d));    box F((a) -> (z));
} connect (A | B) .. (C | D) .. (E | F);
```

The network contains eight routes to be checked. However, if we introduce auxiliary networks such that all networks only contain at most one combinator

```
net N {
  box A((a) -> (b));    box B((a) -> (x));
  box C((b) -> (x));    box D((a) -> (y));
  box E((x) -> (d));    box F((a) -> (z));

  net AB connect A | B;
  net CD connect C | D;
  net EF connect E | F;
  net ABCD connect AB .. CD;
} connect ABCD .. EF;
```

some combinations are ruled out before computing the global signature of network `N`. A bottom-up inference would first compute signatures for networks `AB` and `CD`. Both networks contain two legal routes. However, during the inference for network `ABCD` the result contains only one legal route through the sequential composition `AB .. CD`, which is `a -> x`. The final inference step now only needs to consider this one signature in the sequential composition of `ABCD .. EF`.

The auxiliary networks may be automatically introduced by a simple compilation step before type inference is carried out.

6.10.2 Type Errors on the Box Language Level

The complete separation between the coordination and the computational layer, i.e. the specifcation of coordination programs without considering specifics of the implementation language of boxes, allows for very generic specifications of coordination models. However, the separation entails that we cannot make any statements on data types of the computational language. This opens up an area of potential runtime errors. For boxes that are implemented in C a mismatch between types, e.g. an argument was expected to be an array of `double` values where in fact `int` was provided, typically leads to segmentation faults. When using SAC, the problem is dealt with more gracefully as the SAC runtime system automatically checks for type compatibilty. A descriptive runtime-error is raised whenever a mismatch is detected. In practice this has so far not lead to substantial problems as these errors are straight-forward to debug. However, a cleaner solution is certainly desirable. A possible approach is to extend the code generation for the glue code of boxes in such a way that the box language compiler is able to raise type errors. The required type information would be annotated in metadata blocks that accompany box declarations on the S-Net level. During the glue code generation a prototype of the box function can be generated which is then checked against the actual implementation. A more general approach is currently investigated within the EU FP7 project ADVANCE [ADV11]. An external constraint aggregation language is used to encode auxiliary information, as for example type information, on the coordination level as well as on the computational level. A constraint solver analyses the annotations and issues an error if the annotated constraints of labels cannot be unified, i.e. if the constraints are unsatisfiable. This approach may be used to encode several other properties as well to guide optimisations of the coordination program and the box implementations alike [**?**].

6.10.3 Networks and Separate Compilation

The presented approach so far does not support nested networks. We are going to address and lift this restriction here.

There are many ways to support nested networks during type checking. We could use an approach that employs the same technique as we use for boxes. This would be implemented by simply inlining an inner network into the `connect` expression of its surrounding network. As this requires access to the implementation of the inner network this solution is not ideal if separate compilation is a concern. An approach that works without exposing the internals of a network is preferable as this allows embedding of external networks that have been type checked and compiled earlier.

We can achieve this by using a bottom-up approach that type checks inner networks first. The result of type checking an inner network is a set of types describing the legal routes through the network. These routes need to be inserted into the surrounding network. As type checking has succeeded for the routes that are left in the inferred signature we treat them as atomic entities; an integration of the routes may be modelled by the following steps.

- introduce one box for each type in the inferred signature
- translate the type to a box signature and associate it with a newly introduced box
- introduce a choice combination over the newly introduced boxes
- replace the occurrence of the internal net in the `connect` expression by the constructed choice combination.

If networks are compiled as modules for later use within other networks, the inferred signatures have to be stored together with their marker strings inside the module. When this module is picked up by a compilation process at a later stage, the compiler reads the network signatures of the module to prepares its representation as choice combination as above. Instead of associating single markers with each of the choice operands as was done before, the entire marker string of a route is inserted. From there the remaining process works as described before.

6.10.4 Overwriting and Stacking

The set of predicates that is computed during the type checking process may appear to be too restrictive. The enforcement of $lacks$ constraints ensures that type checking only succeeds if no labels are overwritten.

However, it is because of this restrictiveness that the developed process may be adopted to accommodate other, more liberal approaches to dealing with label clashes by re-interpreting the constraint sets.

- ***Label overwriting.*** If implicit label overwriting is allowed, the result of computations may replace flow-inherited labels. By not adding the *lacks* constraints that are introduced by a function's right-hand side overwriting is made possible. This allows for compositions of the form

  ```
  net overwrite {
    box A((a) -> (b));
    box B((c) -> (b));
  } connect A .. B;
  ```

  to be successfully type checked.

- ***Stacking.*** Stacking semantics, i.e. allowing multiple labels of the same name within one record, may be modelled if *lacks* constraints are not introduced for left-hand sides of functions. This allows compositions of the form

  ```
  net multiset {
    box A((a) -> (b));
    box B((a) -> (x));
  } connect A .. B;
  ```

  to be successfully type checked.

A thorough analysis of the implications of these relaxations on the presented framework remains as future work. Stacking in particular introduces complications for fix-point computations and the coercion mechanism as the set of labels of a record type is effectively extended to a multi-set which may contain an infinite amount of labels of the same name.

6.10.5 From Inferred Type Schemes to Network Signatures

After the type checking process has finished the inferred types are not in a format that adheres to the standard S-Net syntax for network signatures. For the function types as such this is only a matter of syntactic conversion. The predicate set that has been computed with every function type, however, has no counterpart in network signatures in standard S-Net syntax as we have used them so far. There are several ways of dealing with this. The least invasive integration simply drops the predicate set from signatures before presenting the result to a user but keeps the information internally. The predicates are used by the implementation to detect which labels have to be shifted out of the current names space in order to prevent them from being overwritten. For example,

```
net shift {
  box A((a) -> (b));
  box B((b) -> (c));
} connect A .. B;
```

will be inferred the type

$$\forall r.\, \underbrace{(r \setminus a, b, c)}_{\text{predicates}} \Rightarrow \overbrace{\underbrace{(\!|a \mid r|\!)}_{\text{input type}} \rightarrow \underbrace{(\!|c \mid r|\!)}_{\text{output type}}}^{\text{function type}} .$$

The function type may be straight-forwardly mapped into {a} -> {c}. The predicates on labels may be used at runtime when records are fed into the system. All labels that are not part of the input type but have predicates on them may be renamed to avoid clashes. At the output of a network this renaming is reversed to reconstruct the original label names. Labels that are under a predicate and appear in the output type cannot be renamed at the output. In this case the label is discarded instead of renamed and a runtime-warning may be issued to announce that a label was overwritten at the output.

Another approach may choose to expose the full amount of inferred information to the programmer at compile time in order to ensure that only legal input is used as input to the network. If an input record does meet the constraints, it will be rejected and a warning issued.

Of course, other approaches may be different again. Tool-chain implementers have a broad design-space and may choose an approach that best matches their anticipated use-cases.

6.10.6 Alternative Approach to Type Checking in S-Net

The design of the presented approach to type checking in S-Net is influenced by experiences made with earlier work in this area [CESG07, CEG^{+}08]. The cited system employs a custom-made mechanism to model record transformations and flow-inheritance. Record types are expressed as sets, variant record types as for example introduced by boxes with multiple right-hand sides are modelled as a set of sets. This approach proved to make dealing with serial combinations a complex undertaking. The presence of multiple right-hand sides introduces alternatives within a sequential chain which have to be considered when turning a sequence of boxes into function composition for type checking. This is one of the motivations for decomposing variant boxes into several non-variant boxes in the current approach. The fact that the underlying theory of the earlier system is built-up from scratch entails a fair amount of work to establish basic properties such as soundness of the inference algorithm and guarantees for finding the most general types possible. In the current approach we have built the system on top of existing work that established these properties for us.

The main difference between the previous and our approach is the way choice combinations are dealt with. Where now the routing decisions are analysed during type checking taking all available context information into account the previous system assumed that routing decisions are taken based on only local knowledge. A consequence of this is that a program

```
net example {
   box A((a) -> (x));
   box B((a) -> (y));
   box C((x,y) -> (z));
} connect (A | B) .. C;
```

will be inferred the signature `{a,x,y} -> {z}` (in a system that allows implicit label overwriting). The routing decision at the split point between `A` and `B` is solely based on the type information present at that point which are the two box signatures, both requiring only `a` as their input. The current approach postpones a decision about legal routing decisions until later and is able to be more permissive for the input by inferring `{a,x} -> {z}` and `{a,y} -> {z}`. The differences become more apparent if we consider the network

```
net example {
   box A((a) -> (x));
   box B((a) -> (y));
   box C((x,y) -> (z));
   box D((y) -> (c));
} connect (A | (B .. D)) .. C;
```

which is rejected as illegal by the previous system. The current approach infers `{a,y} -> {z}` because the route containing `B .. D` is detected as hazardous in its context and ruled out from the set of possible routes.

7 Semantics of Implicitly Typed S-NET

In this chapter we bring together the information that the type inference system provides us with and the formal semantics of the language that we have developed in Chapter 4.

The type inference system computes required type information from only the box signatures. One of the major benefits we see from this is that we may now drop the manually annotated network signatures. The purpose of a signature is two-fold. First of all, it determines the legal inputs to a network. Only a record whose type matches (or is a subtype of) one of the input types of the inferred signature is acceptable as input. Secondly, signatures influence the behaviour of choice combinators. The decisions at choice points within a network are based on the type of an input record and the signatures of the two operands of the choice combinator. If the type of the record matches an input type of one of the operand's signatures, then the operand may be chosen as the destination for the record.

With the type inference system in place, these vital information are provided automatically. Although this is the most apparent change from a programmer's point of view, it is not just type information that is computed during the inference process. Marker strings are composed during the inference process, and these strings are attached to each individual type of a signature. The information that these marker strings carry are not directly relevant to programmers but they limit the validity of the type they are attached to: An individual type within a signature corresponds to a specific path through the network, namely exactly the path that is encoded by the marker string. As such, the type is only valid as long as routing decisions at runtime are in accordance with the assumed decisions expressed by the marker string. Enforcing consistency between runtime decisions and marker strings has a profound effect on the semantics of the language.

In order to maintain consistency between a statically inferred marker string and the decisions made at runtime for an input record, the routing decisions have to follow the choices that the marker string allows for. Although this strikes us to be more restrictive than in the untyped semantics, it increases the amount of legal inputs, i.e. inputs for which we can statically guarantee that every intermediate result can be further processed. Consider the example of Fig. 7.1.

If decisions are solely based on a record's type at a choice point, the program shown in the figure cannot accept records that only carry a field `{img}`. For an input record that only carries `{img}` the second branch of the choice combination leads to a situation where the result of the operand cannot be processed by the subsequent box. Consequently, this type of input has to be rejected. If, however, we restrict the options of the choice point to only the first branch, we may allow `{img}` as an input, as in this

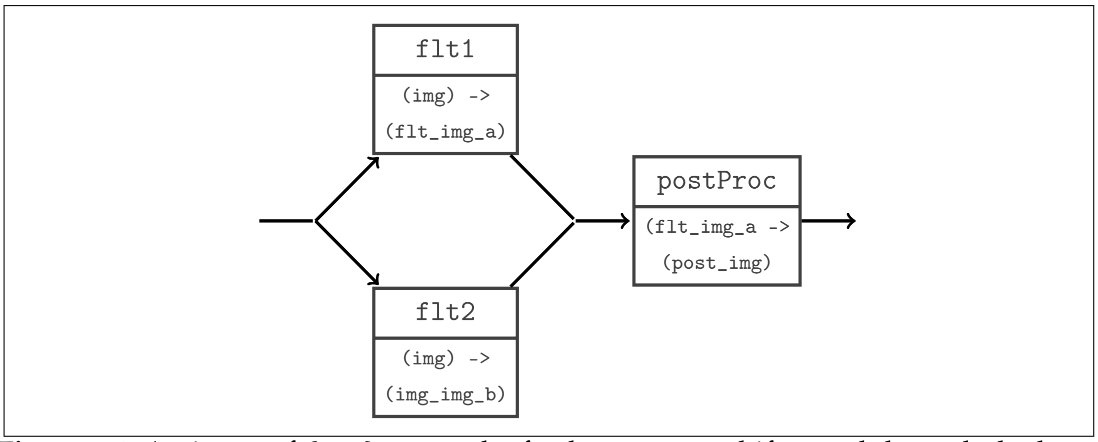


Figure 7.1: An input of {img} cannot be further processed if routed through the lower branch of the choice combination.

case the operand's result can be processed further by the following box.

The restriction of choices limits decisions to cases for which the type inference system can prove that intermediate results are always of a type for which further processing is possible. As this differs from the previous specification of the semantics where decisions were based on record types, the formal semantics that we develop in this chapter takes a different approach. The input to a program is not just a plain record anymore, but a record paired with routing information in the form of marker strings. Instead of testing for subtype relationships we now test for the presence and absence of marker elements in the marker strings that a record carries. As with every decision that is taken the set of remaining decisions may narrow down, the maintenance of routing information also has to be captured in the semantic framework.

In the remainder of this chapter we develop this semantic framework. We introduce a representation of inferred signatures and marker strings in Sect. 7.1. Section 7.2 presents the transition rules. These resemble the rules of Chapter 4 but now take marker strings and their maintenance into account. In Section 7.3 we show that the formal semantics are consistent with the type inference algorithm, i.e. that the application of the transition rules indeed produce results of the type that the inferred signature claims.

7.1 Basic Definitions

The type inference process that we have introduced in Chapter 6 computes a set S of types and marker strings for a given program P. The constraint set that has been computed for each route type in S indicates which labels have to be protected from overwriting. This poses vital information in practice, and we assume that a mechanism is put in place that automatically shifts clashing labels as discussed in Sect. 6.10. This allows us to drop constraint sets from route types for now and we may represent an inferred result set S of n routes for a program P as

$$S = \left\{R_i :: \tau_i^{in} \to \tau_i^{out} \mid i \in \{0, \ldots, n-1\}\right\}$$

where R is a marker string τ^{in} is an input type and τ^{out} is an output type. We will introduce a more compact representation of such sets in Section 7.1.1 and we will discuss how records that are to be processed by a network are enriched by routing information in Sect. 7.1.2.

7.1.1 Marker String Representation and Route Groups

A set of inferred route types for a program may contain several route types of the same input type. As we will be working with such groups of routes that share the same input types it is convenient to represent the different marker strings of these routes as a single object. In order to do this we use a tree representation for sets of marker strings, similar to the approach that we have used during type checking in Sect. 6.7 for marker graphs. In Def. 50 a formal description of such a representation for a set of marker strings is given, Def. 51 introduces a serialisation of the representation.

Definition 50. *A* corresponding marker tree *for a set of marker strings $M = \{m_1, \ldots, m_n\}$ where $\forall_{i=1}^{n} : m_i \in \Sigma_{RV}^{\star}$ is a graph $G = (N, E)$ for which the set of nodes N is defined as*

$$N = \bigcup_{m \in M} \mathit{Pref}(m) \cup M$$

and the set of edges E is defined as

$$E = \left\{(s, s') \mid s, s' \in N \wedge s' - s = x \in \Sigma_{RV}\right\}$$

Definition 51. *The* serialised representation $\mathfrak{M}$ *of a marker tree $G = (N, E)$ is a nested tuple structure of the form*

$$\mathit{MarkerTree} \quad \Rightarrow \quad \textbf{(}\ \mathit{Marker}\ [\ \textbf{,}\ \mathit{MarkerTree}\ ]^{*}\ \textbf{)}$$

The nesting of tuples is determined by a depth-first traversal of G. The node strings $n \in N$ are shortened to their last element in the serialised representation.

As an illustration of the previous two definitions Fig. 7.2 shows a marker tree and its serialised representation. In order for us to check for the presence of a certain symbol within a marker tree we define a membership check on marker trees in Def. 52.

Definition 52. *Let $\mathfrak{M}, \mathfrak{M}'$ be two serialised marker trees. We will say that $\mathfrak{M}'$ is a sub-tree of $\mathfrak{M}$, if $\mathfrak{M}'$ is contained in $\mathfrak{M}$. We define the membership relation $\in$ on marker trees recursively:*

$$\begin{aligned} \mathfrak{M}' \in_0 \mathfrak{M} &\iff \mathfrak{M}' = \mathfrak{M} \\ \mathfrak{M}' \in_{i+1} (v, \mathfrak{M}_1, \ldots, \mathfrak{M}_n) &\iff \exists k \in \{1, \ldots, n\} : \mathfrak{M}' \in_i \mathfrak{M}_k \end{aligned}$$

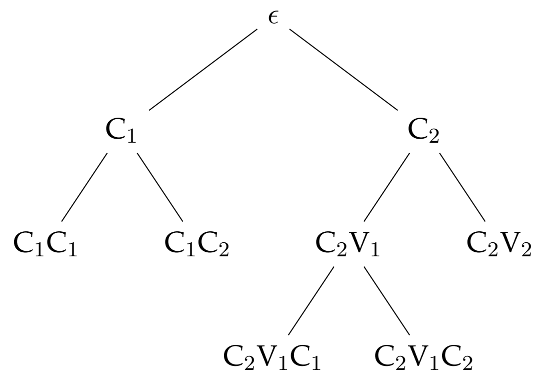
We will write

$$\mathfrak{M}' \in \mathfrak{M} \iff \exists n \in \mathbb{N}_0 : \mathfrak{M}' \in_n \mathfrak{M}$$

and

$$\mathfrak{M}' \in_{\leq n} \mathfrak{M} \iff \exists k \in \mathbb{N}_0, k \leq n : \mathfrak{M}' \in_k \mathfrak{M}$$

$$M = \{\mathrm{C_1C_1},\ \mathrm{C_1C_2},\ \mathrm{C_2V_1},\ \mathrm{C_2V_2},\ \mathrm{C_2V_1C_1},\ \mathrm{C_2V_1C_2}\}$$



$$\mathfrak{M} = (\epsilon, (\mathrm{C_1}, (\mathrm{C_1}), (\mathrm{C_2})), (\mathrm{C_2}, (\mathrm{V_1}, (\mathrm{C_1}), (\mathrm{C_2})), (\mathrm{V_2})))$$

Figure 7.2: A corresponding marker tree for a set of marker strings M and its serialised representation $\mathfrak{M}$.

For example, reconsidering the marker tree $\mathfrak{M}$ of Fig. 7.2, we may now say that $(\mathrm{V_1}, (\mathrm{C_1}), (\mathrm{C_2})) \in \mathfrak{M}$, $(\mathrm{C_1}, (\mathrm{C_1}), (\mathrm{C_2})) \in_1 \mathfrak{M}$ and $(\mathrm{V_2}) \in_{\leq 3} \mathfrak{M}$. As a syntactical convenience we will occasionally replace a list of sub-trees by a single symbol. For example, instead of $(\mathrm{V_1}, (\mathrm{C_1}), (\mathrm{C_2}))$ we write $(\mathrm{V_1}, J)$ and use J as a placeholder for the two sub-trees $(\mathrm{C_1}), (\mathrm{C_2})$.

With the new representation for sets of marker strings in place a set of route types may be written in a more compact way. For this we group all routes that share the same input type and represent their marker strings as a marker tree. The various output types are grouped into a set of outputs such that a single input type may now have a set of possible output types. More specifically, for n inferred route types

$$S = \left\{R_i :: \tau_i^{in} \rightarrow \tau_i^{out} \mid i \in \{0, \ldots, n-1\}\right\}$$

we first partition S into several sets $S_1, \ldots, S_n$ such that

$$\forall i \in \{1, \ldots, n\} \forall s, s' \in S_i : \tau_{in}(s) = \tau_{in}(s')$$

where $\tau_{in}(s)$ denotes the input type of route type s, i.e. all routes in S_i share the same input type $\tau_{in}(S_i)$. Next, we represent all marker strings of the route types in S_i by one marker tree $\mathfrak{M}_i$. The m_i output types $\tau_{out}(s)$ for the $s_1, \ldots, s_{m_i}$ routes in S_i we arrange into one set of outputs $\{\tau_{out}(s_1), \ldots, \tau_{out}(s_m)\}_i$. Finally, we may represent each S_i as

$$\mathfrak{M}_i :: \tau_{in}(S_i) \rightarrow \{\tau_{out}(s_1), \ldots, \tau_{out}(s_{m_i})\}_i$$

7.1.2 Best Representatives for Records

The marker string that has been constructed during the type inference process alongside the input and the output type of a route predetermines a path through a network for a record. The elements of a marker string influence the behaviour of network components in those places where previously subtype relationships determined the behaviour. Because of this we need to ensure that routing information is available alongside each record that a network is supposed to process. In the semantic framework this requirement manifests itself in the definition of transition rules. In the semantics of untyped S-NET each transition rule describes (among other things) how a record is transformed into a stream of results. In the framework that we develop for typed S-NET we extend the definition of rules to also include a transformation of marker information; transition rules now define how a tuple of a record and a marker are transformed into result tuples of the same form. Where previously the transition rules were of the general form

$$\frac{s = \epsilon \triangleright s_1 \triangleright \ldots \triangleright s_n}{(E, r, M) \rightarrow (E', M', \vec{s})}$$

we now use rules of the form

$$\frac{s = \epsilon \triangleright (s_1, \mathfrak{M}_1) \triangleright \ldots \triangleright (s_n, \mathfrak{M}_n)}{(E, (r, \mathfrak{M}), M) \rightarrow (E', M', \vec{s})}$$

The processing steps for a record are not determined consecutively by checking subtype relations anymore. By attaching a set of possible routing decisions, i.e. marker trees, to a record the possible processing steps are limited to those contained in the marker tree. In order to determine which set of routes is to be attached to a record we develop the notion of a "best" representative from a given set of routes; the best representative for a record r is a collection of route types for which the input types of the routes have the largest overlap with the type of r.

Definition 53. *The* best representative *from a group of routes for a record r is constructed from a set of n route types*

$$S = \{\mathfrak{M}_i :: \tau_{in}(S_i) \rightarrow \{\tau_{out}(s_1), \ldots, \tau_{out}(s_{m_i})\}_i \mid i \in \{0, \ldots, n-1\}\}$$

by first selecting those routes for which r is a subtype of the input type:

$$\hat{R}^r_S = \{s \mid s \in S \wedge r \sqsubseteq \tau_{in}(s)\}$$

Let $|\tau|$ denote the number of labels in record type τ and let

$$m_r = max \left\{ |\tau_{in}(s)| \mid s \in \hat{R}^r_S \right\}$$

Then, from the set

$$\begin{aligned} R^r_S &= \left\{ s \mid s \in \hat{R}^r_S \ \wedge \ |\tau_{in}(s)| = m_r \right\} \\ &= \Big\{ \mathfrak{M}_{R_1} :: \tau_{in}(S_{R_1}) \rightarrow \{\tau_{out}(s_{R_1}), \ldots, \tau_{out}(s_{m_{R_1}})\}, \ldots, \\ &\qquad \mathfrak{M}_{R_k} :: \tau_{in}(S_{R_k}) \rightarrow \{\tau_{out}(s_{R_k}), \ldots, \tau_{out}(s_{m_{R_k}})\} \Big\} \end{aligned}$$

the best representative for r is derived from R^r_S by joining the marker trees into one tree for all route types. More precisely, let $G_j = (N_j, E_j)$ be the marker tree of S_j in R^r_S for all $j \in J = \{1, \ldots, k\}$. The combined marker tree $G_R = (\bigcup_{j \in J} N_j, \bigcup_{j \in J} E_j)$ is used (in its serialised representation $\mathfrak{M}_R$) for the construction of the best representative:

$$\mathfrak{M}_R :: \{\tau_{in}(s) \rightarrow \{\tau_{out}(s)_1, \ldots, \tau_{out}(s)_m\} \mid s \in R^r_S\}$$

The routing decisions that are involved for the individual routes in the collection are grouped together into one tree; the decisions that are encoded in this tree are exactly those that represent the routes in the collection. Consequently, a record is paired with the marker tree of its best representative before it is given to a network for processing.

7.2 Transition Rules

The transition rules for typed S-NET are shown in this section. The main difference between these rules and the rules of the untyped language in Chapter 4 is the replacement of subtype checks by marker trees.

7.2.1 Networks

Before a record can be processed by a network it has to be paired with a marker tree. For a given network

```
net netId S connect P;
```

with an inferred signature S for its program P and a record r we attach the best representative of S to r.

Where we have nested networks we need to preserve marker data that a record already carries. The result from an inner network has to continue its originally intended path through the surrounding network after the inner network has finished processing it. In order to do this, the original marker tree of an input record is kept at the network boundary upon entering. The retained marker information is attached to the produced records, replacing any other marker trees that the records may carry. This allows the result records to continue their original path through the rest of the network. We assume that any inbound record on the outermost network is already paired with an empty marker tree which is then replaced by the best representative for the record from the network's signature.

Transition Rule 11. *At network boundaries the marker tree of a record is replaced by a marker tree for the network that the record is about to enter. The marker tree is that of the best representative for the record constructed from the inferred network signature S. The representative is denoted BR^r_S.*

$$\text{tNet} \quad : \quad \frac{\begin{array}{c} BR^r_S = \mathfrak{M}_B :: T \\ (E,\ (r, \mathfrak{M}_B),\ P) \to (E',\ P',\ \vec{s'}) \\ \vec{s'} = \epsilon \triangleright (s_1, \mathfrak{M}_1) \triangleright \ldots \triangleright (s_n, \mathfrak{M}_n) \\ \vec{s} = \epsilon \triangleright (s_1, \mathfrak{M}_o) \triangleright \ldots \triangleright (s_n, \mathfrak{M}_o) \end{array}}{\begin{array}{l} (E,\ (r, \mathfrak{M}_o),\ \texttt{net}\ S\ \texttt{connect}\ P) \to \\ \qquad\qquad (E',\ \texttt{net}\ S\ \texttt{connect}\ P',\ \vec{s}) \end{array}}$$

7.2.2 Boxes and Synchro cells

The behaviour of boxes does not change with the presence of marker information. Quite the opposite is the case; marker information might have to be amended to reflect the behaviour of boxes. For boxes with non-variant output types no amendments are required. The marker information that is attached to an input record is carried over to all result records on the output stream. Variant boxes, however, require us to adapt the marker data of a record depending on the variant that has been produced. The marker data that is attached to a record contains two branches in the marker tree, one for each possible output variant of the box. The marker element V_1 identifies the branch that has to be taken in case the box produced a record of the first variant. The marker element V_2 does the same for the second variant. For each output record that a variant box produces, the marker information of the inbound record is replaced by the sub-tree of the marker tree that is connected to the V_1 resp. V_2 node. This effectively means that the marker strings that are represented by the marker tree are shortened by one element as the first element is consumed by the box decision.

Transition Rule 12. *Boxes are represented by two rules.* tBox *attaches inbound marker information to each output. Variant boxes (*tBoxLR*) amend the marker tree according to the produced variant.*

$$\text{tBox} \quad : \quad \frac{\begin{array}{c} \vec{s} = \epsilon \triangleright (s_0, \mathfrak{M}) \triangleright \ldots \triangleright (s_{n-1}, \mathfrak{M}) \\ \forall (s, \mathfrak{M}) \in \vec{s} : s \sqsubseteq \gamma \end{array}}{\begin{array}{l} (E,\ (r, \mathfrak{M}),\ \texttt{box}\ BoxId\texttt{(}\tau\ \texttt{->}\ \gamma\texttt{)}) \to \\ \qquad\qquad (E,\ \texttt{box}\ BoxId\texttt{(}\tau\ \texttt{->}\ \gamma\texttt{)},\ \vec{s}) \end{array}}$$

$$\text{tBoxLR} \quad : \quad \frac{\begin{array}{c} (V_1, \overbrace{\mathfrak{M}_{L_1}, \ldots, \mathfrak{M}_{L_j}}^{M_L}) \in_1 \mathfrak{M}\ \wedge\ (V_2, \overbrace{\mathfrak{M}_{R_1}, \ldots, \mathfrak{M}_{R_k}}^{M_R}) \in_1 \mathfrak{M} \\ \forall (s, M) \in \vec{s} : s \sqsubseteq \gamma_L \wedge M = (\epsilon, M_L)\ \vee\ s \sqsubseteq \gamma_R \wedge M = (\epsilon, M_R) \end{array}}{\begin{array}{l} (E,\ (r, \mathfrak{M}),\ \texttt{box}\ BoxId\texttt{(}\tau\ \texttt{->}\ \{\gamma_L, \gamma_R\}\texttt{)}) \to \\ \qquad\qquad (E,\ \texttt{box}\ BoxId\texttt{(}\tau\ \texttt{->}\ \{\gamma_L, \gamma_R\}\texttt{)},\ \vec{s}) \end{array}}$$

The behaviour of synchro cells is determined by the current state of the cell and the type of the input record. In untyped S-Net the state transition was dependent on the subtype relation between the synchro cell's pattern and an input record. Here we base such decisions on the presence and absence of markers in the marker tree.

The type inference process has analysed the possible outputs of synchro cells and it has inserted S_{1o} (overflow) and S_{1s} (synchronisation) markers into the marker strings of routes that represent cases where the first pattern is matched by an input. The symmetric cases for the right pattern are marked with S_{2o} and S_{2s}. Because of this we achieve the same behaviour as before by checking for the presence of these markers instead of checking a subtype relationship between input and patterns. Depending on the action that a cell has taken the marker tree is amended accordingly; the marker tree is replaced by the corresponding sub-tree of the case that is handled. If several sub-trees are a possible choice, as is the case after synchronisation in TSYID and for the case where a record matches both patterns of an empty cell, the sub-trees are joined after removing the synchro cell marker.

Transition Rule 13. *The synchro cell implements the same state machine as in untyped* S-NET, *however state transitions are now dependent on the presence and absence of markers in the marker tree of an inbound record rather than on the subtype relationship between input and patterns. For the sake of brevity and readability of the following transition rules we will use an M as shorthand notation for a list of marker trees. For example, the tree $(S_{1o}, (C_1), (C_2, (C_1))$ may be shortened to (S_{1o}, M) as we allow M to represent the two trees $(C_1), (C_2, (C_1))$.*

tSYmatchL :
$$\frac{\begin{array}{c}((S_{1o}, M_{Lo}) \in_1 \mathfrak{M} \vee (S_{1s}, M_{Ls}) \in_1 \mathfrak{M}) \ \wedge \\ \neg((S_{2o}, M_{Ro}) \in_1 \mathfrak{M} \vee (S_{2s}, M_{Rs}) \in_1 \mathfrak{M})\end{array}}{(E,\ (r, \mathfrak{M}),\ \llbracket p_1, p_2 \rrbracket) \rightarrow (E,\ \llbracket p_1, p_2 \rrbracket_{p_1},\ \epsilon)}$$

tSYmatchR :
$$\frac{\begin{array}{c}((S_{2o}, M_{Ro}) \in_1 \mathfrak{M} \vee (S_{2s}, M_{Rs}) \in_1 \mathfrak{M}) \ \wedge \\ \neg((S_{1o}, M_{Lo}) \in_1 \mathfrak{M} \vee (S_{1s}, M_{Ls}) \in_1 \mathfrak{M})\end{array}}{(E,\ (r, \mathfrak{M}),\ \llbracket p_1, p_2 \rrbracket) \rightarrow (E,\ \llbracket p_1, p_2 \rrbracket_{p_2},\ \epsilon)}$$

tSYoflL :
$$\frac{(S_{1o}, M) \in_1 \mathfrak{M}}{(E,\ (r, \mathfrak{M}),\ \llbracket p_1, p_2 \rrbracket_{p_1}) \rightarrow (E,\ \llbracket p_1, p_2 \rrbracket_{p_1},\ \epsilon \triangleright (r, (\epsilon, M)))}$$

tSYoflR :
$$\frac{(S_{2o}, M) \in_1 \mathfrak{M}}{(E,\ (r, \mathfrak{M}),\ \llbracket p_1, p_2 \rrbracket_{p_2}) \rightarrow (E,\ \llbracket p_1, p_2 \rrbracket_{p_2},\ \epsilon \triangleright (r, (\epsilon, M)))}$$

tSYmergeL :
$$\frac{(S_{2s}, M) \in_1 \mathfrak{M} \ \wedge \ (S_{1s}, M') \notin_1 \mathfrak{M}}{\begin{array}{l}(E,\ (r, \mathfrak{M}),\ \llbracket p_1, p_2 \rrbracket_{p_1}) \rightarrow \\ \quad (E,\ Id,\ \epsilon \triangleright (RT_{p_2}^{p_1 \cup p_2}(r), (\epsilon, M)))\end{array}}$$

tSYmergeR :
$$\frac{(S_{1s}, M) \in_1 \mathfrak{M} \ \wedge \ (S_{2s}, M') \notin_1 \mathfrak{M}}{\begin{array}{l}(E,\ (r, \mathfrak{M}),\ \llbracket p_1, p_2 \rrbracket_{p_2}) \rightarrow \\ \quad (E,\ Id,\ \epsilon \triangleright (RT_{p_1}^{p_1 \cup p_2}(r), (\epsilon, M)))\end{array}}$$

tSYmatchLR :
$$\frac{(S_{2s}, M) \in_1 \mathfrak{M} \ \wedge \ (S_{1s}, M') \in_1 \mathfrak{M}}{(E,\ (r, \mathfrak{M}),\ \llbracket p_1, p_2 \rrbracket) \rightarrow (E,\ Id,\ \epsilon \triangleright (r, (\epsilon, M, M')))}$$

tSYid :
$$\frac{\begin{array}{c}X = \{S_{1o}, S_{1s}, S_{2o}, S_{2s}\} \\ \{M_1, \ldots, M_n\} = \{M_x \mid x \in X \wedge (x, M_x) \in_1 \mathfrak{M}\}\end{array}}{(E,\ (r, \mathfrak{M}),\ Id) \rightarrow (E,\ Id,\ \epsilon \triangleright (r, (\epsilon, M_1, \ldots, M_n)))}$$

7.2.3 Combinators

Except for the choice combinator, the combinators in typed S-Net remain unchanged to their untyped counterparts. As such, we can reuse the transition rules of Chapter 4 by extending records to pairs of records and their marker trees, as is done in Transition Rule 14. The rule for the star combinator has additionally been extended by a `net` construct that is wrapped around each recursive unfolding. We do this to attach fresh marker trees to inbound records in each instance.

Transition Rule 14. *The definitions of combinators in untyped* S-Net *are extended to cater for marker trees that are attached to records. In rule* tStar $S_{M_{\star\star}}$ *denotes the inferred type signature for* M `**p` *and* sId *denotes a simple box that takes the pattern* p *as input and produces* p *as output without changing it.*

$$\textsc{tEmpty} \quad : \quad \frac{}{(E,\ \epsilon,\ C) \rightarrow (E,\ C,\ \epsilon)}$$

$$\textsc{tSeq} \quad : \quad \frac{\begin{array}{c}(E,\ (r,\mathfrak{M}),\ L_i) \rightarrow (E'\ ,L_{j\geq i},\ \vec{s}) \\ (E',\ \vec{s},\ R_k) \rightarrow^{\star} (E'',\ R_{l\geq k},\ \vec{t})\end{array}}{(E,\ (r,\mathfrak{M}),\ L_i\ ..\ R_k) \rightarrow (E'',\ L_j\ ..\ R_l,\ \vec{t})}$$

$$\textsc{tSplit} \quad : \quad \frac{\begin{array}{c}v = val(r,\ \texttt{<k>}) \\ M^v_{j_v} \in \left\{M^i_{j_i} \mid i \in \mathbb{Z}\right\} \\ (E,\ (r,\mathfrak{M}),\ M^v_{j_v}) \rightarrow (E', M^v_{j'_v \geq j_v}, \vec{s})\end{array}}{\begin{array}{l}(E,\ (r,\mathfrak{M}),\ \left\{M^i_{j_i} \mid i \in \mathbb{Z}\right\}\texttt{!!<k>}) \\ \qquad \rightarrow (E',\ ((\left\{M^i_{j_i} \mid i \in \mathbb{Z}\right\} \setminus \{M^v_{j_v}\}) \cup \{M^v_{j'_v}\})\texttt{!!<k>},\ \vec{s})\end{array}}$$

$$\textsc{tStar} \quad : \quad \frac{\begin{array}{c}N = \texttt{net nstar}\, S_{M_{\star\star}}\ \texttt{connect}\,((M\ ..\ M\texttt{**p})\ \texttt{||}\ \texttt{SId}\,) \\ (E,\ (r,\mathfrak{M}),\ N) \rightarrow^{\star} (E',\ N',\ \vec{s})\end{array}}{(E,\ (r,\mathfrak{M}),\ M\texttt{**p}) \rightarrow (E',\ N',\ \vec{s})}$$

$$\textsc{tMap} \quad : \quad \frac{\begin{array}{c}\vec{r} = \epsilon \triangleright (r_0, \mathfrak{M}_0) \triangleright \ldots \triangleright (r_{n-1}, \mathfrak{M}_{n-1}) \\ \forall^{n-1}_{j=0} : (E_{i+j},\ (r_j, \mathfrak{M}_j),\ C_{i+j}) \rightarrow (E_{i+j+1},\ C_{i+j+1},\ \vec{s_j}) \\ \vec{s} = flat(++^{n-1}_{j=0}\ \vec{s_j})\end{array}}{(E_i,\ \vec{r},\ C_i) \rightarrow^{\star} (E_{i+n},\ C_{i+n},\ \vec{s})}$$

For the choice combinator the situation is different. Its behaviour is dependent on the type of its input. In order to test which of the two branches of the combinator should be taken for an input record, the marker tree that the record is paired with is inspected. If it contains a branch with a choice marker C_1 and no marker C_2 the first operand is selected for processing, and consequently, in the opposite case the second operand is selected. In case a record is paired with a marker tree that has a branch for C_1 as well as for C_2, then we base a decision on the oracle stream. We use the similar technique as in the untyped semantics and employ a decision function f that selects one of its input arguments depending on the first value on the oracle stream:

$$f((\epsilon \triangleright e) + E, (C_0, M_L), (C_1, M_R)) = \begin{cases} (M_L, E) & \text{if } e = \top \\ (M_R, E) & \text{otherwise} \end{cases}$$

where M_L and M_R are the descending marker trees of node C_0 resp. C_1. After the decision has been made and one of the two sub-trees has been selected either TCHOICEL or TCHOICER is applicable, depending on the outcome of f.

Transition Rule 15. *In typed* S-NET *the choice combinator is modelled by three rules. Rule* TCHOICELR *uses a decision function f to select a branch for processing its input.*

$$\textsc{tChoiceL} \quad : \quad \frac{\begin{array}{c}(C_1, M_L) \in_{\leq 1} \mathfrak{M} \ \wedge \ (C_2, M_R) \notin_{\leq 1} \mathfrak{M} \\ (E,\ (r, (\epsilon, M_L)),\ P) \to (E', P', \vec{s})\end{array}}{(E,\ (r, \mathfrak{M}),\ P \texttt{ || } Q) \to (E',\ P' \texttt{ || } Q,\ \vec{s})}$$

$$\textsc{tChoiceR} \quad : \quad \frac{\begin{array}{c}(C_2, M_R) \in_{\leq 1} \mathfrak{M} \ \wedge \ (C_1, M_L) \notin_{\leq 1} \mathfrak{M} \\ (E,\ (r, (\epsilon, M_R)),\ Q) \to (E', Q', \vec{s})\end{array}}{(E,\ (r, \mathfrak{M}),\ P \texttt{ || } Q) \to (E',\ P \texttt{ || } Q',\ \vec{s})}$$

$$\textsc{tChoiceLR}_f \quad : \quad \frac{\begin{array}{c}(C_1, M_L) \in_{\leq 1} \mathfrak{M} \ \wedge \ (C_2, M_R) \in_{\leq 1} \mathfrak{M} \\ (E', M') = f(E,\ (C_0, M_L),\ (C_1, M_R)) \\ (E',\ (r, (\epsilon, M')),\ P \texttt{||} Q) \to (E', X, \vec{s})\end{array}}{(E,\ (r, \mathfrak{M}),\ P \texttt{ || } Q) \to (E',\ X,\ \vec{s})}$$

The non-deterministic variants of combinators may also be straight-forwardly mapped into rules for typed S-Net by extending records to pairs of records and their marker trees. We use the same approach as in Chapter 4 and explain the non-deterministic combinators by their deterministic counterparts and employ a non-deterministic map rule to cater for reordering of records on the output stream. The definition of the combinators and the map rule are shown in Transition Rule 16.

Transition Rule 16. *The definitions of non-deterministic combinators are based on their deterministic counterparts. The function Σ applies a sub-stream order preserving permutation to its second argument as described in Section 4.4.*

$$\textsc{tNSplit} \quad : \quad \frac{(E,\ (r, \mathfrak{M}),\ \left\{M^i_{j_i} \mid i \in \mathbb{Z}\right\} \texttt{!!<k>}) \to (E',\ M' \texttt{!!<k>},\ \vec{s})}{(E,\ (r, \mathfrak{M}),\ \left\{M^i_{j_i} \mid i \in \mathbb{Z}\right\} \texttt{!<k>}) \to (E',\ M',\ \vec{s})}$$

$$\textsc{tNStar} \quad : \quad \frac{(E,\ (r, \mathfrak{M}),\ M \texttt{**p}) \to (E',\ M',\ \vec{s})}{(E,\ (r, \mathfrak{M}),\ M \texttt{*p}) \to (E',\ M',\ \vec{s})}$$

$$\textsc{tNChoice} \quad : \quad \frac{(E,\ (r, \mathfrak{M}),\ M \texttt{||N}) \to (E',\ M',\ \vec{s})}{(E,\ (r, \mathfrak{M}),\ M \texttt{|N}) \to (E',\ M',\ \vec{s})}$$

$$\textsc{tNmap}_\Sigma \quad : \quad \frac{\begin{array}{c}M \in \{P \texttt{|} Q,\ P \texttt{*p},\ \left\{P^i_{j_i} \mid i \in \mathbb{Z}\right\} \texttt{!<k>}\} \\ (E,\ \vec{r}, M) \to^{\star} (E', M', flat(\texttt{++}_{i=0}^{n-1}(\vec{s_i})) \\ (E'', \texttt{++}_{i=0}^{n-1}\ \vec{q_i}) = \Sigma(E', \texttt{++}_{i=0}^{n-1}(\vec{s_i})) \\ \vec{\sigma} = flat(\texttt{++}_{i=0}^{n-1}(\vec{q_i}))\end{array}}{(E,\ \vec{r},\ M) \to^{N\star} (E'', M', \vec{\sigma})}$$

7.3 Soundness

In this section we show that a well-typed program does not "go wrong". Going wrong in S-Net means that somewhere inside a network a record is produced that cannot be processed. In the context of our developed semantic framework this would be the case if for a given input there is no transition rule that can be applied to it. This situation is usually referred to as "being stuck"; in simply typed lambda calculus, for example, a term is stuck if no further reductions are possible for a term that is not a value [Pie02].

In order to make statements about well-typedness, we distinguish between well-typedness with respect to a given input (Def. 54) and well-typedness with respect to an output (Def. 55).

Definition 54. *Let E be an oracle stream, r be a record of type τ_r, $\mathfrak{M}$ be a marker tree and M be an* S-Net *program. Furthermore, let*

$$S = \{\mathfrak{M}_i :: C_i, \tau_i^{in} \to \{\tau_{i_1}^{out}, \ldots, \tau_{i_{m_i}}^{out}\}\}_{i=1}^{n}$$

be the inferred signature of M as computed by algorithm S. We say that the tuple

$$(E, (r, \mathfrak{M}), M) :: S$$

is well-typed with respect to S *if*

$$\exists (\mathfrak{M}_i, C_i :: \tau_i^{in} \to T) \in S : \tau_r \sqsubseteq \tau_i^{in}$$

Definition 55. *Let E be an oracle stream, $\vec{s} = \epsilon \triangleright (s_1, \mathfrak{M}_1) \triangleright \ldots \triangleright (s_k, \mathfrak{M}_k)$ be a stream of record-marker tree pairs and M be an* S-Net *program. Furthermore, let*

$$S = \{\mathfrak{M}_i :: C_i, \tau_i^{in} \to \{\tau_{i_1}^{out}, \ldots, \tau_{i_{m_i}}^{out}\}\}_{i=1}^{n}$$

be the inferred signature of M as computed by algorithm S. We say that the tuple

$$(E, M, \vec{s}) :: S$$

is well-typed with respect to S *if*

$$\exists (\mathfrak{M}_i, C_i :: \tau_i^{in} \to T) \in S : \forall (s, \mathfrak{M}) \in \vec{s} : \exists \tau_{i_j} \in T : \tau_s \sqsubseteq \tau_{i_j}$$

where τ_s is the type of record s.

The above definitions express what we intuitively expect from well-typed input and output. For a given program the inferred signature contains a set of possible input types. An input to the program whose type is a subtype of one of these input types we consider to be legal, i.e. the program and the input are well-typed with respect to the inferred signature. For each of the possible input types the inferred signature also contains a set of possible output types. Any output that a program produces as response to one input can be checked against the signature again. If the signature contains one set of output types that covers the types of all records in the produced output, then the program's output is well-typed with respect to the inferred signature.

With the definitions of these properties in place we may now ask the core question of this section. If a program and its input are well-typed, will it produce well-typed output, i.e. is well-typedness a property that is preserved across program application? In order to answer this question, we first formally define this property in the following definition.

Definition 56. *Let E be an oracle stream, r be a record of type τ_r, $\mathfrak{M}$ be a marker tree, M be an* S-NET *program and $\vec{s} = \epsilon \triangleright (s_1, \mathfrak{M}_1) \triangleright \ldots \triangleright (s_k, \mathfrak{M}_k)$ be a stream of record-marker tree pairs. Furthermore, let*

$$S = \mathfrak{M} :: \{C_i, \tau_i^{in} \to \{\tau_{i_1}^{out}, \ldots, \tau_{i_{m_i}}^{out}\}\}_{i=1}^{n}$$

be the best representative for r of the inferred signature of M.

We say that M is well-typedness preserving *if*

$$\begin{aligned} &(E, (r, \mathfrak{M}), M) :: S \textit{ is well-typed} \wedge \\ &\quad (E, (r, \mathfrak{M}), M) \to (E', M', \vec{s}) \quad \Rightarrow \quad (E', M', \vec{s}) :: S \textit{ is well-typed} \end{aligned}$$

We will write $((E, (r, \mathfrak{M}), M) \to (E', M', \vec{s})) :: S$ in this case.

We will show that programs are indeed well-typedness preserving with respect to the signature that is inferred by S and the transition rules presented above. We prove this by first showing that the property holds for simple boxes and the combination of simple boxes. After showing that the property also holds for synchro cells and variant boxes we extend the proof to include programs that contain exactly one combinator. These proofs form the basis for the final proof in which we inductively show that the property also holds for programs with more than one combinator.

As a last prerequisite before diving into the details of the proof for preservation we show that for a program and its input there is always a transition rule applicable and that this rule is uniquely determined. We do this in Theorem 11.

Theorem 11. *Let M be an* S-NET *program, E be an oracle stream and r be a record. Furthermore, let $\mathfrak{M}$ be the marker tree of the best representative for r of the inferred signature S of M. If*

$$(E, (r, \mathfrak{M}), M) :: S$$

is well-typed then there is exactly one transition rule that is applicable, i.e.

$$(E, (r, \mathfrak{M}), M) \to (E', M', \vec{s})$$

is uniquely determined.

Proof. The set of applicable rules is determined by the syntactical structure of M. For boxes, variant boxes, sequential composition, serial and parallel replication there is exactly one rule for each case. Where there are more than one rule that match the syntax of M the associated marker tree and the preconditions of the transition rules narrow the set down to one:

- **synchro cells** A further distinction is made by the current state of the cell. In state $[\![p_1, p_2]\!]$ rules tSYMATCHL, tSYMATCHR and tSYMATCHLR are possible candidates. Transformation 6 ensures that the marker elements for each pattern are both present at the same time, i.e. S_{1o} and S_{1s}simultaneously or S_{2o} and S_{2s}. Additionally, all four markers may be present at the same time. The conditions for these rules are as follows:

$$\begin{array}{ll} \text{tSYMATCHL}: & ((\mathsf{S}_{1o}, M_{Lo}) \in_1 \mathfrak{M} \vee (\mathsf{S}_{1s}, M_{Ls}) \in_1 \mathfrak{M}) \;\wedge \\ & \neg((\mathsf{S}_{2o}, M_{Ro}) \in_1 \mathfrak{M} \vee (\mathsf{S}_{2s}, M_{Rs}) \in_1 \mathfrak{M}) \\ \text{tSYMATCHR}: & ((\mathsf{S}_{2o}, M_{Ro}) \in_1 \mathfrak{M} \vee (\mathsf{S}_{2s}, M_{Rs}) \in_1 \mathfrak{M}) \;\wedge \\ & \neg((\mathsf{S}_{1o}, M_{Lo}) \in_1 \mathfrak{M} \vee (\mathsf{S}_{1s}, M_{Ls}) \in_1 \mathfrak{M}) \\ \text{tSYMATCHLR}: & (\mathsf{S}_{2s}, M) \in_1 \mathfrak{M} \;\wedge\; (\mathsf{S}_{1s}, M') \in_1 \mathfrak{M} \end{array}$$

  If the condition for rule tSYMATCHL is met then the condition of rule tSYMATCHR are not and vice versa. If the condition for rule tSYMATCHLR are met, i.e. the markers for both patterns are present, neither of the previous two rules can be applied as their conditions are not met. In state $[\![p_1, p_2]\!]_{p_1}$ the rules tSYOFLL and tSYMERGEL are possible candidates; the conditions of these rules are mutually exclusive, as the conditions are

$$\begin{array}{ll} \text{tSYOFLL}: & (\mathsf{S}_{1o}, M) \in_1 \mathfrak{M} \\ \text{tSYMERGEL}: & (\mathsf{S}_{2s}, M) \in_1 \mathfrak{M} \;\wedge\; (\mathsf{S}_{1s}, M') \notin_1 \mathfrak{M} \end{array}$$

  State $[\![p_1, p_2]\!]_{p_2}$ is symmetric to the previous case with the according rules. For state Id there is exactly one rule applicable, tSYID.

- **choice combinations** The conditions of tCHOICEL and tCHOICER are mutually exclusive, as the conditions are

$$\begin{array}{ll} \text{tCHOICEL}: & (\mathsf{C}_1, M_L) \in_{\leq 1} \mathfrak{M} \;\wedge\; (\mathsf{C}_2, M_R) \notin_{\leq 1} \mathfrak{M} \\ \text{tCHOICER}: & (\mathsf{C}_2, M_R) \in_{\leq 1} \mathfrak{M} \;\wedge\; (\mathsf{C}_1, M_L) \notin_{\leq 1} \mathfrak{M} \end{array}$$

  In case both markers are present at the same time rule tCHOICELR is the only rule applicable.

□

7.3.1 Preservation for Boxes and Single Combinations

We show that boxes are well-typedness preserving in Theorem 12. Combinations over boxes that contain exactly one combinator are shown to be well-typedness preserving as shown in Lemma 2 and Lemma 3. In each proof we will first show which signature the type inference algorithm infers. We then show how the application of the appropriate transition rules produce output that is well-typed with respect to the inferred signature.

Theorem 12. *The rule* tBox *is well-typedness preserving. Let*

$$B = \texttt{box Id(} \{\alpha_i\}_{i=1}^{m} \texttt{ -> } \{\beta_j\}_{j=1}^{n} \texttt{)}$$

and let S be the inferred signature for B.
If $(E, (r, \mathfrak{M}), B) :: S$ is well-typed then so is

$$((E, (r, \mathfrak{M}), B) \to (E, B, \vec{s})) :: S.$$

Proof. The inferred signature for B is

$$
\begin{array}{rl}
 & \texttt{box Id(} \{\alpha_i\}_{i=1}^{m} \texttt{ -> } \{\beta_j\}_{j=1}^{n} \texttt{)} \\
\overset{\text{Trf. 1}}{\to} & (\mathbf{Box_1}, Id, \{\alpha_i\}_{i=1}^{m}, \{\{\beta_j\}_{j=1}^{n}\}) \\
\overset{\text{Trf. 2}}{\to} & (\mathbf{Box_2}, Id, \{\alpha_i\}_{i=1}^{m}, \{\beta_j\}_{j=1}^{n}) \\
\overset{\text{Trf. 3}}{\to} & (\mathbf{Subst_3}, \{((\mathbf{Box_3}, Id, \{\alpha_i\}_{i=1}^{m}, \{\beta_j\}_{j=1}^{n}), \epsilon)\}) \\
\overset{\text{Trf. 4}}{\to} & (\mathbf{Subst_4}, \\
 & \quad \{(\mathbf{Box_4}, Id, \\
 & \qquad \lambda r.(\texttt{add}_{\beta_1}(\ldots(\texttt{add}_{\beta_n} \\
 & \qquad\qquad (\texttt{rem}_{\alpha_1}(\ldots(\texttt{rem}_{\alpha_m}\ r)\ldots), \\
 & \qquad \{(\texttt{rem}_{\alpha_k} :: \tau_{\texttt{rem}_{\alpha_k}})\}_{k=1}^{n} \\
 & \qquad \cup\{(\texttt{add}_{\beta_l} :: \tau_{\texttt{add}_{\alpha_k}})\}_{l=1}^{m})\}) \\
\overset{\text{Trf. 5}}{\to} / \overset{\text{Trf. 6}}{\to} & (\mathbf{Subst_5}, \\
 & \quad \{(\mathbf{Typ}, Id, \\
 & \qquad \forall r.\overbrace{(r \setminus \alpha_1, \ldots, \alpha_m, \beta_1, \ldots, \beta_n)}^{C} \Rightarrow \\
 & \qquad\quad \llparenthesis \alpha_1, \ldots, \alpha_m \mid r \rrparenthesis \to \llparenthesis \beta_1, \ldots, \beta_n \mid r \rrparenthesis, \\
 & \qquad\quad \epsilon)\}) \\
\to & \{\epsilon :: C, \underbrace{\llparenthesis \alpha_1, \ldots, \alpha_m \mid r \rrparenthesis}_{\tau_{in}} \to \{\underbrace{\llparenthesis \beta_1, \ldots, \beta_n \mid r \rrparenthesis}_{\tau_{out_1}}\}\}
\end{array}
$$

If $(E, (r, \mathfrak{M}), B) :: S$ is well-typed it follows that $\tau_r \sqsubseteq \tau_{in}$; the best representative is the only route type of S. For B the only applicable rule is tBox and hence for

$$(E, (r, (\mathfrak{M})), B) \to (E, B, \vec{s})$$

it holds that $\vec{s} = \epsilon \rhd (s_1, (\mathfrak{M})) \rhd \ldots \rhd (s_q, (\mathfrak{M}))$ and $\forall i \in \{1, \ldots, q\} : \tau_{s_i} \sqsubseteq \tau_{out_1}$. Consequently, $(E, B, \vec{s}) :: S$ is well-typed.

□

Lemma 1. *The* tMap *rule (and consequently $\to^{\star}$) is well-typedness preserving: Let $\vec{s} = \epsilon \rhd s_1 \rhd \ldots \rhd s_n$ and let M be well-typedness preserving with inferred signature S. If*

$$\forall s_i \in \vec{s} : (E, (s, \mathfrak{M}_s), M) :: S$$

is well-typed , then so is

$$((E, \vec{s}, M) \to^{\star} (E, M', \vec{t})) :: S$$

Proof. Straight-forward. □

Lemma 2. *Let S be the inferred signature of $M\,..\,N$.*
If $(E,(r,\mathfrak{M}),M\,..\,N) :: S$ is well-typed and M and N are boxes, then so is

$$((E,(r,\mathfrak{M}),M\,..\,N) \to (E',M'\,..\,N'),\vec{s})) :: S.$$

Proof. Assume $(E,(r,\mathfrak{M}),M\,..\,N) :: S$ is well typed. The inferred signature S for $M\,..\,N$ is

$$\begin{array}{ll}
 & M\,..\,N \\
\overset{\text{Trf. 1}}{\to} & (\mathbf{Seq_1}, M', N') \\
 & \text{where } M' \text{ and } N' \text{ are the transformed operands} \\
\overset{\text{Trf. 2}}{\to},\ldots,\overset{\text{Trf. 4}}{\to} & \text{simultaneous for both operands as in proof of Thm. 12} \\
 & (\mathbf{Subst_4}, \\
 & \quad \{((\mathbf{Seq_4}, \\
 & \qquad (\mathbf{Box_4}, id_N, expr_N, env_N), \\
 & \qquad (\mathbf{Box_4}, id_M, expr_M, env_M)), \epsilon)\}) \\
\overset{\text{Trf. 5}}{\to}(1) & (\mathbf{Subst_4}, \{ \\
 & \quad ((\mathbf{Expr}, \{ \\
 & \qquad id_N \mathbin{\char`^} id_M, \\
 & \qquad \texttt{let}\ id_M = expr_M \\
 & \qquad \texttt{in let}\ id_N = expr_N \\
 & \qquad\quad \texttt{in}\ \lambda.r(id_N\ (id_M\ r)), \\
 & \qquad env_M \cup env_N), \epsilon)\}) \\
\overset{\text{Trf. 5}}{\to}(2) & (\mathbf{Subst_5}, \{ \\
 & \quad ((\mathbf{Typ}, \{ \\
 & \qquad id_N \mathbin{\char`^} id_M, \\
 & \qquad \forall r.C \Rightarrow \alpha \to \beta \\
 & \qquad env_M \cup env_N), \epsilon)\}) \\
\to & \{\epsilon :: C, \alpha \to \{\beta\}\}
\end{array}$$

The construction of the function composition

$$\lambda.r(id_N\ (id_M\ r))$$

implies that the result type of M and the argument type of N are compatible: Let S_M be the inferred signature of M and let S_N be the inferred signature of N

$$\begin{array}{l}
S_M = \{\epsilon :: C_M, \tau_{M_{in}} \to \{\tau_{M_{out_1}}\}\} \\
S_N = \{\epsilon :: C_N, \tau_{N_{in}} \to \{\tau_{N_{out_1}}\}\}
\end{array}$$

It holds that $\tau_{M_{out_1}} \sqsubseteq \tau_{N_{in}}$. From the same construction we can derive that $\alpha \sqsubseteq \tau_{M_in}$ and $\beta \sqsubseteq \tau_{N_{out_1}}$. As $(E,(r,\mathfrak{M}),M\,..\,N) :: S$ is well-typed, so is $(E,(r,\mathfrak{M}),M) :: M$ and with Thm. 12 it follows that $((E,(r,\mathfrak{M}),M) \to (E',M',\vec{s})) :: M$ is well-typed. As

$$\begin{array}{rcl}
\forall s \in \vec{s} & : & \tau_s \sqsubseteq \tau_{M_{out_1}} \sqsubseteq \tau_{N_{in}} \\
 & \Rightarrow & (E,(s,\mathfrak{M}'),N) :: S_N \text{ is well-typed} \\
 & \Rightarrow & ((E,(s,\mathfrak{M}'),N) \to (E',N',\vec{t_s}) :: S_N \text{ is well-typed}
\end{array}$$

and with $\beta \sqsubseteq \tau_{N_{out_1}}$ and Lemma 1 the claim follows. □

Lemma 3. *Let M and N be boxes and let S be the inferred signature of $M\,/\,/N$. If $(M\,/\,/N) :: S$ is well-typed, then so is*

$$((E,(r,\mathfrak{M}),M\,/\,/N) \to (E',M'\,/\,/N',\vec{s})) :: S.$$

Proof. The inferred signature of $M\,|\,|\,N$ is

$$\begin{array}{ll}
& M\,|\,|\,N \\
\overset{\text{Trf. 1}}{\to} & (\textbf{Choice}_\mathbf{1}, M', N') \text{ where } M' \text{ and } N' \text{are the transformed operands} \\
\overset{\text{Trf. 2}}{\to} & (\textbf{Choice}_\mathbf{2}, M', \mathrm{C}_1, N', \mathrm{C}_2) \\
\overset{\text{Trf. 3}}{\to} & (\textbf{Subst}_\mathbf{3}, \{(M',\mathrm{C}_1),(N',\mathrm{C}_2)\}) \\
\overset{\text{Trf. 4}}{\to} & (\textbf{Subst}_\mathbf{3}, \{(M'',\mathrm{C}_1),(N'',\mathrm{C}_2)\}) \\
\overset{\text{Trf. 5}}{\to} & (\textbf{Subst}_\mathbf{5}, \{ \\
& \quad (\textbf{Typ}, Id_M, \forall r.C_M \Rightarrow \tau_M^{in} \to \tau_M^{out}, \epsilon), \\
& \quad (\textbf{Typ}, Id_N, \forall r.C_N \Rightarrow \tau_N^{in} \to \tau_N^{out}, \epsilon)\}) \\
\to & \{(\mathrm{C}_1) :: C_M, \tau_M^{in} \to \{\tau_M^{out}\}, (\mathrm{C}_2) :: C_N, \tau_N^{in} \to \{\tau_N^{out}\}\} = S
\end{array}$$

Let $(E,(r,\mathfrak{M}),M\,|\,|\,N)$ be well-typed. We need to distinguish three cases for the best representative from S for r:

- 1. [$\mathfrak{M} = (\mathrm{C}_1)$] Rule tChoiceL is the only one applicable as the precondition of this is $(\mathrm{C}_1, M_L) \in_{\leq 1} \mathfrak{M} \wedge (\mathrm{C}_1, M_R) \notin_{\leq 1} \mathfrak{M}$. The signature S is the union over the inferred signatures S_M, S_N of M and N with (C_1) as prefix to the marker tree of S_M and (C_2) to S_N. Hence, $(E,(r,(\epsilon)),M) :: S_M$ is well-typed. With Lemma 12 it follows that

  $$((E,(r,(\epsilon)),M) \to (E',M',\vec{s}) :: S_M$$

  is well-typed and this implies that $(E, M'\,|\,|\,N, \vec{s}) :: S$ is well-typed.

- 2. [$\mathfrak{M} = (\mathrm{C}_2)$] analogue to case 1 for rule tChoiceR.

- 3. [$\mathfrak{M} = (\epsilon, (\mathrm{C}_1), (\mathrm{C}_2))$] Depending on E one of the two sub-trees is selected in rule tChoiceLR. Analogue to case 1 if the firs tree is selected, case 2 otherwise.

□

7.3.2 Preservation for Variant Boxes and Synchro Cells

The proofs for variant boxes and synchro cell follow the same structure as before. We will first show how the inferred signature is computed and then show that the produced output is well-typed. In Lemma 4 we show that variant boxes are well-typedness preserving and in Lemma 5 we do the same for synchro cells.

Lemma 4. *A variant box is well-typedness preserving, i.e. if*

$$(E,\ (r, \mathfrak{M}),\ \texttt{box Id(}\ \alpha\ \texttt{->}\ \{\gamma_L, \beta_2\}))::S$$

is well-typed then so is

$$(E,\ \texttt{box Id(}\ \alpha\ \texttt{->}\ \{\gamma_L, \beta_2\}),\ \vec{s})$$

Proof. The inferred signature S of the box is

$$\begin{array}{ll}
 & \texttt{box Id(}\ \alpha\ \texttt{->}\ \{\gamma_L, \gamma_R\} \\
\overset{\text{Trf. 1}}{\rightarrow} & (\mathbf{Seq_1}, \\
 & \quad (\mathbf{Box_1}, Idc, \alpha, \gamma_L \cup \gamma_R)) \\
 & \quad (\mathbf{Choice_1}, \\
 & \quad\quad (\mathbf{Box_1}, Ido_1, \gamma_L \cup \gamma_R, \gamma_L), \mathrm{V}_1, \\
 & \quad\quad (\mathbf{Box_1}, Ido_2, \gamma_L \cup \gamma_R, \gamma_R), \mathrm{V}_2)) \\
\overset{\text{Trf. 3}}{\rightarrow}(1) & (\mathbf{Choice_2}, \\
 & \quad \underbrace{\begin{array}{l}(\mathbf{Seq_2}, \\ \quad (\mathbf{Box_2}, Idc, \alpha, \gamma_L \cup \gamma_R), \\ \quad (\mathbf{Box_2}, Ido_1, \gamma_L \cup \gamma_R, \gamma_L))\end{array}}_{R_1}, \mathrm{V}_1, \\
 & \quad \underbrace{\begin{array}{l}(\mathbf{Seq_2}, \\ \quad (\mathbf{Box_2}, Idc, \alpha, \gamma_L \cup \gamma_R), \\ \quad (\mathbf{Box_2}, Ido_2, \gamma_L \cup \gamma_R, \gamma_L))\end{array}}_{R_2}, \mathrm{V}_2) \\
\overset{\text{Trf. 3}}{\rightarrow}(2) & (\mathbf{Subst_3}, \{(R_1, \mathrm{V}_1), (R_2, \mathrm{V}_2)\}) \\
\vdots & \text{remaining transformation as in Lemma 12 and Lemma 2} \\
\rightarrow & \{\mathrm{V}_1 :: C_{IdcIdo_1}, \tau_\alpha \rightarrow \{\tau_{\gamma_L}\}, \mathrm{V}_2 :: C_{IdcIdo_2}, \tau_\alpha \rightarrow \{\tau_{\gamma_R}\}\}
\end{array}$$

Both inferred route types of S are of the same input type. The best representative for any compatible record will consequently be paired with a marker tree that contains a branch for V_1 and V_2 at the same time. This matches the precondition of TBoxLR. Thus

$$(E,\ \texttt{box Id(}\ \alpha\ \texttt{->}\ \{\gamma_L, \gamma_R\}\}),\ \vec{s})::S$$

is well-typed. □

Lemma 5. *The synchro cell is well-typedness preserving. Let S be the inferred type signature of* `[|p_1, p_2|]` *. Let*

$$X = \{[\![p_1, p_2]\!], [\![p_1, p_2]\!]_{p_1}, [\![p_2, p_2]\!]_{p_2}, Id\}$$

If $(E, (r, \mathfrak{M}), x \in X)::S$ is well-typed then so is $(E, y \in X, \vec{s})::S$ where y represents the synchro cell in its new state.

Proof. The inferred signature S of the synchro cell is

$$
\begin{array}{ll}
 & [\![p_1, p_2]\!] \\
\overset{\text{Trf. 1}}{\to} & (\mathbf{Sync_1}, p_1, p_2) \\
\overset{\text{Trf. 2}}{\to} & (\mathbf{Choice_2}, \\
 & \quad (\mathbf{Choice_2}, \\
 & \quad\quad (\mathbf{Box_2}, OflL, p_1, p_1), \mathrm{S}_{1o}, \\
 & \quad\quad (\mathbf{Box_2}, OflR, p_2, p_2), \mathrm{S}_{2o}), \epsilon \\
 & \quad (\mathbf{Choice_2}, \\
 & \quad\quad (\mathbf{Box_2}, SyL, p_1, p_1 \cup p_2), \mathrm{S}_{1s}, \\
 & \quad\quad (\mathbf{Box_2}, SyR, p_2, p_2 \cup p_1), \mathrm{S}_{2s}), \epsilon) \\
\overset{\text{Trf. 2}}{\to} & (\mathbf{Subst_3}, \\
 & \quad \{((\mathbf{Box_3}, OflL, p_1, p_1), \mathrm{S}_{1o}), ((\mathbf{Box_3}, OflR, p_2, p_2), \mathrm{S}_{2o}) \\
 & \quad\ ((\mathbf{Box_3}, SyL, p_1, p_1 \cup p_2), \mathrm{S}_{1s}), ((\mathbf{Box_3}, SyR, p_2, p_2 \cup p_1), \mathrm{S}_{2s})\}) \\
\vdots & \text{remaining transformation as in Lemma 12} \\
\to & \{(\epsilon, (\mathrm{S}_{1o}), (\mathrm{S}_{1s})) :: (C_{\mathrm{S}_{1o} \cup \mathrm{S}_{2o}}), \tau_{p_1} \to \{\tau_{p_1}, \tau_{p_1 \cup p_2}\}, \\
 & \ (\epsilon, (\mathrm{S}_{2o}), (\mathrm{S}_{2s})) :: (C_{\mathrm{S}_{2o} \cup \mathrm{S}_{2s}}), \tau_{p_2} \to \{\tau_{p_2}, \tau_{p_1 \cup p_2}\}\}
\end{array}
$$

For the best representative of a record r we have to distinguish 3 cases for $\tau_r \sqsubseteq \tau_{p_1} \wedge \tau_r \not\sqsubseteq \tau_{p_2}$, $\tau_r \not\sqsubseteq \tau_{p_1} \wedge \tau_r \sqsubseteq \tau_{p_2}$ and $\tau_r \sqsubseteq \tau_{p_1} \wedge \tau_r \sqsubseteq \tau_{p_2}$ as this determines the structure of the marker tree.

- 1. [$\mathfrak{M} = (\epsilon, (\mathrm{S}_{1o}), (\mathrm{S}_{1s}))$] The rules applicable in this case are TSYMATCHL,TSYOFLL, TSYMERGEL, and TSYID. Depending on the current state of the cell, these rules either produce an empty stream (TSYMATCHL) or a one-element stream which contains as output the inbound record without modification (TSYOFLL, TSYID) or the result of merging the inbound record with the record in storage. For the inbound record's type it holds that $\tau_r \sqsubseteq \tau_{p_1}$; the signature S of the cell contains $\{\tau_{p_1}, \tau_{p_1 \cup p_2}\}$ as inferred output types for an input of this type. Thus, $(E, y \in X, \vec{s}) :: S$ is well-typed.
- 2. [$\mathfrak{M} = (\epsilon, (\mathrm{S}_{2o}), (\mathrm{S}_{2s}))$] This is the symmetric case to 1. for the second pattern.
- 3. [$\mathfrak{M} = (\epsilon, (\mathrm{S}_{1o}), (\mathrm{S}_{1s}), (\mathrm{S}_{2o}), (\mathrm{S}_{2s}))$] Applicable rules in this case are TSYOFLL, TSYOFLR, TSYMATCHLR, and TSYID. The inbound record's type is a subtype of $\tau p_1 \cup p_2$, as all rules do not modify the record $(E, y \in X, \vec{s}) :: S$ is well-typed.

□

7.3.3 Preservation for Sequential Combinations and Choice Combinations

In this section we extend the proofs for the combinators to include all of the basic constructs of the language. We limit ourselves here to programs that contain exactly one combinator. We show that sequential combinations are well-typedness preserving in Theorem 13 and do the same for choice combinations in Theorem 14.

Theorem 13. *A typed* S-NET *program that contains exactly one sequential composition and no other combinators is well-typedness preserving. Let S be the inferred signature of $M\,..\,M$ and let $(E,(r,\mathfrak{M}),M\,..\,N) :: S$ be well-typed. Then*

$$((E,(r,\mathfrak{M}),M\,..\,N) \to (E',M'\,..\,N',\vec{t})) :: S$$

is well-typed.

Proof. We have to distinguish several cases to capture the possible choices of M and N.

Cases	Box	var. Box	Synchro cell
Box	1	2	3
var. Box	4	5	6
Synchro cell	7	8	9

From Lemma 12,4,5 we know that M and N are well-typedness preserving.

1. Lemma 2
2. The two separate routes that are created during type inference for a variant box (Lemma 4) are both prefixed with M during Transformation 3. Following the argument of Lemma 2 the output type of M and input of N are compatible. Transformation 6 ensures that both outputs of the variant box are covered by the output type.
3. Analogue to 2. except that Transformation 6 only ensures that at least one pattern is covered for all states of the synchro cell. This consequently ensures that the inferred output type contains $\tau_{p_1 \cup p_2}$ and τ_{p_1} or τ_{p_2} or both; the marker trees mask those rules of the synchro cell transitions that lead to output types not covered in the inferred signature.
4. The variant box is represented by two paths, each of which is postfixed by N. The route stabilisation of Transformation 6 ensures that every acceptable inbound record r is paired with a marker tree that covers both variants of the variant box, i.e. the precondition of TBOXLR is met. The output types of M are either all sub-types of the input type of N or the types are unrelated (otherwise route stabilisation would have removed both branches resulting in an empty signature), i.e. the inferred signature S contains two output types capturing all possible outputs (analogue to Lemma 3 and Lemma 2).
5. Analogue to 4. except that S contains four output types as the two variant boxes establish four conceptual paths.
6. Analogue to 3. for each of the two conceptual branches of M.
7. An inferred type for a synchro cell contains four output types as shown in Lemma 5. During type inference all conceptual branches are suffixed by N resulting in at most four output types in S. Route stabilisation proceeds analogue to case 3. to prevent uncovered outputs from being produced.

8. Analogue to 7. except that route stabilisation additionally ensures that both variants of the box are covered.

9. Analogue to 3. and 7..

□

Theorem 14. *A* S-NET *program that contains exactly one choice combinator and no other combinators is well-typedness preserving. Let S be the inferred signature of $M\,|\,|M$ and let $(E,(r,\mathfrak{M}),M\,|\,|N) :: S$ be well-typed. Then*

$$((E,(r,\mathfrak{M}),M\,|\,|N) \to (E',M'\,|\,|N',\vec{t})) :: S$$

is well-typed.

Proof. The inferred type of $M\,|\,|N$ is the union over the inference results for M and N plus an added prefix of $C_1$to all marker strings of the result for M and $C_2$for the results of N. As M and N are well-typedness preserving the claim follows. □

7.3.4 Preservation for S-NET Programs

In this section we prove by induction that programs are well-typedness preserving. In order to do this we use the results of the previous sections as induction basis. In Theorem 15 we do not cater for parallel and sequential replication. We deal with these in Corollary 1 and Corollary 2.

Theorem 15. *An* S-NET *program P with an inferred signature S is well-typedness preserving, i.e. if $(E,(r,\mathfrak{M}),P) :: S$ is well-typed, then so is*

$$((E,(r,\mathfrak{M}),P) \to (E',P',\vec{s})) :: S$$

Proof. We prove this by induction over the number of combinators in P. The induction basis for 0 and 1 is established by Lemma 12, 4, 5 and Theorem 13 and 14. For the induction step we distinguish two cases 1. $P = M\,.\,.\,N$ and 2. $P = M\,|\,|N$.

1. The inferred signature S is of the form

$$\begin{array}{cl}
 & M\,.\,.\,N \\
\overset{\text{Trf. 1}}{\to} & (\mathbf{Seq_1}, M', N') \text{ where } M', N' \text{ are the transformed operands} \\
\vdots & \text{transformations applied to both operands independently} \\
\overset{\text{Trf. 3}}{\to} & (\mathbf{Subst_3}, \{((\mathbf{Seq_3}, M_i, N_j), R_{M_i} \wedge R_{N_j})\}_{i=j=1}^{i=m,j=n}) \\
 & \text{routes of both operands are pair-wise combined} \\
 & \text{and their marker strings } R \text{ are concatenated} \\
\vdots & \\
\overset{\text{Trf. 5}}{\to} & (\mathbf{Subst_5}, \underbrace{\{(\mathbf{Typ}, Id_i, \forall r.C_i \Rightarrow \tau_i^{in} \to \tau_i^{out}, R_i)\}_{i=1}^{k \leq m \cdot n}}_{T}) \\
\overset{\text{Trf. 6}}{\to} & (\mathbf{Subst_5}, \underbrace{\{(\mathbf{Typ}, Id_i, \forall r.\hat{C}_i \Rightarrow \hat{\tau}_i^{in} \to \hat{\tau}_i^{out}, R_i)\}_{i=1}^{l \leq k}}_{\hat{T}}) \\
\to & \{\mathfrak{M}_i :: \hat{C}_{i_1} \cup \ldots \cup \hat{C}_{i_{n_i}}, \hat{\tau}^{in} \to \{\hat{\tau}_{i_1}^{out}, \ldots, \hat{\tau}_{i_{n_i}}^{out}\}\}_{i=1}^{j \leq l}
\end{array}$$

From the construction of the route stabilisation algorithm used by Transformation 6 we make the following two observations:

Observation 1.

$$\begin{array}{lcl}
\forall(\mathbf{Typ}, Id, \forall r.\hat{C} \Rightarrow \hat{\tau}^{in} \to \hat{\tau}^{out}, R) \in \hat{T} & & \\
\quad \exists(\mathbf{Typ}, Id, \forall r.C \Rightarrow \tau^{in} \to \tau^{out}, R) \in T & : & \hat{\tau}^{in} \sqsubseteq \tau^{in} \;\wedge \\
 & & \quad \hat{\tau}^{out} \sqsubseteq \tau^{out}
\end{array}$$

Observation 2. $\forall(\mathfrak{M} :: \tau^{in} \to \{\tau_1^{out}, \ldots \tau_n^{out}\}) \in S$:

$$\begin{array}{lcl}
(V_1, J) \in_j \mathfrak{M} & \Longleftrightarrow & (V_2, J') \in_j \mathfrak{M} \\
(S_{1o}, K) \in_k \mathfrak{M} & \Longleftrightarrow & (S_{1s}, K') \in_k \mathfrak{M} \\
(S_{2o}, L) \in_l \mathfrak{M} & \Longleftrightarrow & (S_{2s}, L') \in_l \mathfrak{M}
\end{array}$$

for $j, k, l \in \mathbb{N}_0$ and J, K, L are lists of sub-trees.

Let $(E, (r, \mathfrak{M}), M\,.\,.\,N) :: S$ be well-typed and let S_M be the inferred signature of M and S_N of N. From Observation 1 it follows that $(E, (r, \mathfrak{M}), M) :: S_M$ is well-typed; with Observation 2 we know that the assumptions on the presence and absence of labels as used in previous proofs are still valid. By the induction hypothesis it holds that

$$((E, (r, \mathfrak{M}), M) \to (E', M', \vec{s})) :: S_M$$

is well-typed. By construction of the function composition in sequential combinations (proof of Lemma 2) the output types of M and the input types of N are compatible, and thus $(E', \vec{s}, N) :: S_N$ is well-typed; by the induction hypothesis it holds that

$$((E', \vec{s}, N) \to^{\star} (E'', N', \vec{t})) :: S_N$$

is well-typed too. With Observation 1 it follows that

$$(E'', M' \mathbin{..} N', \vec{t}) :: S$$

is well-typed.

2. The inferred signature S is of the form

$$\begin{array}{ll}
 & M \mid\mid N \\
\overset{\text{Trf. 1}}{\rightarrow} & (\mathbf{Choice_1}, M', N') \text{ where } M', N' \text{ are the transformed operands} \\
\overset{\text{Trf. 2}}{\rightarrow} & (\mathbf{Choice_2}, M', \mathrm{C}_1, N', \mathrm{C}_2) \\
\vdots & \\
\overset{\text{Trf. 3}}{\rightarrow} & (\mathbf{Subst_3}, \underbrace{\{(M'_i, \mathrm{C}_1 \wedge J_{M_i})\}_{i=1}^{m}}_{\text{routes of } M} \cup \underbrace{\{(N'_i, \mathrm{C}_1 \wedge J_{N_i})\}_{i=1}^{n}}_{\text{routes of } N}) \\
\vdots & \\
\overset{\text{Trf. 5}}{\rightarrow} & \overbrace{\{\mathfrak{M}_1 :: C_1, \tau_1^{in} \rightarrow \tau_1^{out}, \ldots, \mathfrak{M}_p :: C_p, \tau_p^{in} \rightarrow \tau_p^{out},}^{(\mathrm{C}_1, J_i) \in_{\leq 1} \mathfrak{M}_i,\ i \in \{1,\ldots,p\}} \\
 & \underbrace{\mathfrak{M}_q :: C_q, \tau_p^{in} \rightarrow \tau_q^{out}, \ldots, \mathfrak{M}_n :: C_n, \tau_n^{in} \rightarrow \tau_n^{out}\}}_{(\mathrm{C}_2, J_i) \in_{\leq 1} \mathfrak{M}_i,\ i \in \{q,\ldots,n\}} \\
\rightarrow & \{\mathfrak{M}_i :: C_{i_1} \cup \ldots \cup C_{i_{n_i}}, \tau_i^{in} \rightarrow \{\tau_{i_1}^{out}, \ldots, \tau_{i_{n_i}}^{out}\}\}_{i=1}^{m \leq n}
\end{array}$$

Let $(E, (r, \mathfrak{M}), M \mid\mid N) :: S$ be well-typed. The inferred signature contains the inferred types of both operands where the marker strings of the left operand are prefixed by C_1 and the marker strings of the right operand are prefixed by C_2. Using the same case distinction over the first element in the marker string as in the proof of Lemma 3 the well-typedness is directly dependent on the well-typedness of the two operands; the operands are well-typedness preserving (induction hypothesis). Hence,

$$(E', M' \mid\mid N', \vec{s}) :: S$$

is well-typed.

□

Corollary 1. *Let M be an* S-Net *program with inferred signature S. If M is well-typedness preserving, then so is*

$$\begin{aligned}
&(E,\ (r, \mathfrak{M}),\ \left\{M_{j_i}^{i} \mid i \in \mathbb{Z}\right\} \texttt{!!<k>}) \\
&\quad \rightarrow (E',\ ((\left\{M_{j_i}^{i} \mid i \in \mathbb{Z}\right\} \setminus \{M_{j_v}^{v}\}) \cup \{M_{j'_v}^{v}\})\texttt{!!<k>},\ \vec{s})
\end{aligned}$$

Proof. The signature for the split combinator is computed by prefixing M with a box that consumes and produces tag <k>. The result of applying the transition rule for the combinator is identical with the result of M and hence the claim follows as a direct consequence of Thm. 15. □

Corollary 2. *Let M be an* S-Net *program with inferred signature S. If M is well-typedness preserving, then so is*

$$(E,\ (r, \mathfrak{M}),\ M\,\texttt{**}p) \to (E',\ N',\ \vec{s})$$

if $(E',\ N',\ \vec{s})$ exists, i.e. if $M\,\texttt{**}p$ terminates. If the recursive descent does not terminate all new instances are still well-typed.

Proof. As discussed in Sect. 6.9.1 and Transformation 7 the signature of the combinator is inferred by computing the fix-point of possible input and output types of a sequential chain of $M\,\texttt{||}\,B$ where B is a box that takes p as an input and produces the same as its output. Thus, the claim follows with Thm. 15. □

8 Implementing S-Net

The coordination approach to concurrency management that we have promoted so far leaves the design space for a concrete implementation of S-Net fairly open. This chapter presents a stream-based runtime system implementation and sketches an accompanying tool-chain. The implementation is suitable for multi-threaded execution on shared-memory machines.

The implementation is based on explicitly typed S-Net. However, as all type related runtime decisions are abstracted out into separate functions, amending the implementation to support implicitly typed S-Net is possible with small effort. Instead of compiler generated decision functions that are based on types we employ decision functions that query and update marker structures instead.

An earlier version of this chapter was published in [GP10].

8.1 Implementation Architecture

The implementation architecture of S-Net comprises two major parts: a compiler that translates S-Net source code into an intermediate representation and a collection of runtime systems. Each runtime system implements its own technique to execute the program encoded in the intermediate representation. Fig. 8.1 gives a general view on the architecture.

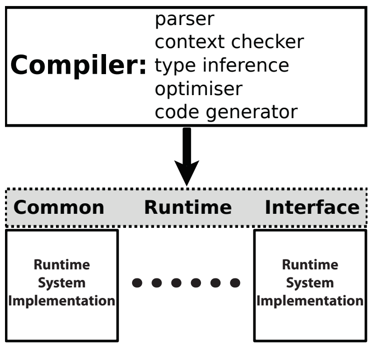


Figure 8.1: The general system architecture of S-Net consists of a compiler that transforms a program into an intermediate representation. Several runtime systems implement a concrete interpretation of the intermediate code.

The compiler parses an S-Net source file and constructs an internal representation from it. On this internal representation the compiler carries out basic consistency checks and it may also apply several optimisation steps to the program. If we are dealing with typed S-Net, then the compiler does also implement the type inference algorithm and it computes all required type information automatically. In this case any user-annotated types are checked for consistency. User-annotated types are also used by the compiler to prune the set of exposed routes at network boundaries where present.

The last stage of the compilation process is code generation. The compiler generates a representation of the original program in a format that we call *common runtime interface* (CRI) code. Although the CRI program if effectively plain C code it is still on a high level of abstraction which is heavily reliant on functionality provided by the runtime system.

The CRI decouples the compilation process from the runtime system. The compiler does not dissolve the hierarchical structure of the network, which leaves this traditional compiler task to the runtime system. This achieves high flexibility for runtime system implementers as the compiled structure still features a high level of abstraction, the actual decomposition, i.e. the interpretation of the CRI format, takes places in the runtime system according to the target architecture. This approach turns the compiler into a universal component which we can re-use for any concrete implementation of a runtime system. Interchanging the runtime system does not even require re-compilation.

The transformation from CRI format to the final and actually executed representation of the network is carried out by a component that any runtime-system implementation has to provide: the *deployer*. The deployer is specific to a runtime-system as it implements the final transformation of the network representation. This approach allows for a concrete implementation of the runtime representation to be entirely decoupled from any stages above the deployer.

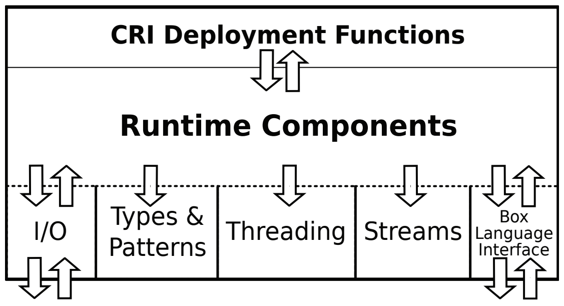


Figure 8.2: The implementation design of the multi-threaded runtime system consists of several modules, each providing separate functionalities. Interaction between modules is indicated by arrows.

In the remainder of this chapter we will develop the ideas that lead to a runtime system that is appropriate for multi-threaded execution of S-Net programs. The main idea of this implementation is to break down the network into smaller runtime components which are connected to each other by one-way fifo channels, i.e. streams. The compiled program which is being executed in this implementation resembles the intuitive view of a network in which data elements are flowing from component to component for processing.

The general implementation design of the runtime system is illustrated in Fig. 8.2: Apart from the deployer, the runtime system consists of several smaller modules for type and pattern representation, thread management, communication, box language interfacing, and general I/O. These modules provide functionality for the runtime components, which form the core of the runtime system. These components implement the runtime behaviour of all S-Net combinators.

8.2 Network Deployment

The network deployer transforms a compiled, but still abstract and hierarchical network specification into a flat network of runtime components. During this process, two main steps are performed. Firstly, the required infrastructure for each S-Net entity is set up. This step is necessary, as most entities are mapped to more than one runtime component. Also, whereas all S-Net entities are strictly single-input and single-output boxes, certain runtime components are required to maintain multiple input or output streams. Secondly, the runtime component(s) of all entities are connected to each other according to the network specification that was compiled.

The deployer also constitutes the boundary between the compiler and the runtime system. In between, the common runtime system interface (CRI) defines the API, to which both sides adhere. The compiler generates a call to the deployer, which in turn calls the generic deployment functions of the runtime system. In case of combinators, the execution of the deployment function is a recursive process: In addition to setting up required runtime components, the deployment function is also applied to the operand(s) of the combinator. Only in case of boxes the recursion ends, as these are the atomic building blocks of a network.

As we will see, the deployment function requires two arguments, a stream and an S-Net expression in CRI format. The stream is connected to the first runtime component that results from processing the S-Net expression. Upon completion, each call of the deployment function yields the outbound stream of their respective last component. It is due to this design that only one stream has to be created outside the generated deployment function call, namely the global inbound stream of the network. The global outbound stream is returned by the outermost deployment function. Once this function returns the static component network has been fully set up and is ready to process records between these two streams.

We will take a closer look at this process for each S-Net entity.

8.2.1 Deployment of Boxes and Synchro Cells

The deployment of boxes and synchro cells (Fig. 8.3) form the base cases for the recursive deployment process. As both entities do not have operands, their deployment does not induce a descent into further deployment functions. Each deployment function connects an inbound stream to the appropriate runtime component. From the stream, the box resp. synchro cell reads inbound records for processing. Resulting output records that are produced by the component are sent via the outbound stream that the deployer has created by calling `new Stream()`. In addition to these streams, both components require auxiliary parameters: For the deployment of the synchro cell, the compiler generates two decision functions, μ_a and μ_b, from the user-defined patterns of the synchro cell. For a box deployment, the user-defined, internal behaviour f of the box (the box implementation) is required. Additionally, the box component also requires a compiler-generated type encoding τ of the box's input type. The purpose of these parameters are explained in greater detail in Sect. 8.3. The runtime components are started by a call to `spawn`.

The `new` and `spawn` functions are high-level abstractions of thread creation functions. We do this for the sake of presentation as code for a concrete threading API, e.g. PTHREADS, would be less concise (but trivial).

```
𝒟(box_τ f, in) =
  let out = new Stream()
      box = spawn Box( in, f, τ, out)
  in out

𝒟([|μ_a, μ_b|], in) =
  let out = new Stream()
      sync = spawn Sync( in, μ_a, μ_b, out)
  in out
```

Figure 8.3: The deployment of boxes and synchro cells does not descend into further deployments. The inbound and outbound streams are created and connected to the runtime component.

8.2.2 Deployment of Combinators

We deal with the deployment of each combinator individually.

Sequential Composition

The simplest case of combinator deployment is the deployment of a *serial* combinator (Fig. 8.4). Both operands are deployed recursively. The inbound stream is connected to the first operand. The outbound stream of the deployed first operand is connected to the second operand, whose outbound stream constitutes the outbound stream of the compound runtime component network, representing this serial combination.

```
$\mathscr{D}$(A .. B, in) =
  let out = $\mathscr{D}$(A, in)
  in  $\mathscr{D}$(B, out)
```

Figure 8.4: The deployment of a *serial* combination connects the outbound stream of its left-hand operand to the inbound stream of the right-hand operand.

Choice Composition

The deployment of the *choice* combinator (Fig. 8.5) requires two runtime components in addition to the components of its operands. These additional components implement the implicit splitting and merging points of streams in an S-NET *choice* combination. The dispatcher, a multi-outbound stream component, forwards inbound records to one of the operands. The output streams of the operands are connected to a complementary multi-inbound stream component, the collector. The collector aggregates the output streams of the operands and bundles these into a single output stream. The CRI representation of the *choice* combinator requires a decision function δ, which the compiler generates. The dispatcher evaluates this function for each inbound record to determine routing destinations. The deployment of the choice combinator also deploys the operands of the combinator. This recursive process only ends once an operand does not have any more operands, i.e., if it is a box or synchro cell.

```
$\mathscr{D}$(A |$_\delta$ B, in) =
  let opin$_1$ = new Stream()
      opin$_2$ = new Stream()
      disp =
        spawn ChoiceDispatch(in, $\delta$, opin$_1$, opin$_2$)
      opout$_1$ = $\mathscr{D}$(A, opin$_1$)
      opout$_2$ = $\mathscr{D}$(B, opin$_2$)
      out = new Stream()
      coll =
        spawn Collector(nil, {opout$_1$,opout$_2$},out)
  in out
```

Figure 8.5: The choice combinator deployment sets up two multi-stream components and also triggers deployment of its operands.

Star Combinator Deployment

Similar to the *choice* deployment, the deployment of a *star* combinator (Fig. 8.6) also sets up a dispatcher and a collector. The operand of the *star*, however, is not deployed yet — its deployment is fully demand-driven and postponed until runtime. For this reason, the operand is passed directly to the dispatcher, in conjunction with a compiler-generated decision function ϵ. The dispatcher calls the deployment function for the operand only when needed, driven by the outcome of the decision function. As the operand network of the *star* combinator may evolve (unfold) over time, the associated collector has to be able to manage a potentially growing set of inbound streams. A control stream between dispatcher and collector is set up to serve the purpose of communicating the appearance of new streams to the collector.

```
𝒟(A *ϵ, in) =
  let ctrl = new Stream()
      byps = new Stream()
      disp =
        spawn StarDispatch(in, A, ϵ, ctrl, nil, byps)
      out = new Stream()
      coll = spawn Collector( ctrl, {byps}, out)
  in out
```

Figure 8.6: During the deployment of the *star* combinator, the operand is not deployed but passed to the dispatcher component. The deployment is demand-driven and determined by the ϵ function.

Split Combinator Deployment

From the deployment perspective, the *split* combinator (Fig. 8.7) is very similar to the *star* combinator. The operand instantiation is demand driven and triggered by a decision function ν. After the initial deployment, no operand instance is present, and thus, the stream set of the collector is empty. A control stream is established between the dispatcher and the collector which is used to register the outbound streams of dynamically created, new instances of the operand with the collector.

Example: Deployment of a Network

To illustrate the deployment process, the resulting component network for the network (for some boxes `A, B, C, D`)

```
  (A | B) .. (C!<t>) * {p} .. D
```

is shown in Fig. 8.8(a). Instances of C and the surrounding splitter are not built by the deployment process until required. Instead, only an initial connection between the

```
$\mathscr{D}$(A !$_\nu$, in) =
  let ctrl = new Stream()
      disp = spawn SplitDispatch( in, A, $\nu$, ctrl, $\emptyset$)
      out = new Stream()
      coll = spawn Collector( ctrl, $\emptyset$, out)
  in out
```

Figure 8.7: After deployment of the *split* combinator, no instance of the operand is present. Merely a control stream connects the dispatcher to the collector.

star dispatcher and its collector ensures that the network is fully connected. While processing records, instances of the star operand and the split operand are spawned demand-driven. The component network as it has developed after one instance of the star operand and three instances of the split operand have been built is shown in Fig. 8.8(b).

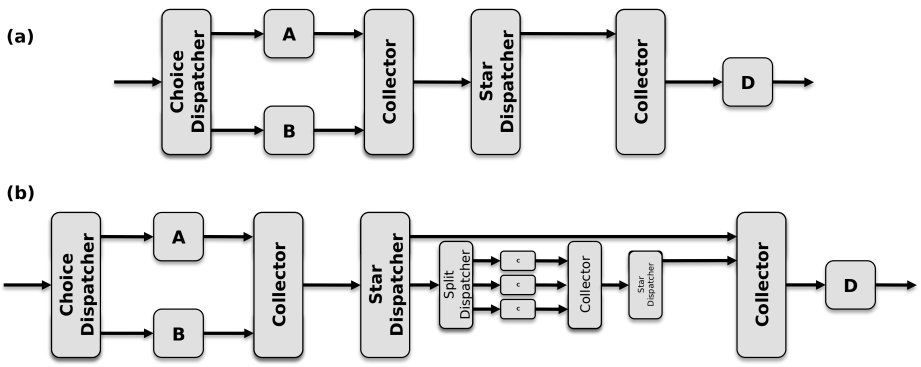


Figure 8.8: The deployed component network of our example network initially only contains A, B, D, and the required split and merge components (a). The operand of the star is only built on demand at runtime. This is shown in (b) where one instance of the star operand and three instances of the split operand have been built.

8.3 Implementation of Components

In this section we develop implementations for the operational behaviour of the components that form the runtime representation of a compiled and deployed S-Net program.

We continue to use ML style pseudo code for the presentation of component implementations. Each component definition starts with the keyword **Thread** to emphasise the fact that the component may be executed as an independent thread. Case differentiations within a function definition are introduced by a | in the source code, guard expressions of cases by the keyword **when**.

Implementation of the Synchro Cell

The synchro cell component implements two main tasks. Firstly, it stores a record if it matches the specified synchronisation pattern. The procedure here again relies on match functions: The compiler generates one match function for each synchro cell pattern. Secondly, the component merges records once all pattern have been matched. We model record storage as parameters to the component function. This serves a dual purpose; it stores the records if they match a pattern and also encodes the state of the synchro cell. The state determines the synchro cell's operation depending on which pattern was matched by an inbound record. A synchro cell of two patterns (see Fig. 8.10; if a storage parameter does not hold a record yet, this is indicated by -) and two match functions μ_a and μ_b, has the following possible states and transitions:

S_i	μ_a	μ_b	Description (current)	Action	S_{i+1}
- -	●		initial state	store record	q -
- -		●	initial state	store record	- q
- -	●	●	initial state	output	*id*
q -	●		first pattern was matched	output	q -
q -		●	first pattern was matched	merge and output	*id*
q -	●	●	first pattern was matched	merge and output	*id*
- q	●		second pattern was matched	merge and output	*id*
- q		●	second pattern was matched	output	- q
- q	●	●	second pattern was matched	merge andoutput	*id*
id	n/a	n/a	sync replaced by identity	output	*id*

When a record is stored in the synchro cell, all constituents of the record which are not part of the pattern, are stripped out. This is implemented by the `strip` function which uses the decision functions to determine which record constituents are to be removed. Only the remainder is stored for the merging process. If a record matches a pattern for which a record has already been stored, the *output* action forwards the record to the outbound stream `out`. A record that matches the last remaining previously unmatched pattern is merged if a record is available in storage and simply output otherwise. Record merging is defined by the flow inheritance operator shown in Fig. 8.9. For clarity, we presented an implementation where the sync component is replaced by an

```
fun infix ⋈ f r = r ∪ (f \ r)
```

Figure 8.9: The flow inheritance operator inserts constituents of the left operand into the right operand, if not already present.

Id component after the last pattern has been matched. The Id component forwards all inbound records directly to the outbound stream. In practice, however, dead synchro cells are completely removed in a garbage collection step, where the cell's inbound stream is directly connected to its successor component.

```
Thread Sync( r◁in, $\mu_a$, $\mu_b$, -, -, out) when $\mu_a \wedge \mu_b$ =
         Id( in, out▷r)
|      Sync( r◁in, $\mu_a$, $\mu_b$, -, -, out) when $\mu_a$ =
       let q = strip( r, $\mu_a$)
       in Sync( in, $\mu_a$, $\mu_b$, q, -, out)
|      Sync( r◁in, $\mu_a$, $\mu_b$, -, -, out) =
       let q = strip( r, $\mu_b$)
       in Sync( in, $\mu_a$, $\mu_b$, -, q, out)
|      Sync( r◁in, $\mu_a$, $\mu_b$, q, -, out) when $\neg\mu_b$ =
         Sync( in, $\mu_a$, $\mu_b$, q, -, out▷r)
|      Sync( r◁in, $\mu_a$, $\mu_b$, q, -, out) =
       let m = r ⋈ q
       in Id( in, out▷m)

Thread Id(r◁in, out) = Id(in, out▷r)
```

Figure 8.10: The sync component stores and merges records. To do this, match functions are employed to check which pattern an inbound record satisfies before storing it. The flow inheritance operator implements the merging process. For brevity, we omit the symmetric cases for state (- q).

Implementation of the Box Component

The box component establishes the boundary between the S-Net domain and the box language domain. The box component (see Fig. 8.11) calls the box function and provides it with an inbound record and the inbound type τ of the S-Net box. The box function may produce an arbitrary amount of records during its execution, each of which needs to flow inherit fields from the original inbound record. To make this process convenient for a box programmer, an SNetOut function is provided. This function expects one result record at a time, carries out the necessary flow inheritance operations and writes the record to the outbound stream. For each output the box produces, it calls SNetOut. After the execution of the box functions finishes, control is returned to the box component.

```
Thread Box( r◁in, f, τ, out) =
       let t' = f( r, τ, out)
       in Box( in, f, τ, t')


fun SNetOut( r, τ, res, out) =
    let f  = r\τ
        rf = f ⋈ res
    in out▷rf
```

Figure 8.11: The box component (left) calls the box implementation f and passes the current inbound record to it. To output records, f calls SNetOut (right) with the original inbound record r and the freshly produces resulting record res. This function carries out flow inheritance and sends the result to the outbound stream of the box component.

Implementation of the Multi-Stream Collector

The collector (Fig. 8.12) is a multi-inbound stream component. This component is used where multiple operand streams are merged into one single outbound stream. The collector keeps all streams that it monitors in a stream set S. When records become available on any of the streams in the set, the record is read from the stream and forwarded to the outbound stream `out`. The collector is always deployed as part of a dispatcher-collector pair, with a control stream connecting these two. The registration of new channels is implemented using this control stream: Dispatchers send streams of dynamically created operand instances via `ctrl` to the collector, where the streams are added to the stream set.

```
Thread Collect(in◁ctrl, S, out) =
         Collect( ctrl, {in}∪S, out)
|      Collect( ctrl, {r◁in} ∪ S, out) =
         Collect( ctrl, {in} ∪ S, out▷r)
```

Figure 8.12: The collector maintains a stream set of monitored streams. If records become available, they are sent to the outbound stream. New streams are received via a control stream.

Implementation of the Choice Dispatcher

The choice dispatcher is a multi-outbound stream component. It reads records from its single inbound stream, and forwards the records over one of the outbound streams to the operand networks. The compiler-generated δ function is an integral part of this process. The compiler generates this function from the input types of the *choice* operands. Applied to a record r, δ returns an integer value n, depending on which operand input type the record matched. The dispatcher shown in Fig. 8.13 reads a record r from the inbound channel `in`. If δ applied to r evaluates to 1, the record is forwarded to the inbound channel o_1 of the first operand, and to the second operand via o_2 otherwise. We chose to design δ as a function to integers and not to a binary set, which would be sufficient for this purpose. The integer domain enables us to implement an optimisation to reduce the overhead that multiple dispatcher would cause. The optimisation maps an n-fold choice combination to a single, n-channel choice dispatcher, as opposed to $n-1$ binary dispatchers.

```
Thread ChoiceDispatch(r◁in, δ, o₁, o₂) when δ(r) = 1 =
           ChoiceDispatch(in, δ, o₁ ▷r, o₂)
|        ChoiceDispatch(r◁in, δ, o₁, o₂) =
             ChoiceDispatch(in, δ, o₁, o₂▷r)
```

Figure 8.13: The choice dispatcher evaluates δ for each inbound record and routes the record to an operand, depending on the result of the decision function.

Implementation of the Star Dispatcher

The main purpose of the star dispatcher (Fig. 8.14) is to decide, whether an inbound record matches the exit pattern of the *star* combinator or not. If the record matches, the dispatcher sends the record to the outbound stream. If the record does not match the pattern, the dispatcher sends the record to the operand network. To make this decision, the dispatcher employs a decision function m. This function is generated by the compiler from the exit pattern of the *star* combinator. When applied to a record, the decision function evaluates to true, if the record matches the exit pattern, and to false otherwise. The instantiation of operands is demand-driven, and hence the star dispatcher is initially not connected to any operand. After deployment, the only connections the dispatcher maintains are an outbound channel (bypass, `bps`) and a control channel (`ctrl`) to the collector. The operand has not been deployed, and the continuation stream `cont` not yet been built. This setup does not change, as long as all inbound records match the exit pattern. The dispatcher immediately sends matching records via `b` to the collector. In case a record does not match the pattern, the operand is deployed.To do this, the `cont` stream is created and connected as inbound stream to the operand. The dispatcher now sends all records that do not match the exit pattern to this continuation stream for processing by the operand. As *star* is a feed-forward combinator, all output of the operand is sent to a new instance of the combinator. This is achieved by instantiating a new dispatcher, in the same way the current dispatcher was set up by the deployment function. No new collector needs to be instantiated:

The already existing collector is notified via the control stream. The new dispatcher instance sends all records that match the exit pattern via stream `bps'` to the collector. If the pattern is not matched, the described process is repeated.

```
Thread StarDispatch(r◁in, N, m, ctrl, cont, bps)
       when m(r) =
         StarDispatch(in, N, m, ctrl, cont, bps▷r)
| StarDispatch(r◁in, N, m, ctrl, nil, bps) =
  let cont = new Stream()
      out = 𝒟(N, cont▷r)
      bps' = new Stream()
      disp =
        spawn StarDispatch(out, N, m, ctrl, nil, bps')
  in StarDispatch(in, N, m, ctrl▷bps', cont, bps)
| StarDispatch(r◁in, N, m, ctrl, cont, bps) =
    StarDispatch(in, N, m, ctrl, cont▷r, bps)
```

Figure 8.14: The star dispatcher starts new instances of the operand and dispatcher demand-driven. If the exit pattern is matched, the dispatcher sends records to its output (bypass) channel `bps` and to the continuation channel otherwise.

Implementation of the Split Dispatcher

The split dispatcher (Fig. 8.15) sends records to one instance of its operand. This instance is determined by the value of the given tag at runtime. The compiler generated function ν is used to extract the tag value from inbound records. This function returns the integer value of the appropriate tag from a record. The split dispatcher deploys instances demand-driven, an thus, no instance is present initially. When a new instance is deployed, the inbound stream instance is added to the set of served channels. The dispatcher associates the tag value with the instance (more specifically, with the inbound stream) and sends the new outbound stream to the collector. All future records that carry the same tag value are forwarded to this instance.

8.4 Maintaining Deterministic Record Order

The asynchronous nature of S-NET and the unspecified execution times of the various components within a network impacts the order of results. As components are able to write results to their output stream as soon as they become available the results of several streams may appear interleaved after merge points. Parallel composition as well as serial and parallel replication involve such merging points at which streams

```
Thread SplitDispatch(r◁in, A, ν, ctrl, {opin_ν(r)}∪S) =
          SplitDispatch(in, A, ν, ctrl, {opin_ν(r)▷r}∪S)
|       SplitDispatch(r◁in, A, ν, ctrl, S) =
          let opin_ν(r) = new Stream()
          opout = 𝒟(A, opin_ν(r)▷r)
          in SplitDispatch(in, A, ν, ctrl▷opout,
                                            {opin_ν(r)▷r}∪S)
```

Figure 8.15: The split dispatcher sends records to the appropriate instance of its operand, depending on the outcome of the ν function. If the instance does not yet exist, it is deployed and the inbound stream is, associated with the tag value, added to the set of served streams.

may be interleaved in a non-deterministic fashion. As a consequence, records travelling on different branches through the network appear to overtake each other as Fig. 8.16 illustrates on the simple example of a parallel composition. While merging streams in a non-deterministic way enables S-Net programs to adapt to load distribution in concurrent systems and leads to efficient runtime behaviour in general, there are situations where non-deterministic system behaviour is undesirable. Unlike their non-deterministic counterparts described so far, their deterministic counterparts guarantee to maintain the causal order along branches of the streaming network: any record created in one branch of the network as a (potentially indirect) response to a record on the compound network's input stream precedes any other such record on the compound network's output stream that stems from a subsequent record on the input stream.

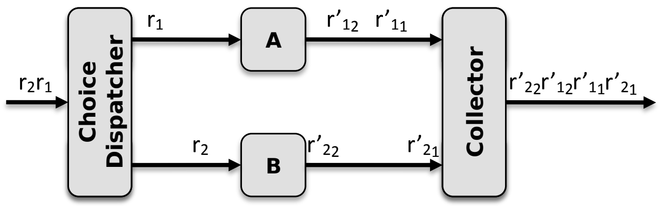


Figure 8.16: Networks A and B output two records for each input. The collector picks up the records when they become available on the stream, which is not necessarily the order in which the original records arrived. The correct deterministic output order for input r_2r_1 (read right to left) would be $r'_{2_2}r'_{2_1}r'_{1_2}r'_{1_1}$.

Both compilation and deployment are largely unaffected by the introduction of deterministic combinators. They merely produce deterministic variants of the dispatcher and collector components with identical argument sets as their non-deterministic counterparts. However, the operational behaviour of deterministic components needs to be extended by mechanisms that allow to maintain order. The required mechanisms are the same for all deterministic combinators. We exemplify their implementation using a deterministic choice dispatcher (Fig. 8.17).

Conceptually, each record that enters a deterministic sub-network is mapped to a separate sub-stream. Within the network, all records that are produced from the inbound record, remain in the same sub-stream. The dispatcher-collector pair ensures, that any sub-stream is completely output before any elements of another sub-stream are forwarded to the merged output stream. We implement sub-streams by means of control records that act as stream delimiters. A control record $[l, c]$ has two attributes: a *level* l and a *counter* c. Only deterministic dispatch components create such control records. The counter value is increased for each new control record to distinguish consecutive sub-streams. The purpose of the level value is to identify correct dispatcher-collector pairs in the presence of recursively nested deterministic and non-deterministic network combinators. When a new record arrives at a deterministic dispatcher, a fresh control record is sent ahead of the data record to the appropriate output stream following consultation of the oracle function δ. Inbound control records are broadcast to all branches with the level value incremented by one.

The deterministic collector, as shown in Fig. 8.17, complements the deterministic choice dispatch (in fact this collector may be used to implement the deterministic replication combinators as well). The collector ensures that different sub-streams appearing on its in-out streams are forwarded to its output stream without interleaving and in the right order. To achieve this, the collector maintains two stream sets: The *ready* set R contains all streams on which the collector actively snoops for input while the *waiting* set W contains those input streams that are currently blocked.

When a control record appears on one of the ready input streams that was emitted by the dispatch component corresponding to this collector (level 0), we check its counter: if the counter coincides with the internal counter of the collector (cnt), it marks the beginning of the next sub-stream to be sent to the collector's output. If so, the corresponding input stream remains in the ready set and the control record is discarded. Otherwise, the input stream is moved from the ready set to the waiting set without consuming the control record.

Any control record that belongs to an outer dispatcher-collector pair (level > 0) appearing on a ready input stream causes that stream to be moved to the waiting set while the control record is stored in the collector. As the corresponding dispatcher had broadcast this control record to all its output streams the collector will receive them sooner or later from all of its input streams. Only after the last replica of the control record has been received by the collector it may issue a single instance on the output stream.

Any regular record appearing on a ready input stream is immediately forwarded to the output stream. Any ordinary data record is preceded by the control record identifying the sub-stream the subsequent regular records belong to. If an input stream is still in the ready set when a regular record arrives this indicates that this is the currently active sub-stream which is issued on the output stream. Only one such

```
𝒟(A ||$_\delta$ B, in) =
let opin$_1$ = new Stream()
    opin$_2$ = new Stream()
    disp = spawn DetChoiceDispatch(in, $\delta$, 1, opin$_1$, opin$_2$)
    opout$_1$ = 𝒟(A, opin$_1$)
    opout$_2$ = 𝒟(B, opin$_2$)
    out = new Stream()
    coll =
     spawn DetCollect(nil,{opout$_1$,opout$_2$},$\emptyset$,1,-,out)
in out

Thread
  DetChoiceDispatch([$l$,$c$]◁in, $\delta$, $cnt$, opin$_1$, opin$_2$)
  when $l > 0$ =
   DetChoiceDispatch(in, $\delta$, $cnt$, opin$_1$▷[$l+1,c$], opin$_2$▷[$l+1,c$])
| DetChoiceDispatch(r◁in, $\delta$, $cnt$, opin$_1$, opin$_2$) =
   if $\delta(r) = 1$
   then
     DetChoiceDispatch(in, $\delta$, $cnt+1$, opin$_1$▷[0,$cnt$]▷r, opin$_2$)
   else
     DetChoiceDispatch(in, $\delta$, $cnt+1$, opin$_1$, opin$_2$▷[0,$cnt$]▷r)

Thread
  DetCollect(ctrl, {[0,$c$]◁in}$\cup R$, $W$, $cnt$, tosend, out) =
    if $c = cnt$
    then DetCollect(ctrl, {in}$\cup R$, $W$, $cnt$, tosend, out)
    else DetCollect(ctrl, $R$, {[0,$c$]◁in}$\cup W$, $cnt$, tosend, out)
| DetCollect(ctrl, {[$l$,$c$]◁in}$\cup R$, $W$, $cnt$, tosend, out) =
    DetCollect(ctrl, $R$, {in}$\cup W$, $cnt$, [$l$,$c$], out)
| DetCollect(ctrl, {r◁in}$\cup R$, $W$, $cnt$, tosend, out) =
      DetCollect(ctrl, {in}$\cup R$, $W$, $cnt$, tosend, out▷r)
| DetCollect(ctrl, $\emptyset$, $W$, $cnt$, -, out) =
    DetCollect(ctrl, $W$, $W$, $cnt+1$, out)
| DetCollect(ctrl, $\emptyset$, $W$, $cnt$, [$l$,$c$], out) =
    DetCollect(ctrl, $W$, $W$, $cnt$, -, out◁[$l-1$,$c$])
| DetCollect(in◁ctrl, $R$, $W$, $cnt$, tosend, out) =
    DetCollect(ctrl, {in}$\cup R$, $W$, $cnt$, tosend, out)
```

Figure 8.17: The deterministic choice dispatcher introduces a new sub-stream for each inbound record by increasing the stream counter. The control records are broadcast to all served streams.

active input stream exists at a time. If there are still further input streams in the ready set, then only because the corresponding control record has not yet arrived.

If the ready set becomes empty, i.e. a followup control record appeared on the previously active input stream indicating the end of that sub-stream, we restore a fresh ready set from the waiting set and increment the internal counter of the collector. This step makes the collector identify the next active sub-stream. In case we have a pending control record belonging to an outer dispatcher-collector pair, we forward it to the output stream with a decremented level counter.

Last not least, we may at any time receive a new input stream via the control stream. In this case we add the new input stream to the ready set. This feature of the collector is only used for implementing the deterministic replication combinators, that lead to dynamically evolving networks.

To make this scheme work, some minor extensions are required for non-deterministic dispatchers and collectors: Dispatchers must broadcast control records to all output streams without touching them. Collectors must gather control records on the various input streams and discard all but one which is issued on the output stream.

9 Use Cases

The toolchain for S-NET consists of a compiler, various runtime system libraries and a set of foreign language interfaces. The system is a vehicle for research and it is continuously further developed and improved. The system is also in practical use; two use-cases of S-NET are presented in this chapter.

9.1 Radar Signal Processing

The first use-case is a parallel signal-processing application for space-time adaptive filtering of RADAR echoes. The project has been developed jointly with Thales Research and has been published in [PHG$^+$10]. This section reproduces the relevant parts of this publication and extends it by a more detailed description of the box implementations in SAC.

9.1.1 The Purpose of the Application

The radar application we present here is an implementation of MTI (Moving Target Indication). The purpose of MTI is to detect moving objects on the ground from an aircraft. The main feature here is the detection of slow moving objects, whereas non-adaptive, classical radar processing is limited to the detection of fast moving objects.

The MTI application (Fig. 9.2) receives a ground echo of a periodic sequence of radar pulses and attempts to distinguish moving objects from all other, generally still, reflecting surfaces (ground clutter) under the radar beam. The position of a target is estimated by measuring the delay between the transmission of a radio pulse and the reception of its echo. The speed of an object is measured by analysing the Doppler effect that affects echoes of several identical pulses which are sent periodically. The movement of the object results in small variations of its distance to the radar. This distance variation is detectable as a phase shift of the radar signal, e.g. at around 10GHz. In this basic approach, Doppler processing consists of a bench of filters, each tuned towards a particular phase shift between successive echoes. This kind of Doppler processing is in some situations sufficient to separate reflecting objects on the basis of their speeds. When the beam is directed towards the ground, the largest part of the echoed energy is assumed to be a reflection of static objects that compose the ground (clutter). The moving object we are interested in send a weak, phase shifted echo. However, as the radar beam is not perfectly sharp and has a width of a few degrees, some still objects at the border of the beam appear to be moving with a speed relative to the aircraft's speed. This causes undesired interference over the moving target's echoes and creates an ambiguity between intrinsic speeds and azimuths of targets.

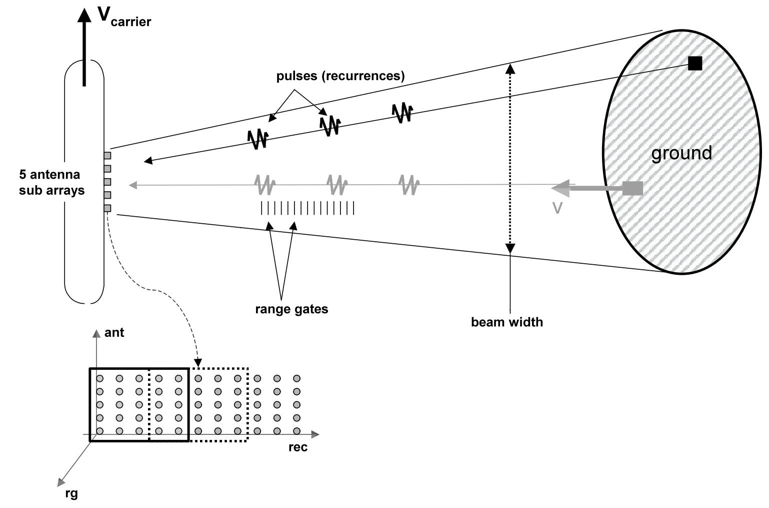


Figure 9.1: The antenna consists of 5 sensors, which receive a ground echo of the radar beam of periodically sent pulses. The signal is a 3D array of axes ant (sensor), rec (pulse sequence) and rg (range gate).

More accurate are adaptive filtering techniques, where filters are computed at runtime to help to minimise the effects of this undesired ambiguity. In this project we use an implementation of 'Space Time Adaptive Processing' (STAP) [CM06, HLM95], which computes a set of filters from signals received by the antenna array at different time steps. Fig. 9.1 shows the setup of the antenna array. The antenna array consists of a number (nant) of equidistant aligned sensors. The aircraft illuminates the ground with a beam orthogonal to its velocity. The sequences of periodic pulses (bursts) may vary in timing and amount of bursts. The reception time of an echoed pulse depends on the number of the pulse and the distance of the reflecting surface on the ground. Measuring the latter is achieved by sampling the received signal at a given frequency, resulting in the distance being sampled into range gates (rg), which is typically 15 meters for 10MHz sampling. The detected signal is received by the radar processing chain as a 3D array with dimensions $Nrg, Nrec, Nant$. The processing chain is subdivided into independent modules, as shown in Fig. 9.2.

For simulation purposes, we also implement a 'Stimuli Generation' module, which simulates the signal received by the radar antenna array. This is achieved by computing a 2D array representing the Radar Cross Section (RCS) of the ground surface situated on range gate rg and angle θ. The clutter model of 'CreateClutter' is computed from random, positive values with a given average and adding peak reflectivity values of a given probability. The returned signal from a burst of pulses of the ground surface to which targets with a given RCS and radial velocity have been added, is computed by 'EchoRaf'. The final processing step in this module is the addition of white noise to the signal.

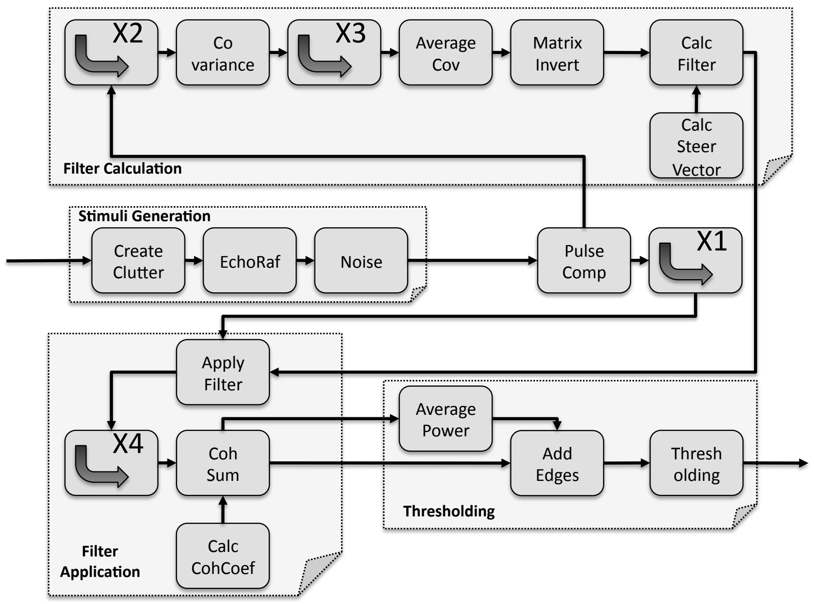


Figure 9.2: The data processing graph of the MTI application is subdivided into several modules. Modules are indicated by boxes with folded bottom right corners. Small boxes with text denote processing functions, boxes containing an arrow and a capital X with a number denote structure transformers. These boxes are used to re-arrange data in a matrix without effecting the actual values.

The presented processing chain contains some naive, well-known radar processing techniques for legacy reasons. Nevertheless, the characteristics, i.e. the main challenges from an implementers point of view, are representative for the important industrial domain of embedded signal-processing applications on parallel hardware, as:

- The processing chain uses multiple operators with different requirements on precision and/or dynamic ranges.
- The static processing graph represents a dynamic processing chain, as algorithm parameters, such as array sizes, loop boundaries, etc., change (multi-mode radar [HHK97]).
- The computational load is high enough to require parallel computing hardware.
- Performance is one of the key requirements, both in terms of computational throughput and latency, which may be due to operational requirements or architecture constraints such as memory limitations.

9.1.2 Modelling STAP in S-Net

The starting point of the design process of the MTI application in S-Net is the dataflow graph of the original implementation shown in Fig. 9.2. We use the structure of this graph to derive the structure of our application: Each signal processing function, i.e. the small boxes in Fig. 9.2, becomes an S-Net box that we build from the existing components. The modules translate to individual networks which connect the boxes using combinators according to the connections within the module. The same combinators also enable us to connect these networks to each other to form the MTI application. This hierarchical approach allows us to implement and test networks, i.e. the modules of the application, independently, as each network is a fully functional application itself when deployed individually.

In the remainder of this section we illustrate the design process of the application modules as networks, using module 'Stimuli Generation' as starting point.

The module in Fig. 9.2 contains three processing functions, and so does the network we implement. The boxes are arranged in a serial combination (*CreateClutter* .. *EchoRaf* .. *Noise*), implementing the function composition of these functions.

On first glance, the definition of the box and network signatures is as straight forward as the definition of the network structure. On second glance, however, there is some design space to be explored. The algorithms of these boxes consume a wealth of parameters. As an example, the parameters of *CreateClutter* are shown in Table 9.1. Most of these parameters are semi-static; they usually do not change during runtime but are only adjusted between runs of the application.

The box returns one single value, a 2D array representing the Radar Cross Section (RCS) of the ground surface. From this knowledge, we may construct the box signature as $\{\sigma\} \rightarrow \{\text{array\_2d}\}$, where σ is shorthand for all entries of the second column of Table 9.1.

The box *EchoRaf* has a similarly extensive input type. Not only does it require *CreateClutter*'s result, it also requires a set of parameters to produce a 3D array of values representing the ground echo including targets. Box *Noise* is more frugal with respect to parameters and requires only two 3-dimensional matrices of random values to compute white noise, which it applies to the result of *EchoRaf*. If we combine these boxes in sequence to form a network, the parameters propagate to the network signature. This results in an extremely unwieldy input type of the network, composed of field 'rnd_values' plus the union of required parameters of all three boxes. The exposure of local parameters to the outside is not only unaesthetic, but also problematic from a software engineering perspective: The parameters propagate through any surrounding context and eventually manifest in the global input type of the application network. To avoid this, we confine parameters to their local contexts.

parameter name	static
rnd_values	no
σ_0	yes
rg_{min},rg_{max},rg_{size}	yes
angle_carrier	yes
beam_width	yes
size	yes
targets	yes
$N\theta$	yes

Table 9.1: Parameters of CreateClutter

One trivial approach to achieve this, is to generate all required parameters within the box implementation itself. The problem is, that this is not always possible: If a box is provided as an opaque object, e.g. as precompiled library from a third party, we do not have access to its implementation. In this case, we have to embed the function call into wrapper code that computes all parameters before invoking the actual box function. By generating the parameters within the box, this approach solves the problem of exploding signatures. The downside of this approach is, that parameters are completely invisible on the S-Net layer. This potentially impedes component reuse, as a user of such a network is now unaware of the presence of the parameters.

Therefore, we take a more flexible approach by generating parameters from within stand-alone S-Net boxes. This is done in such a way, that a parameter generating box does not require any input fields or tags and delivers parameters for one specific box as output. The non-static input is merged to the generating parameter set by exploiting the power of flow-inheritance: For a box A with signature $\alpha \cup \rho \to \beta$, where α is the set of input labels and ρ is the set of static parameters, a corresponding parameter generating box P_A has signature $\{\} \to \rho$. Because of the empty input type, this box can be inserted as predecessor of A without adding any label constraints to surrounding contexts. Any record that arrives at P_A is accepted as input and is enriched by parameter set ρ. More specifically, due to flow-inheritance, a record with label set α as input to P_A results in $\alpha \cup \rho$ (of course, any excess label is carried over as well: $\alpha \cup \gamma \to \alpha \cup \rho \cup \gamma$).

Using this technique, the original box implementation is left untouched and parameters are visible on the S-Net level without being propagated.

The network that implements the 'Filter Calculation' module of the MTI application is a sequence of boxes as defined by the order of tasks shown in Fig. 9.2. Parameters of boxes are, as above, supplied by parameter generators if required. Special treatment is necessary for boxes *CalcSteerVect* and *CalcFilter*, as the former lacks an input

```
net Thresholding
{
  box ApplyFilter((array_3d_signal, array_4d_filter)
                                 -> (array_4d_filtered));
  box X5((array_4d_filtered) -> (array_4d_filtered));
  box CalcCohCoeff(() -> (coh_array_2d));
}
connect [{array_4d_filter, array_3d_signal} ->
           {array_4d_filter, array_3d_signal}; {}]
        .. ((ApplyFilter .. X5)
             | CalcCohCoeff)
        .. [|{array_4d_filtered},{coh_array_2d}|];
```

Figure 9.3: Implementation of the Thresholding network in S-NET

channel, whereas the latter requires two input channels. One possibility to model this, is to assign an empty input type *CalcSteerVect* and place it as direct predecessor of *CalcFilter*. A record that arrives at this box triggers execution of the box without any of the record's fields or tags being read. Flow-inheritance inserts all inbound-record constituents to the resulting record, ensuring that *CalcFilter* receives all required input fields in one record. This approach, however, does not overlap computations where it would be possible: *CalSteerVect* does not require any input, and can therefore begin its computation much earlier. To do this, the box is arranged in parallel to the rest of the network and computation is triggered at the earliest possible moment, i.e. when a record arrives at the first box. The output of the box is then combined to a result record at the latest possible stage, i.e. a synchro-cell merges the box's result to the result of the remaining network. This created record contains all required fields for *CalcFilter*. Fig. 9.4 (a) and (b) illustrate these techniques.

The remaining networks implementing 'Filter Application' and 'Thresholding' use the same techniques. For brevity we refrain from describing them in detail here but show the concrete S-NET implementation of network 'Thresholding' in Fig. 9.3 and its graphical representation in Fig. 9.4(c).

The final step of the implementation phase is to combine all modules to form the MTI application. The complete application network is shown in Fig. 9.4(d).

9.1.3 Implementing Boxes in SaC

SAC is a functional, side-effect free variant of C: we interpret assignment sequences as nested let-expressions, branches as conditional expressions and loops as syntactic sugar for tail-end recursive functions. The meaning of functional SAC code coincides with the state-based semantics of literally identical C code (cf. [Sch03]). This language kernel is extended by n-dimensional state-less arrays: Any expression may evaluate

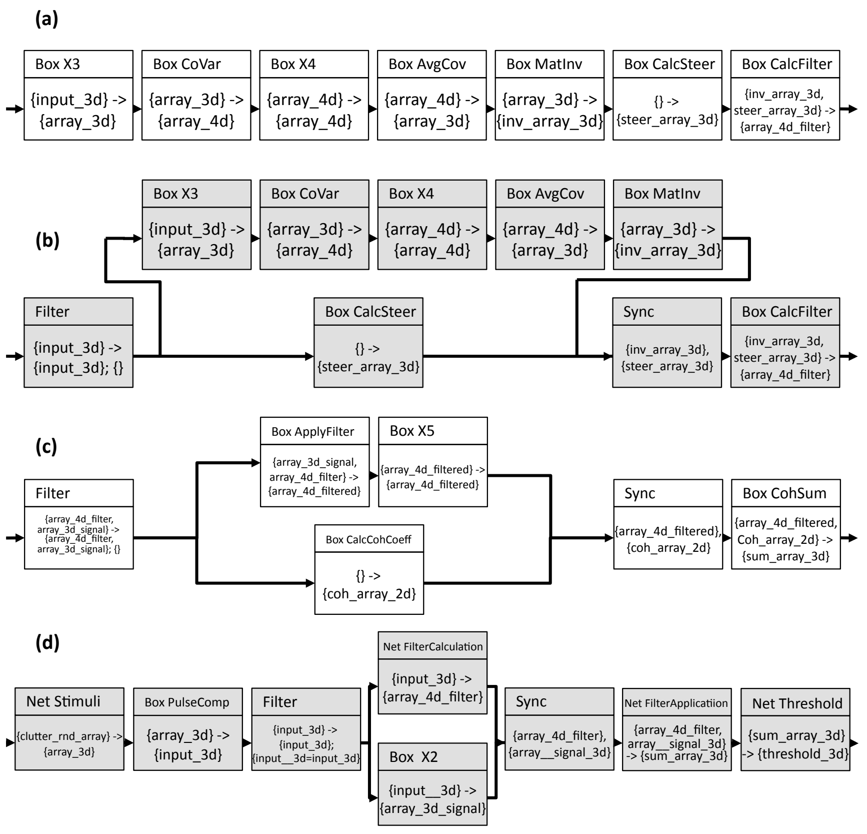

(a)
Box X3
{input_3d} -> {array_3d}
Box CoVar
{array_3d} -> {array_4d}
Box X4
{array_4d} -> {array_4d}
Box AvgCov
{array_4d} -> {array_3d}
Box MatInv
{array_3d} -> {inv_array_3d}
Box CalcSteer
{} -> {steer_array_3d}
Box CalcFilter
{inv_array_3d, steer_array_3d} -> {array_4d_filter}
(b)
Box X3
{input_3d} -> {array_3d}
Box CoVar
{array_3d} -> {array_4d}
Box X4
{array_4d} -> {array_4d}
Box AvgCov
{array_4d} -> {array_3d}
Box MatInv
{array_3d} -> {inv_array_3d}
Filter
{input_3d} -> {input_3d}; {}
Box CalcSteer
{} -> {steer_array_3d}
Sync
{inv_array_3d}, {steer_array_3d}
Box CalcFilter
{inv_array_3d, steer_array_3d} -> {array_4d_filter}
(c)
Filter
{array_4d_filter, array_3d_signal} -> {array_4d_filter, array_3d_signal}; {}
Box ApplyFilter
{array_3d_signal, array_4d_filter} -> {array_4d_filtered}
Box X5
{array_4d_filtered} -> {array_4d_filtered}
Box CalcCohCoeff
{} -> {coh_array_2d}
Sync
{array_4d_filtered}, {coh_array_2d}
Box CohSum
{array_4d_filtered, Coh_array_2d} -> {sum_array_3d}
(d)
Net Stimuli
{clutter_rnd_array} -> {array_3d}
Box PulseComp
{array_3d} -> {input_3d}
Filter
{input_3d} -> {input_3d}; {input__3d=input_3d}
Net FilterCalculation
{input_3d} -> {array_4d_filter}
Box X2
{input__3d} -> {array_3d_signal}
Sync
{array_4d_filter}, {array__signal_3d}
Net FilterApplicatiion
{array_4d_filter, array__signal_3d} -> {sum_array_3d}
Net Threshold
{sum_array_3d} -> {threshold_3d}

Figure 9.4: (a) This variant of 'FilterCalculation' uses *CalcSteerVect* as direct predecessor to *CalcFilter*. (b) shows an alternative implementation of (a) where *CalcSteer* is arranged in parallel to the remaining network to overlap computations. (c) shows the implementation of network 'Thresholding'. (d) shows the final MTI application

to an array, and arrays may be passed between functions without restrictions. Arrays in SaC are neither explicitly allocated nor de-allocated. They exist as long as the associated data is needed, just like scalars in conventional languages.

Array types include arrays of fixed shape, e.g. `int[3,7]`, arrays of fixed rank, e.g. `int[.,.]` and arrays of any rank, e.g. `int[*]`. The latter include scalars which in SaC are considered rank-0 arrays with an empty shape vector. For convenience and equivalence with C we use `int` rather than the equivalent `int[]` as a type notation for scalars. SaC provides a small set of built-in array operations, basically primitives to retrieve data pertaining to the structure and contents of arrays, e.g. an array's rank (`dim(`*array*`)`) or its shape (`shape(`*array*`)`); a selection facility provides access to individual elements or entire sub-arrays: (*array*`[`*idx_vec*`]`).

Compound array operations are specified using WITH-loop expressions, SaC-specific array comprehensions:

```
with { ( lower_bound <= idx_vec < upper_bound) : expr;
}:  genarray( shape, default)
```

where *lower_bound* and *upper_bound* are expressions that must evaluate to integer vectors of equal length. They define a rectangular (multidimensional) index set. The identifier *idx_vec* represents elements of this set, similar to loop variables in FOR-loops. However, we deliberately do not define any order on these index sets. We call the specification of such an index set a *generator* and associate it with an arbitrary expression. By doing so we create a mapping between index vectors and values, in other words an array. A WITH-loop

```
with { ([0,0] <= iv < [3,5]) : 42;
}: genarray( [3,5], 0)
```

defines a 3×5 matrix with all elements uniformly set to 42. The scope of *idx_vec* is confined to the expression associated with the generator. It can be used to access the current index location. For example, the WITH-loop

```
with { ([0] <= iv < [5]) : iv[0];
}: genarray( [5], 0)
```

computes the vector `[0,1,2,3,4]`. Note that `iv` denotes a 1-element vector rather than a scalar. Therefore, we need to select the first (and only) element from `iv` to achieve the desired result. Actually, it is not the generator that defines the shape of the resulting array, but the first expression following the key word `genarray`. In our previous examples they always coincided with the upper bound vectors, but

```
with { ([1] <= iv < [4]) : 42;
}: genarray( [5], 0)
```

computes the vector `[0,42,42,42,0]`. We still create a 5-element vector, but only the three inner elements are defined as 42 while all others are set to the *default value*, which is given by the second expression after the key word `genarray`. WITH-loops are not limited to have a single generator only. For example, the WITH-loop

```
with { ([1] <= iv < [4]) : 1;
```

```
        ([3] <= iv < [5]) : 2;
}: genarray( [6], 0)
```

defines the vector `[0,1,1,2,2,0]`. Whenever the index sets defined by the various generators are not pairwise disjoint, the order of the generators matters: in the example the array's value at index location `[3]`, which is covered by both generators is set to 2 rather than to 1. SaC actually features several variants of WITH-loops. Let us assume we have named the array defined by the previous WITH-loop `A`. Then

```
with { ([0] <= iv < [3]) : 3;
}: modarray( A)
```

defines the vector `[3,3,3,2,2,0]`. More precisely, it computes a new array that has exactly the same shape as the existing array referred to by the expression following the key word `modarray`. The computation of those elements covered by one or more generators follows exactly the same pattern as in the case of `genarray`-WITH-loops. The remaining elements are defined by the values of the corresponding elements in the referred array.

As described in Section 9.1.2, the radar application is hierarchically broken down, with *boxes* as atomic building blocks. These boxes each map an individual transformation function to a stream of incoming data. Each incoming data item is a record holding one or more arrays of data. Hence, we can specify the functionality of each box by means of a single SaC function; its argument-arrays are the components of the input record and its return-arrays are the components of the output record. All the marshalling required can be, and in fact is, compiler generated. This enables a programmer to directly utilise arbitrary SaC libraries without any modification.

As an example, Fig. 9.5 shows the implementation of one of the boxes mentioned in Section 9.1.2. It implements Gauss-Jordan matrix inversion for arrays of complex values. The first three lines identify this code as a separate module providing a function `MatInvert` for external use. As we can see, there are two function definitions that match that name. The first definition expects two-dimensional arrays (or matrices) of complex numbers, whereas the second version accepts arbitrary arrays of complex numbers. This form of overloading in SaC ensures that the first definition of `MatInvert` is applied iff the argument is of rank two. In all other cases, the second definition is used.

Having a look at the second, generically applicable definition, we can see that all it does is to map the rank-2-specific version to all rank-2 sub-arrays of the argument array. As a consequence, any argument array effectively is considered an array of complex matrices which are to be inverted.

The actual matrix inversion is triggered by the rank-2-specific version of `MatInvert`. It extends the argument matrix `input` row-wise by a unit matrix `E( input)` of matching size. The Gauss-Jordan-Elimination (function `GJ`) is applied to the extended matrix and, subsequently, the leading columns that carry the eliminated components are cut off.

Looking at the elimination itself, we can identify a loop which, for each individual

```
module MatInvert;
use StdEnv: all;
export {MatInvert};

complex[.,.] GJ( complex[.,.] A) {
  n = shape(A)[0];
  for (i = 0; i < n; i++) {
    A[i] = A[i] / A[i,i];
    A = with {
      ( [0] <= [j] < [i]): A[j] - A[i] * A[j,i];
      ( [i] < [j] < [n]): A[j] - A[i] * A[j,i];
    } : modarray( A);
  }
  return( A);
}

complex[.,.] MatInvert( complex[.,.] mat) {
  joined = { [i] -> mat[i] ++ E(mat)[i]};
  solved = GJ( joined);
  return( drop( [0,shape(mat)[1]], solved));
}

complex[*] MatInvert( complex[*] input) {
  defaultMatrix = mkarray( take( [-2],
                    shape( input)), toc(0));
  result = with {
             ( . <= iv <= .) :
               MatInvert( input[iv]);
           } : genarray( drop( [-2],
                           shape( input)),
                         defaultMatrix);
  return( result);
}
```

Figure 9.5: Matrix inversion using Gauss Jordan Elimination.

row, first normalises the row under consideration and then triggers a data-parallel elimination step on the remaining elements of the array.

Notice here, that this entire specification is purely declarative. The programmer does not specify how to use memory or whether or not things should be computed in parallel or not. All these choices are up to the compiler. As soon as the compiler finds out about the shapes of input arrays, these things can be suitably arranged. For example, in case we obtain large arrays of very small complex matrices, the compiler may decide to implement the actual inversion sequentially as there is a lot of outer concurrency stemming from the generic version of `MatInvert`. Likewise, the compiler could decide to execute all individual elimination steps in parallel provided the individual matrices to be inverted are large enough. However, this happens completely transparently for the programmer. As a consequence, this very module can be used in various contexts without any modification.

It should also be mentioned here, that the interface between SaC and S-Net is done in a way that enables programmers to plug such boxes together without being required to think about any memory issues such as memory reuse or garbage collection. All these issues are resolved implicitly, enabling the programmer to focus on the most important task, i.e., on getting the specification of the algorithm right.

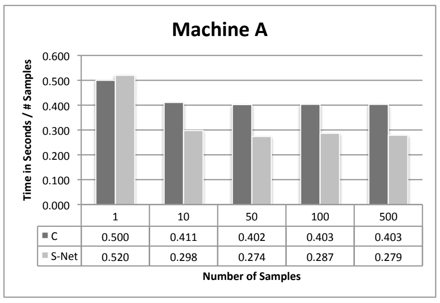
9.1.4 Performance Evaluation

The measurements we present here compare the original, sequential C implementation with the S-Net implementation that we have developed. Both programs were given several sets of input samples and for each set the total runtime was recorded. The numbers presented in Fig. 9.6 show the average amount of time that was required to process *one* input sample, computed by dividing the total runtime by the number of data samples in the input set.

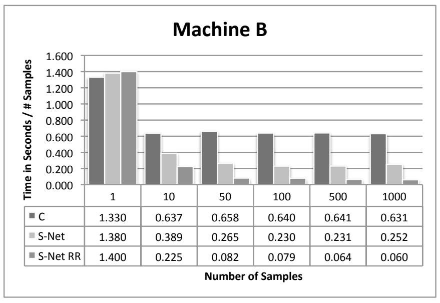
We employed two separate machines for measurements: Representative for consumer-grade hardware, we chose a laptop equipped with an Intel Core 2 Duo processor at 2.4GHz and 4GB of memory running Darwin 10.2.0 (Mac OS X 10.6.2). We refer to this host as *Machine A*. The second machine, *Machine B*, is a computation server featuring four Quad-Core AMD Opteron 8356 processors and 16GB of memory running Linux 2.6.18-128.1.10.el5 (CentOS 5.3).

On both machines we conducted the experiments using input data that was identical in size and value. Time was measured from beginning to end of a program run, i.e., including all I/O operations. However, to keep disk access to a minimum we reused input data from memory after reading it in once from disk. As was to be expected, the S-Net runtime system adds runtime overhead whose effect is most prominently seen on the runtimes for one single input sample. This overhead, however, is amortised by the fact that the S-Net runtime system executes multiple stages of the computational pipeline in parallel on systems with more than one core. As both systems contain multiple cores the S-Net implementation outperforms the original C implementation on both machines from as few as five input samples.

The best speed-up on Machine A is about 1.47 which we deem acceptable for a



	1	10	50	100	500
■ C	0.500	0.411	0.402	0.403	0.403
■ S-Net	0.520	0.298	0.274	0.287	0.279



	1	10	50	100	500	1000
■ C	1.330	0.637	0.658	0.640	0.641	0.631
■ S-Net	1.380	0.389	0.265	0.230	0.231	0.252
■ S-Net RR	1.400	0.225	0.082	0.079	0.064	0.060

Figure 9.6: Timings of the original C and S-NET implementation on a dual-core (left) and 16-core (right) machine

dual-core laptop machine. On Machine B, however, the best speed-up we were able to achieve was 2.78. This disappointing figure is mainly due to sequential box code which does not take advantage of multiple cores. As these strictly sequential code blocks inevitably limit the achievable speed-up, we have also measured a round-robin distribution of input records to five instances of the network, labelled "S-Net RR" in Fig. 9.6. This experiment came with almost no additional development cost, as the `!` combinator can be used to achieve exactly this: By applying the split combinator `!` to the outermost level of the network, the runtime system automatically creates multiple instances of the network (demand driven). The combinator determines based on a tag value which instance a data item is dispatched to. In this case, `stap!<n>` requires all data items to carry a tag `<n>` whose value determines the instance. By tagging the input data with `<n>` using values between 1 and 5, and thereby dispatching data items in a round-robin fashion to the instances, a much better utilisation of the computing resources was possible, and hence the speed-up increased to a more satisfying value of 10.5.

9.2 Raytracing Solver

The second use-case presented in this chapter implements a distributed ray-tracing solver. This work has been developed as a collaborative project with a group at the University of California, Irvine and has been published in [PHS$^+$10]. Here we reproduce the relevant parts of this publication.

9.2.1 The Purpose of the Application

Ray tracing is a well-known technique of rendering a 2D pixel images of a 3D scene model by tracing paths of light back from the eye of an imaginary observer through pixels in an image plane and by simulating the effects of their encounters with the objects of the scene illuminated by a source of light [Whi80].As illustrated in Fig. 9.7, the primary ray is shot through each pixel in the image plane and tested for intersection

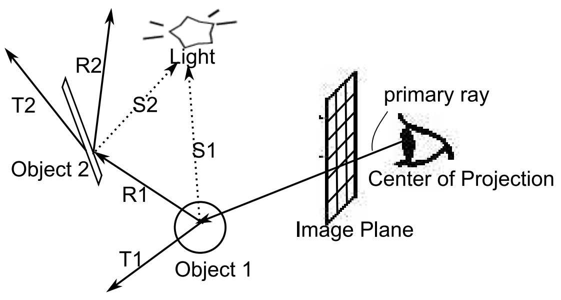


Figure 9.7: Image rendering via ray tracing. A primary ray is cast and tested for intersection with objects. At the closest hit point, reflective ray R1, shadow ray S1, and transmitted ray T1 are generated, each of which is also tested for intersection with the objects in the scene.

against the objects in the scene. When the closest intersection is found, the pixel value is computed based on the characteristic of the object material. Rays are continually generated until either the end of the scene or the maximum ray depth level is reached.

As each ray is cast to every object, the majority of the rendering time is spent calculating intersections. In order to enable efficient ray tracing, we use the Bounding-Volume Hierarchy (BVH) algorithm [KA88]. It builds a hierarchical representation of 3D objects that makes the traversal of the nodes in search for intersections more efficient. More precisely, when adding an object to the BVH, it inserts the bounding volume that contains the object at the optimal place in the hierarchy using a branch-and-bound algorithm, which minimises the cost estimation based on the surface area [GS87].

Algorithm 1 Ray Tracing Algorithm : it loops over the entire image, casting a single ray per pixel.

```
1: /* Input : Scene Information */
2: /* Output: 2D Rendered Image */
3: scene ← construct a Bounding Volume Hierarchy (BVH) based on the input scene
4: for each pixel in the image plane do
5:     ray ← construct the primary ray from the centre of projection through pixel
6:     colour ← Trace( ray, scene ) /* See Algorithm 2 */
7: end for
```

Algorithm 1 provides the pseudo code of ray tracing process that we used in this paper, and Algorithm 2 describes the procedure of tracing rays and deciding the shade of a pixel. The Cast function in Algorithm 2 traverses the BVH data structure, which contains the objects of the scene, to find the closest intersection between the ray and the objects. When it finds a hit, it calculates the shade of the pixel considering the reflective, refractive, shadow ray interactions.

The highly computation intensive nature of ray tracing and the fact that each pixel

Algorithm 2 Trace Algorithm : it follows the ray, and if it finds a closet hit, it decides the pixel colour based on ray interactions with the objects. It selects the background colour by default.

```
1: /* Input : ray, scene */
2: /* Output: a shade of a pixel */
3: if ray_depth < MAX_RAY_DEPTH then
4:    hitinfo ← Cast(ray, scene)
5:    if hitinfo is not null then
6:       colour ← Shader(hitinfo)
7:    end if
8: end if
```

of an image can be rendered independently of all others make ray tracing amenable to aggressive parallelisation. Indeed, many attempts have been made at implementing ray tracing on distributed systems and multi-core architectures [DPH+03, BWSF06]. The implementation we use in this paper distributes an image evenly across all cluster nodes and processes these independently. The root process collects all sub-results and assembles the completed scene.

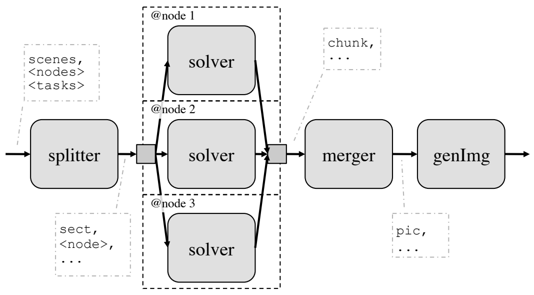
9.2.2 Modelling STAP in S-Net

Raytracing lends itself nicely to concurrent execution. All rays can be traced individually allowing for top-level coarse grain parallelism. To start with, we want to capture this concurrency by a simple fork-join approach.

9.2.3 A simple fork-join model in S-Net

We create three major components, a *splitter*, a *solver*, and a *merger* and we combine these in a pipeline-like fashion as shown in Figure 9.8. The splitter divides the overall task into sections, which are subsequently computed by instances of the solver, each of which is executed on an individual MPI node. After the solvers have dealt with the individual sections, the merger collects all the resulting image chunks into an overall result picture which is written to a file by means of a box *genImg*. Note that the entire distribution of data and re-collection of results on the master node is being triggered by the use of the @-symbol in the index splitter. It ensures that the value of the tag `<node>` is interpreted as the MPI-rank (processing node) that executes the corresponding solver instance. Therefore, the scheduling is controlled by the splitter component, which identifies the individual sections of work and attaches `<node>` tags to the records. Also note that the signatures of the individual boxes only define the parts of the data to be manipulated; the actual records may contain extra fields which are preserved by flow inheritance. The tag `<fst>` is an example of this.

Most of the components could easily be derived from the existing C-code. Using the C interface for S-Net (see [CPG+10] for details), only small wrapper functions



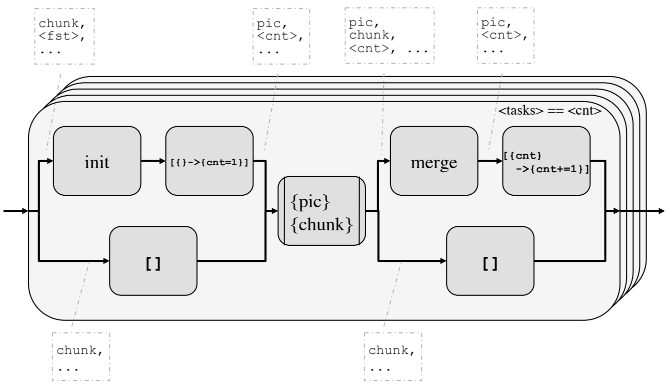
```
net raytracing_stat
{
  box splitter((scene, <nodes>, <tasks>)
      -> (scene, sect, <node>, <tasks>, <fst>)
         | (scene, sect, <node>, <tasks> ));
  box solver  ((scene, sect)  -> (chunk));
  net merger  ((chunk, <fst>) -> (pic),
               (chunk)        -> (pic));
  box genImg  ((pic)          -> ());
} connect

splitter .. solver!@<node> .. merger .. genImg
```

Figure 9.8: Overall design for a simple fork-join model.

needed to be created. The only component that could not be mapped directly to a C-implemented box was the merger. This was due to the fact that the merger needs to combine several chunks arriving asynchronously within separate records while boxes can only ever see one record at a time. Therefore, we created a sub-net named *merger* in which we specified the step-wise synchronisation of chunks in terms of two further components: an *init* box and a *merge* box. The init box creates an initial version of the resulting picture from the first chunk of work (tagged by a flag `<fst>` by the splitter component), which serves as an accumulator throughout the merging process. An n-fold application of the box merge is achieved by placing it under a serial replication combinator. Figure 9.9 shows the details of the merger net. The init box is followed by a filter which adds a flag `<cnt>` initialised by the value 1. This flag is used to count the number of sub-images that have been incorporated into the result image already. Since only the first chunk needs to be processed by the init box, we also provide a bypass to the initialisation path for all the other records containing further chunks. This is done by means of an empty filter box which is parallel to the initialisation path.

After the initialisation, we have a star which implements the merging with the remaining chunks. In each unfolding (iteration) of the star, we first have a synchrocell, which synchronises the accumulator held in `{pic}` with yet another chunk. The resulting joint record, containing the accumulated picture and a chunk to be inserted, is presented to the merge box which outputs the combined picture. The insertion of a



```
net merger
{
  box init  ((chunk, <fst>) -> (pic));
  box merge ((chunk, pic)   -> (pic));
} connect
    ((init .. [ {} -> {<cnt=1>} ] )
     | []
    )
    .. ([| {pic}, {chunk} |]
         .. ((merge
                .. [ {<cnt>} -> {<cnt+=1>}]
             )
             | []
            )
       )*{<tasks> == <cnt>} ;
```

Figure 9.9: Merger network for re-combining sub-images into a complete picture.

new chunk is reflected in an increment of the flag `<cnt>` as defined by the subsequent filter. Once the counter equals the overall number of tasks, which is kept in another, flow-inherited flag `<tasks>`, the accumulated picture is output from the merger network. The reader may wonder why we have a bypass parallel to the merge branch within the star. This is due to the fact that the star combinator does not feed any records back, but instead unrolls into copies of its operand. As a consequence, all but the first chunks not yet processed need to be bypassed to the next instance of the star.

While the network presented so far serves its purpose perfectly well, imbalances in the distribution of objects within any given scene quickly lead to limited scalability on clusters with more than 2 processing nodes. To improve on this situation, a dynamic workload balancing scheme had to be put into place.

9.2.4 Fork-Join with dynamic scheduling

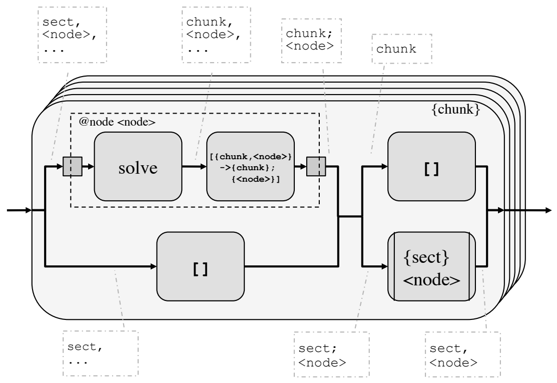
Here, the strict separation of application and concurrency engineering as it is enforced in S-Net pays off. Only very little modification to the basic solution described so far was required in order to achieve dynamic scheduling. The basic idea is to get the `<node>` flag to represent the availability of the processor identified by the value of that flag for computing an arbitrary section on it. We can enable the splitter box to output sections which, initially, do not contain a node flag. As a result, we now have two kinds of computing task streaming towards the solver component: those that do have a node flag attached to them, and those that do not. We can process the former straight-away by using the solve box while we need to queue the others to wait for node tokens. These node tokens can then be taken from tasks that arrive back from solvers. This modification of the S-Net solution presented so far can be achieved by simply replacing the `solver@<node>` component from Figure 9.8 by the network segment shown in Figure 9.10.

Here, we see that the solve box directly links into a filter which separates the resulting image chunk from the node token. Both these activities happen now on the individual nodes for all those sections that actually do come with a node token. All other sections just bypass the distributed solver. The reunited streams of computed chunks, node tokens, and remaining sections are then fed into either a synchrocell, which combines the remaining section with a just released node token, or a bypass, which enables the computed chunks to leave the solver segment. Those sections that have been combined with a new node flag are brought to a new instance of the distributed solver by means of a surrounding star. This unfolding of the start takes place until all sections have been transformed into chunks of the resulting picture.

Since the remaining part of the S-Net presented in Section 9.2.3 is oblivious of the node tag, it can be utilised in the dynamic setting without modification.

9.2.5 Distributed S-Net

The coordination model of S-Net leaves the choice of implementation strategy for a runtime system open.



```
( ( ( solve .. [ {chunk, <node>}
                 -> {chunk}; {<node>} ]
    )!@<node>
    | []
  )
  .. ( [] | [| {sect}, {<node>} |] )
) * {chunk}
```

Figure 9.10: Solver segment for dynamically scheduled work load distribution.

In particular, there is no notion of computing resources in S-NET, nor does S-NET make any specific assumptions about the execution environment. Distributed S-NET [GJP09] is a conservative extension of S-NET that introduces the concept of abstract compute nodes as an organisational layer on top of the logic network of boxes defined by standard S-NET. Again, let `A` denote an S-NET network or box. we introduce two additional *placement* combinators as follows. *Static placement* `A@num` places the box or network `A` onto compute node `num` for execution. *Indexed dynamic placement* `A!@<tag>` places the execution of network or box `A` onto the node identified by the value of `tag` on a per-record basis. More precisely, every incoming record is routed through a replica of `A` instantiated on the compute node determined by the value of `tag`.

We deliberately restrict ourselves to plain integer values for identifying compute nodes to retain the advantages of an abstract model as far as possible. The concrete mapping of numbers to machines is implementation-dependent. Our prototype implementation of Distributed S-NET is based on MPI where numbers correspond to MPI task identifiers.

Implementation details of the runtime system for distributed execution may be found in [GJP09].

9.2.6 Performance Evaluation

We have conducted several experiments using the original C/MPI implementation and S-NET solutions with and without dynamic load balancing, and recorded the runtime of each using a scene of 3000×3000 pixels.

The experiments were run on an 8-node cluster where each node contains two Intel PIII 1.4GHz CPUs and 1024MB of RAM. The nodes are connected by a standard 100Mbit ethernet network and all nodes have access to a shared file system.

For the dynamic load balancing solution we have experimented with several scheduling algorithms and found that block scheduling and a simple variant of factoring [HSF92] produces the best results. In the latter case, the scheduler divides the problem into several batches of sections, where in each batch the sections are of the same size. The section size decreases from batch to batch by a certain factor. For example, suppose a scene of 3000×3000 pixels is split along the y axis by dividing it into 48 section. One possible scheduling is to split the scene into two batches with the first batch containing 24 sections of size 93 and the second batch the remaining 24 section of size 32.

The results for several task sizes with varying token numbers are shown in Fig. 9.11(left) for factoring scheduling and in Fig. 9.11(right) for block scheduling. As can be seen in the diagrams, performance was generally best when 16 tokens were made available to the system. With this number of tokens each node holds two tokens on average which maps one solver instance to each CPU of the node. In the block scheduling case 32 tokens could be beneficial, but a further investigation is required to establish this. Performance is generally at its worst when the number of tasks equals the number of tokens. In this case all sections are immediately mapped to the nodes and the benefits of dynamic scheduling are lost.

To test the scalability of our approach, we compared the runtimes of all variants on one to eight nodes. Fig. 9.12 shows the results of these experiments. The runtimes on one single node clearly show the overhead the S-Net runtime system adds to the application when compared to the original MPI implementation (labelled MPI in the figure). However, from only two nodes onwards the overheads are amortised. For better utilisation of the computing resources we added two more static variants that spawn two instances of the solver per node (one per CPU). These experiments came at almost no additional development cost: by adding one more index split combinator to the solver of Fig. 9.8 (`(solver!<cpu>)!@<node>`) and marking input data with a `<cpu>` tag of values 0 and 1, the desired effect was achieved for the S-Net implementation. The MPI implementation did not require any code changes but the experiments were re-run with two processes per node by starting $2n$ MPI jobs on n nodes. The runtimes of this experiment are labelled "S-Net Static 2CPU" and "MPI 2 Proc/Node" in Fig. 9.12. As the S-Net runtime system automatically utilises multiple cores if available, the limited performance gain was to be expected. The MPI implementation however could benefit substantially from utilising the second CPU of each node. We also include runtimes for dynamic scheduling using number of nodes $\cdot\ 8$ tasks and tasks/2 tokens with block scheduling (labelled S-Net best dynamic) to present the compelling improvements on runtimes this technique offers. For comparison between these implementations, Fig. 9.12(right) shows the speed-up of the 2 CPU MPI version versus the 2 CPU static S-Net implementation and the best dynamic scheduling run.

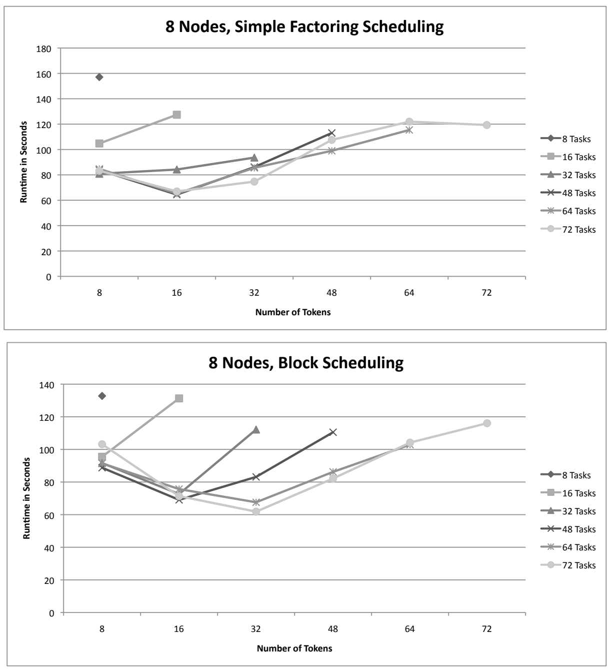
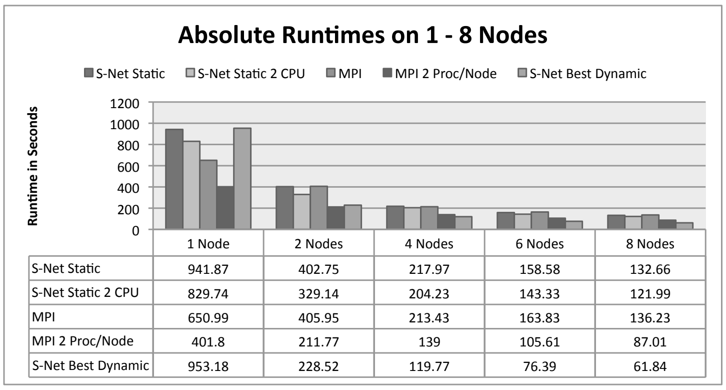


Figure 9.11: Runtimes on 8 nodes using simple factoring scheduling (left) and block scheduling (right) on a 3000 by 3000 pixels scene



	1 Node	2 Nodes	4 Nodes	6 Nodes	8 Nodes
S-Net Static	941.87	402.75	217.97	158.58	132.66
S-Net Static 2 CPU	829.74	329.14	204.23	143.33	121.99
MPI	650.99	405.95	213.43	163.83	136.23
MPI 2 Proc/Node	401.8	211.77	139	105.61	87.01
S-Net Best Dynamic	953.18	228.52	119.77	76.39	61.84

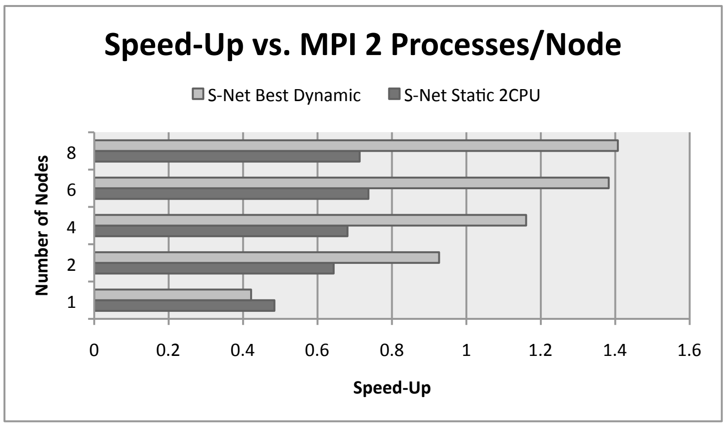


Figure 9.12: Runtimes on 1 - 8 nodes, comparing the original MPI implementation against S-Net variants (left) and speed-up of each implementation measured against the original MPI implementation with 2 processes per node (right)

10 Conclusion and Outlook

This dissertation demonstrates that a complete separation between the computational aspects and the coordination aspects of an application unlocks powerful analysis opportunities. Indeed, by capturing all concurrency related issues in a dedicated language and by isolating them from all other aspects we were able to devise a comprehensive set of analysis algorithms for concurrent programming pattern. The analysis steps ensure that a concurrent program will not fail because of ill-formed compositions of its concurrently executing parts. If we were to make a bold statement we could say that by following Dijkstra's principle of separating concerns to the maximal extent we were able to take another step towards Knuth's formulated goal of provably correct software systems, even when exploiting the seemingly steadily increasing number of available cores in our computing machinery.

A key factor in being able to achieve this was the insight that complex coordination programs can be decomposed into a set of independent data-flow paths. Although components within a program typically belong to several such paths we were able to analyse these paths independently of each other. This deconstruction made it possible for us to infer a most-general type for each route in isolation using Gaster's system of extensible records and then to reconstruct the original program without loosing any information of the original program. A concise semantics framework that allows for step-wise interpretation of coordination programs even in the absence of any concrete implementation of its components was devised. This served a dual purpose: Firstly, it provided the basis for giving a formal meaning to each of the combinators and programs as a whole. It also allowed us to prove that the inference and analysis algorithms indeed compute type constraints that restrict the input of program in such a way that at any stage a further reduction step is possible. In short, we were able to show progress and preservation for well-typed coordination programs. Secondly, the semantics framework provides clear guidance for an implementation of the language without prescribing to a particular execution model. One such implementation using stream-based communication and multi-threaded execution of components has been presented as a high-level, functional specification. The tool-chain of S-NET that is in use today is directly based on the presented specification. Several applications, some in collaboration with industry, have been used to confirm that the approach as a whole presents a viable alternative to more convential practices of parallel programming.

It is worth pointing out that the ability of the presented approach to infer all type information statically leads to a situation where type checks in a runtime system implementation are made completely redunant: All routing decisions at runtime do not require any type analysis but can be implemented as trivial updates to the routing vectors that have been computed at compile time. This allows for almost arbitrary network combinations, including parallel compositions containing branches that are only safe for certain inputs, without requiring any context information at split points. Routing decisions are completely localised and type-agnostic, based on the path decomposition technique that we have developed as part of the type inference process. Although the complexity of the algorithm is in general exponential in the number of decisions, several specification techniques may be used (as discussed in Chapter 6.10) to restrict the potential choices to a managable amount.

Before discussing alternative implementations and future research directions, we shall direct a final remark at the technically inclined reader: You might be excited to learn that since recently the entire S-Net implementation is available as an open-source project[1] to encourage interested parties to freely experiment with it and actively participate in future research and development work.

10.1 Implementation Aspects Beyond the Scope of the Thesis

The formal specification of the S-Net model determines the possible ways in which computational tasks interact, which inputs a program can accepts and what output it may produce. The formalisation deliberately does not define a specific execution model for programs to allow for flexibility in runtime system designs. A stream-based runtime system we have seen in this dissertation, and in fact, the implementation provided the basis of the S-Net tool-chain since its initial version. A distributed memory implementation of the runtime system is also available [GJP10] that allows to execute programs on MPI clusters. The ray-tracing use case is based on this. Currently, further development is taking place that targets Intel's Single Chip Cloud Computer [HDH+10] as an execution platform for S-Net programs [VGvT+11]. Alternative implementations that are not based on streams and follow a data-centric exploitation strategy for concurrency, as alternative to the task-centric approach, are explored [THSS08], as well as implementations that target new architecture concepts such as the Self-Adaptive Virtual Processor [Jes06, BJT+08].

Further to these development efforts, an additional threading and monitoring layer has been developed [Pro10] to enable extensive performance analysis and profiling of S-Net applications. The threading layer transparently inserts into the runtime system implementation described in Chapter 8 and records various runtime measurements as trace files for post-run analysis. Making these measurements available at runtime forms the basis for the work carried out in project ADVANCE. The objective of the project is to develop statistical analyses mechanisms of runtime measurements to guide the optimisations of computational code as well as optimisations of the coordination model.

The lessons learnt from the work on performance analysis will also help to improve the overall efficiency of system implementations. The thinner the actual implementation of the coordination layer is, the more attractive it becomes as an alternative to explicit and manual thread management. Using threads as an abstraction to model

[1]snetdev.github.com

concurrency is still a wide-spread technique as it promises highest-possible performance with threads being natively supported by many operating systems and hardware architectures [IG99, Inc10, DHJS10, MBH$^+$02]. Working with threads, however, can be a challenge even for experienced programmers (see for example [Lee06]). As the complexity grows where several threading abstractions are used simultaneously, for example in applications that are written to execute on a clusters of SMP machines using MPI and PTHREADS, the verification of correctness quickly becomes intractable. As mentioned above, the current S-NET tool-chain already targets this kind of system; future research has to show what further requirements of a unified programming model for heterogeneous systems are, how these can be addressed from an S-NET perspective and how further targets like CUDA may be included.

10.2 Runtime Reconfiguration

Within the existing system it should be fairly straight-forward to provide a template mechanism where typical design pattern, e.g. fork-join based networks and load balancing schemes, are made readily available. Once a model has been checked for consistency it is stored as a generic template and instantiated with a new set of tasks when required. Experimenting with various implementations of coordination models would be a quick and easy task.

Providing mechanisms for reconfiguration at runtime opens up an entire area of future research. The reasons to reconfigure networks at runtime may be manifold and range from simple software updates, for example when an improved version of a computational task becomes available, over bottle-neck resolution through dynamic resource management, to recovery from failure when tasks have crashed. One may argue that reconfiguration and adaptation of applications to changes in the runtime environment are intrinsic aspects of concurrency engineering. As such, a coordination language should provide flexible means to address these issues.

In the setting of S-NET we have an easy way of introducing reconfiguration mechanism seamlessly with the rest of the language by means of a dedicated replacement combinator. The reconfiguration combinator introduces "cutting points" on the input stream and on the output stream of its operand. Between these cutting points the network is replaceable, i.e. the combinator may discard its current operand and instantiate a new one in its place. In order to provide a replacement combinator with new operands we promote networks to first-class citizens of the language; where records previously could carry labels that reference field data and tag values, records may now additionally carry fields that reference S-NET programs. A label referencing a network has the network's type signature a its label type. The transmission of new operands to replacement combinators is carried out through the existing network infrastructure in the same way as fields and tags are routed to tasks.

When a record reaches the replacement combinator, the combinator may treat it in two different ways. If the record contains a compatible network, i.e. a new operand, the combinator reads and removes the network from the record and replaces its current operand by the new network. If the record contains an incompatible network or no network at all, the record is sent to the operand of the replacement combinator where the record is processed as usual.

The replacement combinator allows us to refine networks from the outside. The next step is to enable networks to trigger reconfiguration from the inside; this enables us to define self-adaptive networks, i.e. networks reconfigure themselves. A self-adaptive network collects runtime information about itself while it is processing records. Runtime information are stored in ordinary records, possibly just as ordinary fields and tags, and are treated in the usual way by the rest of the system. A box may analyse the collected runtime information and, for example, output a refined network layout, an optimised task implementation or it may decide that no reconfiguration is necessary. The output of such a box, however, needs to be sent back to the entry point of the network about which the runtime information was collected. More specifically, the record has to be sent back to he replacement combinator that surrounds the network so that a replacement with a refined network can take place. The feedback combinator takes a network and a pattern as arguments. Every record that the operand network emits is matched against the pattern. Records that do not match the pattern are output as usual. If a record matches the pattern however, the record is sent back to the feedback operand. We guarantee a deterministic record order for the feedback combinator: If a record is sent back, all results from this record will be output before any new record is read from the inbound stream.

The combination of replacement and feedback combinators allows for the implementation of user-defined adaptation strategies on the level of S-Net. The mechanisms seamlessly integrate with the rest of the language design and the impact on runtime system implementations are limited as only two new combinators need to be supported.

A full formalisation of the combinators, and more importantly, the introduction of higher-order types and inference rules into the existing type system need to remain as future work at this point.

References

[AA05] Benjamin A. Allan and Robert Armstrong. The ccaffeine framework: Composing and debugging applications interactively and running them statically. In *Compframe 2005*. Atlanta, Georgia, June 2005. (extended abstract),. Available from: `http://www.cca-forum.org/~baallan/ccafe-cf05.pdf`.

[AAB+06] Benjamin A. Allan, Robert Armstrong, David E. Bernholdt, Felipe Bertrand, Kenneth Chiu, Tamara L. Dahlgren, Kostadin Damevski, Wael R. Elwasif, Thomas G. W. Epperly, Madhusudhan Govindaraju, Daniel S. Katz, James A. Kohl, Manoj Krishnan, Gary Kumfert, J. Walter Larson, Sophia Lefantzi, Michael J. Lewis, Allen D. Malony, Lois C. Mclnnes, Jarek Nieplocha, Boyana Norris, Steven G. Parker, Jaideep Ray, Sameer Shende, Theresa L. Windus, and Shujia Zhou. A component architecture for high-performance scientific computing. *Int. J. High Perform. Comput. Appl.*, 20:163–202, May 2006. ISSN 1094-3420. `doi:10.1177/1094342006064488`.

[ABD+09] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52:56–67, October 2009. ISSN 0001-0782. `doi:http://doi.acm.org/10.1145/1562764.1562783`.

[AC86] Arvind and David E. Culler. *Dataflow architectures*. Annual Reviews Inc., Palo Alto, CA, USA, 1986. ISBN 0-8243-3201-6. 225–253 pp. Available from: `http://portal.acm.org/citation.cfm?id=17814.17824`.

[ACc+03] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003. ISSN 1066-8888.

[ACG94] Shakil Ahmed, Nicholas Carriero, and David Gelernter. A program building tool for parallel applications. In *DIMACS Workshop on Specification of Parallel Algorithms*, pages 161–178. Princeton University, May 1994.

[Ack82] W.B. Ackerman. Data flow languages. *Computer*, 15(2):15 – 25, feb 1982. ISSN 0018-9162. `doi:10.1109/MC.1982.1653938`.

[ACPR95] Martin Abadi, Luca Cardelli, Benjamin Pierce, and Didier Remy. Dynamic typing in polymorphic languages. *JOURNAL OF FUNCTIONAL PROGRAMMING*, 5:92–103, 1995.

[ACW$^+$09] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. Petabricks: a language and compiler for algorithmic choice. *SIGPLAN Not.*, 44:38–49, June 2009. ISSN 0362-1340. Available from: `http://doi.acm.org/10.1145/1543135.1542481`, `doi:http://doi.acm.org/10.1145/1543135.1542481`.

[ADV11] ADVANCE. EU FP7 project website, 2011. accessed Oct. 2011. Available from: `http://www.project-advance.eu`.

[AHS93] F. Arbab, I. Herman, and P. Spilling. An overview of manifold and its implementation. *Concurrency: Pract. Exper.*, 5:23–70, February 1993. ISSN 1040-3108. Available from: `http://portal.acm.org/citation.cfm?id=157274.157276`, `doi:10.1002/cpe.4330050103`.

[AKM$^+$06] Rob Armstrong, Gary Kumfert, Lois Curfman McInnes, Steven Parker, Ben Allan, Matt Sottile, Thomas Epperly, and Tamara Dahlgren. The cca component model for high-performance scientific computing. *Concurrency and Computation: Practice and Experience*, 18(2):215–229, 2006. ISSN 1532-0634. `doi:http://dx.doi.org/10.1002/cpe.911`.

[ALSU07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools with Gradiance*. Addison-Wesley Publishing Company, USA, 2nd edition, 2007. ISBN 0321547985, 9780321547989.

[AN89] Arvind and Rishiyur S. Nikhil. A dataflow approach to general-purpose parallel computing. Technical Report CSG Memo 302, Massachusetts Institute of Technology, Laboratory for Computer Science, July 1989.

[AN90] R.S. Arvind; Nikhil. Executing a program on the mit tagged-token dataflow architecture. *Transactions on Computers*, 39(3):300–318, 1990. ISSN 0018-9340.

[AN05] Franz Achermann and Oscar Nierstrasz. A calculus for reasoning about software composition. *Journal of Theoretical Computer Science*, 331(2-3):367–396, 2005. ISSN 0304-3975. Available from: `http://dx.doi.org/10.1016/j.tcs.2004.09.022`, `doi:10.1016/j.tcs.2004.09.022`.

[ANP89] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: data structures for parallel computing. *ACM Trans. Program. Lang. Syst.*, 11(4): 598–632, 1989. ISSN 0164-0925.

[AR03] F. Arbab and J. Rutten. A coinductive calculus of component connectors. In Martin Wirsing, Dirk Pattinson, and Rolf Hennicker, editors, *Recent*

Trends in Algebraic Development Techniques, volume 2755 of *Lecture Notes in Computer Science*, pages 34–55. Springer Berlin / Heidelberg, 2003. ISBN 978-3-540-20537-1. 10.1007/978-3-540-40020-2_2. Available from: `http://dx.doi.org/10.1007/978-3-540-40020-2_2`.

[Arb04] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical. Structures in Comp. Sci.*, 14(3):329–366, 2004. ISSN 0960-1295.

[Ass97] Claus Assmann. Coordinating functional processes using petri nets. In *Selected Papers from the 8th International Workshop on Implementation of Functional Languages*, pages 162–183. Springer-Verlag, London, UK, 1997. ISBN 3-540-63237-9.

[AW77] E. A. Ashcroft and W. W. Wadge. Lucid, a nonprocedural language with iteration. *Commun. ACM*, 20(7):519–526, 1977. ISSN 0001-0782.

[Bac78] John Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Commun. ACM*, 21:613–641, August 1978. ISSN 0001-0782. Available from: `http://doi.acm.org/10.1145/359576.359579`, `doi:http://doi.acm.org/10.1145/359576.359579`.

[BAC06] Matthias Blume, Umut A. Acar, and Wonseok Chae. Extensible programming with first-class cases. In *Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, ICFP '06, pages 239–250. ACM, New York, NY, USA, 2006. ISBN 1-59593-309-3.

[BBC10] Marco Barbosa, Luis Barbosa, and JosÃľ Campos. A coordination model for interactive components. In Farhad Arbab and Marjan Sirjani, editors, *Fundamentals of Software Engineering*, volume 5961 of *Lecture Notes in Computer Science*, pages 416–430. Springer Berlin / Heidelberg, 2010. ISBN 978-3-642-11622-3. Available from: `http://dx.doi.org/10.1007/978-3-642-11623-0_25`.

[BBD$^+$02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16. ACM, New York, NY, USA, 2002. ISBN 1-58113-507-6.

[BBH$^+$08] U. Bondhugula, M. Baskaran, A. Hartono, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Towards effective automatic parallelization for multicore systems. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1 –5, april 2008. ISSN 1530-2075. `doi:10.1109/IPDPS.2008.4536401`.

[BC85] Gérard Berry and Laurent Cosserat. The esterel synchronous programming language and its mathematical semantics. In *Seminar on Concurrency, Carnegie-Mellon University*, pages 389–448. Springer-Verlag, London, UK, 1985. ISBN 3-540-15670-4.

[BC11] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Commun. ACM*, 54:67–77, May 2011. ISSN 0001-0782. Available from: `http://doi.acm.org/10.1145/1941487.1941507`, `doi:http://doi.acm.org/10.1145/1941487.1941507`.

[BCE+03] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003. ISSN 0018-9219.

[BCG88] G. Berry, P. Couronne, and G. Gonthier. Synchronous programming of reactive systems: an introduction to esterel. In *Proceedings of the first Franco-Japanese Symposium on Programming of future generation computers*, pages 35–56. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, 1988. ISBN 0-444-70410-8. Available from: `http://portal.acm.org/citation.cfm?id=60661.60664`.

[BCM88] J.-P. Banâtre, A. Coutant, and D. Le Metayer. A parallel machine for multiset transformation and its programming style. *Future Generation Computer Systems*, 4(2):133 – 144, 1988. ISSN 0167-739X. Available from: `http://www.sciencedirect.com/science/article/pii/0167739X8890012X`, `doi:10.1016/0167-739X(88)90012-X`.

[BCS02] E. Bruneton, T. Coupaye, and J. B. Stefani. Recursive and dynamic software composition with sharing. In *Seventh International Workshop on Component-Oriented Programming at ECOOP*, June 2002.

[BCS+09a] Robert Bjornson, Nicholas Carriero, Martin Schultz, Patrick Shields, and Stephen Weston. Networkspace: A coordination system for high-productivity environments. *International Journal of Parallel Programming*, 37:106–125, 2009. ISSN 0885-7458. 10.1007/s10766-008-0091-4. Available from: `http://dx.doi.org/10.1007/s10766-008-0091-4`.

[BCS09b] Gordon Blair, Thierry Coupaye, and Jean-Bernard Stefani. Component-based architecture: the fractal initiative. *Annals of Telecommunications*, 64: 1–4, 2009. ISSN 0003-4347. 10.1007/s12243-009-0086-1. Available from: `http://dx.doi.org/10.1007/s12243-009-0086-1`.

[Bet00] Gustavo Betarte. Type checking dependent (record) types and subtyping. *J. Funct. Program.*, 10(2):137–166, 2000. ISSN 0956-7968.

[BFR08] Jean-Pierre Banatre, Pascal Fradet, and Yann Radenac. The chemical reaction model recent developments and prospects. In Martin Wirsing, Jean-Pierre Banatre, Matthias Hoelzl, and Axel Rauschmayer, editors, *Software-Intensive Systems and New Computing Paradigms*, volume 5380 of *Lecture Notes in Computer Science*, pages 209–234. Springer Berlin / Heidelberg, 2008. ISBN 978-3-540-89436-0. Available from: `http://dx.doi.org/10.1007/978-3-540-89437-7_14`.

[BGS94] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26:345–420, December 1994. ISSN 0360-0300.

[BGvN89] Arthur W. Burks, Herman H. Goldstine, and John von Neumann. *Perspectives on the computer revolution*, chapter Preliminary discussion of the logical design of an electronic computing instrument (1946), pages 39–48. Ablex Publishing Corp., Norwood, NJ, USA, 1989. ISBN 0-89391-369-3. Available from: `http://dl.acm.org/citation.cfm?id=98326.98337`.

[BJT$^+$08] K. Bousias, C.R. Jesshope, J. Thiyagalingam, S-B. Scholz, and A. Shafarenko. Graph Walker: Implementing S-Net on the Self-adaptive Virtual Processor. In *Proceedings of the Æther-Morpheus Workshop From Reconfigurable to Self-Adaptive Computing (AMWAS'08), Lugano, Switzerland*. ALaRI, University of Lugano, 2008.

[BLM93] Jean-Pierre Banâtre and Daniel Le Métayer. Programming by multiset transformation. *Commun. ACM*, 36:98–111, January 1993. ISSN 0001-0782. `doi:http://doi.acm.org/10.1145/151233.151242`.

[BM86] Jean-Pierre Banâtre and Daniel L. Metayer. A New Computational Model and its Discipline of Programming. Technical Report RR0566, INRIA, September 1986.

[Boa11] OpenMP Achitecture Review Board. *OpenMP Application Program Interface 3.1*, 2011. Available from: `http://www.openmp.org/mp-documents/OpenMP3.1.pdf`.

[Buc03] Ian Buck. Brook spec v0.2. Technical report, Stanford University, 2003.

[Bur75] W. H. Burge. Stream processing functions. *j-IBM-JRD*, 19(1):12–25, January 1975. CODEN IBMJAE. ISSN 0018-8646.

[BVZ$^+$08] Matthew J. Bridges, Neil Vachharajani, Yun Zhang, Thomas Jablin, and David I. August. Revisiting the sequential programming model for the multicore era. *IEEE Micro*, 28:12–20, January 2008. ISSN 0272-1732. Available from: `http://dl.acm.org/citation.cfm?id=1345865.1345875`, `doi:10.1109/MM.2008.13`.

[BWSF06] C. Benthin, I. Wald, M. Scheerbaum, and H. Friedrich. Ray tracing on the cell processor. In *IEEE Symposium on Interactive Ray Tracing (RT'06), Salt Lake City, USA*, pages 15–23. IEEE Computer Society, 2006.

[BYR$^+$11] V. Basupalli, T. Yuki, S. Rajopadhye, A. Morvan, S. Derrien, P. Quinton, and D. Wonnacott. ompverify: Polyhedral analysis for the openmp programmer. In Barbara Chapman, William Gropp, Kalyan Kumaran, and Matthias MÃijller, editors, *OpenMP in the Petascale Era*, volume 6665 of *Lecture Notes in Computer Science*, pages 37–53. Springer Berlin / Heidelberg, 2011. ISBN 978-3-642-21486-8. Available from: `http://dx.doi.org/10.1007/978-3-642-21487-5_4`.

[CBB$^+$03] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Çetintemel, Ying Xing, and Stanley B. Zdonik. Scalable distributed stream processing. In *CIDR*, 2003.

[CcC$^+$02] Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. Monitoring streams - a new class of data management applications. In *VLDB*, pages 215–226, 2002.

[CDMM10] James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. The gentle art of levitation. *SIGPLAN Not.*, 45:3–14, September 2010. ISSN 0362-1340.

[CDW01] Eylon Caspi, AndrÃl' Dehon, and John Wawrzynek. A streaming multithreaded model. In *In Proceedings of the Third Workshop on Media and Stream Processors*, pages 21–28, 2001.

[CEG$^+$08] Haoxan Cai, Susan Eisenbach, Clemens Grelck, Frank Penczek, Sven-Bodo Scholz, and Alex Shafarenko. S-Net type system and operational semantics. In *Proceedings of the Æther-Morpheus Workshop From Reconfigurable to Self-Adaptive Computing (AMWAS'08), Lugano, Switzerland*, 10 2008.

[Cer98] Paul E. Ceruzzi. *A history of modern computing*. MIT Press, Cambridge, MA, USA, 1998. ISBN 0-262-03255-4.

[CESG07] Haoxan Cai, Susan Eisenbach, Alex Shafarenko, and Clemens Grelck. Extending the S-Net Type System. In *Proceedings of the Æther-Morpheus Workshop From Reconfigurable to Self-Adaptive Computing (AMWAS'07), Paris, France*, 2007.

[CG86] Keith Clark and Steve Gregory. Parlog: parallel programming in logic. *ACM Trans. Program. Lang. Syst.*, 8:1–49, January 1986. ISSN 0164-0925. Available from: `http://doi.acm.org/10.1145/5001.5390`, `doi:http://doi.acm.org/10.1145/5001.5390`.

[CG89] Nicholas Carriero and David Gelernter. Linda in context. *Commun. ACM*, 32:444–458, April 1989. ISSN 0001-0782. `doi:http://doi.acm.org/10.1145/63334.63337`.

[Che91] Doreen Y Cheng. A survey of parallel programming tools, NAS systems division technical report rnd93005. Technical report, NASA Ames Research Center, 1991.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940. ISSN 00224812. Available from: `http://www.jstor.org/stable/2266170`.

[Cla88] K.L. Clark. Parlog and its applications. *Software Engineering, IEEE Transactions on*, 14(12):1792 –1804, dec 1988. ISSN 0098-5589. `doi:10.1109/32.9064`.

[CLJ+07] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel haskell: a status report. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, DAMP '07, pages 10–18. ACM, New York, NY, USA, 2007. ISBN 978-1-59593-690-5. `doi:http://doi.acm.org/10.1145/1248648.1248652`.

[CLL02] F. H. Carvalho, Jr., R. M. F. Lima, and R. D. Lins. Coordinating functional processes with haskell#. In *Proceedings of the 2002 ACM symposium on Applied computing*, SAC '02, pages 393–400. ACM, New York, NY, USA, 2002. ISBN 1-58113-445-2. `doi:http://doi.acm.org/10.1145/508791.508865`.

[CM90] Luca Cardelli and John C. Mitchell. Operations on records. In *Proceedings of the fifth international conference on Mathematical foundations of programming semantics*, pages 22–52. Springer-Verlag New York, Inc., New York, NY, USA, 1990. ISBN 0-387-97375-3.

[CM06] F.L. Chevalier and S. Maria. Stap processing without noise-only reference: requirements and solutions. In *Radar, 2006. CIE '06. International Conference on*, pages 1–4, 10 2006.

[CP98] Erick Cantu-Paz. A survey of parallel genetic algorithms. *CALCULATEURS PARALLELES*, 10:1–30, 1998.

[CPG+10] C. Grelck and A. Shafarenko (eds):, F. Penczek, C. Grelck, H. Cai, J. Julku, P. Hölzenspies, S.B. Scholz, and A. Shafarenko. S-Net Language Report 2.0. Technical Report 499, University of Hertfordshire, School of Computer Science, Hatfield, England, United Kingdom, 2010.

[CPHP87] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre: A declarative language for programming synchronous systems. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 178–188, 1987.

[CPLA11] Dave Clarke, Jose Proenca, Alexander Lazovik, and Farhad Arbab. Channel-based coordination via constraint satisfaction. *Science of Computer Programming*, 76(8):681 – 710, 2011. ISSN 0167-6423. Available from: `http://www.sciencedirect.com/science/article/pii/S0167642310000948`, `doi:10.1016/j.scico.2010.05.004`.

[DAK+11] Minh Ngoc Dinh, D. Abramson, D. Kurniawan, Chao Jin, B. Moench, and L. DeRose. Assertion based parallel debugging. In *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*, pages 63 –72, may 2011. `doi:10.1109/CCGrid.2011.44`.

[Dav11] Michael Davis. Will software engineering ever be engineering? *Commun. ACM*, 54:32–34, November 2011. ISSN 0001-0782. Available from: `http://doi.acm.org/10.1145/2018396.2018407`, `doi:http://doi.acm.org/10.1145/2018396.2018407`.

[dCJL10] Francisco Heron de Carvalho-Junior and Rafael Dueire Lins. Compositional specification of parallel components using circus. *Electron. Notes Theor. Comput. Sci.*, 260:47–72, January 2010. ISSN 1571-0661. `doi:http://dx.doi.org/10.1016/j.entcs.2009.12.031`.

[DEKL09] Tamara Dahlgren, Thomas Epperly, Gary Kumfert, and James Leek. Babel users' guide, ucrl-sm-230026. Technical report, Lawrence Livermore National Laboratory, 2009.

[Den74] J. B. Dennis. First version of a data flow procedure language. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 362–376. Springer-Verlag, London, UK, 1974. ISBN 3-540-06859-7. Available from: `http://portal.acm.org/citation.cfm?id=647323.721501`.

[Dev11] Analog Devices. SHARC processor programming reference. Technical report, Analog Devices Inc., 2011.

[DGS11] Cinzia Di Giusto and Jean-Bernard Stefani. Revisiting glue expressiveness in component-based systems. In *Proceedings of the 13th international conference on Coordination models and languages*, COORDINATION'11, pages 16–30. Springer-Verlag, Berlin, Heidelberg, 2011. ISBN 978-3-642-21463-9. Available from: `http://dl.acm.org/citation.cfm?id=2022052.2022054`.

[DHE+03] W. J. Dally, P. Hanrahan, M. Erez, T. J. Knight, F. Labonte, J-H A., N. Jayasena, U. J. Kapasi, A. Das, J. Gummaraju, and I. Buck. Merrimac: Supercomputing with streams. In *SC'03*. Phoenix, Arizona, November 2003.

[DHJS10] Martin Dixon, Per Hammerlund, Stephan Jourdan, and Ronak Singhal. The next-generation intel core microarchitecture. *Intel Technology Journal*, 14:8–28, 2010.

[Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. ISSN 0029-599X. 10.1007/BF01386390. Available from: `http://dx.doi.org/10.1007/BF01386390`.

[Dij65] Edsger W. Dijkstra. EWD 123: cooperating sequential processes. Technical report, Department of Mathematics, Technological University of Eindhoven, 1965. ISBN 0-387-95401-5. 65–138 pp. Available from: `http://dl.acm.org/citation.cfm?id=762971.762974`.

[Dij82] Edsger W. Dijkstra. *Selected writings on computing: a personal perspective*, chapter EWD447, August 1974: On the Role of Scientific Thought, pages 60–66. Springer-Verlag New York, Inc., New York, NY, USA, 1982. ISBN 0-387-90652-5.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM, New York, NY, USA, 1982. ISBN 0-89791-065-6.

[DPH$^+$03] D.E. DeMarle, S. Parker, M. Hartner, C. Gribble, and C. Hansen. Distributed interactive ray tracing for large volume visualization. In *IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG'03), Seattle, USA*. IEEE Computer Society, 2003.

[Dre07] Ulrich Drepper. What Every Programmer Should Know About Memory. Technical report, Red Hat Inc., 2007. Available from: `http://people.redhat.com/drepper/cpumemory.pdf`.

[EAH97] Kemal Ebcioglu, Erik Altman, and Erdem Hokenek. A java ilp machine based on fast dynamic compilation. In *In IEEE MASCOTS International Workshop on Security and E ciency Aspects of Java*, 1997.

[EJL$^+$03] J. Eker, J.W. Janneck, E.A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Yuhong Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127 – 144, jan 2003. ISSN 0018-9219. `doi:10.1109/JPROC.2002.805829`.

[EKK01] Tom Epperly, Scott R. Kohn, and Gary Kumfert. Component technology for high-performance scientific simulation software. In *Proceedings of the IFIP TC2/WG2.5 Working Conference on the Architecture of Scientific Software*, pages 69–86. Kluwer, B.V., Deventer, The Netherlands, The Netherlands, 2001. ISBN 0-7923-7339-1. Available from: `http://dl.acm.org/citation.cfm?id=647102.717427`.

[EWC94] Aaron Garth Enright, Linda M. Wilkens, and James T. Canning. An alternative computer architecture course. *SIGCSE Bull.*, 26:9–12, December

1994. ISSN 0097-8418. Available from: `http://doi.acm.org/10.1145/190650.190653`, `doi:http://doi.acm.org/10.1145/190650.190653`.

[FLR98] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33:212–223, May 1998. ISSN 0362-1340. `doi:http://doi.acm.org/10.1145/277652.277725`.

[For09] Message Passing Interface Forum. Mpi: A message-passing interface standard, version 2.2. Technical report, University of Tennessee, Knoxville, 2009.

[Fos99] Ian Foster. Compositional parallel programming languages. *ACM Transactions on Programming Languages and Systems*, 8(1):111–134, January 1999. ISSN 0164-0925.

[FOT92] Ian Foster, Robert Olson, and Steven Tuecke. Productive parallel programming: The pcn approach. *Sci. Program.*, 1:51–66, January 1992. ISSN 1058-9244. Available from: `http://dl.acm.org/citation.cfm?id=1402583.1402587`.

[FP98] Jr. Fenwick, J.B. and L.L. Pollock. Data flow analysis across tuplespace process boundaries. In *Computer Languages, 1998. Proceedings. 1998 International Conference on*, pages 272 –281, may 1998. ISSN 1074-8970. `doi:10.1109/ICCL.1998.674177`.

[FP99] James Fenwick and Lori Pollock. Tuple counting data flow analysis and its use in communication optimization. In Peter Sloot, Marian Bubak, Alfons Hoekstra, and Bob Hertzberger, editors, *High-Performance Computing and Networking*, volume 1593 of *Lecture Notes in Computer Science*, pages 1282–1285. Springer Berlin / Heidelberg, 1999. ISBN 978-3-540-65821-4. 10.1007/BFb0100709. Available from: `http://dx.doi.org/10.1007/BFb0100709`.

[Gar70] M. Gardner. The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, 223:120–123, Oct. 1970.

[Gas98] Benedict R. Gaster. *Records, variants and qualified types*. PhD thesis, University of Nottingham, July 1998.

[GBN+97] J.-L. Gaudiot, W. Bohm, W. Najjar, T. DeBoni, J. Feo, and P. Miller. The sisal model of functional programming and its implementation. *Parallel Algorithms/Architecture Synthesis, 1997. Proceedings. Second Aizu International Symposium*, 1:112–123, 1997.

[GC92] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35:97–107, February 1992. ISSN 0001-0782. `doi:http://doi.acm.org/10.1145/129630.129635`.

[Gel85] David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985. ISSN 0164-0925.

[Gen34] Gerhard Gentzen. Untersuchungen über das logische schliessen. *Mathematische Zeitschrift*, 39:176–210, 1934.

[GGB87] Thierry Gautier, Paul Le Guernic, and Löic Besnard. Signal: A declarative language for synchronous programming of real-time systems. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 257–277. Springer-Verlag, London, UK, 1987. ISBN 0-387-18317-5.

[GHKW08] Thorsten Groetker, Ulrich Holtmann, Holger Keding, and Markus Wloka. Debugging parallel programs. In *The DeveloperâĂŹs Guide to Debugging*, pages 87–99. Springer Netherlands, 2008. ISBN 978-1-4020-5540-9.

[GJ96] Benedict R. Gaster and Mark P. Jones. A Polymorphic Type System for Extensible Records and Variants. Technical Report NOTTCS-TR-96-3, Department of Computer Science, University of Nottingham, November 1996.

[GJP09] Clemens Grelck, Jukka Julku, and Frank Penczek. Distributed S-Net. In M. Morazan, editor, *Implementation and Application of Functional Languages, 21st International Symposium, IFL'09, Seton Hall University*. Seton Hall University, 2009.

[GJP10] Clemens Grelck, Jukka Julku, and Frank Penczek. S-Net for Multi-Memory Multicores. In Leaf Peterson and Enrico Pontelli, editors, *5th ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming, Madrid, Spain, 2010*, pages 25–34. ACM Press, New York City, New York, USA, 2010. ISBN 978-1-60558-859-9.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Prentice Hall; 3 edition, 2005.

[GP10] C. Grelck and F. Penczek. Implementation Architecture and Multithreaded Runtime System of S-Net. In S.B. Scholz and O. Chitil, editors, *Implementation and Application of Functional Languages, 20th International Symposium, IFL'08, Hatfield, United Kingdom, Revised Selected Papers*, volume 5836 of *Lecture Notes in Computer Science*. Springer-Verlag, 2010.

[Gre01] Clemens Grelck. *Implicit Shared Memory Multiprocessor Support for the Functional Programming Language SAC — Single Assignment C*. PhD thesis, Institute of Computer Science and Applied Mathematics, University of Kiel, Germany, 2001. Logos Verlag, Berlin, 2001.

[GS87] J. Goldsmith and J. Salmon. Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Comput. Graph. Appl.*, 7(5):14–20, 1987.

[GS06] Clemens Grelck and Sven-Bodo Scholz. SAC: A functional array language for efficient multithreaded execution. *International Journal of Parallel Programming*, 34(4):383–427, 2006.

[GSS$^+$07] G.R. Gao, T. Sterling, R. Stevens, M. Hereld, and W. Zhu. Parallex: A study of a new parallel computation model. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1 –6, march 2007. `doi:10.1109/IPDPS.2007.370484`.

[GVL10] Horacio González-Vélez and Mario Leyton. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Softw. Pract. Exper.*, 40:1135–1160, November 2010. ISSN 0038-0644. `doi:http://dx.doi.org/10.1002/spe.v40:12`.

[Ham06] Kevin Hammond. Exploiting purely functional programming to obtain bounded resource behaviour: The hume approach. In ZoltÃạn HorvÃạth, editor, *Central European Functional Programming School*, volume 4164 of *Lecture Notes in Computer Science*, pages 100–134. Springer Berlin / Heidelberg, 2006.

[HB84] Kai Hwang and Faye A. Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill Book Company, 1984.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9): 1305 –1320, sep 1991. ISSN 0018-9219. `doi:10.1109/5.97300`.

[HDH$^+$10] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 108 –109. IEEE International, feb. 2010. ISSN 0193-6530. `doi:10.1109/ISSCC.2010.5434077`.

[Hel78] Don Heller. A survey of parallel algorithms in numerical linear algebra. *SIAM Review*, 20(4):pp. 740–777, 1978. ISSN 00361445. Available from: `http://www.jstor.org/stable/2029741`.

[Hen94] Fritz Henglein. Dynamic typing: syntax and proof theory. *Science of Computer Programming*, 22(3):197 – 230, 1994. ISSN 0167-6423.

[Hen04] M. Henning. A new approach to object-oriented middleware. *Internet Computing, IEEE*, 8(1):66 – 75, jan-feb 2004. ISSN 1089-7801. `doi:10.1109/MIC.2004.1260706`.

[Her10] Stephan A. Herhut. *Auxiliary Computations: A Framework for a Step-Wise, Non-Disruptive Introduction of Static Guarantees to Untyped Programs Using Partial Evaluation Techniques*. PhD thesis, University of Hertfordshire, UK, 2010.

[HHK97] Y.C. Hwang, D.H. Hong, and Y.K. Kwag. Real-time o.s. based radar controller for multi-mode phased array radar system. In *Radar 97 (Conf. Publ. No. 449)*, pages 558–562, Oct 1997. ISSN 0537-9989.

[HJ88] R. W. Hockney and C. R. Jesshope. *Parallel Computers 2*. IOP Publishing Ltd (Adam Hilger Imlprint), 1988.

[HLM95] J.-P. Hardange, P. Lacomme, and J.-C. Marchais. *Radars aéroportés et spatiaux*. Masson, 1995.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21: 666–677, August 1978. ISSN 0001-0782.

[HP91] Robert Harper and Benjamin Pierce. A record calculus based on symmetric concatenation. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 131–142. ACM, New York, NY, USA, 1991. ISBN 0-89791-419-8.

[HP06a] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*, chapter Introduction, pages 2–8. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 2006.

[HP06b] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*, chapter Instruction-Level Parallelism and its Exploitation, pages 65–151. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 2006.

[HPP09] Mary Hall, David Padua, and Keshav Pingali. Compiler research: the next 50 years. *Commun. ACM*, 52:60–67, February 2009. ISSN 0001-0782. Available from: `http://doi.acm.org/10.1145/1461928.1461946`, `doi:http://doi.acm.org/10.1145/1461928.1461946`.

[HS94] Andreas V. Hense and Gert Smolka. A record calculus with principal types. In *CCL '94: Proceedings of the First International Conference on Constraints in Computational Logics*, pages 219–236. Springer-Verlag, London, UK, 1994. ISBN 3-540-58403-X.

[HS09] Jason Howarth and Edward Stow. A post-mortem javaspaces debugger. In Nikos Mastorakis, Valeri Mladenov, and Vassiliki T. Kontargyri, editors, *Proceedings of the European Computing Conference*, volume 28 of *Lecture Notes in Electrical Engineering*, pages 553–561. Springer US, 2009. ISBN 978-0-387-85437-3. Available from: `http://dx.doi.org/10.1007/978-0-387-85437-3_55`.

[HSF92] Susan Flynn Hummel, Edith Schonberg, and Lawrence E. Flynn. Factoring: a method for scheduling parallel loops. *Commun. ACM*, 35(8):90–101, 1992. ISSN 0001-0782. doi:http://doi.acm.org/10.1145/135226.135232.

[IG99] IEEE and Open Group. Ieee std 1003.1c, threads extensions. Technical report, IEEE / Open Group, 1999.

[Inc10] Oracle Inc. SPARC T4 Processor, Data Sheet 497205. http://www.oracle.com/us/products/servers-storage/servers/sparc-enterprise/t-series/sparc-t4-processor-ds-497205.pdf, 2010. accessed Oct. 2011.

[Int99] International Organisation for Standardization. Iso/iec 9899:1999 programming languages – C. Technical report, JTC 1 / SC 22, 1999.

[Int00] International Organisation for Standardization. Iso/iec 13211-2:2000. Technical report, JTC 1 / SC 22, 2000.

[Int06] International Organisation for Standardization. Iso/iec 23270:2006 information technology – c# language specification. Technical report, JTC 1 / SC 22, 2006.

[Int07] International Organisation for Standardization. Iso/iec 8652:1995/amd 1:2007. Technical report, JTC 1 / SC 22, 2007.

[Int10] International Organisation for Standardization. Iso/iec 1539-1:2010. Technical report, JTC 1 / SC 22, 2010.

[ISO96] ISO/IEC. International Standard: Information technology – Syntactic metalanguage – Extended BNF. ISO/IEC 14977: 1996(E), First Edition, December 15, 1996.

[Jes06] Chris R. Jesshope. Microthreading a model for distributed instruction-level concurrency. *Parallel Processing Letters*, 16(2):209–228, 2006.

[JHM04] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36:1–34, March 2004. ISSN 0360-0300.

[Jon92] Mark P. Jones. A theory of qualified types. In Bernd K. Bruckner, editor, *ESOP '92, 4th European Symposium on Programming, Rennes, France, February 1992, Proceedings*, volume 582, pages 287–306. Springer-Verlag, New York, N.Y., 1992.

[Jon95a] Mark P. Jones. *Qualified Types: Theory and Practice*. Distinguished Dissertations in Computer Science. Cambridge University Press, January 1995.

[Jon95b] Mark P. Jones. Simplifying and improving qualified types. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 160–169. ACM, New York, NY, USA, 1995. ISBN 0-89791-719-7.

[Jon97] Mark P. Jones. A prototype implementation of extensible records for Hugs. Part of the documentation for Hugs, which is available at http://www.haskell.org/hugs/ (accessed May 2011), March 1997.

[KA88] D. Kirk and J. Arvo. The Ray Tracing Kernel. In *Ausgraph'88, Melbourne, Australia*, pages 75–82, 1988.

[Kah73] Gilles Kahn. A Preliminary Theory for Parallel Programs. Research Report, INRIA, 1973. 19 pp. Rapport IRIA.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475. North Holland, Amsterdam, Stockholm, Sweden, Aug 1974.

[KHH$^+$08] Bettina Krammer, Tobias Hilbrich, Valentin Himmler, Blasius Czink, Kiril Dichev, and Matthias S. MÃijller. Mpi correctness checking with marmot. In Michael Resch, Rainer Keller, Valentin Himmler, Bettina Krammer, and Alexander Schulz, editors, *Tools for High Performance Computing*, pages 61–78. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-68564-7. Available from: `http://dx.doi.org/10.1007/978-3-540-68564-7_5`.

[KKKV00] Dieter Kranzlmueller, Rene Kobler, Rainer Koppler, and Jens Volkert. Debugging mpi programs with array visualization. In Marian Bubak, Hamideh Afsarmanesh, Bob Hertzberger, and Roy Williams, editors, *High Performance Computing and Networking*, volume 1823 of *Lecture Notes in Computer Science*, pages 597–600. Springer Berlin / Heidelberg, 2000. ISBN 978-3-540-67553-2. 10.1007/3-540-45492-6_71. Available from: `http://dx.doi.org/10.1007/3-540-45492-6_71`.

[KL81] Robert M. Keller and Gary Lindstrom. Applications of feedback in functional programming. In *FPCA '81: Proceedings of the 1981 conference on Functional programming languages and computer architecture*, pages 123–130. ACM, New York, NY, USA, 1981. ISBN 0-89791-060-5.

[Klo87] C. Delgado Kloos. Stream: a scheme language for formally describing digital circuits. In *Volume II: Parallel Languages on PARLE: Parallel Architectures and Languages Europe*, pages 333–350. Springer-Verlag, London, UK, 1987. ISBN 0-387-17945-3.

[KM66] Richard M. Karp and Raymond E. Miller. Properties of a model for parallel computations: Determinancy, termination, queueing. *SIAM Journal on Applied Mathematics*, 14(6):1390–1411, 1966. ISSN 00361399.

[KM69] Richard M. Karp and Raymond E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3 (2):147 – 195, 1969. ISSN 0022-0000. Available from: `http://www.sciencedirect.com/science/article/pii/S0022000069800115`, `doi:DOI:10.1016/S0022-0000(69)80011-5`.

[KMZS08] Henry Kasim, Verdi March, Rita Zhang, and Simon See. Survey on parallel programming model. In Jian Cao, Minglu Li, Min-You Wu, and Jinjun Chen, editors, *Network and Parallel Computing*, volume 5245 of *Lecture Notes in Computer Science*, pages 266–275. Springer Berlin / Heidelberg, 2008. ISBN 978-3-540-88139-1. 10.1007/978-3-540-88140-7_24. Available from: `http://dx.doi.org/10.1007/978-3-540-88140-7_24`.

[Knu70] Donald E. Knuth. Von neumann's first computer program. *ACM Comput. Surv.*, 2:247–260, December 1970. ISSN 0360-0300. Available from: `http://doi.acm.org/10.1145/356580.356581`, `doi:http://doi.acm.org/10.1145/356580.356581`.

[Knu74] Donald E. Knuth. Computer programming as an art. *Communications of the ACM*, 17:667–673, December 1974.

[Kos73] Paul R. Kosinski. A data flow language for operating systems programming. *SIGPLAN Not.*, 8:89–94, January 1973. ISSN 0362-1340. Available from: `http://doi.acm.org/10.1145/390014.808289`, `doi:http://doi.acm.org/10.1145/390014.808289`.

[Kos78] Paul R. Kosinski. A straightforward denotational semantics for non-determinate data flow programs. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '78, pages 214–221. ACM, New York, NY, USA, 1978.

[KRD$^+$03] Ujval J. Kapasi, Scott Rixner, William J. Dally, Brucek Khailany, Jung Ho Ahn, Peter Mattson, and John D. Owens. Programmable stream processors. *IEEE Computer*, 36:54–62, Aug 2003.

[KRSG94] L. V. Kalé, B. Ramkumar, A. B. Sinha, and A. Gursoy. The CHARM Parallel Programming Language and System: Part I – Description of Language Features. Technical report, Parallel Programming Laboratory, University of Illinois, 1994.

[KSA09] Rajesh K. Karmani, Amin Shali, and Gul Agha. Actor frameworks for the jvm platform: a comparative analysis. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 11–20. ACM, New York, NY, USA, 2009. ISBN 978-1-60558-598-7. Available from: `http://doi.acm.org/10.1145/1596655.1596658`, `doi:http://doi.acm.org/10.1145/1596655.1596658`.

[KV07] Alexey Kukanov and Michael J. Voss. The foundations for scalable multi-core software in intel threading building blocks. *Intel*, 11:309–322, 2007.

[Lan65a] P. J. Landin. Correspondence between algol 60 and church's lambda-notation: part i. *Commun. ACM*, 8(2):89–101, 1965. ISSN 0001-0782.

[Lan65b] P. J. Landin. A correspondence between algol 60 and church's lambda-notations: Part ii. *Commun. ACM*, 8(3):158–167, 1965. ISSN 0001-0782.

[LAN00] Markus Lumpe, Franz Achermann, and Oscar Nierstrasz. A formal language for composition. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of component-based systems*, pages 69–90. Cambridge University Press, New York, NY, USA, 2000. ISBN 0-521-77164-1.

[Lee06] Edward A. Lee. The problem with threads. Technical Report UCB/EECS-2006-1, EECS Department, University of California, Berkeley, Jan 2006. The published version of this paper is in IEEE Computer 39(5):33-42, May 2006.

[Lei05] Daan Leijen. Extensible records with scoped labels. In *Trends in Functional Programming*, pages 179–194, 2005.

[LHHW10] K. Leung, Z. Huang, Q. Huang, and P. Werstein. Data race: tame the beast. *The Journal of Supercomputing*, 51:258–278, 2010. ISSN 0920-8542. 10.1007/s11227-009-0370-x. Available from: `http://dx.doi.org/10.1007/s11227-009-0370-x`.

[Lim95] SGS-THOMSON Microelectronics Limited. occam 2.1 reference manual. Technical report, SGS, 1995.

[LM87] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235 – 1245, sept. 1987. ISSN 0018-9219. `doi:10.1109/PROC.1987.13876`.

[LM91] Xavier Leroy and Michel Mauny. Dynamics in ml. In *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*, pages 406–426. Springer-Verlag New York, Inc., New York, NY, USA, 1991. ISBN 0-387-54396-1.

[LO09] Kung-Kiu Lau and Mario Ornaghi. Control encapsulation: A calculus for exogenous composition of software components. In *Proceedings of the 12th International Symposium on Component-Based Software Engineering*, CBSE '09, pages 121–139. Springer-Verlag, Berlin, Heidelberg, 2009. ISBN 978-3-642-02413-9. Available from: `http://dx.doi.org/10.1007/978-3-642-02414-6_8`.

[LOQS07] Matthieu Leclercq, Ali Erdem Ozcan, Vivien Quema, and Jean-Bernard Stefani. Supporting heterogeneous architecture descriptions in an extensible toolset. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 209–219. IEEE Computer Society, Washington, DC, USA, 2007. ISBN 0-7695-2828-7. doi:http://dx.doi.org/10.1109/ICSE.2007.82.

[LP10] Mario Leyton and Jose M. Piquer. Skandium: Multi-core programming with algorithmic skeletons. *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, 0:289–296, 2010. ISSN 1066-6192. doi:http://doi.ieeecomputersociety.org/10.1109/PDP.2010.26.

[LSS08] Michael Lienhardt, Alan Schmitt, and Jean-Bernard Stefani. Typing communicating component assemblages. In *Proceedings of the 7th international conference on Generative programming and component engineering*, GPCE '08, pages 125–136. ACM, New York, NY, USA, 2008. ISBN 978-1-60558-267-2. Available from: http://doi.acm.org/10.1145/1449913.1449933.

[Lum07] Markus Lumpe. Applications = components + gloo. *Electron. Notes Theor. Comput. Sci.*, 182:123–138, June 2007. ISSN 1571-0661. doi:10.1016/j.entcs.2006.09.035.

[LW07] Kung-Kiu Lau and Zheng Wang. Software component models. *IEEE Trans. Softw. Eng.*, 33:709–724, October 2007. ISSN 0098-5589. doi:10.1109/TSE.2007.70726.

[MBH+02] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller, and Michael Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 06:4–16, 2002.

[McG82] James R. McGraw. The val language: Description and analysis. *ACM Trans. Program. Lang. Syst.*, 4:44–82, January 1982. ISSN 0164-0925. Available from: http://doi.acm.org/10.1145/357153.357157, doi:http://doi.acm.org/10.1145/357153.357157.

[McI68] M. Douglas McIlroy. Mass-produced software components. In J. M. Buxton, Peter Naur, and Brian Randell, editors, *Software Engineering Concepts and Techniques (1968 NATO Conference of Software Engineering)*, pages 88–98. NATO Science Committee, Oct. 1968.

[MDB04] Nasim Mahmood, Guosheng Deng, and James Browne. Compositional development of parallel programs. In Lawrence Rauchwerger, editor, *Languages and Compilers for Parallel Computing*, volume 2958 of *Lecture Notes in Computer Science*, pages 109–126. Springer Berlin / Heidelberg, 2004. ISBN 978-3-540-21199-0. Available from: http://dx.doi.org/10.1007/978-3-540-24644-2_8.

[MDK93] J. Magee, N. Dulay, and J. Kramer. Structuring parallel and distributed programs. *Software Engineering Journal*, 8(2):73 –82, mar 1993. ISSN 0268-6961.

[Mea55] George H Mealy. A method to synthesizing sequential circuits. *Bell Systems Technical Journal*, 34:1045âĂŞ1079, 1955.

[MF07] Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. *SIGPLAN Not.*, 42(1):3–10, 2007. ISSN 0362-1340.

[MH00] Greg Michaelson and Kevin Hammond. Hume: a functionally-inspired language for safety-critical systems. In *Draft proceedings from the 2nd Scottish Functional Programming Workshop (SFP00), University of St Andrews, Scotland, July 26th to 28th, 2000*, volume 2 of *Trends in Functional Programming*, 2000.

[Mil06] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[MK98] Manibrata Mukherji and Dennis Kafura. A process-calculus-based abstraction for coordinating multi-agent groups. *Theoretical Computer Science*, 192(2):287 – 314, 1998. ISSN 0304-3975. Available from: `http://www.sciencedirect.com/science/article/pii/S0304397597001539`, `doi:10.1016/S0304-3975(97)00153-9`.

[MKD93] J. Magee, J. Kramer, and N. Dulay. Darwin/mp: an environment for parallel and distributed programming. In *System Sciences, 1993, Proceeding of the Twenty-Sixth Hawaii International Conference on*, volume ii, pages 337 –346 vol.2, jan 1993. `doi:10.1109/HICSS.1993.284094`.

[MKJ08] Shin-Cheng Mu, Hsiang-Shang Ko, and Patrik Jansson. Algebra of programming using dependent types. In *Proceedings of the 9th international conference on Mathematics of Program Construction*, MPC '08, pages 268–283. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 978-3-540-70593-2.

[Moo56] E. F. Moore. Gedanken-experiments on sequential machines. *Annals of Mathematical Studies*, 34:129–153, 1956.

[Mor10] J. Paul Morrison. *Flow-Based Programming: A New Approach to Application*. CreateSpace, 2010.

[MPR06] Amy L. Murphy, Gian Pietro Picco, and Gruia-Catalin Roman. Lime: A coordination model and middleware supporting mobility of hosts and agents. *ACM Trans. Softw. Eng. Methodol.*, 15:279–328, July 2006. ISSN 1049-331X. `doi:http://doi.acm.org/10.1145/1151695.1151698`.

[MQB$^+$08] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 267–280. USENIX Association, Berkeley, CA, USA, 2008. Available from: `http://dl.acm.org/citation.cfm?id=1855741.1855760`.

[MR98] P.J. McCann and G.-C. Roman. Compositional programming abstractions for mobile computing. *Software Engineering, IEEE Transactions on*, 24(2):97 –110, feb 1998. ISSN 0098-5589. `doi:10.1109/32.666824`.

[MSA$^+$83] J. McGraw, S. Skedzielewski, S. Allan, D. Grit, R. Oldehoeft, J. Glauert, I. Dobes, and P. Hohensee. Sisal: streams and iteration in a single-assignment language. language reference manual, version 1. 1. Technical report, Lawrence Livermore National Lab., CA (USA), 1983.

[MTM97] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997. ISBN 0262631814.

[MVM09] Frederic P. Miller, Agnes F. Vandome, and John McBrewster. *AMD FireStream*. Alpha Press, 2009. ISBN 6130267754, 9786130267759.

[NA90] R. Nikhil and Arvind. Id: A language with implicit parallelism. CSG Memo 305, MIT Lab. for Computer Science, 1990.

[NB92] Peter Newton and James C. Browne. The code 2.0 graphical parallel programming language. In *Proceedings of the 6th international conference on Supercomputing*, ICS '92, pages 167–177. ACM, New York, NY, USA, 1992. ISBN 0-89791-485-6. `doi:http://doi.acm.org/10.1145/143369.143405`.

[NBGS08] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6:40–53, March 2008. ISSN 1542-7730. Available from: `http://doi.acm.org/10.1145/1365490.1365500`, `doi:http://doi.acm.org/10.1145/1365490.1365500`.

[NCG$^+$08] Ryan Newton, Michael Craig, Lewis Girod, Sam Madden, and Greg Morrisett. Wavescript: A case-study in applying a distributed stream-processing language. Technical report, MIT CSAIL, 2008.

[Nik89] R. S. Nikhil. Can dataflow subsume von neumann computing? *SIGARCH Comput. Archit. News*, 17:262–272, April 1989. ISSN 0163-5964. `doi:http://doi.acm.org/10.1145/74926.74955`.

[NMT10] Adrian Nistor, Darko Marinov, and Josep Torrellas. Instantcheck: Checking the determinism of parallel programs using on-the-fly incremental hashing. *Microarchitecture, IEEE/ACM International Symposium on*, 0:251–262, 2010. ISSN 1072-4451. doi:http://doi.ieeecomputersociety.org/10.1109/MICRO.2010.55.

[NPA86] R. Nikhil, Keshav Pingali, and Arvind. Id nouveau. CSG Memo 265, MIT Lab. for Computer Science, 1986.

[Obj08] Object Management Group. Common object request broker architecture (CORBA) specification 3.1, 2008. accessed Oct. 2011. Available from: http://www.omg.org/spec/CORBA/3.1/Interfaces/PDF.

[OH05] Kunle Olukotun and Lance Hammond. The future of microprocessors. *Queue*, 3:26–29, September 2005. ISSN 1542-7730. Available from: http://doi.acm.org/10.1145/1095408.1095418, doi:http://doi.acm.org/10.1145/1095408.1095418.

[Oho95] Atsushi Ohori. A polymorphic record calculus and its compilation. *ACM Trans. Program. Lang. Syst.*, 17(6):844–895, 1995. ISSN 0164-0925.

[OV11] Andrea Omicini and Mirko Viroli. Review: coordination models and languages: From parallel computing to self-organisation. *Knowl. Eng. Rev.*, 26:53–59, 2011. ISSN 0269-8889. doi:http://dx.doi.org/10.1017/S026988891000041X.

[PA98] George A. Papadopoulos and Farhad Arbab. Coordination models and languages. Technical report, Department of Software Engineering, Amsterdam, The Netherlands, The Netherlands, 1998.

[Pap90] G.M. Papadopoulos. Monsoon: a dataflow computing architecture suitable for intelligent control. In *Intelligent Control, 1990. Proceedings., 5th IEEE International Symposium on*, pages 292 –298 vol.1, sep 1990. doi:10.1109/ISIC.1990.128471.

[PBCV07] Louis-Noel Pouchet, Cedric Bastoul, Albert Cohen, and Nicolas Vasilache. Iterative optimization in the polyhedral model: Part i, one-dimensional time. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 144–156. IEEE Computer Society, Washington, DC, USA, 2007. ISBN 0-7695-2764-7. Available from: http://dx.doi.org/10.1109/CGO.2007.21, doi:http://dx.doi.org/10.1109/CGO.2007.21.

[PBG84] L. M. Patnaik, Prabal Bhattacharya, and R. Ganesh. Dfl: A data flow language. *Computer Languages*, 9(2):97 – 106, 1984. ISSN 0096-0551. doi:DOI:10.1016/0096-0551(84)90017-1.

[Pet61] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Fakultät für Mathematik und Physik, Technische Hochschule Darmstadt, Bonn, 1961.

[Pet77] James L. Peterson. Petri nets. *ACM Comput. Surv.*, 9:223–252, September 1977. ISSN 0360-0300. doi:http://doi.acm.org/10.1145/356698.356702.

[Pey03] Simon (editor) Peyton Jones. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):1–255, Jan 2003.

[PHG+10] Frank Penczek, Stephan Herhut, Clemens Grelck, Sven-Bodo Scholz, Alex Shafarenko, Remi Barrere, and Eric Lenormand. Parallel Signal Processing with S-Net. *Procedia Computer Science*, 1 (1):2079 – 2088, 2010. ISSN 1877-0509. ICCS 2010. Available from: http://www.sciencedirect.com/science/article/B9865-506HM1Y-88/2/87fcf1cee7899f0eeaadc90bd0d56cd3, doi:DOI:10.1016/j.procs.2010.04.233.

[PHS+10] Frank Penczek, Stephan Herhut, Sven-Bodo Scholz, Alex Shafarenko, JungSook Yang, Chun-Yi Chen, Nader Bagherzadeh, and Clemens Grelck. Message Driven Programming with S-Net: Methodology and Performance. *Parallel Processing Workshops, International Conference on*, 0:405–412, 2010. ISSN 1530-2016. doi:http://doi.ieeecomputersociety.org/10.1109/ICPPW.2010.61.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002. ISBN 0-262-16209-1.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Report DAIMI FN-19 DAIMI FN-19, Computer Science Department, Aarhus University, University of Aarhus, 1981.

[PMR99] G.P. Picco, A.L. Murphy, and G.-C. Roman. Lime: Linda meets mobility. In *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, pages 368 –377, may 1999. ISSN 0270-5257.

[Pro10] Daniel Prokesch. A light-weight parallel execution layer for shared-memory stream processing. Master's thesis, Technische Universität Wien, Vienna, Austria, Feb. 2010.

[Pur94] James M. Purtilo. The polylith software bus. *ACM Trans. Program. Lang. Syst.*, 16(1):151–174, 1994. ISSN 0164-0925.

[RE96] Matthias Radestock and Susan Eisenbach. Semantics of a higher-order coordination language. In Paolo Ciancarini and Chris Hankin, editors, *Coordination Languages and Models*, volume 1061 of *Lecture Notes in Computer Science*, pages 339–356. Springer Berlin / Heidelberg, 1996.

[RF93] B. Ramakrishna Rau and Joseph A. Fisher. Instruction-level parallel processing: history, overview, and perspective. *J. Supercomput.*, 7:9–50, May

1993. ISSN 0920-8542. Available from: `http://dx.doi.org/10.1007/BF01205181`, `doi:http://dx.doi.org/10.1007/BF01205181`.

[Ric84] H. Richards. An overview of arc sasl. *SIGPLAN Not.*, 19(10):40–45, 1984. ISSN 0362-1340.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965. ISSN 0004-5411.

[RR10] Thomas Rauber and Gudula Ruenger. *Parallel Programming: for Multicore and Cluster Systems*. Springer, 2010.

[RRMP08] Andreas Rausch, Ralf Reussner, Raffaela Mirandola, and Frantisek Plasil. *The Common Component Modeling Example: Comparing Software Component Models*. Springer Publishing Company, Incorporated, 1st edition, 2008. ISBN 3540852883, 9783540852889.

[Rus08] Bertrand Russell. Mathematical logic as based on the theory of types. *American Journal of Mathematics*, 30(3):222–262, 1908. ISSN 00029327.

[Rut64] J. D. Rutledge. On ianov's program schemata. *J. ACM*, 11:1–9, January 1964. ISSN 0004-5411. Available from: `http://doi.acm.org/10.1145/321203.321204`, `doi:http://doi.acm.org/10.1145/321203.321204`.

[RWH90] Benjamin C. Pierce Robert W. Harper. Extensible records without subsumption. Technical Report CMU-CS-90-102, Carnegie Mellon University, School of Computer Science, February 1990.

[Sca08] Matthew Scarpino. *Programming the Cell Processor: For Games, Graphics, and Computation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edition, 2008. ISBN 0136008860, 9780136008866.

[SCD+97] M. Schlansker, T.M. Conte, J. Dehnert, K. Ebcioglu, J.Z. Fang, and C.L. Thompson. Compilers for instruction-level parallelism. *Computer*, 30(12): 63 –69, dec 1997. ISSN 0018-9162. `doi:10.1109/2.642817`.

[Sch03] Sven-Bodo Scholz. Single Assignment C — efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 13(6):1005–1059, 2003.

[SGS10] John E. Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science and Engineering*, 12:66–73, 2010. ISSN 1521-9615. `doi:http://doi.ieeecomputersociety.org/10.1109/MCSE.2010.69`.

[SPSP87] Y. Shi, N. Prywes, B. Szymanski, and A. Pnueli. Very high level concurrent programming. *Software Engineering, IEEE Transactions on*, SE-13(9):1038 – 1046, sept. 1987. ISSN 0098-5589. `doi:10.1109/TSE.1987.233791`.

[Ste95] R. Stephens. Stream processing ii: an alternative algebraic approach and the language astral. Technical report, Submitted to Acta Informatica, 1995. Available from: `citeseer.ist.psu.edu/stephens95stream.html`.

[Ste97] Robert Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997. Available from: `citeseer.ist.psu.edu/stephens95survey.html`.

[Ste06] Thomas Sterling. The Gilgamesh MIND Processor-in-Memory Architecture for Petaflops-Scale Computing. In Hans Zima, Kazuki Joe, Mitsuhisa Sato, Yoshiki Seo, and Masaaki Shimasaki, editors, *High Performance Computing*, volume 2327 of *Lecture Notes in Computer Science*, pages 493–498. Springer Berlin / Heidelberg, 2006. ISBN 978-3-540-43674-4. Available from: `http://dx.doi.org/10.1007/3-540-47847-7_1`.

[THSS08] Jeyarajan Thiyagalingam, Philip Hölzenspies, Sven-Bodo Scholz, and Alex Shafarenko. A Stream-Order Relaxed Execution Model forAsynchronous Stream Languages. In Sven-Bodo Scholz, editor, *Implementation and Application of Functional Languages, 20th international symposium, IFL'08, Hatfield, Hertfordshire, UK*, Technical Report 474, pages 316–329. University of Hertfordshire, England, UK, 2008.

[TKA02] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *Computational Complexity*, pages 179–196, 2002. Available from: `citeseer.ist.psu.edu/thies01streamit.html`.

[TSR11] Carolyn Talcott, Marjan Sirjani, and Shangping Ren. Comparing three coordination models: Reo, arc, and pbrd. *Science of Computer Programming*, 76(1):3–22, 2011. ISSN 0167-6423. Available from: `http://www.sciencedirect.com/science/article/pii/S016764230900166X`, `doi:10.1016/j.scico.2009.11.006`.

[Tur83] D. A. Turner. Sasl language manual. Technical report, Computer Laboratory, University of Kent, Canterbury, 1983.

[UM03] Jop Sibeyn Ulrich Meyer, Peter Sanders. *Algorithms for Memory Hierarchies*, volume 2625 of *Lecture Notes in Computer Science*. Springer Verlag, 2003.

[vdS06] Aad J. van der Steen. Issues in computational frameworks. *Concurrency and Computation: Practice and Experience*, 18(2):141–150, 2006. ISSN 1532-0634. Available from: `http://dx.doi.org/10.1002/cpe.908`.

[VGvT$^+$11] Merijn Verstraaten, Clemens Grelck, Michiel W. van Tol, Roy Bakker, and Chris R. Jesshope. On mapping distributed S-NET to the 48-core intel SCC processor. In *3rd Many-core Applications Research Community (MARC) Symposium. (KIT Scientific Reports 7598) `http://digbib.ubka.`*

uni-karlsruhe. de/ volltexte/ 1000023937, pages 41–46. KIT Scientific Publishing, 2011.

[Vo11] Anh Vo. *Scalable formal dynamic verification of MPI programs through distributed causlity tracking*. PhD thesis, School of Computing, The University of Utah, August 2011.

[Wan87] Mitchell Wand. Complete type inference for simple objects. In *Logic in Computer Science*, pages 37–44, 1987.

[Wan88] Mitchell Wand. Corrigendum: Complete type inference for simple objects. In *LICS*, page 132, 1988.

[WBS^{+}92] John Werth, James C. Browne, S. Sobek, Taejae Lee, Peter Newton, and Ravi Jain. The interaction of the formal and the practical in parallel programming environment development: Code. In *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*, pages 35–49. Springer-Verlag, London, UK, 1992. ISBN 3-540-55422-X. Available from: `http://dl.acm.org/citation.cfm?id=645669.665215`.

[WG82] I. Watson and J. Gurd. A practical data flow computer. *Computer*, 15(2): 51 – 57, feb 1982. ISSN 0018-9162. `doi:10.1109/MC.1982.1653941`.

[Whi80] T. Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, 1980.

[WHW^{+}11] Michael Wilde, Mihael Hategan, Justin M. Wozniak, Ben Clifford, Daniel S. Katz, and Ian Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9):633 – 652, 2011. ISSN 0167-8191. Available from: `http://www.sciencedirect.com/science/article/pii/S0167819111000524`, `doi:10.1016/j.parco.2011.05.005`.

[Wir71] Niklaus Wirth. The programming language pascal. *Acta Inf.*, 1:35–63, 1971.

[WP94] P.G. Whiting and R.S.V. Pascoe. A history of data-flow languages. *Annals of the History of Computing, IEEE*, 16(4):38–59, Winter 1994. ISSN 1058-6180.

[Xia10] H. Xiao. Towards parallel and distributed computing in large-scale data mining: A survey. Technical report, Technical University of Munich, 2010.

[YAM^{+}11] Fan Yang, Tomoyuki Aotani, Hidehiko Masuhara, Flemming Nielson, and Hanne Riis Nielson. Combining static analysis and runtime checking in security aspects for distributed tuple spaces. In *COORDINATION*, pages 202–218, 2011.