School of Computer Science
University of Hertfordshire

Clemens Grelck, Alex Shafarenko (Eds.):

S-Net Language Report

Version 1.0

Frank Penczek, Clemens Grelck, Haoxuan Cai, Jukka Julku, Philip Hölzenspies, Sven-Bodo Scholz, Alex Shafarenko

Technical Report No. 487
June 15, 2009

Published by the School of Computer Science
University of Hertfordshire
Hatfield, AL10 9AB, United Kingdom

Preface

S-Net is a novel coordination language and component technology for the era of multi-core and many-core computing. It turns sequential legacy code written in conventional languages into asynchronous components that interact with each other via a stream-processing network. The specification of these networks, a core feature of S-Net, follows an algebraic approach: only four different network combinators allow us the concise specification of complex streaming networks through a simple expression language. Routing of data packages is defined via a record type system with structural subtyping and (flow) inheritance.

The development of S-Net, both in terms of theory and implementation, has been funded by the European Union through the Integrated Project Æther (Self-adaptive Embedded Technologies for Pervasive Computing Architectures), where S-Net was adopted as the main contribution of the subproject on software architectures.

While the development of S-Net has been led by the Compiler Technology and Computer Architecture Group (CTCA) in the University of Hertfordshire's School of Computer Science, other partners both from within the Æther project as well as from outside have made substantial contributions to S-Net in general and to this report in particular: Imperial College London, United, Kingdom, VTT Research Center Helsinki, Finland, the University of Twente, Netherlands, and the University of Lübeck, Germany.

After almost three years of active development, the project and this language report have reached a level of maturity that justifies the 1.0 version number. Nevertheless, the S-Net language report remains a document in progress: we will add further chapters and sections on novel language features or additional examples as well as clarifications and useful changes to existing features as time goes by.

The S-Net language report presents S-Net from an external perspective. It is the document of choice to anyone who wants to learn more about S-Net and eventually become proficient in S-Net: from novice to expert, from informal descriptions to formal semantics to application examples. However, the language report does not make any statements on (potential) implementations of S-Net on different target hardware architectures. For this internal perspective on S-Net we refer to the S-Net Implementation Report.

We take this opportunity to thank all those who have contributed to the S-Net project in whatever way: implementation, documentation, applications or just intense discussions on any of these issues.

Hatfield, United Kingdom
June 2009

Clemens Grelck
Alex Shafarenko

Document History

New in revision 1.0 of June 15, 2009

- New preface
- Reorganisation of editorships and authorships
- Author portraits
- New title page with official technical report status
- Improved layout
- New flow inheritance rule for synchrocells explained and motivated
- Support for inlined box language code removed, feature to be realised through metadata
- Initialiser boxes added (boxes with no input type)
- Boxes without output type added
- Broadcast index split combinator (dual tag) removed
- Metadata namespace changed to snet-home.org
- Chapter on interfaces completely rewritten
- New section on network interfaces
- New sections on C and SAC interface implementations
- New chapter on runtime inspection and observers
- Outdated chapter on semantics and type inference removed
- New chapter on formal operational semantics
- New chapter on denotational semantics
- New chapter on type inference
- New chapter on language extensions for reconfiguration and self-adaptivity
- Formal operational semantics of proposed new combinators for reconfiguration and self-adaptivity

New in revision 0.7 of June 03, 2008

- New syntax for underspecified net signature that only provide an input type
- Explained introduction of name spaces through annotation of net signatures
- Combinator associativity clarified

New in revision 0.6 of December 17, 2007

- New chapter on metadata
- New syntax for filter boxes

New in revision 0.5 of June 27, 2007

- This document history
- Introductory parts rewritten to meet ongoing language design process
- Type signature definitions
- Clear separation between types and type signatures
- Synchrocell extended by guarded pattern
- Serial replication combinator extended by guarded termination pattern
- Filter extended by guarded pattern match and expression language
- Clarification of associativity and priority rules for network combinators
- Disambiguation of deterministic and non-deterministic combinators
- Multi-file S-Net programs
- New graphical representation of serial and parallel replication combinators
- New example: factorial from an imperative perspective
- New example: implementing Sudokus with S-Net and SaC

New in revision 0.4 of November 28, 2006

- Mostly minor formatting improvements and clarifications

New in revision 0.3 of October 18, 2006

- New syntax for types
- New syntax for binding tags
- More illustrating examples of type language
- Discussion of S-Net subtyping vs object-oriented programming
- Primitive boxes `driver`, `plug` and `link` abandoned
- Synchrocell redesigned
- New primitive box `filter` for housekeeping tasks
- All network operators (`pass`, `strip`, `set`) abandoned
- Example: computing factorial numbers with S-Net
- Complete syntax of S-Net in appendix

New in revision 0.2 of August 25, 2006

- New layout of document
- Concept of field types and homomorphic subtyping on fields temporarily abandoned and deferred to future Part II of the S-Net report

Revision 0.1 of June 13, 2006

- First version of report made public within the Æther project consortium

Contents

Chapter 1

Introduction

1.1 S-Net as a Component Technology

This document provides a comprehensive introduction to S-Net. S-Net is a novel coordination language that orchestrates asynchronous components that communicate with each other and their execution environment solely via typed streams.

The concept of coordination language arises wherever an application has to be presented as a set of concurrent communicating activities, each defined in application-specific terms as a meaningful program unit, while all together representing a concurrently executing, parallel (and potentially distributed) application. The application program units are presented in an appropriate fully-fledged programming language, such as C, Java, etc., while the aspects of communication, concurrency and synchronisation (referred to by the term *coordination*) are captured by a separate, coordination, language. The whole idea of coordination hinges on the principle that the integration between the coordination and application languages is loose: coordination constructs have little access, if at all, to the facilities of the application program.

A complete separation between computation and coordination language is always desirable, but rarely achieved in practice. Nevertheless, there must be a rigorously defined contract between them. The usefulness of the coordination language comes from the fact that coordination minimally disturbs the application code. In our approach, which is rather extreme in this sense, the application program units merely use a special output function (which is in fact part of the coordination/application interface) instead of a standard function return, and even that is additional to simply using those units as is, whenever the application language is reach enough for aggregated return values (e.g., a list of records). Another great advantage of coordination is that the programmer responsible for concurrency could be a system integrator without specialist algorithmic knowledge in the application area. This obviously provides for the wider adoption of distributed and parallel computing in practical software engineering.

1.2 S-Net and Stream Processing

The approach taken by S-Net is targeted at stream processing. This is a well-established area, which, at the time when distributed computing, multimedia and signal processing permeate the computing and telecommunication sectors, is very important. This paper focuses on asynchronous stream processing, which on the one hand, enables the philosophy of data-flow synchronisation developed in the 1980s to be taken on board (thanks to the coordination aspect, which assumes course granularity), whilst on the other hand, develop a whole host of analysis techniques thanks to

the regular nature of stream communication (as opposed to general message-passing). As a result S-Net is a very compact and powerful coordination language. It reflects the modern notions of subtyping, encapsulation and inheritance, while completely separating all communication and concurrency concerns from the application code.

The concept of stream processing has a long history. The view of a program as a set of processing blocks connected by a static network of channels goes back at least as far as Kahn's seminal work [1] and the language Lucid [2]. Kahn introduced the model of infinite-capacity, deterministic process network and proved that it had properties useful for parallel processing. Lucid was apparently the first language to introduce the basic idea of a block that transforms input sequences into output ones. A variable would represent such a sequence, acting as a stream of values of that variable in time. Ordinary operators in Lucid acted on variables point-wise, by effectively synchronising streams and applying the operation across pairs of corresponding stream elements. Additionally there were also some "temporal" operators, which were intended for altering the order of elements in a sequence.

Somewhat later, in the 1980s, a whole host of synchronous dataflow languages sprouted, notably the languages Lustre [3] and Esterel[4], which introduced explicit recurrence relations over streams and further developed the concept of synchronous networks. These languages are still being used for programming reactive systems and signal processing algorithms today, including industrial applications such as the recent Airbus flight control system and various other aerospace applications [5]. The authors of Lustre broadened their work towards what they termed synchronous Kahn's networks[6, 7], i.e functional programs where the connection between functions, although expressed as lists, is in fact 'listless': as soon as a list element is produced, the consumer of the list is ready to process it, so that there is no queue and no memory management is required.

A nonfunctional interpretation of Kahn's networks is also receiving attention, the latest stream processing language of this category being, to the best of our knowledge, the MIT's StreamIt [8]. The latest comprehensive survey of stream processing and the underlying theory for it can be found in [9]. There is also a growing activity in *database stream processing* [10] , which concerns itself with the problem of responding to a database query "on the fly", using the same limited-memory, sliding-window view of processing blocks that started with Lucid and continued through the aforementioned stream-processing languages. Still, despite much work having been done in various niche areas, stream processing has yet to be recognised as a general-purpose paradigm in the same sense as, for instance, object-oriented or functional programming.

Around the time that Lustre was introduced, David Turner[11] remarked that streams could be used as software glue for complex parallel software systems, even operating systems. In his interpretation, streams were lazy lists, which were produced on demand for their consumers. The lists were seen as an interface between the deterministic parts of a parallel system, which were pure stream-processing functions (indeed they could have been any self-contained procedures as long as the only access they had to each other's state was via stream communication), and the external interleavers/mergers that realise the inter-process communication and capture its nondeterministic behaviour.

This arrangement is sketched in fig 1.1. Note that each processing box has a single input and a single output. This does not lead to a loss of generality due to the fact that a function requiring multiple input streams can be represented as a function of a single stream argument where the elements of the multiple streams are somehow merged into a single sequence of records. Similarly, a single output stream can be split into any given number of secondary output streams by picking out records for each of the output sequences. The issue of how exactly the inputs are merged is a delicate one; an efficient solution would depend on the properties of the function in question. The merging usually benefits from being nondeterministic, as this accommodates the delays incurred in receiving the contributing streams by allowing the first message that arrives to be passed on to

the processing function without waiting for its turn. On the other hand, the processing block can be required to be deterministic, in which case it may not be ready to accept a given input at an arbitrary time.

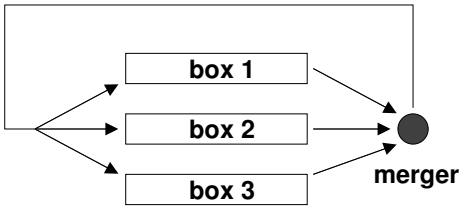

Figure 1.1: The Turner scheme

Note that a merged stream has no overall order: only records belonging to a single tributary stream have a precedence relation defined on them. To allow that order to be recovered from the merged stream, the provenance information can be preserved by, for example, tagging the ordered records by the same tag.

Overall, the Turner scheme seems very attractive as it neatly separates the computational aspect of stream processing from the communication aspect; it confines non-determinism to the part of the system where no value processing takes place (since merging, filtering and splitting only re-package streams without computing new values of basic types); and it uniformly represents an application as a set of interconnected, side-effect-free, single-input, single-output stream functions. The only quality that it seems to lack is satisfactory support for modularity. The problem is that streams in complex systems tend to be record-based, and the processing functions expect a certain set of fields to be present in the records. Moreover, rather than streams having a single record layout, variant records are often required, so that a number of different algorithms can be carried out by a single block. In addition, certain "control" records can be used for exception handling, load balancing, etc. The boxes can be usefully *extended* by adding more variants and passing the unused fields downstream to further, perhaps newly inserted, boxes which provide additional functionality. Those are examples of network structuring, subtyping and inheritance that one would expect to find in a practical stream-processing paradigm.

Besides these pragmatic considerations, we must mention here equally important theoretical advances in streaming networks. The key work in this area appears to have been done by Stefanescu, who has developed several semantic models for streaming networks starting from flowcharts [12] and recently including models for nondeterministic stream processing developed collaboratively with Broy [13]. This work aims to provide an algebraic language for denotational semantics of stream processing and as such is not focused on pragmatic issues. It nevertheless offers important structuring primitives, which are used as the basis for a network algebra (see [14]). It is interesting to note that apparently the StreamIt team [8] as well as ourselves [15] were unaware of those and had to re-invent them, albeit for purely pragmatic reasons.

1.3 S-Net at a Glance

As mentioned earlier, S-Net is a coordination language that orchestrates asynchronous components interconnected by typed streams. Atomic components, named *boxes* in S-Net terminology, are connected to their environment by exactly one input and one output stream. Boxes communicate with each other and with the execution environment solely by means of data received from the input stream and data sent to the output stream.

Boxes are entirely stateless and strictly operate in a input-process-output work cycle. Upon receiving a data item on its input stream a box produces none, one or more data items on its output stream. The functional properties of S-Net boxes enable them to be be deployed cheaply and moved and replicated at will, without giving rise to data integrity concerns.

We deliberately restrict S-Net to coordination aspects. Boxes are implemented externally using an appropriate *box language*. Functional languages are particularly suitable for this purpose as they inherently adhere to the restrictions imposed by the interface. Nevertheless, imperative box languages may be used as well, but require some discipline by the programmer. S-Net boxes are in fact "black boxes". Being implemented in a different language they do not expose any internal content to the S-Net level. In principle, a single S-Net may combine boxes implemented using different box languages and, thus, also provides a structured approach towards multi-language programming.

S-Net defines streaming networks of asynchronous components inductively through algebraic formulae rather than error-prone port specifications and wire lists. We have identified only four essential composition techniques for SISO components: serial and parallel composition of different components as well as the serial and the parallel replication of an individual component. Each composition technique is expressed by a dedicated *network combinator* that takes one or two SISO operand components and produces a new (compound) SISO component. Our use of algebraic formulae for describing streaming networks is inspired by Stefanescu's network algebra [13].

S-Net networks are generally asynchronous: A component's output is sent to the input buffer of the recipient component. These buffers are bounded; their size determines the degree of asynchrony between components. Whenever the output of several components needs to be combined, we require some kind of synchronisation facility. It is introduced in the form of a SISO *synchrocell*, which is the only kind of "stateful" box in an S-Net. A synchrocell expects records of several types to appear at its input; it combines them into a joint record and outputs the result. The internal state of a synchrocell is made up by the records waiting to be synchronised. Note that synchrocells, though "stateful", have no computation to perform, whereas boxes have no state, but can compute. This solution also clearly separates two memory aspects which are usually combined in conventional programming: memory as work storage for computations and memory as a means of inter-process communication. It is this separation that promotes flexible specification, which assists generic parallel and distributed computing.

Boxes communicate with each other by sending data items over streams. These data items are organised as non-recursive, tagged records with arbitrary non-record fields. Types associated with streams in an S-Net network are non-recursive, tagged variant record types. Like the function that actually implements a box elementary types are effectively opaque to S-Net. Since all actual data is produced and consumed by box language programs, only the box language code can interpret the data.

We use (record) subtyping as an important adaptation mechanism within our component technology. It allows component designers to provide several versions of a box with different (sub-)types. An S-Net box may be capable of processing more record variants than there are in the incoming typed stream. Likewise, boxes accept records that have additional fields in excess of those required by and known to the box. Structural subtyping on records provides the formal foundation for this operational behaviour.

Excess fields are stripped off a record before a box starts processing it. However, such fields are not discarded, but rather attached to each record that the box produces in response. We call this behaviour *flow inheritance*. It is fundamental to S-Net as an adaptation technology for components that were developed in isolation. Whereas structural record subtyping is well known, the concept of flow inheritance, to the best of our knowledge, is a new concept. Crucially, S-Net does not require explicit subtype declarations, but applies type inference instead.

Chapter 2

Types and Subtyping

2.1 Record Types

The type system of S-Net supports non-recursive variant records with *record subtyping*. As defined syntactically in Fig. 2.1, a *type* in S-Net is a non-empty set of anonymous *record variants* separated by vertical bars. Each record variant is a possibly empty set of named *record entries*, enclosed in curly brackets.

Type	⇒	*RecordType* ⌈ **|** *Type* ⌋
	|	*TypeName* ⌈ **|** *Type* ⌋
RecordType	⇒	**{** ⌈*RecordEntry* ⌈ **,** *RecordEntry* ⌋* ⌋ **}**
RecordEntry	⇒	*Field* | *Tag*
Field	⇒	*FieldName*
Tag	⇒	*SimpleTag* | *BindingTag*
SimpleTag	⇒	**<** *SimpleTagName* **>**
BindingTag	⇒	**< #** *BindingTagName* **>**
TypeDef	⇒	**type** *TypeName* **=** *Type* **;**

Figure 2.1: Syntax definition of S-Net types and type definitions. The non-terminal symbols *TypeName*, *FieldName* and *TagName* uniformly refer to identifiers. We only distinguish them here for the purpose of illustration.

We introduce two kinds of record entry: *fields* and *tags*. A field is characterised by its *field name*; at runtime it is associated with a value, but the value is opaque to S-Net and may only be generated, inspected or manipulated by a box (which is written in a box language). A tag is also represented by its name, which is enclosed in angular brackets. However, at runtime the name is associated with an integer value that is visible to both the box language code and S-Net. The intention of tags is to control the flow of records through a network, rather than to serve as ordinary containers of integer values. Furthermore, we distinguish between *simple tags* and *binding*

tags, which are distinguished by the lexical marker #. Binding tags have a different behaviour under subtyping, as explained in Section 2.2.

We illustrate S-Net types by a simple example from 2-dimensional geometry. A rectangle can be represented by the S-Net type

```
{x, y, dx, dy}
```

that contains its coordinates and dimensions. Likewise, we can represent a circle by the centre point coordinates and its radius:

```
{x, y, radius}
```

Using the S-Net support for variant records we can now define a common type for plain figures, which consists of rectangles and circles:

```
{x, y, dx, dy} | {x, y, radius}
```

It is often convenient to use tags to identify anonymous variants, for instance:

```
{<rectangle>, x, y, dx, dy} | {<circle>, x, y, radius}
```

We refer to types that consist of a single variant as *record types* because a record at runtime has a clear variant choice.

S-Net also supports non-recursive abstractions on types. Using the key word `type` an identifier may be bound to a type specification. Such an identifier may afterwards be used instead of a complete type specification in any syntactical position that requires a type. For example, we may first define type abstractions for rectangles and circles:

```
type rectangle = {<rectangle>, x, y, dx, dy};
type circle    = {<circle>, x, y, r};
```

and use them later on to define the more general type `figure` to represent plane figures of either kind:

```
type figure = rectangle | circle;
```

Type abstractions in S-Net are purely syntactical: Given the above definitions, the types

```
body
```

```
rectangle | circle
```

and

```
{<rectangle>, x, y, dx, dy} | {<circle>, x, y, r}
```

are identical.

S-Net does not support recursive record types for a good reason. A non-recursive record is a mere collection of fields accessible at once, in no particular order. The fields may themselves be records, so in fact a non recursive record can be thought of as a finite tree, whose leaves are named by the (unique) path from root to leaf. These path names are static and, hence, all leaves are accessible at once in any order. Since there are no operations on whole records in S-Net, the subrecord structure is not important and may easily be resolved in a preprocessing step.

By contrast, a recursive record type can be thought of as a set of non-recursive records in which some fields represent cross references, and where each record has a special statically unknown label for use in cross referencing. The data structure as a whole is characterised by (partial) access order, so it cannot be accessed at once, but rather one group of (non-recursive) records at a time by following references. When a recursive data structure is to be communicated, it is common to stream only relevant parts of it, under the control of a client-server protocol, rather than the whole data structure at once. Even when the latter is unavoidable, such data structures are not send in their natural form, but rather in a serialised, 'marshalled' stream, which is a stream of non-recursive records with abstract label fields.

This is not to say that S-Net would be unable to process lists, trees and other inductively defined data structures. As long as the box language provides the necessary support, such data may be associated with a single S-Net record entry. The marshalling and unmarshalling will need to be supported by the box language as well, as part of its interface with S-Net.

2.2 Record Subtyping

As mentioned earlier, S-Net supports subtyping on record types. Record subtyping is based essentially on the subset relationship between collections of record fields.

Definition 2.1 (record subtyping) *Let $BT(x)$ denote the set of binding tags in a record type x. Record subtyping is defined by the following rules:*

1. *A record type r_1 is a subtype of a record type r_2, $r_1 \sqsubseteq r_2$, if*

$$r_1 \supseteq r_2 \wedge BT(r_1) = BT(r_2)\,.$$

2. *A type t_1 is a subtype of a type t_2, $t_1 \sqsubseteq t_2$, if*

$$(\forall r_1 \in t_1 \exists r_2 \in t_2) r_1 \sqsubseteq r_2\,.$$

Informally, one type is a subtype of another type if it has additional record entries in the variants or fewer variants. For example, the type

```
{<circle>, x, y, radius, colour}
```

representing coloured circles is a subtype of the previously defined type `circle`. Likewise, we may add another type to represent triangles:

```
type triangle = {<triangle>, x, y, dx1, dy1, dx2, dy2};
```

Now, the combined type

```
type figure2 = triangle | rectangle | circle
```

is a supertype of the previously defined type `figure`.

Our definition of record subtyping coincides with the intuitive understanding that a subtype is more specific than its supertype(s) while a supertype is more general than its subtype(s). In the first example, the subtype contains additional information concerning the geometric body (i.e. its colour) that allows us to distinguish for instance green circles from blue circles, whereas the more general supertype identifies all circles regardless of their colour. In the second example, the supertype `figure2` is again more general than its subtype `figure` as it encompasses all three plane figures. In turn, `figure` is more specific than `figure2` in ruling out triangles from the set of plane figures covered.

With the definition of record subtyping, the purpose of binding tags becomes apparent: They provide a means to exercise explicit control over record subtyping. One record type is a subtype of another one only if the two have the same set of binding tags. In contrast, non-binding tags behave just like record fields with respect to record subtyping. For instance, with the above definition of type `circle` using a simple tag `<circle>` for identification the following type

```
type position = {x, y};
```

would be a supertype of `circle` as it contains fewer record entries. This, however, is less intuitive. We would rather like to see the position being a part of the definition of the geometric body circle than a circle being a specific position. Changing our definitions of types representing individual plane shapes to

```
type rectangle = {<#rectangle>, x1, y1, dx2, dy2};
type circle    = {<#circle>, x1, y1, r};
type triangle  = {<#triangle>, x, y, dx1, dy1, dx2, dy2};
type figure3     = triangle | rectangle | circle
```

using binding tags prevents this and helps to create a better hierarchy.

Unlike many object-oriented languages like C++ or Java our definition of record subtyping allows any type to have multiple supertypes (which are not subtypes of one another). When binding tags are not used, the type {}, i.e. the empty record, is the greatest supertype of all record types. Otherwise, for each set of binding tags B, B itself is the greatest supertype of any types τ for which $BT(\tau) = B$.

2.3 Type Signatures

Now, we are ready to define the concept of a *type signature*, i.e. the type associated with an S-Net box. As defined in Fig. 2.2, an S-Net type signature is a non-empty set of type mappings each relating an *input type* to an *output type*. The input type specifies the records a box accepts for processing; the output type characterises the records that the box may produce as as response. As the box may choose not to produce any records when receiving records of certain input types, the output type of type mappings is optional.

TypeSignature ⇒ *TypeMapping* [, *TypeSignature*]*
| *TypeSigName* [, *TypeSignature*]

TypeMapping ⇒ *Type* -> *Type*

TypeSigDef ⇒ **typesig** *TypeSigName* = *TypeSignature* ;

Figure 2.2: Grammar for S-Net type signatures

An input type that consists of multiple variants is nothing but syntactic sugar for a set of type mappings each relating one of the variants to the common output type. For example, the type signature

```
{a,b} | {c,d} -> {x} | {y}
```

is equivalent to the type signature

```
{a,b} -> {x} | {y},
{c,d} -> {x} | {y}
```

Therefore, we assume (single variant) record types as input types from here on, we call these type signatures *normalised*. A multi-variant output type means that a box may produce any of the records specified in response to receiving an input record that fits the associated input type. However, it is important here to note that S-Net boxes may produce as many output records in response to a single input record as they like, including none at all. Multiple output records may follow the same output variant or be all different from each other.

In analogy to types, S-Net supports abstractions on type signatures using the key word `typesig` (cf. Fig. 2.2). Explicit type mappings and predefined type signature symbols may freely be mixed with each other.

Despite the representation of type signatures as sets of type mappings, we define the (global) input type of the type signature to be the union of input types of all mappings. Likewise, we define the (global) output type to be the union of the output types of all mappings.

2.4 Type Coercion

The introduction of type signatures in the previous section raises the issue of *type coercion*. Informally, we are concerned with the question of which type mapping to choose for a given type of incoming records in order to determine the potential type of records output in response. An S-Net box accepts any record whose type is a subtype of its type signature's input type. In general, this requires an up-coercion to the most appropriate supertype.

As an example, let us assume the input type of our type signature to be the type `body3`, as defined in Section 2.2 using binding tags. The necessary up-coercion of a record type

```
{<#circle>, x, y, radius, colour}
```

of coloured circles is simply done by eliminating the additional colour field. We always coerce to the least common supertype. In other words, we aim at disposing of as few record entries as possible. If we would enrich the input type `body3` by an additional variant for coloured circles as above, we would choose that more specific mapping for coloured circles rather than the less specific for circles in general, although both would fit with respect to subtyping.

However, unlike in single-inheritance object-oriented languages up-coercion may be ambiguous. Consider

```
{x, y} | {dx, dy}
```

as another example of an input type. An incoming record of type `rectangle`, as defined in Section 2.1 (without binding tags), would match both variants equally well. Only some targets for coercion can cause such ambiguities; the following definition introduces a uniqueness condition for type coercions:

Definition 2.2 (complete record type) *A record type τ is called complete iff*

$$\forall v, w \in \tau : BT(v) = BT(w) \implies v \cup w \in \tau \,.$$

As in the definition of record subtyping in Section 2.2, $BT(x)$ denotes the set of binding tags of a type x. For any pair of variants with the same set of binding tags a complete record type must have a third variant combining their fields. Consequently, (non-variant) record types are automatically complete.

In order to disambiguate coercion we require type signatures to have complete input types.

2.5 Flow Inheritance

Streaming networks promote pipelining whereby a record travels along a chain of boxes that apply various processing algorithms to its content. Since a box can legally be fed with a subtype of the input type, this would result in the loss of all fields that are not required by the input type, but these fields could possibly be required by another box further down the pipeline. For example, we may have a box that manipulates the position of a geometric body regardless of its type. The associated type signature could be just `({x,y}->{x,y})`. Using simple tags instead of binding tags for variant identification, this box would accept circles, rectangles and triangles focussing on their common data (i.e. the position) and ignoring their specific record entries.

Unfortunately, such a box would be completely useless because following the necessary up-coercion to type `{x,y}` we lose all specific information on the geometric bodies. What is intended to be a pure position manipulation effectively destroys the records making proper processing further down the stream effectively impossible. To remedy this misbehaviour, we introduce the following type rule that complements the up-coercion with an automatic down-coercion.

Definition 2.3 (flow inheritance) *Let $v^{[i]} \rightarrow \tau^{[i]}$, $i \in [1, \ldots, n]$, 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 subtyped by flow inheritance *to the type $X' : V^{[i]} \rightarrow T^{[i]}$, where*

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

and

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

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

Informally, an input variant can be extended with a new field or non-binding tag (but not binding tag) ϕ, if it does not clash with any other variant. The output type associated with this input variant is extended with the field named ϕ in each of its variants unless it is present there already. Any number of flow inheritance extensions can be applied to a box, resulting in several fields being added. Value-wise, the extension is in terms of copying the value of the input record field ϕ over to the output record field with the same name. (Obviously, an implementation is free to simply switch references.) If the output already contains an identically named field, then that field's value supersedes the inherited one. For convenience, we shall write box signatures in the form $(n, m)v^{[i]} \rightarrow w_j^{[i]}$, which signifies a box with input variants $v^{[i]}$ and the corresponding output types $\tau^{[i]} = \{w_1^{[i]}, \ldots, w_{m_i}^{[i]}\}$, $i \in \{1, \ldots, n\}$. Note that n is a scalar integer that describes the number of input variants, whereas m denotes a vector of n integers describing the number of variants in each output type associated with one input variant.

Flow inheritance creates a subtyping hierarchy for boxes. For example, a box that accepts records with a single field named x and which produces records with a single field name y is a supertype of a box that accepts $\{x, z\}$ and returns $\{y, z\}$. As a side effect, flow inheritance can be a source of redundancy in type signatures. Indeed, in the above example if the signature of the *same* box contains the rules $\{x\} \rightarrow \{y\}$ and $\{x, a\} \rightarrow \{y, a\}$, then clearly the second rule can be deleted without changing the effective box type. Value-wise, the second rule carries additional information, namely that a record $\{x, a\}$ if presented to the input, will cause a record $\{y, a\}$ to appear with a potentially *different value* of a, while, assuming that b does not occur anywhere in the signature, if $\{x, b\}$ is presented at the input it would cause the output of $\{y, b\}$ with the output value of b being exactly the same as its input value. Still, as far as types are concerned, we can always assume that the signature is non-redundant, since the redundant rules change nothing in the type transformation defined by it.

2.6 Box Subtyping

Other forms of subtyping come from the conventional subtyping rules for a function:

$$\frac{f : \tau_1 \rightarrow \tau_2,\ \tau_1 \sqsubseteq \tau_1'\ \tau_2' \sqsubseteq \tau_2}{f : \tau_1' \rightarrow \tau_2'}$$

and our concept of records that allows a subtype to have fewer variants and more fields in each variant. Accordingly, we state four subtyping rules. The rules may violate the topological order of

the left-hand sides as fields and variants are inserted at arbitrary positions. To restore the order we use the topological permutation T_S defined on any set of variants S=$\{v_i\}$ as a permutation of the index range $[1, |S|]$ such that $T_S(j)$ enumerates the indices of the variants $v_{T_S(j)}$ in topological order as j traverses the index range in ascending order. Here is the summary of the subtyping rules:

Definition 2.4 (box subtyping) *Let box X have the type signature $(n, m)v^{[i]} \to w_j^{[i]}$. Then for any $k \le n$, the following are subtypes of X:*

input field: *the type $(n, m)V^{[T_V(i)]} \to W_j^{[T_V(i)]}$, where for some field name $\phi \in v^{[k]}$*

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

provided that $\phi \neq v^{[k]} \setminus v^{[l]}$ for all $l > k$; otherwise, for any $l > k$ such that $\phi = v^{[k]} \setminus v^{[l]}$

$$V^{[i]} = \begin{cases} v^{[i]} & if\, i \neq k \wedge i < l, \\ v^{[k]} \setminus \{\phi\} & if\, i = k, \\ v^{[i-1]} & if\, i \ge l \end{cases}, \; W_j^{[i]} = \begin{cases} w_j^{[i]} & if\, i \neq k \wedge i < l, \\ w_j^{[k]} \cup w_j^{[l]} & if\, i = k, \\ w_j^{[i-1]} & if\, i \ge l \end{cases},$$

input variant: *for any variant $\pi \notin \{v^{[i]}\}$, the type $(n+1, M)V^{[i]} \to W_j^{[i]}$, where*

$$V^{[i]} = \begin{cases} v^{[i]} & if\, i < k, \\ v^{[i-1]} & if\, i > k, \\ \pi & otherwise \end{cases},$$

$$W_j^{[i]} = \begin{cases} w_j^{[i]} & if\, i < k, \\ w_j^{[i-1]} & if\, i > k, \\ \tau & otherwise \end{cases},$$

provided that $(\forall i > k)v^{[i]} \not\sqsubseteq \pi$ and τ is such that for all i for which $\pi \sqsubseteq v^{[i]}$, the relation $\tau \sqsubseteq \{w_j^{[i]} \cup (\pi \setminus v^{[i]})\}$ holds as well. Note that this rule accounts for a newly introduced variant having extra fields over an existing one, so that these fields would have been flow inherited given the original type. Here

$$M^{[i]} = \begin{cases} m^{[i]} & if\, i < k, \\ m^{[i-1]} & if\, i > k, \\ \mu & otherwise \end{cases},$$

and μ is the number of variants in τ.

output field: $(n, m)v^{[i]} \to W_j^{[i]}$, *where for all $j \le n$, $r \le m^{[j]}$, some $l \le m^{[k]}$ and a field name ϕ*

$$W_r^{[j]} = \begin{cases} w_r^{[j]} & if\, j \neq k\, or\, r \neq l, \\ w_l^{[k]} \cup \{\phi\} & otherwise; \end{cases}$$

output variant: $(n, M)v^{[i]} \to W_j^{[i]}$, *where for some $l \le m^{[k]}$*

$$W_j^{[i]} = \begin{cases} w_j^{[i]} & if\, i \neq k, \\ w_j^{[k]} & if\, i = k\, and\, j < l, \\ w_{j+1}^{[k]} & if\, i = k\, and\, l \le j \le m^{[k]} - 1 \end{cases},$$

and

$$M^{[i]} = \begin{cases} m^{[i]} & \text{if } i \neq k, \\ m^{[k]} - 1 & \text{otherwise} \end{cases},$$

flow inheritance: *for any field name* $\phi \notin v^{[k]}$, $(n, m)V^{[i]} \to W_j^{[i]}$, *where*

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

and

$$W_j^{[i]} = \begin{cases} w_j^{[i]} & \text{if } i \neq k, \\ w_j^{[k]} \cup \{\phi\} & \text{if } i = k \text{ and } j \leq m^{[k]} \end{cases},$$

provided that $V^{[i]}$ *is in topological order.*

The above subtyping rules are general and consequently quite complex, although their meaning and application in most situations would be straightforward. Still two problems with subtyping remain at this point. Firstly, suppose that when a record of type v is sent to a box, the box responds with a certain output type τ; if the record is of a subtype $v' \sqsubseteq v$, the output type τ' can be completely unrelated to τ, even though the intention could have been to just use the fields common with v and ignore any fields in $v' \setminus v$. One could argue that the type signature of the box is quite clear about what the response is to any given type, and so if the additional fields are 'caught' by one of the alternatives, this would be deliberate and the user of the box would know about it. However, the second-order version of this problem causes a serious difficulty: in a network of boxes, how does the type signature of the network change if a box is replaced by its subtype? The answer potentially depends on every box in the network and cannot be abstracted easily. Even if we could obtain it, the 'second-order' signature of the network with respect to one participating box would, in general, be quite unwieldy, as it would have the type signature of the box in question as parameters. It is unlikely that such a device would be practical. To avoid problems of this kind, we constrain all box signatures to be monotonic, a property that we illustrate in the following.

2.7 Monotonicity

Informally, monotonicity means that one can use a subtype in place of a supertype and still assume that the output type is the same or coercible to the same. When there are more than one possible supertypes at the input, e.g. when the variant $\{a, b\}$ is present alongside the variants $\{a\}$ and $\{b\}$, the output type in response to each supertype must be included. In other words, a type signature is monotonic provided that for each input variant $v^{[i]}$ that is a subtype of some other input variant $v^{[j]}$, its associated output type $\tau^{[i]}$ is also a subtype of the output type $\tau^{[j]}$.

Definition 2.5 (monotonicity) *The type signature* $(n, m)v^{[i]} \to \tau^{[i]}$ *is considered monotonic iff*

$$(\forall i \in \{1, \ldots, n\})\, \tau^{[i]} \sqsubseteq \bigcup_{j \in \sigma v_i} \tau^{[j]}$$

where σv_i *denotes the set of indices* j *of all supertypes* $v_j \sqsupseteq v_i$ *of* v_i *in the given type signature.*

There is of course no guarantee of value consistency. For instance, a monotonic type signature that takes a single-field record x to a single-field record y can catch $\{x, z\}$ and produce a different value y as well as further fields in the output record. This, however, is not a problem since the

input record with field z carries more information which can be expected to affect the output value. The only thing that monotonicity guarantees is that the field y will not disappear merely because one has additionally supplied z at the input.

Monotonicity appears to be a useful property, but it does not come without a price. Consider an output type τ as a response to input v. If $v' \sqsubseteq v$ causes the box to yield output of type $\tau' \sqsubseteq \tau$, it follows that τ' cannot have variants essentially different from those that τ is made up of, in particular, one cannot introduce a nonempty variant that has no common fields with any variant of τ. However, imagine that the processing of v' sometimes raises certain exceptions that never arise when processing v, and so a variant is required to encode those. Then τ must include that variant (or a subset thereof) even though it will never be used at run-time as a response to v. Adding a variant to τ, would raise the box type, so such an alteration may cause a complete re-design of the network. Hence, some account should be taken of possible extensions already when designing the initial version of a box, which is undesirable as it prevents extensibility of the network. The solution is in exploiting the multiplicity of supertypes. One could, for example, add a rule such as $\langle x \rangle \rightarrow \tau''$ to the box signature and then include tag $\langle x \rangle$ into v'. Then τ' would be allowed to "inherit" any variants from τ'' and extend them as appropriate. Direct use of the $\langle x \rangle$ input can be guarded against by including a unique binding tag into one of the variants of τ'' which is not used by τ'. If the environment supplies $\langle x \rangle$, then it will not be able to match the unique tag appearing at the output and the resulting type error will alert the user. Such schemes could get as complex and secure as necessary and desirable, and the basic type infrastructure of S-Net will provide the required type guarantee.

It is interesting to note that flow inheritance is itself a form of monotonic subtyping. Indeed, it adds the same field to the input record and to each of the output records, thus replacing every record by its subtype. It is therefore obvious that if a signature is monotonic, applying flow inheritance to it will keep it monotonic. The same is true of subtyping by input variant (see above); the rest of the subtyping rules: input/output field and the output variant should be further constrained by the condition that the resulting signature is monotonic. It is possible to state such constraints explicitly as a restriction on the choice of ϕ, and where appropriate output τ, but since the modifications required are straightforward we shall leave them out to save space.

2.8 S-Net and Object-Orientation

Object-oriented languages have a more restrictive concept of subtyping whereby fields are sequentially ordered and only the tail fields can be ignored to produce a supertype. The motivation here is to preserve static field offset and thus to efficiently compile field access. S-Net deliberately does not restrict subtyping this way, and could pay the penalty of up to a single additional reference per field. The penalty would have been payable even without the liberal subtyping, since we accept that fields can have statically unknown size, which is the case in our component technology with opaque record fields. Still, with the added reference the record structure is fully static, due to the static topology of S-Net networks, which ensures that the type relationship between the producer and the consumer is resolved at compile time. More precisely, S-Net does have a dynamic connectivity mechanism (the indexed parallel replication combinator), but the dynamic connection is always with a member of a type-homogeneous box collection. Hence, the type relationship between the producer and consumer is always statically known.

Variant records in conventional languages have named variants and, hence, identification of an individual variant is by its names. In contrast, records in S-Net have anonymous variants containing named fields. Thus, variant identification is done primarily by analysis of the field set. As a consequence, input types of type signatures must be complete to ensure proper variant identification. Completeness may, for example, be achieved by the use of binding tags, which

effectively mimic the named variants of conventional languages.

Whilst the conventional approach completely avoids the variant ambiguity, it also precludes subtyping on the basis of field names only. For instance, in our running example of a type for geometric bodies, conventional subtyping would preclude the construction of a generic box that alters the body position expressed in terms of fields `x` and `y` that are common to all variants. As a result a box would require variants to be identified individually and specifically, even when the processing of fields `x` and `y` is the same for all variants (e.g. a simple shift operation).

The conventional object-oriented approach to this problem would be to use a base class with fields `x` and `y` and a method `shift` to be inherited by all subclasses. This restricts the design to at most one set of common fields (without multiple inheritance). More importantly, the significance of a common group of fields may only become apparent at a late stage in the design process or due the re-design of a system. With traditional object-oriented technology this usually requires a re-design of the class hierarchy.

Subtyping by subsetting as introduced in Section 2.2 supports *a-posteriori* introduction of a supertype (equivalent to a base class). The price to pay in implementation is the price of a run-time coercion, since we may no longer assume that the fields to be processed necessarily form a prefix of the field list.

Chapter 3

Network Description Language

3.1 Overview

S-Net essentially is a language for specifying hierarchical networks of boxes statically interconnected by typed streams. As a pure coordination language S-Net does not provide any means for the specification of computations, i.e. the concrete behaviour of boxes. This is left to existing computation or box languages like C or SaC [16, 17]. Likewise the concrete data travelling along the typed streams is opaque to S-Net and is only meaningful to the box language(s).

SNet	⇒	⌈ *Definition* ⌉*
Definition	⇒	*TypeDef* | *TypeSigDef* | *NetDef* | *BoxDef*
NetDef	⇒	**net** *NetName* ⌈ (*NetSignature*) ⌉ *NetBody*
NetSignature	⇒	*TypeSignature*
	|	*Type* `->` `...`
NetBody	⇒	⌈ { ⌈ *Definition* ⌉* } ⌉ **connect** *TopoExpr* ;

Figure 3.1: Grammar of S-Net specifications

Fig. 3.1 provides the root definition of S-Net syntax. An S-Net program is a sequence of definitions of types, type signatures, (atomic) boxes and (compound) networks. Type and type signature definitions have already been described in detail in Chapter 2. The definition of a network starts with the key word `net` followed by the network name, an optional network type, an optional network body (with further local S-Net definitions) and a network topology specification. The concrete type signature of a network is generally inferred by the S-Net compiler based on the network topology. Nevertheless, the programmer may provide an explicit type signature. Any given type signature must be in box subtype relationship with the inferred type signature. In other words, the given type signature must be a subsignature of the inferred signature. Otherwise, the S-Net compiler will raise a type error. Details on box subtyping can be found in Section 2.6.

Although unnecessary from a purely technical perspective, providing explicit type information makes sense for two reasons. First of all, it may serve as a documentation of the network, explaining its extensional behaviour in a concise way. More importantly, subtyping on type signatures provides an opportunity to specialise or customise given networks to particular needs. For example,

a given network may support more input variants than are actually needed in a certain context. Restricting the input type of the network as desired, allows the S-Net compiler to optimise the given network a-posteriori. Likewise, we may add additional record entries to variants of the input type. Although the operand network does not process these entries, flow inheritance ensures that they are added to each outgoing record before leaving the operand network. If the given output type of the type signature (in some variant) contains less record entries than the inferred output type, the additional entries will be discarded. Hence, if a type signature is provided, the network will behave exactly as specified. Furthermore, an annotated type signature also creates a private name space for the encapsulated network. Consequently, labels used internally in the network cannot conflict with labels flow-inherited on the outer network level.

Instead of providing a complete type signature, one may also choose to provide an input type only and to leave the corresponding output type unspecified. If so, the compiler infers the proper corresponding output type from the network definitions and the user simply foregoes the opportunity to manipulate the inferred output type by annotation.

The specification of a network is completed by the definition of the interconnection topology following the key word `connect`. In S-Net we specify interconnection topologies by an expression language. These *topology expressions* are made up of instances of networks defined in the current scope as well as instances of two kinds of built-in boxes: *filter boxes* for housekeeping tasks and *synchrocells* for synchronisation of records in the streaming network. Filter boxes and synchrocells are explained in detail in Sections 3.3 and 3.5, respectively. The operators of our network topology expression language are named *network combinators*; we elaborate on them in Sections 3.4 and 3.6.

The scoping rules of S-Net follow a pattern common to many other programming languages with local definitions. The scope of a type, box or network definition begins immediately behind the end of the definition. This deliberately precludes any recursive network definition: a network cannot refer to itself in its own connect expression. The scope of local definitions in network bodies include the connect expression of the network definition, but end immediately thereafter. For reasons of simplicity we disallow re-definitions of identifiers within the same scope.

Last but not least, S-Net allows for comments in the C/C++ style, i.e., single line comments start with `//` and last until the following end-of-line, multi-line comments start with `/*` and end with `*/`.

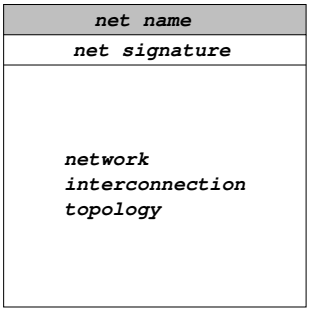

Figure 3.2: Graphical representation of S-Net networks

Fig. 3.2 sketches out a graphical representation of (compound) networks. It takes the form of a box with the network name in a head line, followed by a second line for the associated type signature. Since the explicit specification of a network's type signature is optional, this field may well be empty. Graphical representations of partially compiled S-Net programs may feature the inferred type signature here. The remaining field shows a graphical representation of the network

constituents and their interconnection topology, as outlined in the remainder of this chapter. Fully-fledged examples can be found in Chapter 11.

3.2 Box Declarations

In the design of S-Net boxes are the atomic units that are assembled to streaming networks in an inductive process. They also form the interface between the coordination layer of S-Net and the (sequential) computing layer implemented in a conventional language of choice. An S-Net box declaration promotes conventional legacy code to an asynchronous component within a streaming network.

BoxDef ⇒ **box** *BoxName* (*BoxSignature*) ;

BoxSignature ⇒ ⌈*BoxType* ⌉ `->` ⌈*BoxType* ⌈ `|` *BoxType* ⌉* ⌉

BoxType ⇒ (⌈*RecordEntry* ⌈ `,` *RecordEntry* ⌉* ⌉)

Figure 3.3: Grammar of S-Net box declarations

Fig. 3.3 shows the S-Net syntax for declaring user-defined boxes. A box is declared by the key word `box` followed by the box name and the box signature. The box signature looks very much like a type signature (cf. Section 2.3). However, it is restricted to a single mapping and a non-variant input type. Furthermore, we use round brackets instead of curly brackets to emphasise the fact that the order of fields and tags does matter in box signatures. The reason is that box signatures serve a dual purpose: in addition to describing the extensional behaviour of the box in the sense of what kinds od records it accepts and what kinds it emits in response they also describe the function call interface to the box language implementation. However, a type signature can trivially be derived from a box signature as is done by the type inference subsystem of the S-Net compiler.

Both the input type and the output type(s) of a box are optional. Whereas all regular boxes feature both input and output box types, two special cases can be realised omitting either of them. A missing input box type characterises a *initialiser box*. While a regular box is triggered by an appropriate record on its input stream, an initialiser box, as the box signature suggests, does not wait for any token on the input stream. Instead it is triggered automatically upon instantiation as part of a streaming network. Initialiser boxes allow for the introduction of constant box language domain values anywhere in a network. This is an often useful alternative to having such values propagated into the network through its global input stream. A missing output box type simply expresses the fact that the box does not emit any records in response to receiving input records of the appropriate type. Effectively, such a box discards all records it receives.

The specification of the intensional behaviour of a box is outside the scope of S-Net; a box is implemented using a conventional compute language, as opposed to S-Net as a coordination language. It is up to the S-Net compiler to locate the compiled box code in the file system and up to the programmer to provide the code accordingly in the first place.

Fig. 3.4 sketches out a graphical representation of a user-defined box in S-Net. It consists of a head line for the box name followed by second row containing the box signature. The remaining field features further meta data associated with the box, e.g. box language (pseudo) code.

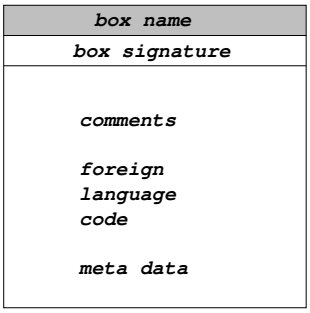

Figure 3.4: Graphical representation of user-defined boxes

3.3 The Filter Box

The primitive filter box in S-Net is devoted to all kinds of housekeeping operations. Effectively, any operation that does not require knowledge of field values can be expressed by this versatile primitive box in a simpler and more elegant way than using a box and deferring the actual operational behaviour to a separate implementation in a box language. Among these operations are

- eliminating fields and tags from passing records,
- copying fields and tags,
- adding tags,
- duplicating record fields,
- splitting records,
- simple computations on tag values.

Fig. 3.5 shows the syntax of S-Net filter boxes. Enclosed in square brackets their main syntactic constituents are a pattern and a potentially empty sequence of *guarded actions*. The pattern syntactically resembles a record type. Like the input type of boxes, the pattern describes what records the filter accepts as input. More precisely, the filter box only accepts a record as input whose type is a subtype of the pattern; excess fields and tags are handled by flow inheritance.

A guarded action consists of an optional *tag expression* and and an associated list of actions. We define a simple expression language on integer values that allows us to define computations involving integer constants as well as the values of tags matched in the pattern. Aggregate expressions can be formed using a fixed set of operators. The usual operator associativities and priorities apply. Hence, operator priorities are essentially in the order of appearance in Fig. 3.5. A simple type inference mechanism properly distinguishes between integer and boolean intermediate values.

The type system of S-Net ensures that any incoming record matches the pattern. Hence, the filter starts with evaluating the guard expressions in their order of appearance. The first guard expression that evaluates to `true` defines the filter's reaction on the incoming record. A guarded action without a guard expression, effectively an unguarded action, serves as a default action in case that non of the guard conditions in preceding guarded actions holds. If none of the guard expressions holds and there is also no default action, the filter consumes the incoming record without any response.

Filter ⇒ `[` *Pattern* `->` ⌈*GuardedAction* ⌋* *Action* `]`
| `[ ]`

Pattern ⇒ `{` ⌈*RecordEntry* ⌈ `,` *RecordEntry* ⌋* ⌋ `}`

GuardedAction ⇒ **if** `<` *TagExpr* `>` **then** *Action* **else**

Action ⇒ ⌈*RecordOutput* ⌈ `;` *RecordOutput* ⌋* ⌋

RecordOutput ⇒ `{` ⌈*OutputField* ⌈ `,` *OutputField* ⌋* ⌋ `}`

OutputField ⇒ *FieldName* ⌈ `=` *FieldName* ⌋
| `<` *TagName* ⌈ `=` *TagExpr* ⌋ `>`

TagName ⇒ *SimpleTagName* | *BindingTagName*

TagExpr ⇒ *TagName*
| *IntegerConst*
| `(` *TagExpr* `)`
| *UnaryOperator TagExpr*
| *TagExpr BinaryOperator TagExpr*
| *TagExpr* `?` *TagExpr* `:` *TagExpr*

UnaryOp ⇒ `!` | `abs`

BinaryOp ⇒ *ArithmeticOp*
| *ComparisonOp*
| *RelationalOp*
| *LogicalOp*

ArithmeticOp ⇒ `*` | `/` | `%` | `+` | `-`

RelationalOp ⇒ `==` | `!=` | `<` | `<=` | `>` | `>=`

LogicalOp ⇒ `&&` | `||`

ComparisonOp ⇒ `min` | `max`

Figure 3.5: Grammar of S-Net filter box

The action itself is defined by a sequence of record specifications to be emitted on the output stream. Again, there may be an empty action associated with some guard in which case the filter only consumes the input record. Otherwise, the length of the list determines the number of records produced in response to any incoming record.

Each output record specification is of a set of record entries enclosed in curly brackets. If a record entry also occurs in the pattern, the corresponding field or tag is left untouched by the filter box and is forwarded from the incoming record to the outgoing record. If a record entry occurs in the pattern, but not in the output record specification, it is discarded (at least for this output record specification; it may still be used in other output record specifications). If a record

entry occurs in the output record specification, but not in the pattern, it is new and requires initialisation. In the case of a field the only way of initialisation is by reference to another field from the pattern. This may, for instance, be used to rename fields without changing their values. In the case of a tag, we can use the same simple expression language that we have introduced for guard expressions to define values of new tags based on the values of tags from the pattern. Initialisation of new tags is still optional; zero serves as a default tag value.

For example, the following filter box

```
[{a,b} -> ]
```

accepts only records that contain fields `a` and `b` and discards them unconditionally, whereas the filter box

```
[{a,b} -> {c=a, <d=42>}]
```

renames field `a` to `c`, discards field `b` and adds a new tag `<d>` that is set to 42. Furthermore, the filter box

```
[{c,<d>} if <d==42> -> {c} ; {<d>}
         if <d==43> -> {c}
                    -> ]
```

takes on records with field `c` and tag `<d>`. If the value of `<d>` equals 42, the filter splits the record and first sends field `c` and then tag `<d>` to the output stream. If the value of `<d>` equals 43, it sends field `c` to the output stream, but discards tag `<d>`. In all other cases, the filter discards the input record entirely.

In its simplest possible variant the filter `[{}->{}]` behaves as an identity function and simply forwards any incoming record without binding tags to its output stream without touching it. This behaviour can be very useful when combining the identity filter in parallel with some other SNet. In such a configuration the filter acts as a default case with the parallel SNet handling specific cases of records on the stream while all records not matching the input type of that SNet are bypassed via the identity filter. Given the relevance of the identity filter to construct SNets of this kind we introduce "`[ ]`" as a notational simplification.

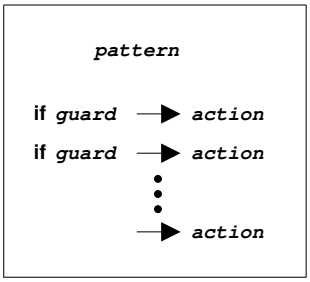

Figure 3.6: Graphical representation of S-Net filters

Type inference for filter boxes is rather straightforward. Essentially, the pattern acts as an input type while the sequence of guarded actions makes up the output type by stripping of record entry initialisations as necessary.

Fig. 3.6 sketches out a graphical representation of S-Net filter boxes.

3.4 Network Combinators

A distinctive feature of S-Net is the fact that complex network topologies are not defined by some form of wire list, but instead by an expression language. Each network definition contains such a topology expression following the key word `connect`. Atomic expressions are made up of box and network names defined in the current scope as well as of built-in filter boxes and synchrocells. Complex expressions are inductively defined using a set of network combinators that represent the four essential construction principles in S-Net: serial and parallel composition of two (different) networks as well as serial and parallel replication of a single network. We give a formal definition of the network topology expression language in Fig. 3.7.

TopoExpr	⇒	*BoxName*
	|	*NetName*
	|	*Sync*
	|	*Filter*
	|	*Combination*
	|	(*TopoExpr*)
Combination	⇒	*Serial* | *Star* | *Parallel* | *Split*
Serial	⇒	*TopoExpr SerialComb TopoExpr*
Parallel	⇒	*TopoExpr ParallelComb TopoExpr*
Star	⇒	*TopoExpr StarComb Terminator*
Terminator	⇒	*GuardPattern* [, *GuardPattern*]*
Split	⇒	*TopoExpr SplitComb Tag*
SerialComb	⇒	..
ParallelComb	⇒	| | ||
StarComb	⇒	* | **
SplitComb	⇒	! | !!

Figure 3.7: Grammar of S-Net topology expression language

The binary serial combinator "`..`" connects the output stream of the left operand to the input stream of the right operand. The input stream of the left operand and the output stream of the right operand become those of the combined network. The serial combinator establishes computational pipelines, as illustrated in Fig. 3.8.

Figure 3.8: Illustration of serial composition of networks: `foo..bar`

As a simple example of a network definition take the following network `example` that contains two user-defined boxes, `foo` and `bar`, and connects both using the serial combinator:

```
net example {
  box foo ((a,b)->(c,d));
  box bar ((c)->(e));
}
connect foo..bar;
```

All output from box `foo` goes into box `bar`. This example nicely demonstrates the power of flow inheritance: In fact the output type of box `foo` is not identical to the input type of box `bar`, but rather is a subtype of it. By means of flow inheritance, any field `d` originating from box `foo` is stripped of the record before it goes into box `bar`, and any record emitted by box `bar` will have this field be added to field `e`.

The binary parallel combinator "`|`" combines its operand networks or boxes in parallel. Any incoming record is sent to exactly one operand depending on its type and the type signatures of the operand networks or boxes. Fig. 3.9 illustrates the parallel composition of two networks `foo` and `bar`, i.e. `foo|bar`.

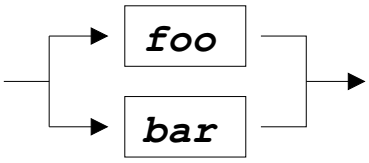

Figure 3.9: Illustration of parallel composition of networks: `foo|bar`

To be precise, any incoming record is sent to that operand network whose type signature's input type is matched best by the record's type. Let us assume the type signature of `foo` is {a}->{b} and that of `bar` is {a,c}->{b,d}. An incoming record {a,<t>} would go to `foo` because it does not match the input type of `bar`, but thanks to record subtyping does match the input type of `foo`. In contrast, an incoming record {a,b,c} would go to `bar`. Although it matches in fact both input types, the input type of `bar` scores higher (2 matches) than the input type of `foo` (1 match).

If a record's type matches both type signatures under consideration equally well, the record is non-deterministically sent to one of the operand networks. In this case, an S-Net implementation is free to choose an appropriate scheduling technique. For example, it may send the record to the less loaded operand for proper workload balancing. The parallel combinator is also referred to as *choice combinator* stressing the property that an input record chooses exactly one branch.

The output streams of the operand networks (or boxes) are merged into a single stream, which becomes the output stream of the combined network. By default, merging of output streams is done non-deterministically, i.e., as soon as a record is available in any of the operand output streams, it is immediately forwarded to the combined output stream. This behaviour can be implemented rather efficiently, but it does not preserve any order induced from the combined input stream of the network. In fact, an input record may effectively overtake an earlier one when taking the other branch of a parallel composition.

There is a restriction on the type signatures of networks that may be combined in parallel: *covariance*. Whenever the input type of one operand network is a subtype of the input type of the other operand network, their output types must be in the same subtyping relationship. This restriction is motivated by the following observation. If one operand network provides a general solution to some problem and the other operand network a specific solution for a subset of potential incoming records then the resulting records emitted by the specialised solution should likewise be

more specific than those emitted by the general solution. Hence, the restriction helps to construct orderly behaving networks.

The serial replication combinator "`*`" replicates the operand network (the left operand) infinitely many times and connects the replicas by serial composition. The right operand of the combinator defines a set of type patterns. As soon as a record matches one of them, i.e., the record's type is subtype of the type pattern, the record is released and sent to the global output stream. In fact, an incoming record that matches one of the termination patterns right away is immediately passed to the output stream without being processed by the operand network. This coincidence with the meaning of star in regular expressions particularly motivates our choice of the star symbol, and we sometimes refer to the serial replication combinator as the *star combinator*. Fig. 3.10 illustrates the operational behaviour of the star combinator for a network `foo*{<stop>}`: Records travel through serially combined replicas of `foo` until they contain a tag `<stop>`. Actual replication of the operand network is demand-driven. Hence, networks in S-Net are not static, but generally evolve dynamically.

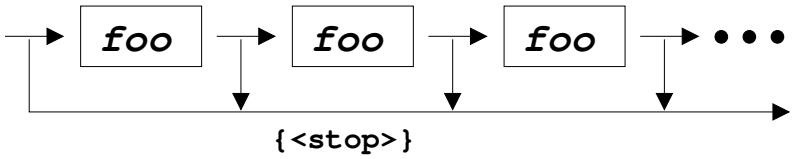

Figure 3.10: Illustration of serial replication of networks: `foo*{<stop>}`

Last but not least, the parallel replication combinator "`!`" takes a network or box as its left operand and a tag as its right operand. Like the star combinator, it replicates the operand, but connects the replicas using parallel rather than serial composition. The number of replicas is conceptually infinite. Each replica is identified by an integer index. Any incoming record goes to the replica identified by the value associated with the given tag, i.e., all records that have the same tag value will be processed by the same replica of the operand network. Since parallel replication actually splits a stream of records depending on a certain tag, we also refer to "`!`" as the *index split combinator*. Fig. 3.11 illustrates the operational behaviour of the index split combinator for a network `foo!<tag>`. In analogy to serial replication, the instantiation of replicas of the operand network is demand-driven.

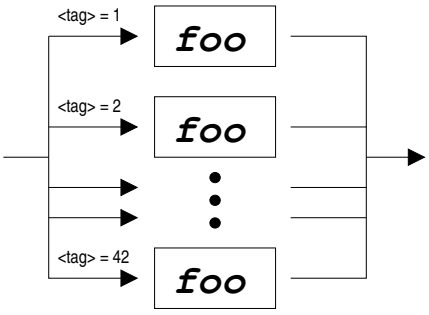

Figure 3.11: Illustration of indexed parallel replication of networks: `foo!<tag>`

While Figures 3.8 and 3.9 immediately insinuate the graphical representation of serial and parallel composition, the illustrations of serial and parallel replication are less suitable as graphical

representations. Instead, we use the network-like representations shown in Fig. 3.12

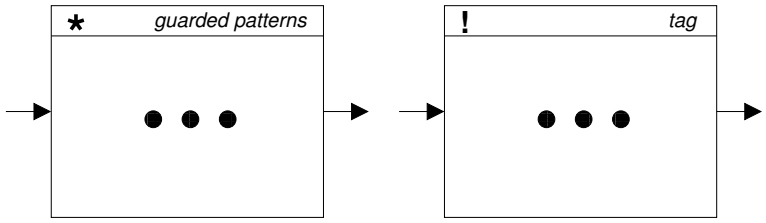

Figure 3.12: Graphical representations of serial (left) and parallel (right) replication

3.5 The Synchrocell

The synchronisation cell, or synchrocell for short, is the only "stateful" box in S-Net. Its concrete syntax is given in Fig. 3.13. Embedded within `[|` and `|]` parentheses, we find an at least 2-element list of possibly guarded patterns. The concept of a pattern, syntactically resembling a record type, is alread familar from Section 3.3. A guarded pattern is associated with a guard expression defined using our simple expression language introduced for filter boxes. The principle idea behind the synchrocell is that it keeps incoming records which match one of the patterns until all patterns have been matched. Only then the records are merged into a single one that is released to the output stream. Matching here means that the type of the record is a subtype of the pattern and, if a guard expression is present, it evaluates to `true`. The pattern also acts as an input type specification of the synchrocell: a synchrocell only accepts records that match at least one of the patterns.

Sync ⇒ `[|` *GuardPattern* ⌈ `,` *GuardPattern* ⌋+ `|]`

GuardPattern ⇒ *Pattern* ⌈ **if** `<` *TagExpr* `>` ⌋

Figure 3.13: Grammar of S-Net synchrocells

More precisely, a synchrocell has storage for exactly one record of each pattern. When a record arrives at a fresh synchrocell, it is kept in this storage and is associated with each pattern that it matches. Any record arriving thereafter is only kept in the synchrocell if it matches a previously unmatched pattern. Otherwise, it is immediately sent to the output stream without alteration. As soon as a record arrives that matches the last remaining previously unmatched variant, all stored records are released. The output record is created by merging the fields of all stored records into the last matching record. This requires patterns of a synchrocell to be pairwise disjoint. Otherwise, we had indistinguishable fields in the output record. If an incoming record matches all patterns of a fresh synchrocell right away, it is immediately passed to the output stream without delay.

Once a synchrocell has received incoming records for each of its input, its purpose is fulfilled and the cell effectively dies. More precisely, all records received after a full match are immediately passed to the output stream.

The type signature of a synchrocell `[|`$v_1, \ldots, v_n$`|]` is

$$\begin{array}{lll} \{v_1\} & \texttt{->} & \{v_1\} \texttt{ | } \{v_1, \ldots, v_n\} \\ \{v_2\} & \texttt{->} & \{v_2\} \\ & \ldots & \\ \{v_n\} & \texttt{->} & \{v_n\} \end{array}$$

It reflects the fact that any incoming record may either be passed through in case of an overflow or it may trigger synchronisation, in which case the output record contains fields from all patterns. The asymmetry between the first type mapping and all other type mappings stems from the specific handling of the first pattern with respect to flow inheritance.

Synchrocells require a special treatment with respect to flow inheritance: At first glance, one may say that if a synchrocell stores a matching input record, it produces no output in response to this record. Hence, excess record fields, which would bypass the synchrocell otherwise, should be discarded. Any record output after successful synchronisation should be extended by the excess fields of the last incoming record because the synchrocell produces this output as a response to the input of this record. Last but not least, if a record is passed through the synchrocell in the case of overflow, there is output in response to input and, therefore, the excess fields bypass the synchrocell as usual. However, this behaviour leads to very irregular and difficult to control behaviour of synchrocells where the sequence of arrival of records to be matched is non-deterministic (and so for good reason). In this case, flow inheritance would keep excess fields in a non-deterministic way as well, and that makes orderly processing of synchronised records in the subsequent network extremely difficult. Hence, it is not the record that triggers synchronisation which keeps its excess fields, but the record that matched the (lexically) first pattern in the definition of the synchrocell. Excess fields of all records that match other than the lexically first pattern are discarded immediately.

An alternative flow inheritance rule for synchrocells would be to keep the excess fields of all synchronised records, but this rule has its downsides as well: Typically, excess fields of different records have the same labels, but the labels may be associated with different values. Since all values are entirely opaque to S-Net as a matter of design this situation cannot be detected and leads to further undesirable non-determinism.

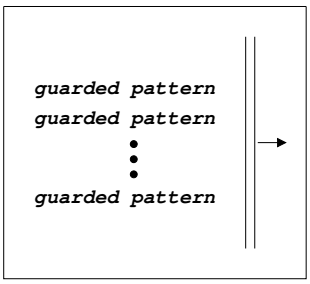

Figure 3.14: Graphical representation of S-Net synchrocells

Fig. 3.14 shows the graphical representation of a synchrocell. Following the name “sync” in the status line, the main field contains the patterns line by line. The double line to the right of the patterns insinuates the synchronisation.

3.6 Deterministic Combinators

In three out of four network combinators, the parallel combinator, the star combinator and the index split combinator, streams are merged non-deterministically. This operational behaviour is

efficient as records are forwarded to the joint output stream as soon as they are available on one of the operand networks' output streams. The downside of this approach is that records may effectively overtake each other. Take the parallel composition of `foo` and `bar` in Fig. 3.9 as an example. Let us assume the first record on the joint input stream is routed to `foo` and the second record to `bar`. There is no reason why box `bar` should not emit its response record(s) to its output stream before box `foo` does so. As a consequence, the order of records in the joint input stream is not preserved.

Whether or not the non-deterministic merging of streams is considered a problem, very much depends on the concrete application. Therefore, S-Net actually features two variants of each of the network combinators that involve merging of streams: we have a deterministic parallel composition combinator "`||`", a deterministic star combinator "`**`" and a deterministic index split combinator "`!!`". They are otherwise identical to their non-deterministic counterparts introduced in the previous section, but do preserve the causal order of records.

Providing both a non-deterministic and a deterministic variant of each relevant network combinator is motivated by the observation that different application scenarios require different operational behaviours of choice. The non-deterministic variant usually is more efficient since it allows the network to continue processing records as soon as they are available. However, in many situations it is crucial that a network behaves more like a box with respect to causality and ensures that records do not overtake others. This comes at the price of holding back readily processed records from the output stream and waiting for other records to be sent first.

Our motivation for using the symbols "`||`", "`**`" and "`!!`" to denote the deterministic variants of our network combinators is twofold. Firstly, the additional character reminds us that some additional effort is required to achieve deterministic behaviour. Secondly, the serial combinator "`..`", which also consists of two characters, is always deterministic as it trivially preserves the order of records.

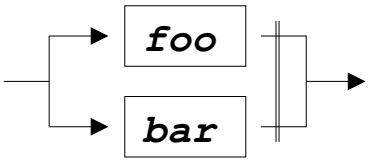

Figure 3.15: Graphical representation of deterministic parallel composition of networks: `foo||bar`

Fig. 3.15 shows the graphical representation of the deterministic parallel combinator. A vertical double line that crosses both output streams symbolises the additional synchronisation required to achieve deterministic behaviour. Graphical representations of the deterministic star and index split combinators simply use the deterministic combinator symbol in the upper left corner of the box instead of the standard non-deterministic one (cf. Fig. 3.12).

3.7 Combinator Associativity and Priority

The definition of an expression language based on unary and binary infix combinators immediately raises the questions of associativity and priority rules. The binary combinators ("`..`", "`|`" and "`||`") are in fact associative. For example, the two expressions `A..(B..C)` and `(A..B)..C` are semantically and operationally equivalent. If brackets are left out in complex expressions, we assume left-associativity for all binary combinators. For example, the expression `A..B..C` is equivalent to `(A..B)..C`.

In order to facilitate the construction of complex topology expressions brackets may be left out according to the following order of combinator priorities:

$$\text{“|”} \prec \text{“||”} \prec \text{“..”} \prec \text{“*”, “**”, “!”, “!!”}$$

The rationale behind these priorities is the following: The serial and parallel replication combinators are effectively unary combinators as far as network construction is concerned. They all have the same highest priority.

The fact that parallel combination has lower priority than serial combination reflects their similarity to boolean disjunction and conjunction: in parallel combination a record is either routed through one operand network *or* through the other; in serial combination a record is routed through the first *and* through the second operand network.

Last but not least, deterministic parallel combination has a higher priority than its non-deterministic counterpart because non-deterministic behaviour allows for a looser form of coupling between the operand networks.

3.8 S-Net Programming in the Large

S-Net provides support for programming-in-the-large, i.e. S-Net specifications that stretch over multiple files. Any S-Net file or module implicitly exports the top-level network definition that bears the same name as the file (i.e. module) itself.

Any network identifier in a connect expression that is unbound within the current file is considered to refer to an external network definition. The correspondence between network name and file name allows the S-Net compiler to search for the definition of such a network within a set of directories that may be defined by environment variables and compiler parameters in an implementation-dependent way.

Effectively, each S-Net file itself defines a scope within the set of S-Net files in a local file system. The scoping rules sketched out in Section 3.1 apply within each individual file. On top of that level there is a global scope of exported networks. Any locally unbound network identifier is supposed to be bound in the global file system level scope.

Chapter 4

Type System

4.1 Introduction

S-Net is not a conventional language, and its type system differs from the common understanding of the type systems in object-oriented or functional languages. The types in S-Net not only specifies what should be input and what may be output, they also influence the decision of routing, in front of every parallel composition, for each input record.

In an S-Net program, the programmer provides box signatures and the signatures of some networks, but no routing information. Its absence indicates that we are not yet able to describe how parallel compositions behave. However, assuming the routing information exists, we can observe the complete typewise behaviour of the program, and therefore the type system can now check whether the program is type-safe.

To populate the routing information, the type system performs inference on each parallel branch to find out what types of records are useful. The resulting routing information attracts the appropriate types into the branch, although the procedure cannot guarantee that all attracted types are safe within the branch, and a type check is still due.

In this chapter, we first present the type checking procedure on an abstract interpretation of an S-Net program populated with the routing information, and then introduce the route inference procedure.

4.2 S-Net Abstract Interpretation

The type system does not deal with individual records, and using the concrete semantics is an overkill. In this section, we present an abstract interpretation of S-Net programs adapted to the purposes of the type system. We call the resulting language S-Net$_{\mathrm{A}}$.

S-Net$_{\mathrm{A}}$ Syntax

Figure 4.1 shows the syntax after the adaptation. All syntactic elements dealing with individual records or label values are removed or simplified. Comparing to the standard syntax, S-Net$_{\mathrm{A}}$ has the following differences:

- Tags and fields are grouped into one single set *Field*.

$$
\begin{array}{lcll}
P \in Program & ::= & N & \\
N \in Net & ::= & \texttt{net}\ (\sigma)\ \texttt{connect}\ e & \\
e \in TopoExpr & ::= & \texttt{box}\ (r \to \tau) & \text{(Box)} \\
& | & e_1..e_2 & \text{(Dotdot)} \\
& | & e_1|e_2 & (\text{Bar}_0) \\
& | & (\tau_1)e_1|(\tau_2)e_2 & \text{(Bar)} \\
& | & e_0 * \gamma & \text{(Star)} \\
& | & e_0!\lambda & \text{(Ex)} \\
& | & N & \text{(Subnet)}
\end{array}
$$

$$
\begin{array}{rcccl}
\lambda & \in & Label & = & Field \cup BTag \\
r, v & \in & Rtype & = & \mathcal{P}(Label) \\
\tau & \in & Vtype & = & \mathcal{P}(Rtype) \\
\gamma & \in & Gtype & = & Rtype \rightharpoonup Bool \\
\sigma & \in & Signature & = & Rtype \rightharpoonup Vtype
\end{array}
$$

$$
Field,\ BTag \text{ finite},\quad Field \cap BTag = \emptyset,\quad \sigma \neq \emptyset.
$$

Figure 4.1: S-Net$_A$ syntax. $\mathcal{P}(x)$ denotes the powerset of a set x.

- For structural simplicity, boxes and named subnetworks are written inline within the topology expressions, instead of defined separately and referred to by names. Moreover, subnetworks without signatures are simply replaced by their topology expressions.
- Boxes do not have names because the external function referred to by box names, which manipulate the values in the records, are not checked by the type system.
- Filters are constructed as boxes, where the output types may be empty, and all the conditionals are removed.
- Synchrocells are constructed as a parallel composition of as many boxes as necessary: the first box takes in the main record type and responds with the main record type and the synchronised record type, and the remaining boxes take in and pass through each auxiliary record type defined in the synchrocell.
- After route inference (see Section 4.4), every parallel branch will be associated with the routing information, i.e. a set of record types it is deemed to attract.
- There are no deterministic operator variants.
- The termination pattern γ of a serial replication is represented by a partial function, whose domain, $\tau_0 = \mathrm{dom}(\gamma)$ is the set of record types defined in the terminating pattern, and for each $r \in \tau_0$, $\gamma(r)$ means the record type is associated with a guard, and $\neg\gamma(r)$ means the record type is unconditional.

Despite the list above, even manually translating a standard S-Net program into its abstract interpretation should not inflict any troubles, except the routing information, which the route inference procedure explaned in Section 4.4 will help. Unchanged is Definition 2.1 about subtype relation, reprised below using S-Net$_\mathrm{A}$ terms: r_1 is a subtype of r_2, denoted as $r_1 \sqsubseteq r_2$, iff

$$r_2 \subseteq r_1 \ \wedge \ (r_1 \cap BTag) = (r_2 \cap BTag).$$

For conciseness, throughout the rest of this chapter, an instance of the class *Rtype* is called a *type* instead of a record type, and a *Vtype* instance is called a *variant type* instead of a type.

Type Transformations

Every topology expression in a standard S-Net program can accept certain types of records and produce other types of records in response. In the abstract interpretation, we view it as the topology expression transforming some types into other types. In reality, the declared output types of a box or network only show what *may* be output: the box or network needs not produce all the declared types; it needs not even produce a single record to comply with the declaration. However, in the abstract interpretation, we consider all *may*s as *must*s, for the type system to capture all possible cases and perform a complete type check.

The following judgement denotes that the topology expression e can transform the input type r to any output types in τ (may be an empty set):

$$e \vdash r \rightsquigarrow \tau.$$

The judgement is true if and only if there is a deduction process that concludes it, using only the type transformation rules in the remaining parts of this section. In case one cannot deduce any conclusions with a topology expression, the expression is said to cause a *type error*.

Boxes

A box declares an input type and zero or more output types. Naturally, it transforms the declared input type into the declared set of output types.

Flow Inheritance A box also accepts all subtypes of the declared input type. For each such type, where there are excess labels, it behaves as if these labels are stripped off from the type before it is fed into the box, and added back to each of the output types. In other words, the excess labels *bypass* the box. On adding back the labels, duplicates are naturally eliminated by the nature of simple sets.

$$\frac{r' \sqsubseteq r}{\texttt{box}\ (r \to \{\overrightarrow{v_i}\}) \vdash r' \rightsquigarrow \left\{\overrightarrow{v_i \cup (r' \backslash r)}\right\}} \quad \textsc{tt-Box}$$

In the rule above, $\{\overrightarrow{\ }\}$ is a "set-of" notation whose usage is as follows: $\{\overrightarrow{v_i}\}$ is a shortcut notation of $\{v_i \mid i = 1..?_i\}$. When this is pattern-matched with a concrete set, it essentially assigns an index to every element of the set (in arbitrary order) and binds $?_i$ to a concrete integer. It is invalid to refer to $?_i$ directly, but later uses of the variable i are confined within other set-of notations or must be universally quantified, to enforce the same range $i = 1..?_i$. For example, $\left\{\overrightarrow{v_i \cup (r' \backslash r)}\right\}$ in the rule above is short for $\{v_i \cup (r' \backslash r) \mid i = 1..?_i\}$, referring to the same upper bound.

Serial Compositions

A serial composition $e_1..e_2$ behaves like a function composition: e_1 transforms an input type into a set of intermediate types, and e_2 transforms each of them to a set of output types. The union of all sets of output types become the result of the transformation.

$$\frac{e_1 \vdash r \rightsquigarrow \{\overrightarrow{v_i}\} \quad \forall i : (e_2 \vdash v_i \rightsquigarrow \tau_i)}{e_1..e_2 \vdash r \rightsquigarrow \bigcup_i \tau_i} \quad \textsc{tt-Dotdot}$$

We also define that the big union operator on a nonexistent range results in an empty set. This definition is used by the rule above as well as other rules with big union operators in this chapter. This means that $e_1 \vdash r \rightsquigarrow \emptyset$ immediately implies $e_1..e_2 \vdash r \rightsquigarrow \emptyset$ without having to look into e_2.

Parallel Compositions

A parallel composition, with routing information, $(\tau_1)e_1|(\tau_2)e_2$ lets either e_1 or e_2, whichever better attracts the input type, to perform the transformation. The function below selects the best matching types relative to the actual input type r in τ_1 or τ_2:

$$\mathrm{BM}(r, \tau) \stackrel{\text{def}}{=} \{v \in \tau' \mid |v| = \max\{|v_0| \mid v_0 \in \tau'\}\},$$

where

$$\tau' = \{v_0 \in \tau \mid r \sqsubseteq v_0\}.$$

The topology expression e_1 better attracts r compared to the other topology expression e_2, iff the types in the best match set $\mathrm{BM}(r, \tau_1)$ have more labels than in $\mathrm{BM}(r, \tau_2)$:

$$\frac{\exists r_1 \in \mathrm{BM}(r, \tau_1),\ \forall r_2 \in \mathrm{BM}(r, \tau_2) : |r_1| > |r_2| \quad e_1 \vdash r \rightsquigarrow \tau}{(\tau_1)e_1|(\tau_2)e_2 \vdash r \rightsquigarrow \tau} \quad \textsc{tt-Bar1}$$

Note that the function BM guarantees all types in the result set have the same cardinality, i.e. number of labels. The use of qualifiers $\exists$ and $\forall$ in the rule above only helps guarantee that $\mathrm{BM}(r, \tau_1)$ is nonempty, and that e_1 can be chosen when $\mathrm{BM}(r, \tau_2)$ is empty.

The other route can be covered by equivalence:

$$(\tau_1)e_1|(\tau_2)e_2 \equiv (\tau_2)e_2|(\tau_1)e_1.$$

In case the function has collected types of equal cardinality from both branches, the parallel composition lets both topology expressions transform the input type, and then merges the output types:

$$\frac{\begin{array}{c}\exists r_1 \in \mathrm{BM}(r, \tau_1), r_2 \in \mathrm{BM}(r, \tau_2) : |r_1| = |r_2| \\ e_1 \vdash r \rightsquigarrow \tau_1 \\ e_2 \vdash r \rightsquigarrow \tau_2\end{array}}{(\tau_1)e_1|(\tau_2)e_2 \vdash r \rightsquigarrow \tau_1 \cup \tau_2} \quad \text{TT-BAR2}$$

Careful readers may think that the last rule does not comply with the standard S-Net semantics, where each input record is delivered to only one of the parallel branches nondeterministically. However, the decision process is done per record, and both branches have the probability to be selected. As explained at the beginning of Section 4.2, all possibilities are considered in the abstract interpretation, and, therefore, both cases are included in the result.

Serial Replications

The abstract interpretation of a serial replication $e * \gamma$ is rather tricky to define. It involves a serial composition chain of as many replicas of e as necessary to transform all non-terminating intermediate types into terminating output types. We will employ a template approach to find the overall type transformation.

For each serial replication $e * \gamma$, define one and only one special binding tag $\Lambda_{e*\gamma} \in \mathit{BTag}$, which is unused throughout the topology of e and the entries in γ. This is to tag the types that match the terminating pattern and are due to output from the serial chain. Adding such binding tag into the type guarantees that it will never be transformed by another replica of e.

Define a selection filter $f_{e*\gamma} \in \mathit{TopoExpr}$ that tags any terminating types with $\Lambda_{e*\gamma}$ and passes through any other types. If a type is conditionally terminating, then the filter generates both tagged and non-tagged copies:

$$\frac{\forall r_0 \in \mathrm{dom}(\gamma) : r \not\sqsubseteq r_0}{f_{e*\gamma} \vdash r \rightsquigarrow \{r\}} \quad \text{TT-STARFNT}$$

$$\frac{\exists r_0 \in \mathrm{dom}(\gamma) : r \sqsubseteq r_0 \wedge \neg\gamma(r_0)}{f_{e*\gamma} \vdash r \rightsquigarrow \{r \cup \{\Lambda_{e*\gamma}\}\}} \quad \text{TT-STARFUT}$$

$$\frac{\begin{array}{c}\exists r_0 \in \mathrm{dom}(\gamma) : r \sqsubseteq r_0 \\ \forall r_0 \in \mathrm{dom}(\gamma) : r \sqsubseteq r_0 \implies \gamma(r_0)\end{array}}{f_{e*\gamma} \vdash r \rightsquigarrow \{r \cup \{\Lambda_{e*\gamma}\}, r\}} \quad \text{TT-STARFCT}$$

Note that any type with $\Lambda_{e*\gamma}$ falls into the non-terminating category and is carried over unchanged according to the first rule.

Define a variation of the topology expression e, $e_{e*\gamma} \in \mathit{TopoExpr}$, which in addition to the type transformations by e, can pass through any type with $\Lambda_{e*\gamma}$ unchanged:

$$\frac{e \vdash r \rightsquigarrow \tau}{e_{e*\gamma} \vdash r \rightsquigarrow \tau} \quad \text{TT-STAREN}$$

$$\frac{\Lambda_{e*\gamma} \in r}{e_{e*\gamma} \vdash r \rightsquigarrow \{r\}} \quad \text{TT-StarEp}$$

Define a new topology expression form $e *^n \gamma$, with the following meaning:

$$e *^n \gamma \stackrel{\text{def}}{=} f_{e*\gamma} .. \underbrace{(e_{e*\gamma} .. f_{e*\gamma}) .. (e_{e*\gamma} .. f_{e*\gamma}) .. \cdots}_{n \text{ times}},$$

for which we can deduce type transformations using the rule for serial composition. Every element in this serial chain can pass through $\Lambda_{e*\gamma}$-tagged types, which without the tag are the real outputs. This implies that we will be able to collect all the output types by infinitely increasing n.

There exists a fixed point, i.e. an n, where further unfolding does not produce more output types. The collection of output types is complete once this fixed point is detected. Then, the output types, minus the tag, become the transformation result of the serial replication. This process is formalised below, where unfolding is done at least twice to capture any possible type errors:

$$\frac{\begin{array}{c}\Lambda_{e*\gamma} \notin r \\ i > 1 \\ e *^i \gamma \vdash r \rightsquigarrow \tau_1 \\ e *^{i+1} \gamma \vdash r \rightsquigarrow \tau_2 \\ \tau_1 = \emptyset \vee \exists v_0 \in \tau_1 : \Lambda_{e*\gamma} \in v_0 \\ \{v_0 \in \tau_1 \mid \Lambda_{e*\gamma} \in v_0\} = \{v_0 \in \tau_2 \mid \Lambda_{e*\gamma} \in v_0\}\end{array}}{e * \gamma \vdash r \rightsquigarrow \{v_0 \backslash \{\Lambda_{e*\tau}\} \mid v_0 \in \tau_1 \wedge \Lambda_{e*\gamma} \in v_0\}} \quad \text{TT-Star}$$

Parallel Replications

As far as the type system is concerned, a parallel replication simply demands the label λ in the input types. The layout of the parallel replication and the delivery of records are insignificant in the abstract interpretation.

$$\frac{\begin{array}{c}\lambda \in r \\ e \vdash r \rightsquigarrow \tau\end{array}}{e!\lambda \vdash r \rightsquigarrow \tau} \quad \text{TT-Ex}$$

Subnetworks

A subnetwork defines a closed environment, only exposing a user-declared interface, i.e. the signature. Provided the subnetwork passes the type check, we simply use the signature to perform our type transformations.

Because a network signature may present multiple type transformations, a procedure similar to the best match choice for parallel compositions (see Section 4.2) is applied at the entrance of the subnetwork. Also, a higher level of flow inheritance takes place at the borders of a subnetwork: after the best match decision, excess labels that are in the input but not in the chosen declared input type are flow inherited to the exit of the network.

$$\frac{\begin{array}{c}e \lhd \sigma \\ \{\overrightarrow{r_i}\} = \mathrm{BM}(r, \mathrm{dom}(\sigma)) \neq \emptyset\end{array}}{\texttt{net}\ (\sigma)\ \texttt{connect}\ e \vdash r \rightsquigarrow \bigcup_i \{v \cup (r \backslash r_i) \mid v \in \sigma(r_i)\}} \quad \text{TT-Subnet}$$

In the rule above, $e \lhd \sigma$ means the topology expression agrees with the signature. Full definition will be covered in Section 4.3.

The same rule also describes the abstract interpretation of the execution of the whole program, which itself is a network.

4.3 Type Check

The judgement $e \triangleleft \sigma$ denotes that the topology expression e agrees with the signature σ. Again, it is true if and only if there exists a deduction procedure that concludes it, using the type transformations in Section 4.2 and the type check rule defined below:

$$\frac{\begin{array}{l}\forall r, \tau_0 : \sigma(r) = \tau_0 \implies \\ \qquad \exists \tau : (e \vdash r \rightsquigarrow \tau) \wedge \tau \sqsubseteq \tau_0\end{array}}{e \triangleleft \sigma} \text{TC}$$

where we used the subtype relation between two variant types, defined as follows: τ_1 is a subtype of τ_2, denoted as $\tau_1 \sqsubseteq \tau_2$, iff

$$\forall r_1 \in \tau_1 : \exists r_2 \in \tau_2 . r_1 \sqsubseteq r_2.$$

The rule requires that all mappings in the signature are backed up by type transformations, but the reverse is not necessary.

If for a network, the topology expression agrees with the declared signature, we say the network passes the type check. An S-Net$_A$ program is type-safe if its top-level network passes the type check, meaning that type errors will never occur.

4.4 Route Inference

The parallel composition type transformation (see Section 4.2) can be performed only with the routing information. It is the task of the route inference to provide such information.

The route inference procedure finds out what types of records are *useful* for every parallel branch. A type is useful if all labels in the type are potentially used, i.e. consumed by a box, within the topology of the branch. A set of these types is then used as the routing information of the branch, for it to attract the appropriate records in competition with the other branch of the same parallel composition.

The route inference consists of two types of computation: signature inference to collect the useful types, and the actual route inference to transform every instance of parallel composition in the form of $e_1|e_2$ into $(\tau_1)e_1|(\tau_2)e_2$.

Foundations

Required, Guaranteed and Discarded Labels

Consider a box A with input type r and output type set $\{v\}$. The box accepts any subtype of r, where r being the minimum, as in the number of labels in the set. The labels defined in r are *required* by the box. Obviously, a type where all labels are required is useful.

To find out the useful types for the whole topology expression as a parallel branch, we need to consider other parts of the topology, for example, the downstream. Suppose the downstream of box A is box B requiring r'. The labels might come from two origins: *guaranteed* by box A, or provided in the combined input and bypassed across box A with flow inheritance (see Section 4.2). If a required label is guaranteed by the upstream, no additional demand needs to be made. Now, the output from box A contains the labels in v plus the excess labels from the input, which means the labels in v are guaranteed even in the minimum input case.

We then add the labels in $r' \backslash v$ to the overall required label set and depend on flow inheritance to carry them over, so that the output from box A is acceptable by box B. However, some labels cannot be carried over, namely those in $r \backslash v$, because the upstream has consumed them. We mark these labels *discarded* to remind ourselves that they are unavailable downstream, and if the downstream indeed requires any of them, we have found a type error.

$$\begin{array}{ccccc} \Sigma & \in & Sig & = & \mathcal{P}(Map) \\ (r \to \omega) & \in & Map & = & Type \times Otype \\ \omega & \in & Otype & = & \mathcal{P}(Ovar) \\ v[\delta] & \in & Ovar & = & Rtype \times \mathcal{P}(Field) \end{array}$$

$$\forall \Sigma : (r \to \omega_1) \in \Sigma \wedge (r \to \omega_2) \in \Sigma \implies \omega_1 = \omega_2$$

$$\forall \Sigma, \forall (r \to \omega) \in \Sigma, \forall v[\delta] \in \omega : v \cap \delta = \emptyset \wedge (r \backslash BTag) \subseteq v \cup \delta.$$

Figure 4.2: Signature for route inference structure.

Signatures for Route Inference

A box may feature multiple output types. Because the discarded label set is calculated by subtracting the output type from the input type, each of the guaranteed label sets will be accompanied by its own discarded label set. The combination of one required label set with a set of guaranteed-discarded label set pairs is like an extended version of the box signature.

Multiple boxes may be combined in a parallel composition, where each branch has its own extended box signature. The topology can be further extended with more compositions into a general topology expression, but the idea of a set of extended box signatures suffices. And again, it looks like an extended version of the network signature.

In conclusion, the structure for a signature for route inference, abbreviated *sig*, as in Figure 4.2. We also name the extended box signature as a *map*. The difference between a sig and a network signature is that the discarded label sets can be found only in the output types of a sig. We also use a set structure instead of a partial function to make it easier to formalise the construction of a sig, while still maintaining the input uniqueness by a constraint, and use the format $r \to \omega$ instead of (r, ω) to denote the maps of a sig. Furthermore, we use $v[\delta]$ instead of (v, δ) to denote the individual variants of the output type.

Signature Inference Rules

The judgement $e : \Sigma$ means the route inference assigns the topology expression e with the sig Σ. Sig inference rules have a name prefix SI.

Boxes

For boxes, the inference rule merely extends the box signature with the discarded label sets to obtain the sig.

$$\frac{}{\texttt{box}\ (r \to \tau) : \{r \to \{v[(r \backslash BTag) \backslash v] \mid v \in \tau\}\}} \quad \text{SI-BOX}$$

Parallel Compositions

The sig inference of a parallel composition uses the routing information, if present, to filter the branch sigs. The union of the branch sigs is the result of the sig inference. To maintain the sig invariants, a custom union is used. The rules are as follows.

$$\frac{(Rtype)e_1 | (Rtype)e_2 : \Sigma}{e_1 | e_2 : \Sigma} \quad \text{SI-BAR1}$$

In the rule above, the whole class *Rtype*, i.e. the set containing all possible types, is used as the initial routing information to aid the sig inference. This effectively causes the rule below not to filter any maps away from the final sig.

$$\frac{\begin{array}{c} e_1 : \Sigma_1 \\ e_2 : \Sigma_2 \\ \Sigma_i' = \{(r \to \omega) \in \Sigma_i \mid \{r\} \sqsubseteq \tau_i\} \quad (i = 1, 2) \end{array}}{(\tau_1)e_1|(\tau_2)e_2 : \Sigma_1' \sqcup \Sigma_2'} \quad \text{SI-BAR2}$$

where

$$\Sigma_1 \sqcup \Sigma_2 \stackrel{\text{def}}{=} \left\{r \to \bigcup_{(r\to\omega)\in\Sigma_1\cup\Sigma_2} \omega \mid r \in \text{Ins}(\Sigma_1) \cup \text{Ins}(\Sigma_2)\right\}.$$

and

$$\text{Ins}(\Sigma) \stackrel{\text{def}}{=} \{r \mid (r \to \omega) \in \Sigma\}.$$

The two definitions above will also be used in various situations later.

Note that only the parallel composition inference rules will increase the number of maps in a sig. In another word, the presence of multiple maps in a sig implies the existence of parallel compositions in the topology.

Serial Compositions

Unlike the type transformation rules, dealing with sigs for serial compositions is a lot more complex. To make the wording more concise, we assume the serial composition $e_1..e_2$ where $e_1 : \Sigma_1$ and $e_2 : \Sigma_2$. For each map $(r_1 \to \omega_1) \in \Sigma_1$, we need to augment r_1, the required label set, to make all output types accepted by Σ_2. Four questions need to be answered: 1) what labels to add, 2) whether the augmented input creates a reroute, 3) what behaviour can be observed with Σ_1 and the augmented input, and 4) what are the final outputs. We will answer them one by one, and finally present the route inference rule.

The first question, what labels to add, is a tough one. ω_1 may contain multiple output variants. To adapt an output variant $v[\delta] \in \omega_1$ to fit into Σ_2, we add into v zero or more labels not present in $v \cup \delta$, so the resulting v' is a subtype of some input types in Σ_2. We keep the added labels minimum, so every label in v' is required by Σ_2, or otherwise guaranteed by Σ_1.

However, because Σ_2 may contain multiple maps, each variant in turn may be adapted in multiple ways. Collecting these options of added labels results in a set of label sets per output variant. We name this set *requirement variants* of the corresponding output variant, and define the function below for calculating it:

$$\text{RV}(v[\delta], \Sigma_2) \stackrel{\text{def}}{=} \{\epsilon \subseteq \mathit{Field} \mid \epsilon \cap (v \cup \delta) = \emptyset \wedge \{v \cup \epsilon\} \sqsubseteq \text{Ins}(\Sigma_2)\}.$$

Now, to make the whole map valid to combine with Σ_2, the augmentation to r_1 should satisfy at least one requirement variant per output variant. By selecting an arbitrary requirement variant for each output variant and calculating the union of them, we have one sample of augmentation. All possible augmentations are captured by the function below:

$$\text{PA}(\left\{\overrightarrow{v_i[\delta_i]}\right\}, \Sigma_2) \stackrel{\text{def}}{=} \left\{\bigcup_i \epsilon_i \mid \forall i : \epsilon_i \in \text{RV}(v_i[\delta_i], \Sigma_2)\right\},$$

where the set-of construct $\{\overrightarrow{\ }\}$ is defined in Section 4.2. Note that

$$\text{PA}(\emptyset, \Sigma_2) = \{\emptyset\}$$

according to the formula. This is because when there is no range for i, the $\forall$-quantified criterion always holds, generating infinite cases to evaluate the pattern, which is a union over an nonexistent range, resulting in $\emptyset$ by definition. The set of an infinite number of $\emptyset$ is $\{\emptyset\}$. The meaning of this result is as follows: for a map with no outputs, there is one possible augmentation, which is to add nothing to the input. The fact that e_1 produces no outputs does not indicate a type error. Only when one of the output variants cannot match with Σ_2 by any means will the result be $\emptyset$, which signals a type error.

The second question, whether a reroute is created, originates from the nature of parallel compositions. As mentioned in Section 4.4, multiple maps are a sign of parallel compositions. When we augment the input type, we may have created a better match elsewhere in Σ_1, rendering the selected map $r_1 \to \omega_1$ ineligible. These cases should be avoided. As a result, the eligible augmented inputs relative to the map $(r_1 \to \omega_1) \in \Sigma_1$ and the sig Σ_2 are

$$\mathrm{AI}_0(\Sigma_1, (r_1 \to \omega_1), \Sigma_2) \stackrel{\text{def}}{=} \{r_1 \cup \epsilon \mid \epsilon \in \mathrm{PA}(\omega_1, \Sigma_2) \wedge r_1 \in \mathrm{BM}(r_1 \cup \epsilon, \mathrm{Ins}(\Sigma_1)\}\,,$$

where the function BM has been defined in Section 4.2. If the function PA returns $\emptyset$, AI_0 also evaluates to $\emptyset$, effectively disabling the map in question.

We can now also move upwards to the sig level and produce the overall eligible augmented inputs with the function below:

$$\mathrm{AI}(\Sigma_1, \Sigma_2) \stackrel{\text{def}}{=} \{r \mid r \in \mathrm{AI}_0(\Sigma_1, (r_1 \to \omega_1), \Sigma_2) \wedge (r_1 \to \omega_1) \in \Sigma_1\}\,.$$

The third question, how Σ_1 transforms the augmented inputs, is answered with flow inheritance. Unlike in the type transformations where all possible cases, with or without flow inheritance, are covered, we only have the base cases available in a sig, where the input types are the required label sets. To imply other cases where flow inheritance carries unused fields across, we employ the evaluation judgement $r \xrightarrow{\Sigma} \omega$ to mean the actual input r produces the outputs ω according to the sig Σ, which is governed by the rule below:

$$\frac{\begin{array}{l}\Sigma_0 = \{(r_0 \to \omega_0) \in \Sigma \mid r_0 \in \mathrm{BM}(r, \mathrm{Ins}(\Sigma))\} \\ \omega = \bigcup_{(r_0 \to \left\{\overrightarrow{v_i[\delta_i]}\right\}) \in \Sigma_0} \left\{\overrightarrow{(v_i \cup ((r \backslash r_0) \backslash \delta_i))[\delta_i]}\right\}\end{array}}{r \xrightarrow{\Sigma} \omega} \quad \textsc{si-Eval}$$

The rule above considers the parallel composition behaviour and passes the input through only the eligible routes. At the output, it also takes into account the discarded label sets, and adds only those labels in the input type that are neither used nor discarded.

The last question, what the final outputs are, can be answered by applying the rule above in a nested nature, as what we have done in Section 4.2. However, the intermediate outputs now contain the discarded label sets. Apparantly they should be merged with the discarded label sets in Σ_2, but when Σ_2 reproduces any labels in the intermediate discarded label sets, they are excluded from the result.

We have so far figured out what are the new, augmented input types, and how one type transforms into outputs according to the sig of the topology expression. Finally, the sig inference

rule for serial compositions is presented as follows:

$$\frac{\begin{array}{c} e_1 : \Sigma_1 \\ e_2 : \Sigma_2 \\ \Sigma = \left\{ r \to \bigcup_i \omega_i \mid r \in \mathrm{AI}(\Sigma_1, \Sigma_2) \ \wedge \ r \overset{\Sigma_1}{\to} \left\{ \overrightarrow{v_i[\delta_i]} \right\} \ \wedge \right. \\ \left. \forall i : \left(v_i \overset{\Sigma_2}{\to} \left\{ \overrightarrow{v_{j_i}[\delta_{j_i}]} \right\} \ \wedge \ \omega_i = \left\{ \overrightarrow{v_{j_i}[\delta_{j_i} \cup (\delta_i \backslash v_{j_i})} \right\} \right) \right\} \\ \Sigma \neq \emptyset \end{array}}{e_1 .. e_2 : \Sigma} \quad \text{SI-DOTDOT}$$

Serial Replications

Similar to the approach in Section 4.2, we define the route inference for serial replications using that for serial compositions. The benefit we have now, comparing to then in Section 4.2, is that we can directly manipulate the sig, avoiding the necessity of the special binding tag $\Lambda_{e*\gamma}$.

First of all, trim the sig of the operand topology expression so it does not contain any maps requiring inputs that are unconditionally terminated:

$$\mathrm{C}(\Sigma, \gamma) \overset{\text{def}}{=} \{ (r \to \omega) \in \Sigma \mid \nexists r_0 \in \mathrm{dom}(\gamma) : \ r \sqsubseteq r_0 \ \wedge \ \neg\gamma(r_0) \} .$$

Then, extract the part of any intermediate sig that terminates, including dead ends where there is no output:

$$\begin{aligned} \mathrm{T}(\Sigma, \gamma) \overset{\text{def}}{=} \Big\{ r \to \omega' \mid (r \to \omega) \in \Sigma \ \wedge \ (\omega = \omega' = \emptyset \vee \\ \omega' = \{ v[\delta] \in \omega \mid \exists r_0 \in \mathrm{dom}(\gamma) : \ v \sqsubseteq r_0 \} \neq \emptyset \Big\} . \end{aligned}$$

The part of the sig that does not terminate should continue onto the serial composition with downstream replicas:

$$\begin{aligned} \mathrm{NT}(\Sigma, \gamma) \overset{\text{def}}{=} \Big\{ r \to \{ v[\delta] \in \omega \mid \forall r_0 \in \mathrm{dom}(\gamma) : \ r \sqsubseteq r_0 \implies \gamma(r_0) \} \\ \mid (r \to \omega) \in \Sigma \ \wedge \ \omega \neq \emptyset \Big\} . \end{aligned}$$

The process goes on as long as the combined resulting sig keeps growing. When the growth stops, the process ends by adding into the result the special cases where the input types match the terminating pattern immediately:

$$\mathrm{TI}(\gamma) \overset{\text{def}}{=} \{ r \to \{ r[\emptyset] \} \mid r \in \mathrm{dom}(\gamma) \} .$$

The complete sig inference rule is as follows:

$$\frac{\begin{array}{c} e : \Sigma \\ \Sigma_1 = \mathrm{C}(\Sigma, \gamma) \\ \Sigma'_1 = \mathrm{T}(\Sigma_1, \gamma) \\ \forall i > 1 : \ \Sigma'_i = \mathrm{T}(\Sigma_i, \gamma) \sqcup \Sigma'_{i-1} \\ \forall i \geq 1 : \ \Sigma''_i = \mathrm{NT}(\Sigma_i, \gamma) \\ \forall i \geq 1 : \ (e_1 : \Sigma''_i \ \wedge \ e_2 : \Sigma_1) \implies (e_1 .. e_2 : \Sigma_{i+1}) \\ n > 1 \\ \Sigma'_n = \Sigma'_{n-1} \end{array}}{e * \gamma : \Sigma'_n \sqcup \mathrm{TI}(\gamma)} \quad \text{SI-STAR}$$

Parallel Replications

The sig inference for a parallel replication $e!\lambda$ helps require the label λ from the input. It then reuses the evaluation judgement $r \xrightarrow{\Sigma} \omega$ defined in Page 37 to obtain the result. Note that special care is needed if $\lambda \in BTag$ which affects the subtype relation, hence the term $\{\lambda\} \cup r \sqsubseteq r$.

$$\frac{e : \Sigma \qquad \tau = \{\{\lambda\} \cup r \mid r \in \mathrm{Ins}(\Sigma) \wedge \{\lambda\} \cup r \sqsubseteq r\} \neq \emptyset}{e!\lambda : \left\{ r \to \omega \mid r \in \tau \wedge r \xrightarrow{\Sigma} \omega \right\}} \quad \textsc{si-Ex}$$

Subnetworks

The programmer provides a signature for each subnetwork in the program. To meet the expectation of the programmer, the route inference for subnetworks simply extends the signature with discarded label sets to form a sig, and leaves its correctness to be checked by the type check procedure.

The rule below reuses the sig inference rules for boxes to construct the sig.

$$\frac{\mathrm{dom}(\sigma) = \{\overrightarrow{r_i}\} \qquad \forall i : \texttt{box}\ (r_i \to \sigma(r_i)) : \Sigma_i}{\texttt{net}\ (\sigma)\ \texttt{connect}\ e : \bigsqcup_i \Sigma_i} \quad \textsc{si-Subnet}$$

Route Inference Rules

The judgement $\tau, e \longrightarrow_{\mathrm{RI}} e'$ expresses that the topology expression e, provided the set of applicable input types τ, is transformed to e' by route inference. In e', all parallel compositions have the form $(\pi_1)e_1|(\pi_2)e_2$.

The set τ does not only contain the types accepted by e: it is in fact the complement of the set of types that the route inference determines inapplicable, and will be used to exclude certain types from the routing information. In some cases where exclusion should be suppressed, τ will be the universal set *Rtype*, as can be seen from some rules below.

Route inference rules have a name prefix RI.

Boxes

No transformation is required to perform the route inference for boxes, the basic unit of the topology.

$$\frac{\exists r_0 \in \tau_0 : r_0 \sqsubseteq r}{\tau_0, \texttt{box}\ (r \to \tau) \longrightarrow_{\mathrm{RI}} \texttt{box}\ (r \to \tau)} \quad \textsc{ri-Box}$$

Parallel Compositions

The route inference rules for parallel compositions trigger the route inference for each branch and then collect the inputs of the branch sigs as the routing information. The set of applicable types is used to initiate the route inference, which filters the accepted input types of each branch.

$$\frac{\tau_0, (\tau_0)e_1|(\tau_0)e_2 \longrightarrow_{\mathrm{RI}} (\tau_1)e_1'|(\tau_2)e_2'}{\tau_0, e_1|e_2 \longrightarrow_{\mathrm{RI}} (\tau_1)e_1'|(\tau_2)e_2'} \quad \textsc{ri-Bar1}$$

$$\frac{\begin{array}{c}\tau_i'' = \{r \in \tau_i \mid \exists r_0 \in \tau_0 : \; r_0 \sqsubseteq r\} \quad (i = 1, 2) \\ \tau_i'', e_i \longrightarrow_{\text{RI}} e_i' \quad (i = 1, 2) \\ e_i' : \Sigma_i \quad (i = 1, 2) \\ \tau_i' = \{r \in \text{Ins}(\Sigma_i) \mid \exists r_0 \in \tau_0 : \; r_0 \sqsubseteq r\} \quad (i = 1, 2)\end{array}}{\tau_0, (\tau_1)e_1|(\tau_2)e_2 \longrightarrow_{\text{RI}} (\tau_1')e_1'|(\tau_2')e_2'} \quad \text{RI-BAR2}$$

Serial Compositions

Instead of merely triggering the route inference for the two operand topology expressions, we will perform some optimisations during the route inference for serial compositions.

From the discussion in Section 4.4, we are able to find out which maps in Σ_1, if any, cannot match with Σ_2 by any means, using the function PA which computes possible augmentations. Because multiple maps originate from the underlying parallel compositions, by removing the input types of the non-matchable maps from the appropriate underlying routing information, we would be able to delete them completely from the sig.

To do so, we first infer the signatures of both operand topology expressions, and figure out which input types should be removed. Using the reduced input types, we trigger the route inference for the left operand topology expression. If the underlying parallel composition that produced the non-matching branch is not encapsulated in a subnetwork, the map would be removed from the updated sig of e_1'. However, the rule below still succeeds if the map cannot be removed, and potential errors, if any, will be detected during type check.

$$\frac{\begin{array}{c}e_1 : \Sigma_1 \\ e_2 : \Sigma_2 \\ \tau_1 = \{r \mid (r \to \omega) \in \Sigma_1 \wedge \text{PA}(\omega, \Sigma_2) \neq \emptyset \wedge \exists r_0 \in \tau_0 : \; r_0 \sqsubseteq r\} \\ \tau_1, e_1 \longrightarrow_{\text{RI}} e_1' \\ \mathit{Rtype}, e_2 \longrightarrow_{\text{RI}} e_2'\end{array}}{\tau_0, (e_1..e_2) \longrightarrow_{\text{RI}} e_1'..e_2'} \quad \text{RI-DOTDOT}$$

Serial Replications

The route inference for serial replications performs the same optimisation step as for serial compositions. The functions C and NT are as defined in Section 4.4 and PA in Section 4.4.

$$\frac{\begin{array}{c}e : \Sigma \\ \Sigma_1 = \text{C}(\Sigma, \gamma) \\ \Sigma_1'' = \text{NT}(\Sigma_1, \gamma) \\ \tau_1 = \{r \mid (r \to \omega) \in \Sigma_1'' \wedge \text{PA}(\omega, \Sigma_1) \neq \emptyset \wedge \exists r_0 \in \tau_0 : \; r_0 \sqsubseteq r\} \\ \tau_1, e \longrightarrow_{\text{RI}} e'\end{array}}{\tau_0, e * \gamma \longrightarrow_{\text{RI}} e' * \gamma} \quad \text{RI-STAR}$$

Parallel Replications

No special care is needed for the route inference of parallel replications.

$$\frac{\pi_0, e \longrightarrow_{\text{RI}} e'}{\pi_0, e!\lambda \longrightarrow_{\text{RI}} e'!\lambda} \quad \text{RI-EX}$$

Subnetworks

Subnetworks define boundaries of components. The route inference does not propagate the input type information into the subnetworks, but uses the declared input types to perform the route inference.

$$\frac{\mathrm{dom}(\sigma), e \longrightarrow_{\mathrm{RI}} e'}{\tau_0, \texttt{net}\ (\sigma)\ \texttt{connect}\ e \longrightarrow_{\mathrm{RI}} \texttt{net}\ (\sigma)\ \texttt{connect}\ e'} \quad \text{RI-SUBNET}$$

The rule above doubles as the process to prepare an initial S-NET$_{\mathrm{A}}$ program with the routing information, before performing the type check.

Chapter 5

Operational Semantics

5.1 Notation

We define the operational semantics of S-Net by a binary transition relation '$\rightarrow$' on tuples (p, M), where p denotes a record (or record stream) and M an S-Net program. Two tuples (p, M) and (q, M') are related with respect to '$\rightarrow$' iff the S-Net programm M on input r and produces s. We use '$\rightarrow$' in infix notation and we reverse the order of elements of the right-hand side tuple. The latter is a purely syntactical measure to illustrate the flow of data: A typical transition of the form $(p, M) \rightarrow (M', q)$ is to be read as

- On input p to M
- M computes on p and
- M turns into M' due to changes of internal states.
- M' outputs q.

As a program without synchrocells does not have a mutable internal state, we obviously allow $M = M'$.

Throughout this chapter we shall also use the following notation: A single italic letter, as for example p, denotes a single record. A single letter with an arrow on top denotes a stream of records, e.g. $\vec{p}$. The concatenation of two streams is denoted by $\vec{p} ++ \vec{q}$ (appending $\vec{q}$ to $\vec{p}$), concatenation of an element and a stream by $p \triangleleft \vec{q}$ (prefixing $\vec{q}$ by p) and $\vec{q} \triangleright p$ (appending p to $\vec{q}$). An n-fold concatenation, denoted by $++_{i=1}^{n}(\vec{p}_i)$, expands to $\vec{p}_1 ++ \vec{p}_1 ++ ... ++ \vec{p}_n$.

5.2 Id, Empty and Map Rule

The following rules are merely 'helper-rules'. The Id rule allows records to be passed on without any alteration. An empty stream has no effect on any component, as shown by the Empty rule. The Map rule allows to derive a semantics for streams if component rules are defined on single records.

$$\textsc{Id} \quad : \quad \frac{}{(p, \texttt{id}) \rightarrow (\texttt{id}, p)}$$

$$\textsc{Empty} \quad : \quad \frac{}{(\epsilon, M) \rightarrow (M, \epsilon)}$$

$$\textsc{Map} \quad : \quad \frac{\forall_{i=1}^{n} : (p_i, M_i) \rightarrow (M_{i+1}, \vec{q_i}) \quad \vec{s} = +\!+_{i=1}^{n}(\vec{q_i})}{(\vec{p}, M_1) \rightarrow (M_{n+1}, \vec{s})}$$

5.3 Boxes and Primitive Boxes

Boxes are the only components in S-Net that are able to process and modify data of record fields. On input of a single record, a box may produce an arbitrary number (including zero) of result records. A distinctive feature of S-Net is the fact that function f of rule Box may be implemented in an arbitrary programming language.

$$\textsc{Box} \quad : \quad \frac{f(p) = \vec{q}}{(p, \texttt{box}f) \rightarrow (\texttt{box}f, \vec{q})}$$

A synchrocell, the only stateful component in S-Net, is a facility to store and merge records that match a user-defined pattern. For this, synchrocells are parametrised over two patterns. Any inbound record is matched against these patterns. If a record matches a previously unmatched pattern, it is stored by the synchrocell, the pattern is marked as matched and no output is produced (rule Syncs). An index at the lower-right corner of the synchrocell indicates this. If a pattern was matched before, the record is simply output again, as described by the Syncn rule. A record that matches the last remaining unmatched pattern triggers a merge of stored and inbound record. After merging and outputting the resulting record, the synchrocell will act as an identity component (Syncm rule). If a record immediately matches both patterns, the record passes unmodified and the synchrocell again will act as an identity component, as shown by the Synci rule.

$$\textsc{Syncs} \quad : \quad \frac{\text{ismatch}(\sigma_a, p_a) \wedge \neg\text{ismatch}(\sigma_b, p_a)}{(p_a, [\![\sigma_a, \sigma_b]\!]) \rightarrow ([\![\sigma_a, \sigma_b]\!]_{p_a}, \epsilon)}$$

$$\textsc{Syncn} \quad : \quad \frac{\text{ismatch}(\sigma_a, p) \wedge \neg\text{ismatch}(\sigma_b, p)}{(p, [\![\sigma_a, \sigma_b]\!]_{p_a}) \rightarrow ([\![\sigma_a, \sigma_b]\!]_{p_a}, p)}$$

$$\textsc{Synci} \quad : \quad \frac{\text{ismatch}(\sigma_a, p) \wedge \text{ismatch}(\sigma_b, p)}{(p, [\![\sigma_a, \sigma_b]\!]) \rightarrow (\texttt{id}, p)}$$

$$\textsc{Syncm} \quad : \quad \frac{\text{ismatch}(\sigma_b, p)}{(p, [\![\sigma_a, \sigma_b]\!]_{p_a}) \rightarrow (\texttt{id}, \text{merge}(p_a, p))}$$

The filter (see rule Filter) is a versatile instrument to modify the structure of records. Filters can split records, rename, copy or discard fields and tags and even insert new tags and modify their value. The behaviour of each filter is controlled by a pattern σ and by a user-defined list α of aforementioned actions. The pattern defines which constituents of inbound records are accessible by the filter actions. Only fields and tags that are present in the pattern may be used

in the action list. On arrival of a record, the actions are applied to the inbound record and all resulting records are output.

$$\textsc{Filter} \quad : \quad \frac{\vec{q} = \text{apply}(\alpha, p)}{(p, [\sigma \texttt{ -> } \alpha]) \rightarrow ([\sigma \texttt{ -> } \alpha], \vec{q})}$$

5.4 S-Net Combinators

The four S-Net combinators, which are used to construct networks from boxes, primitive boxes and networks, are very briefly introduced here.

The serial combinator connects the output of its left operand to the input of its right operand. The output of the right operand thereby forms the output of the newly constructed network.

$$\textsc{Ser} \quad : \quad \frac{(\vec{p}, M) \rightarrow (M', \vec{p'}) \quad (\vec{p'}, N) \rightarrow (N', \vec{q})}{(\vec{p}, M..N) \rightarrow (M'..N', \vec{q})}$$

The choice combinator creates a new network by connecting its two operands in parallel. Records are routed to either of the operands depending on which operand is a more specific match. The specificity of the match is determined by analysing the type of the inbound record and the input type of the operands. If both operands match equally well, one is chosen non-deterministically. The choice combinator comes in two variants, deterministic and non-deterministic. The deterministic variant preserves the order of inbound and resulting outbound records. If record order is not a concern, the non-deterministic variant may be used. The Dchoice rule formalises the behaviour of the deterministic choice combinator.

$$\textsc{Dchoice} \quad : \quad \frac{\begin{array}{c}(\vec{p_{l_i}}, \vec{p_{r_i}})_{i \in \{1,..,n\}} = \text{lrpsplit}(\vec{p}, \tau_M, \tau_N) \\ \forall i \in \{1,..,n\} : (\vec{p_{l_i}}, M_i) \rightarrow (M'_{i+1}, \vec{q_{l_i}}) \quad (\vec{p_{r_i}}, N_i) \rightarrow (N'_{i+1}, \vec{q_{r_i}})\end{array}}{(\vec{p}, M_1||N_1) \rightarrow (M_{n+1}||N_{n+1}, {+\!\!+}_{i=1}^{n}(\vec{q_{l_i}} {+\!\!+} \vec{q_{r_i}}))}$$

The inbound stream $\vec{p}$ is divided into pairs of sub-streams $(\vec{p_{l_i}}, \vec{p_{r_i}})$ such that each (potentially empty stream) $\vec{p_{l_i}}$ (resp. $\vec{p_{r_i}}$)) contains records matching the input type of the left (resp. right) operand network. These pairs are processed by the operand networks, whose internal state may change due to synchrocells, and produce streams $\vec{q_{l_i}}$ and $\vec{q_{r_i}}$ as result. The output streams are concatenated to a result stream and represent the result for one input pair. The overall outbound stream is constructed by concatenating result streams of all input pairs. The behaviour of the non-deterministic choice combinator is expressed by the NDchoice rule.

$$\textsc{NDchoice} \quad : \quad \frac{\begin{array}{c}\vec{p_l}, \vec{p_r} = \text{lrsplit}(\vec{p}, \tau_M, \tau_N) \\ (\vec{p_l}, M) \rightarrow (M', \vec{q_l}) \quad (\vec{p_r}, N) \rightarrow (N', \vec{q_r})\end{array}}{(\vec{p}, M|N) \rightarrow (M'|N', \text{ndzip}(\vec{q_l}, \vec{q_r}))}$$

Any inbound stream of records $\vec{p}$ is divided into two separate sub-streams $\vec{p_l}$ and $\vec{p_r}$, such that each record in $\vec{p_l}$ (resp. $\vec{p_r}$) matches the inbound type of the left (resp. right) operand of the choice combinator. The operand networks, whose internal state may change due to synchrocells, produce $\vec{q_l}$ and $\vec{q_r}$ as results. These result streams are non-deterministically merged, i.e. both streams may be arbitrarily interleaved.

The star combinator requires an operand and a pattern for operation. Any inbound record that matches the pattern is immediately output. If the record does not match the pattern, it is sent to the operand. At the output of the operand, the same process repeats. If the result matches the patten, it is output, otherwise a new instance of the operand is spawned and the record is sent to it. The star combinator is available as deterministic and non-deterministic combinator. The deterministic variant guarantees that any result of an earlier input will exit the network before any other result of a later input, i.e. outbound streams of multiple inbound records are not interleaved and in the same order of their corresponding inbound records.

$$\textsc{Dstar} \quad : \quad \frac{(\vec{p}, (M..M*\{\sigma\})|||[\sigma \texttt{ -> } \tau_\sigma]) \rightarrow (M', \vec{q})}{(\vec{p}, M**\{\sigma\}) \rightarrow (M', \vec{q})}$$

$$\textsc{NDstar} \quad : \quad \frac{(\vec{p}, (M..M*\{\sigma\})|[\sigma \texttt{ -> } \tau_\sigma]) \rightarrow (M', \vec{q})}{(\vec{p}, M*\{\sigma\}) \rightarrow (M', \vec{q})}$$

The split combinator routes inbound records to different instances of its operand depending on the value of a specified tag. The name κ of the tag that determines the instance of the operand is given as parameter to the split combinator. Each instance of the operand stays associated with the combinator and is reused whenever a record is processed that holds the appropriate instance value.

$$\textsc{Dsplit} \quad : \quad \frac{\text{value}(\kappa, p) = j \quad (p, N_j) \rightarrow (N'_j, \vec{q})}{(p, N!!\kappa) \rightarrow (N!!\kappa, \vec{q})}$$

$$\textsc{NDsplit} \quad : \quad \frac{\forall_{i=1}^{n} p_i \in \vec{p} : \quad \text{value}(\kappa, p_i) = v_i \quad (p_i, N_{v_i}) \rightarrow (N'_{v_i}, \vec{q_i})}{(\vec{p}, N!\kappa) \rightarrow (N!\kappa, \text{ndzip}(\vec{q_1}, ..., \vec{q_n}))}$$

5.5 Algorithms

Here we present the algorithms that are used wthin the rules. Where an implementation is not providing any more insight, either because it is trivial or too technical, it is left out and only the signature is shown.

eval

eval :: pattern → record → Bool

ismatch

ismatch :: pattern → record → Bool
ismatch σ p =
  $p \preccurlyeq \sigma \wedge$ eval σ r

score

score :: pattern → record → $\mathbb{N}_0 \cup \{-1\}$

bestmatch

bestmatch :: record → pattern → pattern → set of pattern
bestmatch $r\ \sigma_l\ \sigma_r$ =
  $\{\sigma_i \mid i,j \in \{l,r\} \wedge \text{score}\ \sigma_i\ r \geq \text{score}\ \sigma_j\ r\}$

prefix

prefix :: stream → pattern → pattern → (stream, stream)
prefix $\vec{p}\ \sigma_l\ \sigma_r$ =
  case $\vec{p}$ of
    $\epsilon \rightarrow (\epsilon,\epsilon)$
    $p : \vec{ps} \mid \text{ndsel (bestmatch}\ p\ \sigma_l\ \sigma_r) == \sigma_l \rightarrow (p{:}\vec{q}, \vec{r})$
    $p : \vec{ps} \mid \textit{otherwise} \rightarrow (\epsilon, \vec{p})$
    where
      $(\vec{q}, \vec{r}) = \text{prefix}\ \vec{ps}\ \sigma_l\ \sigma_r$

splitOnePair

splitOnePair :: stream → pattern → pattern → (stream, stream, stream)
splitOnePair $\vec{p}\ \sigma_l\ \sigma_r$ =
  case $\vec{p}$ of
    $\epsilon \rightarrow (\epsilon, \epsilon, \epsilon)$
    $p : \vec{ps} \rightarrow (\vec{q_l}, \vec{q_r}, \vec{r})$
    where
      $(\vec{q_l}, \vec{t}) = \text{prefix}\ \vec{p}\ \sigma_l\ \sigma_r$
      $(\vec{q_r}, \vec{r}) = \text{prefix}\ \vec{t}\ \sigma_r\ \sigma_l$

lrpsplit

lrpsplit :: stream → pattern → pattern → (stream of streams, stream of streams)
lrpsplit $\vec{p}\ \sigma_l\ \sigma_r$ =
  case $\vec{p}$ of
    $\epsilon \rightarrow (\vec{\epsilon},\vec{\epsilon})$
    $p : \vec{ps} \rightarrow (\vec{l_p} : \vec{l_{pl}}, \vec{r_p} : \vec{r_{pl}})$
    where
      $(\vec{l_p}, \vec{r_p}, \vec{q}) = \text{splitOnePair}\ \vec{p}\ \sigma_l\ \sigma_r$
      $(\vec{l_{pl}}, \vec{r_{pl}}) = \text{lrpsplit}\ \vec{q}\ \sigma_l\ \sigma_r$

lrsplit

lrsplit :: stream → pattern → pattern → (stream, stream)
lrsplit $\vec{p}\ \sigma_l\ \sigma_r$ =
  let $(\vec{p_l}, \vec{p_r}) = \text{lrpsplit}\ \vec{p}\ \sigma_l\ \sigma_r$ in
    (flatten $\vec{p_l}$, flatten $\vec{p_r}$)

merge

merge :: record → record → record
record $\vec{p}\ \vec{q}$ =
  $\vec{p} \cup ((\vec{q} \setminus BT(\vec{q})) \setminus (\vec{p} \cap \vec{q}))$

apply

apply :: record → action → stream

Chapter 6

Denotational Semantics

6.1 Foundations

In this chapter on denotational semantics of S-Net we describe how an S-Net program can systematically be associated with a Haskell function that maps an input stream of data in form of a lazy list into an output stream, again represented by a lazy list.

First of all, we establish an abstract representation of S-Net networks that is amenable to this undertaking. The Haskell module `SNet` described below is the outcome of this work.

```
module SNet where

import Data.Set hiding (filter,map)
import qualified Data.Set as Set (filter)
```

A few types are required for the structural representation of S-Net types and data structures. Records consist of record elements. Record elements are label-value pairs. To fully qualify a label, it needs to be annotated to be referring to a field (`F`), a tag (`T`) or a binding tag (`B`).

```
data Ord label ⇒ Element label = F label | T label | B label deriving (Eq,Ord)
data Ord label ⇒ RecEl label value = RecEl
   { label :: Element label
   , value :: Either value ℤ
   }
type Record label value = Set (RecEl label value)

isBindingTag :: Ord label ⇒ Element label → Bool
isBindingTag (B _) = True
isBindingTag  _    = False
```

Because only the values of fields and tags are handed into box functions (i.e. not the labels), the order in which values are specified for the box function is relevant. The output type of a box function is a list of alternatives. Actual output must thus be tagged with a number to specify which of the output alternatives is the type of the actual record produced.

```
type BoxFunc value = [Either value ℤ] → [(ℕ,[Either value ℤ])]
```

A record type is simply the set of (annotated) labels. The type for boxes reflect the orderedness of the arguments of box functions. What is referred to as a *function type* in this text is a generalization of the box types; the order constraint is dropped. Finally, a *full type* is a set of alternative function types.

```
data Ord label ⇒ RecType label = RecType { recTpEls :: Set (Element label) }
   deriving (Eq,Ord)
```

```
data Ord label ⇒ BoxType label = [Element label] ↦ [[Element label]]
data Ord label ⇒ FunType label = RecType label ⇝ Set (RecType label)
  deriving (Eq,Ord)
data Ord label ⇒ FulType label = FulType { funTypes :: Set (FunType label) }

generalize :: Ord label ⇒ BoxType label → FunType label
generalize (dom↦ran) = RecType (fromList dom) ⇝ fromList (map (RecType ∘ fromList) ran)
```

The order constraint on `label` is required for the use of `Set`s. When a record element is inserted into a record, any old record element with the same label is overwritten. This behaviour can be implemented by defining the order (and equality) of record elements as the order on their labels.

```
instance Ord label ⇒ Eq (RecEl label value) where
  x = y = label x = label y
instance Ord label ⇒ Ord (RecEl label value) where
  compare x y = compare (label x) (label y)
```

Deriving the type of a record can be implemented as, for every record element, keeping only the label and throwing away the value. A subtype is defined conform the S-Net typesystem definition. The operator `<:=` checks whether its left-hand side is a subtype of its right-hand side. The function `match` calculates the 'match strength' of a record with a type. This can be used for routing where the network alternative with the 'strongest' match should be the recipient of the matched record.

```
typeOf :: Ord label ⇒ Record label value → RecType label
typeOf = RecType ∘ mapMonotonic label

(≺) :: Ord label ⇒ RecType label → RecType label → Bool
(RecType sub) ≺ (RecType super)
  = Set.filter isBindingTag sub = Set.filter isBindingTag super
  ∧ super ⊆ sub

match :: Ord label ⇒ FulType label → Record label value → ℤ
match (FulType nettp) rec = let rt = typeOf rec in maximum
  [ if rt ≺ t then size (recTpEls t) else −1
  | (t ⇝ _) ← elems nettp ]
```

Records can be seen as lookup tables. Given a label and a record, the function `element` performs that lookup. The function does not check whether the record actually contains an element with the given label, so the presence of said label must be determined in the context in which `element` is used.

```
element :: Ord label ⇒ Element label → Record label value → RecEl label value
element e = head ∘ toList ∘ Set.filter ((= e) ∘ label)
```

Records must be translated to and from box input and output respectively. The function `toBox` looks at the input type of a box and selects the required elements in the required order from a given record. The function `fromBox` looks up the correct output type from the list of output types of a box and translates the box output back into a record (according to the output type).

```
toBox :: Ord label ⇒ BoxType label → Record label value → [Either value ℤ]
toBox (dom↦_) r = map (value ∘ flip element r) dom

fromBox :: Ord label ⇒ BoxType label → (ℕ,[Either value ℤ]) → Record label value
fromBox (_↦ran) (i,es) = fromList $ zipWith RecEl (ran!!i) es
```

For flow inheritance, records must be split into two parts: the 'through' part—that matches the input type of a network—and the 'around' part. For convenience, the operators `-|->` and `>-|-` are defined to break up records into through and around and to join them back together again. Note that the union in `>-|-` favours its left-hand side, i.e. if a label occurs in both its operands, the one from the right-hand operand is discarded.

```
data Ord label ⇒ Inh label value = Inh { through, around :: Record label value }

class Inheritable τ where
   (≻) :: Ord label ⇒ τ label → Record label value → Inh label value
instance Inheritable RecType where
   (≻) (RecType t) = uncurry Inh ∘ partition (flip member t ∘ label)
instance Inheritable BoxType where (≻) t           = (generalize t ≻)
instance Inheritable FunType where (≻) (dom⇝ _) = (dom ≻)

(≺) :: Ord label ⇒ Inh label value → Record label value → Record label value
inh ≺ res = res ∪ around inh
```

Finally, the type for networks can be defined. It is parametric in the annotation of every network element and in the representation for boxes, filters, patterns and record element labels. This way, the data structure can be reused for different compilation stages and run-time environments.

```
data Ord label ⇒ Net annot box fil pat label
  = Box      { notes :: annot, nettype :: FulType label,
               boxtype :: BoxType label, box :: box }
  | Filter   { notes :: annot, nettype :: FulType label,
               fil :: fil }
  | Sync     { notes :: annot, nettype :: FulType label,
               inheritType :: RecType label, patterns :: [pat] }
  | C_Seq    { notes :: annot, nettype :: FulType label,
               subs :: [Net annot box fil pat label] }
  | C_Par    { notes :: annot, nettype :: FulType label,
               subs :: [Net annot box fil pat label], deterministic :: Bool }
  | C_Star   { notes :: annot, nettype :: FulType label,
               body :: Net annot box fil pat label,  deterministic :: Bool,
               terminator :: [pat] }
  | C_Split  { notes :: annot, nettype :: FulType label,
               body :: Net annot box fil pat label,  deterministic :: Bool,
               selector :: Element label }
```

6.2 Indices

In order to identify different positions in a network, we add an indexing scheme to the language implementation. To this end, we define a hierarchical data structure for indices and some operators to translate index transactions to possibly more familiar terminology.

Although this module does not require anything from any library, it has one import statement, to hide the definition of `break` coming from the prelude.

```
{-# LANGUAGE RecordWildCards #-}
module Index (
   Index,
   initialIndex,
   isPrefixOf,
   start, step, stop,
   loop, continue, break,
   idxSplit, idxMerge,
   idxInf,
   POrdering(..),
   idxCompare,
   uncheckedIterators
) where

import Prelude hiding (break)
```

The `Index` data type reflects the hierarchical nature of S-Net networks. The simplest index

form, just enumerates the sequential compositions. If the output of network a is the input of network b then b's index is a's index plus one. However, since there are stars and parallel constructions (parallel composition and bling), there are more constructors required.

```
data Index
  = Seq   { value :: Int, next :: Index }
  | Iter  { value :: Int, next :: Index }
  | Split { alt   :: Int, next :: Index }
  | EndIndex
  deriving Eq
```

`Seq` denotes a position within a sequential composition. `Iter` denotes the iteration (or unfolding instance) of a star combinator. `Split` denotes the choice out of numbered alternatives made when flowing through a bling or parallel composition. Because all these constructors have a `next` field in which the index within a subnet is placed, a terminator is required, which is the role of `EndIndex`.

First, we define a few functions for internal use to implement the interface functions. Some straightforward property checkers make the implementation easier.

```
hasNext, isSeq, isIter, isSplit :: Index → Bool
hasNext = (EndIndex /=)
isSeq   i = case i of Seq   {} → True; _ → False
isIter  i = case i of Iter  {} → True; _ → False
isSplit i = case i of Split {} → True; _ → False
```

Because the inner-most index element describes the inner most network element, it is the most frequently changed element. The function `applyToDeepest` takes a predicate, a function and an index. It recurses down through the index to find occurrances where the predicate holds. If the predicate holds for an index element, and it does *not* hold for any of its children (`next`), the function is applied to this element. If the predicate holds for one of the children of a node, `applyToDeepest` recurses into the children and alters them.

```
applyToDeepest :: (Index → Bool) → (Index → Index) → Index → Index
applyToDeepest p f = fst ∘ applyToDeepest' p f
  where
  applyToDeepest' p f i | hasNext i ∧ r = (i { next = n },r)
                        | p i           = (f i, True)
                        | otherwise     = (i, False)
    where
    (n,r) = applyToDeepest' p f (next i)
```

With this function, we can now define `append` and `prune` operations to add or take away levels of hierarchy.

```
append :: Index → Index → Index
append t' = applyToDeepest (not ∘ hasNext) (λEndIndex → t')

prune :: (Index → Bool) → Index → Index
prune p = applyToDeepest p (λi → EndIndex)
```

We can make a 'new' index, by starting with an empty one:

```
initialIndex = EndIndex
```

One index is considered the prefix of another when it occurs entirely in the other, except for its terminator. In other words, i is a prefix of j when j is equal to i, or is i with a deeper index.

```
isPrefixOf :: Index → Index → Bool
EndIndex `isPrefixOf` j        = True
i        `isPrefixOf` EndIndex = False
i        `isPrefixOf` j        = i { next = EndIndex } = j { next = EndIndex } ∧
  (next i) `isPrefixOf` (next j)
```

The `start`, `step` and `stop` functions transform the index to, respectively, add an element to

signify a sequence, increment the deepest sequential element of the index (pruning its children) and prune the deepest sequential index.

```
start, step, stop :: Index → Index
start = append (Seq { value = 0, next = EndIndex })
step  = applyToDeepest isSeq (λi → i { value = succ (value i), next = EndIndex })
stop  = prune isSeq
```

Unfolding stars can be seen as loops. The indices reflect this by keeping an iteration count. The functions `loop`, `continue` and `break` perform the correct transformations on indices.

```
loop, continue, break :: Index → Index
loop     = append (Iter { value = 0, next = EndIndex })
continue = applyToDeepest isIter (λi → i { value = succ (value i), next = EndIndex })
break    = prune isIter
```

Finally, to facilitate parallel compositions and blings, the indices can be transformed by `idxSplit` (which takes the number of the alternative branch a subnet is in) and `idxMerge`.

```
idxSplit :: Int → Index → Index
idxSplit a = append (Split { alt = a, next = EndIndex })

idxMerge :: Index → Index
idxMerge = prune isSplit
```

Given two indices, the minimum is defined as the greatest common prefix. This may be slightly counterintuitive, because normally $\min(i,j) \in \{i,j\}$. In this case, we say that $\min(i,j) \leq i,j$ and $\nexists k(\min(i,j) < k < i,j)$.

```
idxInf :: Index → Index → Index
idxInf EndIndex j = EndIndex
idxInf i EndIndex = EndIndex
idxInf i•Split { alt = ai } j•Split { alt = aj } | ai /= aj = EndIndex
                                                 | ai = aj = i { next = idxInf (next i) (next j) }
idxInf i•Iter { value = vi } j•Iter { value = vj } = case compare vi vj of
  LT → i
  EQ → i { next = idxInf (next i) (next j) }
  GT → j
idxInf i•Seq { value = vi } j•Seq { value = vj } = case compare vi vj of
  LT → i
  EQ → i { next = idxInf (next i) (next j) }
  GT → j
idxInf i j = error $ "Index difference that shouldn't occur: " ++ show (i,j)

data POrdering = Less | Equal | Equivalent | Greater deriving (Eq,Show)
idxCompare :: Index → Index → POrdering
idxCompare EndIndex EndIndex = Equal
idxCompare EndIndex j = Less
idxCompare i EndIndex = Greater
idxCompare i•Split { alt = ai } j•Split { alt = aj } | ai /= aj = Equivalent
                                                    | ai = aj = idxCompare (next i) (next j)
idxCompare i•Iter { value = vi } j•Iter { value = vj } = case compare vi vj of
  LT → Less
  EQ → idxCompare (next i) (next j)
  GT → Greater
idxCompare i•Seq { value = vi } j•Seq { value = vj } = case compare vi vj of
  LT → Less
  EQ → idxCompare (next i) (next j)
  GT → Greater
idxCompare i j = error $ "Index difference that shouldn't occur: " ++ show (i,j)

uncheckedIterators :: Index → [Int]
uncheckedIterators  EndIndex = []
```

```
uncheckedIterators i•Iter {} = value i : uncheckedIterators (next i)
uncheckedIterators i         = uncheckedIterators (next i)
```

6.3 Putting it all together

The purpose of this module is to ultimately translate an S-Net network into a function that transforms an input stream into an output stream. In- and output streams are represented as lists of records.

```
{-# LANGUAGE RecordWildCards #-}
module Semantics where

import Data.Set (elems,empty,union,singleton)
import Data.List (partition)
import Data.Either (partitionEithers)

import SNet
import Index
```

The idea behind this formulation of the semantics of S-Net is that non-determinism can be expressed as a deterministic choice made outside of the network. This choice is represented by an *oracle*.

```
type Oracle = [(Index,ℕ)]
```

Every tuple in the oracle represents a single choice for a specific element of the network (indicated by the element's index). The semantics of a network can now be defined by stating that for all desirable behaviours (i.e. outputs), there exists an oracle, such that this output is produced. Similarly, for all oracles, there is an unambiguous output.

The representation for S-Net networks required to describe the semantics has index annotations on every network element, boxes are represented by box functions, filters are normal functions and patterns are functions that project records onto zero-or-one type that was matched by the record. Throughout the semantic description, labels are assumed to be represented by strings.

```
type SemNet value = Net
  Index                                  — annotation
  (BoxFunc value)                        — repr. boxes
  (SemRec value → [SemRec value])        — repr. filters
  (SemRec value → [RecType String])      — repr. patterns
  String                                 — label type
type SemRec value = Record String value
```

Non-deterministic mergers are constrained by the outside world. When there is a non-deterministic parallel composition *inside* a deterministic parallel composition, the non-deterministic merger must respect the order of *substreams* caused by the splitter of the deterministic parallel composition. As an example, suppose that the input stream of a deterministic parallel composition consists of the records a, b, c and d. Furthermore, suppose that a, b and d all flow into the same branch of the composition, whereas c flows into another. The responses to a and b may be arbitrarily mixed up, according to the non-determinism inside their branch. However, any response to a or b can never come out *after* any response to d. There should, thus, be some sort of barrier between a and b on the one hand and d on the other. This is done by modelling streams as lists of substreams (and substreams as lists of records). The stream from this example for the branch mentioned would look like this: $[[a, b], [d]]$.

```
type Stream value = [[SemRec value]]
```

Having a representation for a stream, the semantics of a network can now be written as a function of an oracle and that network. In other words, the function `lift` describes unambiguously the behaviour of any network.

```
lift :: Oracle → SemNet value → (Stream value → Stream value)
```

Because there are network elements with a state (most notably the SynchroCell), a few helper functions are defined. The function `stateT` takes a function of some state and an input to the output and a new state and it produces a function that takes an entire substream of input. The function `liftState` generalizes this, so that a state transforming function and an initial state are translated into a function from an input stream to an output stream.

```
stateT :: (s → i → (o,s)) → (s → [i] → ([o],s))
stateT _ s [] = ([],s)
stateT f s (i:is) = (o:os,s'')
  where
    (o ,s' ) = f s i
    (os,s'') = stateT f s' is

liftState :: (s → i → ([o],s)) → s → [[i]] → [[o]]
liftState f s = map concat ∘ fst ∘ (stateT ∘ stateT) f s
```

The function `lift` can now be defined. Firstly, the primitive networks can be lifted. None of the primitive networks can contain any non-determinism, so the oracle has no influence on their lifting. Record elements not required as input are flow inherited over boxes and filters. Since boxes and filters can produce zero-or-many records and SynchroCells can produce zero-or-one record in responce to any input record, the resulting *lists* of records must be concatenated. This concatenation does not change the barriers that substreams represent, because only the responses to records from the same substream are concatenated (i.e. `concat` is applied inside the outer `map`). Since Synchrocells are stateful, `liftState` is used, so that the SynchroCell semantics can be formulated on a per-input basis. This is delegated to the function `sync`, which is described below.

```
lift _ Box     {..} = map (concat ∘ map (applyBox ∘ (boxtype ≻)))
  where
  applyBox inh = map ((inh ≺) ∘ fromBox boxtype)
      (box $ toBox boxtype $ through inh)
lift _ Filter  {..} = map (concat ∘ map (applyFilter ∘ selectInput))
  where
  applyFilter inh = (map (inh ≺) ∘ fil ∘ through) inh
  selectInput rec = (head ∘ elems ∘ funTypes) nettype ≻ rec
lift _ Sync    {..} = liftState sync (SyncS empty patterns (Just undefined))
```

Secondly, sequential composition can be lifted. This is now the very simple case of applying all functions resulting from lifting the subnets in order.

```
lift o C_Seq   {..} = foldr (.) id $ reverse $ map (lift o) subs
```

Lastly, the compositions that may involve non-determinism are all delegated to their respective functions.

```
lift o C_Par   {..} = liftPar   notes o deterministic subs
lift o C_Star  {..} = liftStar o (C_Star {..})
lift o C_Split {..} = liftSplit notes o deterministic body selector
```

Before dealing with specific combinator lifting, splitting and merging of streams is discussed. All remaining combinators use splitters and mergers in some way.

Splitting requires a function `router` that determins where every record should go. For every record in a stream, `router` gives the branch number the record should flow into and the record itself. The determinism of the corresponding merger is important for the behaviour of the splitter. If the merger (and thus the splitter) is deterministic, barriers should be introduced whenever the

next record flows into a different branch than the current. Non-deterministic splitters may simply divide all records of a substream over the branches, one substream per branch. Every branch filters the stream for records that are relevant to it, i.e. have been tagged with the branch's number. Finally, all branches are lifted and applied to their filtered input. The result of the split is a list of decisions per substream, i.e. the branch numbers, and the list of resulting streams, one for every branch.

```
splitStream
   :: Oracle
   → (Stream value → [[(ℕ,SemRec value)]]) -- router
   → Bool                                  -- deterministic?
   → [SemNet value]                        -- branches
   → Stream value                          -- in-stream
   → ([[ℕ]],[Stream value])                -- split decisions and branch results
splitStream o router det nets ss = (cs,bs)
   where
   ssR = (map runs ∘ router) ss
   cs  = map (map fst) ssR
   bs  = zipWith filterLift [0..] (map (lift o) nets)
   filterLift bn f = f $ (if det then concat else map concat) $ map (map snd ∘ filter ((≡bn).fst)) ssR
   runs [] = []
   runs rs•((i,_):_) = (i,map snd run) : runs rem
      where
      (run,rem) = span ((i≡) ∘ fst) rs
```

Deterministic merging is quite straightforward. Given a list of splitter decisions per substream and a list of branch results, the latter must be merged one substream at a time. The decision for every substream consists of as many branch numbers as there were consecutive sequences the original substream was cut up into by the splitter. This means that when the first decision in the list of decisions is an empty list, the end of the original substream has been reached.

```
detMerge :: [[ℕ]] → [[[a]]] → [[a]]
detMerge [] _ = []
detMerge ([]:bs) ss = [] : detMerge bs ss
detMerge ((c:cs):bs) ss = (s++os) : oss
   where
   (pss,(s:css):nss) = splitAt c ss
   (os:oss) = detMerge (cs:bs) (pss ++ css:nss)
```

Non-deterministic merging is best understood on a per-substream basis. For every substream coming into a non-deterministic splitter, every branch receives exactly one (possibly empty) substream. Thus, merging the head of every branch result, results in the first substream coming out of this merger. To facilitate for infinite branches (which may occur in case of a ! combinator) in the face of lazyness, the function `nonDetMerge` is given the highest branch number for every substream (`ms`) as an argument. Notice that when the oracle dictates taking an element from an empty substream, this substream is discarded from consideration and the oracle's dicision is reinterpreted in the face of one less substream.

```
nonDetMerge :: [ℕ] → [ℕ] → [[[a]]] → [[a]]
nonDetMerge _     []    _ = []
nonDetMerge o (m:ms) ss = s : nonDetMerge (drop (length s) o) ms (map tail ss)
   where
   s = mergeSubs o (take (m + 1) $ map head ss)
   mergeSubs    _     [] = []
   mergeSubs (c:cs) ss | null rs   = mergeSubs (c:cs) (pss++nss)
                       | otherwise = head rs : mergeSubs cs (pss ++ tail rs:nss)
      where
      c' = c `mod` (length ss)
      (pss,rs:nss) = splitAt c' ss
```

In the parallel composition, there is non-determinism both at both sides; splitter and merger. When more than one branch in the parallel composition matches the type of the input record, the oracle dictates the choice for a branch. Non-determinism on the side of the merger is more comprehensive, i.e. *all* choices are dictated by the oracle. In order to distinguish between oracle decisions for the splitter and the merger, the splitter is controlled by positive integers in the oracle, whereas the merger is controlled by the negative numbers. Because 0 would be a valid choice for both, merger decisions are offset by one. The router function `sigma` makes the routing choices according to S-Net's type-driven routing rules.

```
liftPar idx o determin nets ss | determin  = detMerge cs bs
                               | otherwise = nonDetMerge o_merge (map maximum cs) bs
  where
  (o',o'')           = partition ((=idx).fst) $ filter (isPrefixOf idx.fst) o
  (o_split,o_merge') = partition (≥ 0) (map snd o')
  o_merge            = map (abs ∘ succ) o_merge'

  (cs,bs) = splitStream o'' router determin nets ss
  router  = liftState σ o_split

  σ :: [ℕ] → SemRec value → ([(ℕ,SemRec value)],[ℕ])
  σ (o:os) r = if null ms then ([(m,r)],o:os) else
     ([((m:ms)!!(o `mod` 1 + length ms),r)], os)
    where
    (m:ms) = [ i | (s,i) ← zip scores [0..], s = best ]
    scores = map (flip match r) ts
    best   = maximum scores
    ts     = map nettype nets
```

The lifting of the star is very similar to the lifting of the parallel composition. Elements that match the exit patterns of a star are always routed to bypass the route into the further-unfolding star. Besides the different routing semantics and a bit of plumbing network indices, the implementation of `liftStar` is very similar to that of `liftPar`.

```
liftStar :: Oracle → SemNet value → Stream value → Stream value
liftStar o net•C_Star {..} ss | deterministic = detMerge cs bs
                              | otherwise     = nonDetMerge (map snd o') (repeat 1) bs
  where
  (o',o'') = partition ((= notes).fst) o
  (cs,bs)  = splitStream o'' router deterministic [body',idNet notes] ss
  router   = map $ map (λr → (fromEnum $ any (not ∘ null ∘ ($ r)) terminator,r))
  index    = continue notes
  body'    = let notes = index in C_Seq { subs = [body,net { notes = index }], ∘ . }

idNet :: Index → SemNet value
idNet i = Filter
  { notes   = i
  , nettype = FulType $ singleton $ RecType empty⇝ singleton (RecType empty)
  , fil     = return
  }
```

The lifting of split combinators, again, is very similar to that of parallel composition. The only actual differences are the fact that splitting is always deterministic and the routing information is taken from the records themselves.

```
liftSplit :: Index → Oracle → Bool → SemNet value → Element String → Stream value → Stream value
liftSplit idx o determin net sel ss | determin  = detMerge cs bs
                                    | otherwise = nonDetMerge (map snd o') (map maximum cs) bs
  where
  (o',o'') = partition ((=idx).fst) $ filter (isPrefixOf idx.fst) o
  (cs,bs)  = splitStream o'' router determin
```

```
                    (zipWith (λi n → n { notes = idxSplit i idx }) [0..] $ repeat net)
                    ss
  router = map $ map $ λr → (((λ(↘ i) → i) ∘ value ∘ element sel) r, r)

data SyncState value = SyncS { accum :: SemRec value,
                               rempats :: [SemRec value → [RecType String]],
                               inheritFirst :: Maybe (RecType String) }

sync :: SyncState value → SemRec value → ([SemRec value], SyncState value)
sync s r | null (rempats s) = ([r], s)
sync SyncS {..} r
  = (if null rp' then [accum'] else [], SyncS accum' rp' iF')
  where
  accum'     = if null ms then accum else (accum ∪ add)
  stripped   = (through ∘ (RecType (foldr union empty ms)) ≻)) r
  (ms, rp')  = partitionEithers [ if null pr then ↘ p else ↙ (recTpEls $ head pr) | p ← rempats,
                                  let pr = p r ]
  (iF', add) = maybe (Nothing, stripped)
     (λt → if typeOf r ≺ t then (Nothing, r) else (Just t, stripped))
     inheritFirst
```

Chapter 7

Metadata

7.1 Purpose

Since S-Net is not limited to a specific box language, inlined box language code needs to be compiled with a specific box language compiler. To do this the S-Net compiler requires more information than just box names: organisation and location of source code files need to be specified just as the name and location of the compiler executable or specific compiler flags to name just a few issues. Likewise the compilation of S-Net code itself often requires manipulation beyond the functional properties described by the S-Net semantics. For example, we may want to plug-in graphical observer components into certain streams to snoop for records. For all these purposes we introduce metadata.

S-Net metadata is written in XML format and must be in the S-Net namespace

`snet-home.org`.

Sections 7.3, 7.4 and 7.5 describe the form of metadata and the currently supported metadata elements in greater detail.

Any S-Net source code file may include any number of metadata XML-trees. Metadata can be included either in S-Net source code file or in a separate metadata file(s). If metadata is provided within the S-Net source code file, then the metadata can be declared in any position in the S-Net code where the S-Net syntax would allow an S-Net definition. Hence, metadata can exist either outside of any S-Net definitions or inside the body of a net definition. Metadata included from other files is considered to be in the same level as metadata written outside any definitions.

7.2 Organisation of Metadata

This section outlines how individual metadata elements are associated with S-Net definitions in an S-Net code file. Each net and box element must have a name attribute. The value of the name attribute is considered as location of the related S-Net definition. The name value consists of one S-Net net or box name identifying the net or box the metadata should be attached to. The name may be preceded by any number of S-Net net names separated by slashes (/). These names represent the path from current scope of the metadata element to the scope of the associated S-Net definition. A value that does not contain any slashes refers to the corresponding S-Net definition in the current scope. Elements with empty name attribute or no name attribute at all are erroneous.

Each net or box element under another net element is associated to element according to the name attribute starting from the scope in which the metadata element is declared. Scopes are defined exactly as they are in normal S-Net code. If a name does not point to any S-Netdefinition then the metadata element and all elements within its scope are ignored.

One S-Net entity may have any number of metadata elements attached to it. Meta data related to single entity, but which is declared in different locations, is merged into one metadata element. In case of conflicting metadata values, the compilation process must be aborted. Any unknown XML-elements or known element in wrong location in the metadata tree must be ignored.

Fig. 7.1 gives an example of how metadata is associated to the S-Net definitions. The metadata inside network `C` in file `A.snet` is associated with the box `B`. It solely contains an interface declaration for the box, which differs from the default interface specified before. Metadata from another file `metadata.md` could be used to change the name of the box given to the language interface implementation.

```
file A.snet:

<?xml version="1.0"?>
<metadata xmlns="snet-home.org" >
  <interface name="C4SNet" default="true" />
</metadata>

net A {
  net C {
    box B( ( N, <T>) -> ( M, <T>));

    <metadata xmlns="snet-home.org">
      <box name="B">
        <interface value="SAC4SNet">
      </box>
    </metadata>

  } connect B;
} connect C;


file metadata.md:

<metadata xmlns="snet-home.org" >
  <box name="A/C/B">
    <boxfun value="funB">
  </box>
</metadata>
```

Figure 7.1: Example: S-Net with metadata both inlined and located in a separate file

The XML element `metadata`, which merely acts as a textual container to separate metadata from regular S-Net code, can be omitted if it contains a single `net` or `box` element. If metadata lacks an XML-declaration, it is assumed to be XML version 1.0 by default.

7.3 Network Metadata

The XML element `net` is a container for metadata attached to a specific network definition; it is associated with exactly one S-Net network definition. This network definition is either determined by the textual location of the metadata or by the `name` attribute, as described in Section 7.2. Net

elements themselves may contain one or more `net`, `box`, `interface` or `observer` elements as their children. So, network metadata serves a dual purpose: attaching "real" metadata, i.e. additional information beyond the S-Net semantics, to the network definition and reflecting the scoping that S-Net network definitions create through nesting on the level of metadata.

7.4 Box Metadata

The XML element `box` acts as a container for metadata attached to a specific S-Net box declaration. As in the case of the `net` element, the association of the XML data to an S-Net box declaration is defined either through textual location in the S-Net source code or by the element's `name` attribute. Box elements may contain the following box-specific metadata elements. In each case the individual metadata value may either be defined through the element's `value` attribute or as character data enclosed within the element.

The interface element

The `interface` element describes the box language interface used by the box implementation. The box language interface must have been declared before using the `interface` element, as described in section 7.5. A box language interface specification can be omitted if a default interface has been declared before.

The boxfun element

The `boxfun` element can be used to map the S-Net level box name to a box implementation language level function name. By default both are expected to have the same name, but this metadata element allows any mapping between the two.

Other XML-elements

As box language interfaces might need language specific information at compile time for box code wrapper generation, S-Net metadata also offers means to provide this data.

All XML-elements declared as immediate child elements of a box element that are not explicitly known to the compiler are passed to the box language interface wrapper code generation function as a special data structure. The elements are stored as key-value pairs, where the key is the name of the element and the value is either the character data inserted as a child of the element or value of attribute named `value`. All other attributes or child elements of the element are ignored.

All element names defined above and all elements whose names begin with *snet_* are reserved for use of S-Net language and S-Net compiler implementations and should not be used for language interfaces.

7.5 Interface Metadata

The XML element `interface` is used to declare box language interfaces used in the S-Net application. Any box language interface in S-Net must be declared with this element prior to its use in box metadata (see Section 7.4). More information on S-Net interfaces in general can be found in Chapter 8.

An interface element must have the attribute `name` whose value is the name of the box language interface. The interface element may have an attribute `default`, whose value can be either `true` or `false`. This attribute is used to declare a default box language interface, which is used for

any box with no explicitly defined box language interface. By default a declared box language interface is not the default interface, hence the default value of the `default` attribute is `false`. If multiple box language interfaces are defined, only one of the interfaces may be declared the default interface.

Chapter 8

Interfaces

8.1 Introduction

As a pure coordination language, S-Net provides no means to describe computations, apart from the simple operations on tag values supported by filters (see Section 3.3 for details). For the description of non-trivial computations as represented by user-defined boxes S-Net relies on an external compute language, called the *box language*. S-Net is not bound to a specific box language and a single network may well combine boxes implemented in different box languages.

The freedom in the choice of a box language and the proper separation of concerns between conventional box code and the S-Net coordination layer require a well defined interface to function properly in practice. This *box interface* implements a contract between S-Net and the box language. It determines such issues as how S-Net calls box language functions to process records, how fields, tags and binding tags are accessed by the box language code and how the box language creates output records and makes them available to S-Net in turn. We describe the box interface in more detail in Section 8.2.

Yet, the box interface is not the only interface relevant to S-Net. S-Net networks are constructed hierarchically in a compositional style. Regardless, of how many layers of composition a network has, it always remains a SISO or Single-Input-Single-Output streaming network. At some point any S-Net network is completed and ready to be *deployed*. Deployment here means that the network is turned into executable (binary) code ready to accept input data from the execution environment. Deployment of an S-Net network is similar to linking in a conventional compiler toolchain. However, deployment immediately raises the question as how the deployed network is connected to the execution environment in order to accept data on its global input stream and to emit data back to the execution environment via the network's global output stream. This is the task of the *network interface*, which connects S-Net networks to the outside world. We describe the network interface in more detail in Section 8.3.

In fact, network interface and box interface are much more related than it seems at first glance: The network interface must retrieve data from the execution environment and store it in memory such that the box language code can properly access it. Likewise, memory data structures created by box language code need to be serialised by the network interface before the data can leave the world of S-Net. Consequently, a complete box language binding must implement both the box interface and the network interface. We describe two such interface implementations, one for the machine-level imperative language C and one for the functional array programming language SaC [20] in Sections 8.4 and 8.5, respectively.

8.2 The Box Interface

Proper interaction between S-Net and box languages requires that box languages provide a certain infrastructure and box code obeys to certain rules and restrictions. For example, a box implementation is expected to compute a function mapping an input record to none, one or multiple output records. In particular, the box code must not interact with its execution environment, although a box language may well provide the necessary means to do so. Furthermore, the box code must be reentrant and refrain from leaving any information in persistent storage. Whereas purely functional languages encourage such a programming style in a natural way, the utilisation of imperative languages requires some discipline from the programmer.

The type signature of a box serves as the only specification of the box's behaviour and its interface to S-Net. In particular, concrete types of data and, hence, the representation of data in memory are unknown to S-Net. To overcome this lack of information all data sent via S-Net streams must be boxed. In fact, S-Net only transfers pointers or references into a shared heap memory while only the box language code is actually able to interpret the data behind a pointer. The only exception to this rule are tags, for which S-Net and box languages share a common integer representation.

Fig. 8.1 demonstrates the interplay between the S-Net box signature and a C-binding box implementation. Let us consider an S-Net box declaration

```
box example ((a,b,<t>) -> (c,<t>) | (x,y,z));
```

that declares a box named `example`, which maps records containing fields `a` and `b` and a tag `t` to none, one or more records that either contain a field `c` and a tag `t` or fields `x`, `y` and `z`.

```
snethandle_t *example( snethandle_t *snethandle,
                       void *a,
                       void *b,
                       int t)
{

  /* regular C code to compute c and t */

  snethandle = snetout( snethandle, 0, c, t);

  /* more C code to compute x, y and z */

  snethandle = snetout( snethandle, 1, x, y, z);

  return( snethandle);
}
```

Figure 8.1: Example of a pseudo C implementation of an S-Net box

Fig. 8.1 shows the corresponding pseudo C function definition. We assume a function that is named like the S-Net box. The first parameter of the function, regardless of the box signature, is a handle to an S-Net control data structure. This handle is opaque to the box language programmer and must be provided by the box language interface implementation. The following parameters are the record fields of the input record type in the sequence of their specification in the box signature. Here, it becomes immediately apparent why the sequence of record entries in a box signature does matter and why we distinguish between type signatures and box signatures in S-Net. As explained before, record fields are of some pointer type while tags are of type integer. Although it may be useful to name the parameters according to the field names in the box signature, this is not a formal requirement.

The box language function may contain any box language code, but must obey to the rules stated above: it neither must draw information from external sources other than its arguments nor must it interact with the outside world in any way.

S-Net boxes may return none, one or multiple records in response to a single input record. As a consequence, we cannot utilise the normal function result for producing an output record because that would enforce a one-to-one correspondence between input and output records. Instead, we use a special function `snetout` that must be provided by the S-Net box language binding. This function receives the S-Net handle provided as an argument to the box language function as its first argument. The second argument is a number referring to the output variant of the box signature. This information is vital for any implementation to interpret the output record. The remaining arguments are the record fields in the sequence of their declaration in the corresponding output variant of the box signature.

Calls to the `snetout` function may occur anywhere in the box language code. Taking C as box language as in our example means that these calls may occur in loops and branches. Depending on the values of the input record the number of calls to `snetout` may vary and so the number of records produced in response. The `snetout` function returns the S-Net handle. Eventually, the box language function returns the given handle.

8.3 The Network Interface

The network interface connects S-Net networks to the outside world by connecting the global input and the global output stream of a deployed network to its execution environment. Assume a standard UNIX-like execution environment, the network interface allows us to choose between different communication modes when a deployed network is run: standard Unix I/O streams (`stdin` for the global input stream, `stdout` for the global output stream), specific input/output files or socket communication. The choice is made via a command line option of when instantiating an S-Net network, the option `-h` shows the available options.

Regardless of the mode of communication with the execution environment the network interface makes use of an XML-like textual representation of records both for input and for output. We consistently use the XML namespace

`snet-home.org`

for all XML-like specifications. Any XML element with no child elements can be shortened according to the XML specification. White space between elements included in the examples below is introduced solely for the purpose of readability; any white space between elements may be omitted in actual messages. Each message consist of the XML elements explained below. Any extra elements and/or attributes are ignored. Missing attributes cause the element to be ignored. Fig. 8.2 shows an example.

Each message contains the XML element `record` as its root element. If the message does not contain an XML-declaration, it is assumed to be XML version 1.0. The `record` element represents one single S-Net record. The `type` attribute specifies the type of the record. Currently, two record types are supported: `data` and `terminate`. While the former represents a (data) record in the sense of S-Net, i.e. a set of label-value pairs, the latter must always be the final record in any sequence of input records. It signals the end of the input data sequence to the S-Net network and causes its orderly termination.

Whereas `terminate` records do not contain any further information, data records have attributes describing individual fields, tags and binding tags as well as technical information that determines the way data received on the input string is deserialised into memory data structures or serialised back on the output stream.

```
<?xml version="1.0" ?>
<record  xmlns="snet-home.org" type="data" mode="textual" interface="SAC4SNet" >
  <field label="F">data</field>
  <field label="X" mode="binary" interface="C4SNet">binary data</field>
  <tag   label="T">0</tag>
  <btag  label="B">1</btag>
</record>

<?xml version="1.0" ?>
<record xmlns="snet-home.org" type="terminate" />
```

Figure 8.2: Example: S-Net message format for reading data on the input stream and for encoding data on the output stream

- The `field` element represents a single S-Net record field. Any `field` element must have a `label` attribute and contain data defining the value of the field. How this data is interpreted and deserialised into a data structure in memory depends on two further attributes of the `field` element:
  - The `interface` attribute specifies the box language interface used for deserialisation and reserialisation; the box language interface establishes the connection between the interpretation of data received on the input stream, the manipulation of data by box implementations and the encoding of data on the output stream.
  - The `mode` attribute determines whether the data is in textual or in binary form; supported modes are `textual` and `binary`. The default mode in case this attribute is omitted is binary. A complete box language interface must support both textual and binary serialisation and deserialisation.
- The `tag` element represents a single S-Net tag. Any `tag` element must have a `label` attribute. If the tag element does contain data, that data is interpreted as the textual representation of an integer number; it determines the tag value. If the data cannot be interpreted as an integer number or the tag element does not contain any data, the tag value is interpreted as zero.
- The `btag` element represents a single S-Net binding tag. It is handled exactly as non-binding tags.

While the `interface` and `mode` attributes can be set for each field element in a data record individually, it is often handy in practice to specify default interfaces and modes for entire records. To achieve this the two attributes can be used with the record element as well. If a field is not associated with a box language interface at all because the attribute is neither used with the field element itself nor with the record element (default), the field is discarded from the record as the associated data cannot be interpreted without this information. In case no mode can be determined for some field element the default mode is binary.

8.4 Interface Implementation: C4SNet

To complement the high-level description of box and network interfaces in the previous sections, we have implemented a simple (and still incomplete) C box language and network interface (`C4SNet`). We utilise this implementation to demonstrate how to use interfaces in practice and to give a general feel for the technicalities involved.

A box interface is implemented by means of two libraries. A data handling library provides all box language specific data handling functionality, which includes aforementioned `snetout` function, as well as copy functions, de-allocation functions and data serialisation and deserialisation functions. In addition to these functions, the second library implements a code generator which produces necessary glue code to enable successful integration of foreign-language box functions into the S-Net runtime system. Whereas the former library provides generic runtime functions (it is therefore linked to the resulting executable file), the code generator is only required at compile time. It is dynamically loaded by the S-Net compiler using a versatile plug-in mechanism.

We will not go into more technical detail here — for a more thorough treatment of interface implementation and its requirements, we refer to the S-Net Implementation Report [18].

To make the S-Net compiler use a specific language interface, the only information a source file needs to provide is the name of the interface implementation. From this the compiler derives library names and the required function names of the interface.

The metadata element `<interface>` makes the name of the interface known to the compiler. It may be specified on a per-box basis inside a `box` metadata block (see Chapter 7 for details). However, in the common scenario where all boxes within a network share the same box interface, it is also possible to conveniently define a default interface. This is achieved by specifying the interface in the top-level metadata block and adding `default="true"` to the element. In this case it is not necessary to annotate boxes with their respective language interface name, as boxes inherit the default interface element automatically. The example in Fig. 8.3 shows how to specify `C4SNet` as default language interface for all boxes.

```
<metadata xmlns="snet-home.org" >
  <interface name="C4SNet" default="true" />
</metadata>

net mini
{
  box do( (A) -> (B, <X>));
  box something( (B, <X>) -> (C));
} connect do .. something;
```

Figure 8.3: The name of a language interface is made known to the S-Net compiler using the metadata element `interface`.

With the interface specified, the S-Net compiler is now able to generate glue code and to link against the interface library. The interface library does not only provide functions that are used for communication with the runtime system, it does also implement functions for data handling inside a box implementation. Data handling in a language like C, where data is referenced by pointers, often has its pitfalls when it comes to passing data from one box to another: As pointers to data do not include any information about the data it is actually pointing to, additional information is often required. This is, for example, the case for arrays, where the extend of axes and the amount of data elements inside the array are important information. Rather then asking a box programmer to implement a custom made solution to this problem, `C4SNet` provides abstract data containers of type `c4snet_data_t`. A data container not only stores the pointer but also encodes information about the referenced data, as for example the base type and, if need be, array shape information. Functions to create, access and modify abstract data containers are provided by the box interface library as well.

In Fig. 8.7 we show sample box implementations for the network shown in 8.6 and demonstrate how to work with data containers. The box functions do not compute anything useful; they are solely designed for illustrative purposes in this document.

Note that for name space separation all functions that are provided by one interface library should share the name of the interface as common prefix.

The `C4SNet` interface supports all C data types (see Fig. 8.4) and provides the functions shown in Fig. 8.5.

Data type	Description
`CTYPE_uchar`	unsigned char
`CTYPE_char`	signed char
`CTYPE_ushort`	unsigned short int
`CTYPE_short`	signed short int
`CTYPE_uint`	unsigned int
`CTYPE_int`	signed int
`CTYPE_ulong`	unsigned long int
`CTYPE_long`	signed long int
`CTYPE_float`	float
`CTYPE_double`	double
`CTYPE_ldouble`	long double
`CTYPE_unknown`	unkown (invalid) data

Figure 8.4: The C interface supports all C base types.

Function	Description
`void C4SNetOut( void *hnd, int variant, ...)`	Send data back to S-Net
`int C4SNetSizeof(c4snet_data_t *data)`	Size of base type
`c4snet_data_t*` `C4SNetDataCreate( c4snet_type_t type, const void *data)`	Create new data container for scalar values
`c4snet_data_t*` `C4SNetDataCreateArray( c4snet_type_t type, int size, void *data)`	Create new data container for arrays
`void *C4SNetDataCopy( void *ptr)`	Copy data container
`void C4SNetDataFree( void *ptr)`	Destroy data container
`void *C4SNetDataGetData( c4snet_data_t *ptr)`	Retrieve data
`c4snet_type_t C4SNetDataGetType( c4snet_data_t *c)`	Return base type
`c4snet_vtype_t C4SNetDataGetVType( c4snet_data_t *c)`	Returns either `VTYPE_simple` or `VTYPE_array`
`int C4SNetDataGetArraySize( c4snet_data_t *c)`	Returns -1 for scalars and number of elements otherwise

Figure 8.5: The C interface provides various functions for data container handling.

For integration with network interfaces for input and output of data, `C4SNet` implements serialisation and de-serialisation functions. The structural record information is encoded in an XML-like syntax as described in Section 8.3. The data that is embedded into this XML record description is passed to the de-serialisation function of the box interface In `C4SNet` data is required to be prefixed with type information. Valid prefixes for supported data types are `(unsigned char)`, `(char)`,

```
#include "C4SNet.h"

void *do( void *handle, c4snet_data_t *A)
{
  c4snet_data_t *B;
  void *result;
  int *A_int;
  double *A_dbl;
  int X;

  /* Using the C4SNet data containers, we can determine the
   * type of data we have received. If need be, we can implement
   * different behaviour depending on argument types, as sketched
   * out below.
   */
  switch( C4SNetDataGetType( A) {
    case CTYPE_int:
      A_int = C4SNetDataGetData( A);
      /* do something using A_int, for example generate a 10-element
       * vector of integers
       * ...
       * and then create a new data container to send it on */
      B = C4SNetDataCreateArray( CTYPE_int, 10, result);
      break;
    case CTYPE_double:
      A_dbl = C4SNetDataGetData( A);
      /* do something else */
      B = C4SNetDataCreate( CTYPE_double, result);
      break;
    default:
      /* something else... */
  }
  /* C4SNetOut is the function to communicate back any
   * results to the S-Net runtime system. Fields are always
   * of type *c4snet_data_t, tags are passed as plain integers.
   *
   * The second argument determines the variant of the output type.
   */
  C4SNetOut( handle, 1, B, X);

  return( handle);
}
```

Figure 8.6: This shows the implementation of box `do`. The `C4SNet` interface provides functions for data handling. To send results back to the runtime system, the function `C4SNetOut` is used.

`(unsigned short)`, `(short)`, `(unsigned int)`, `(int)`, `(unsigned long)`, `(long)`, `(float)`, `(double)`, and `(long double)`. Example record encodings for our example network are shown in Fig. 8.8.

8.5 Interface Implementation: SAC4SNet

The SaC interface `SAC4SNet` is implemented as two separate libraries in the same way as the `C4SNet` interface described above. The integration of SaC as box language, however, requires more information because of the way the SaC and `sac4c` compiler generates C libraries from SaC modules (this process is beyond the scope of this document). For a fully automated compilation process, the `SAC4SNet` code generator requires the metadata elements shown in Fig. 8.9.

```
void *something( void *handle, c4snet_data_t *B, int X)
{
  int i;
  c4snet_data_t *C;
  int *array_B, *result;
  int size;

  size = C4SNetDataGetArraySize( B);
  if( size > -1) { /* C4SNetDataGetArraySize returns -1 for scalars */

    if( C4SNetDataGetType( B) == CTYPE_int)  {
      array_B  = C4SNetDataGetData( B);
      /* do some computation and produce a resulting array */
      C = C4SNetDataCreateArray( CTYPE_int, size, result);
    }
    else {
      /* do something else, for example, create a scalar result */
      C = C4SNetDataCreate( CTYPE_int, result);
    }
  } else {
    /* do something else */
  }

  for( i=0; i < X; i++) {
    C4SNetOut( handle, 1, C);
  }

  return( handle);
}
```

Figure 8.7: Example implementation of S-Net box `something`

The implementation of a box function in SaC requires this function to accept the S-Net handle object as first argument. Results are sent back to the runtime system by a call to the `SNet::out` which follows the same convention as its `C4SNet` counterpart. A sample S-Net and an implementation of a SaC module which contains boxes for the network are shown in Fig. 8.10 and Fig. 8.11.

Data serialisation and de-serialisation is realised using the Fibre print and Fibre scan functions of SaC's standard library. The data in Fibre format follows a `SAC4SNet` specific header (`:::SACFIBRE:` *type dim sizes* `:::`) that encodes the base type of elements, the dimensionality of the array and the extent of axes. Fig. 8.12 shows two examples of record encodings.

```
<record xmlns="snet-home.org" type="data" mode="textual" interface="C4SNet">
  <field label="A" >(int)5</field>
</record>

<record xmlns="snet-home.org" type="data" mode="textual" interface="C4SNet">
  <field label="A" >(double)5.678</field>
  <field label="F" >(int[4])17,18,37,13</field>
</record>
```

Figure 8.8: The type of data is given in parentheses. For arrays the amount of elements is given in square brackets.

Element	Description
`<interface name = "SAC4SNet">`	instructs the compiler to load the `SAC4SNet` code generator and to link against the matching box interface library
`<SACmodule>` *name* `</SACmodule>`	this information is required by the code generator to construct the actual name of of the function as generated by the SaC compiler. Currently this element is mandatory for each box.
`<SACboxfun>` *name* `</SACboxfun>`	this element is only required if the box name as it appears in the network's source code does not coincide with the function name in the SaC source code. In this case one specifies the SaC function name here.

Figure 8.9: The SaC interface specific metadata elements are used by the code generator to automatically construct required function names.

```
net mini
{
  <metadata xmlns="snet-home.org" >
    <box name="do" >
      <SACmodule>mini</SACmodule>
    </box>
    <box name="something" >
      <SACmodule>mini</SACmodule>
      <SACboxfun>foo</SACboxfun>
    </box>
  </metadata>

  box do( (A) -> (B, <X>));
  box something( (B, <X>) -> (C));
} connect do .. something;
```

Figure 8.10: The SaC interface is called `SAC4SNet`. Currently the SaC module name needs to be annotated for each box function using the metadata element `<SACmodule>`
.

```
/* This is the SAC module that implements boxes for the example S-Net */
module mini;

use Structures: all;
use sacprelude: {wrap};

export {do, foo};


/* When implementing boxes in SAC we can make use of (most) SAC
 * features. Overloading, for example, allows us
 * to implement different behaviour for various input (SAC) types,
 * as sketched out below: */
void do( SNet::SNet &handle, int[*] A)
{
  /* do some computation */
  B = A + 1;
  X = 17;

  /* Any field we output has to be converted into SACargs again.
   * This is done by calling the wrap() function from the SAC
   * standard prelude. In contrast, tags can be passed as scalar
   * integers without wrapping.
   */
  SNet::out( handle, wrap( B), X);
}

/* Same function name, this time for arbitrary double arrays: */
void do( SNet::SNet &handle, double[*] A)
{
  /* do some computation */
  B = A + 2.34;

  SNet::out( handle, wrap( B), 24);
}

/* Here, the function name "foo" differs from the
 * box name in the S-Net source code. To still be able to correlate
 * the S-Net box to this function, the metadata element "SACboxfun"
 * is used.
 */
void foo( SNet::SNet &handle, int[*] B, int X)
{
  for( i=0; i < X; i++) {
    /* do some meaningful computation */
    SNet::out( handle, wrap( B), X);
  }
}

void foo( SNet::SNet &handle, double[*] B, int X)
{
  for( i=0; i < X; i++) {
    /* do some meaningful computation */
    SNet::out( handle, wrap( B), X);
  }
}
```

Figure 8.11: Using SAC as box language for S-NET is straight forward. Only very few modifications to existing SAC code is necessary: Functions have to accept the S-NET handle as their first argument and results have to be output using `SNet::out` rather then `return`.

```
<record xmlns="snet-home.org" type="data" mode="textual" >
<field label="A" interface="SAC4SNet">
:::SACFIBRE: float 1 2  :::
[ 0,1:
    1.800000e+01
    1.800000e+01
]
</field>
</record>

<record xmlns="snet-home.org" type="data" mode="textual" >
<field label="A" interface="SAC4SNet">
:::SACFIBRE: int 3 4 1 1  :::
[ 0,3:
 [ 0,0:
  [ 0,0:
   20
  ]
 ]
 [ 0,0:
  [ 0,0:
   20
  ]
 ]
 [ 0,0:
  [ 0,0:
   20
  ]
 ]
 [ 0,0:
  [ 0,0:
   20
  ]
 ]
]
</field>
</record>
```

Figure 8.12: Record data is read and output by `SAC4SNet` in Fibre format.

Chapter 9

Runtime Inspection through Observers

9.1 Introduction

An observer is an S-Net component that makes dynamic information from the execution of an S-Net network available, for example for debugging or tracing. Observers are designed to intervene with the operational behaviour of S-Net as little as possible and to be entirely transparent from an untimed semantic perspective. As the name suggests, observers cannot modify S-Net records, their content or their order in anyway. Observers are not meant to be used as output components of the S-Net network either.

The basic action of an observer is as follows: an observer can be attached to any S-Net stream such that the stream is redirected from its target component to the observer and a new stream connects the observer with the target component. When the observer receives a record it creates a message describing the content of the record and sends it to predefined *listener* via socket communication. After this the record itself is sent to the subsequent runtime component. Instead of sending data to an active listener, the message can likewise be dumped into a file on demand.

Alternatively, S-Net features *interactive observers*. An interactive observer mainly acts just as a regular observer, but it waits for an acknowledgement from the listener before it passes a record on. As a consequence, interactive observers hold back a stream of records for some time and, hence, influence the operational behaviour but not the semantics of an S-Net network.

The level of information that an observer provides can be chosen at compile time. S-Net currently supports three levels: On the lowest level, only the labels of fields, tags and binding tags are sent to the listener. The middle level include values of tags and binding tags. The highest level additionally includes field values making use of the serialisation facility coming with the network interface definition (see Section 8.3 for further information). S-Net observers do not make any assumptions about their listener: Whatever application they communicate with is opaque to them. Listeners are expected to understand the message formats as defined in Section 9.3.

The correspondence between observers and listeners or files can be chosen in various ways. For example, each observer may be connected to its own private listener (file) or multiple observers may share the same listener (file). S-Net takes care of proper synchronisation where necessary; messages are received by listeners according to the semantic rules for sequencing of data flow in S-Net.

9.2 Observer Metadata Declarations

Observers are deliberately not part of the S-Net language itself, but instead they are defined via metadata (see Chapter 7 for complete coverage of metadata). Observers can be attached to any S-Net network definition or box declaration by adding an `observer` element to the corresponding metadata `net` or `box` element.

An `observer` element may have any of the attributes listed below. Default values are used if the attribute is omitted or the attribute value is illegal. Illegal attribute values cause compiler warnings. Due to the dynamic nature of S-Net networks (serial replication and indexed parallel replication) any observer may have multiple dynamic instances. Each instance of an observer inherits exactly the same attributes in this case. Nevertheless, S-Net provides means to distinguish between messages sent by different dynamic instances of observers. Fig. 9.1 shows an example of an observer declaration.

```
net A {

  <?xml version="1.0"?>
  <metadata xmlns="snet-home.org" >
    <box name="B" >
      <observer interactive="true"
                data="tags"
                type="before"
                port="6667"
                address="localhost"
                code="6A6DACC1" />

      <observer file="output.xml"
                data="full"
                type="after"
                code="6A6DACC2" />
    </box>
  </metadata>

  box B( ( N, <T>) -> ( M, <T>));


} connect B;
```

Figure 9.1: Example: S-Net observer metadata.

Any XML element without child elements can be shortened according to the XML specification. White space between elements included in example figures is introduced solely to make the examples more readable and may be omitted.

Attribute `interactive`

The `interactive` attribute specifies whether the observer is an interactive observer or not. Legal values for this attribute are `true` and `false`; default value is "false", i.e. by default observers are non-interactive.

Attribute `file`

The `file` attribute determines whether the observer uses file for dumping information or actually sends the data to a listener. The value of the attribute must be the name of the file. If this

attribute is omitted, an active listener is anticipated and socket communication is used instead. If the file specified in the value of the attribute cannot be opened for writing at runtime, the observer will simply pass records on to the following S-Net component.

If the `file` attribute is specified, then the `interactive` attribute is ignored and the observer is always non-interactive.

Attribute `address`

The `address` attribute can be used to specify the address where the observer sends all data in case socket communication to some listener component is chosen as observer modus operandi. The data must be a proper URL or ip-address. If the observer cannot connect to the address at runtime, it will simply pass records on to the following S-Net component. Default value is `localhost`.

If both the `file` and the `address` attributes are specified, then the `file` attribute takes precedence.

Attribute `port`

The `port` attribute can be used to specify the port used for socket communication. The default port is 6555. If both the `file` and the `port` attributes are specified, then the `file` attribute takes precedence.

Attribute `data`

The `data` attribute specifies the level of information to be sent to the listener or to be dumped into a file. As explained in Section 9.1, S-Net supports three levels of information. With the attribute value `labels` only the labels of fields, tags and binding tags are communicated. The attribute value `tagvalues` additionally includes values of tags and binding tags, and with `allvalues` even the values of fields are included.

Attribute `type`

The `type` attribute specifies the location of the observer with respect to the component it is attached to. Legal values are `before` and `after`. The third value `both` can be used as an abbreviation of adding observers both before and after the component. Both the observers will have exactly the same attributes, apart from the type attribute, of course. Default value is `both`.

Attribute `code`

The `code` attribute can be used to provide any application specific compile time defined data element with the observer message. This attribute can for example be used to identify the observer instance in the listener application or in the dump file.

9.3 Observer Message Formats

We now define the message formats used by S-Net observers for communication with listener applications. Observer messages are in XML format and use the S-Net namespace. Any XML element with no child elements can be shortened according to the XML specification. White space between elements included in example figures is introduced solely to make the examples more readable and may be omitted.

Outgoing messages consist of the XML elements explained below. Fig. 9.2 gives an example of a message sent by an observer.

```
<?xml version="1.0"?>
<observer xmlns="snet-home.org" oid="128" position="MyNet" type="before" code="MyCode" >
  <record type="data" mode="textual" >
    <field label="F" interface="C4SNet">1</field>
    <tag label="T" >0</tag>
    <btag label="B" >0</btag>
  </record>
</observer>
```

Figure 9.2: Example: observer message

Observer element

Each message contains the XML element `observer` as its root element. If the message does not contain an XML-declaration, it is assumed to be XML version 1.0 by default. The `observer` element identifies the observer sending the message, it has the following attributes: The `oid` attribute contains a unique observer instance identifier. This can be used to distinguish between observers and their instances. The `position` attribute contains the original textual name of the S-Net component to which the observer is attached to. This is exactly the same as the name attribute of the metadata net or box element where the observer was declared. The `type` attribute specifies the location of the observer with respect to the S-Net component it is attached to. Potential values are `before` and `after`, but not `both`, which rather serves as a shortcut for declaring two observers at once. The `code` attribute contains the user defined data optionally given in the observer declaration. This attribute is omitted if no code was specified in observer declaration. Last not least, the `observer` element contains one or more `record` elements.

Record element

The `record` element represents a single S-Net record. The `type` attribute specifies the type of the record. In analogy to the network interface data format (see Section 8.3, supported types are `data` and `terminate`. Data records may contain zero or more `field`, `tag` and `btag` elements. All of these elements have an attribute `label`, which contains the label of the entity. Data records may also have a `mode` attribute, which is used to declare the mode of the data inside the `field` elements. Values for the `mode` attribute are `binary` and `textual`. In the absence of a mode specification, binary mode is assumed as default.

The contents of `field`, `tag` and `btag` elements depends on the level of information specified in the observer declaration. According to the levels defined in Section 9.1 values are included or not. Field values are serialised using the corresponding box language interface facility. The attribute `interface` disclosing the box language interface is attached to each `field` element if the chosen level of information is `allvalues`.

Acknowledge messages

Interactive observers expect to receive an acknowledge message in response to each outgoing message. This message is expected to have the format exemplified in Fig. 9.3. The `oid` attribute must have exactly the same value that was included in the corresponding observer message. If the same listener listens to multiple observers, the order of acknowledge messages is irrelevant.

```
<?xml version="1.0"?>
<observer xmlns="snet-home.org" oid="128" />
```

Figure 9.3: Example: interactive observer acknowledge message

Chapter 10

Reconfiguration and Self-Adaptivity

10.1 Motivation

Reconfigurable software systems are useful for a variety of reasons... These systems allow for reconfiguration of deployed applications to adapt to changing computing environments. Environments are influenced, for example, by computing resources that become available (or unavailable) and communication links that change in quality and speed during the runtime of an application. Even, in stable environments the ability to reconfigure is desirable: Ideally, the design of an application is based on the anticipated computational intensity of input data. Often, this knowledge is not available until runtime, and even if it is, the input data may change over time. Here, reconfiguration after deployment of the system is key to maintain a well-performing application. This is even more the case if energy efficiency is a concern. Energy consuming computing resources are often wasted by employing non-optimised algorithms that are not specifically tailored towards the problem at hand. Replacing these algorithms by highly specialised versions at runtime is worthwhile but requires the underlying system to be flexible enough to allow for this.

Projected onto the S-Net domain, the reasons to reconfigure a network may be manifold: A box has to be replaced, because an improved version of an algorithm a box is currently running became available (software update). A network suffers from poor throughput, because of imbalanced execution times and bottlenecks (resource management), or even worse, a box has crashed and does not process any records anymore (recovery from failure). There are many more scenarios in which reconfiguration is key to maintain, improve or restore system stability and performance. In all of the above cases, it is relatively easy to come up with a simple solution for each of them. A software update can be achieved by recompiling the network with a new box, bottlenecks can be detected by channel monitoring at runtime and resolved by a runtime-system that acts accordingly, and crashed components can be detected by a watch-dog timer which also triggers a re-start of components.

Each of these approaches tackles a specific symptom rather then providing a general cure. The next section offers a brief design-space exploration and presents our approach of providing a cure in S-Net.

10.2 Introducing Reconfiguration and Self-Adaptation Mechanisms

As mentioned in the previous section, providing separate mechanisms for each possible situation is not very satisfactory. It renders the system very complex and not flexible, as we have to add a potentially new feature for each additional scenario we would like to control. We need to identify a common, essential and yet powerful enough mechanism to solve these problems. It turns out, that a replacement mechanism for networks and boxes satisfies all needs. If we are able to replace components *at runtime*, a software update is nothing more than a replacement of a box, a bottleneck is resolved by replacing an under-performing network by more instances of the network, and any crashed component is resurrected by replacing it with a fresh instance of itself. This replacement mechanism can be provided by a runtime system. If done so, mechanism to detect where replacement is necessary, have to be implemented in the runtime system as well — which immediately exposes the weakness of this approach. For any scenario that was not anticipated when designing the runtime system, the implementation has to be extended. In many cases this is not possible, as it requires expert knowledge about the implementation. A user that discovers the need for a new reconfiguration scenario, read replacement, will have to invest time and effort to implement what is needed in the runtime system.

It is much more flexible to promote the replacement mechanism to the language level. If the replacement mechanism is available on the language level, a user may use it to implement replacement scenarios explicitly — in the language itself! The runtime system has to provide only the basic replacement functionality. Any further runtime behaviour is specified outside the runtime system, intriguingly, by the application the runtime system executing.

Before introducing a replacement mechanism, we have to ask ourselves how new networks are made available. The design space is endless, but our aim is to provide this feature explicitly for greatest possible flexibility. Consequently, networks are made first-class citizens. A network may be part of a record, just like tags and data fields. This immediately solves the problem of how to communicate new networks to their intended destination: Records are routed on the same basis as before. Exactly as fields and tags, a network influences the routing decisions made by combinators. The record containing a network finds its destination travelling along the existing network infrastructure.

$$\textsc{UnWrap} \quad : \quad \overline{(p, \rho N) \rightarrow [(p, \varrho N || N)]}$$

$$\textsc{Replace} \quad : \quad \frac{N' \in p}{(p, \varrho N) \rightarrow (\varrho N', \{N\} \cup (p \setminus N'))}$$

$$\textsc{Wrap} \quad : \quad \begin{cases} \dfrac{\vec{q_{l_1}} \neq \epsilon}{[(\varrho N' || N, \vec{q_{l_1}} +\!\!+ \vec{q_{r_1}})] \rightarrow (\rho N', \vec{q_{l_1}})} \\ \dfrac{\vec{q_{l_1}} = \epsilon}{[(\varrho N || N', \vec{q_{l_1}} +\!\!+ \vec{q_{r_1}})] \rightarrow (\rho N', \vec{q_{r_1}})} \end{cases}$$

Figure 10.1: The operational semantics of the replacement combinator defines the replacement process itself and defines routing of records by the choice combinator.

But what is the destination of a record containing a new network? As we intend to replace a specific, existing network, this network would be the obvious destination. However, a network has

no concept of replacement.

To add this concept of replacement, we follow the language design and provide a combinator to do this. As with the other combinators, the replacement combinator may be applied to any network. By applying it to a network, the network is made replaceable. Thereby, it is not the network that replaces itself, but it is the combinator that replaces its operand.

When a record arrives at this combinator, the record may be treated in two different ways. If the records contains a network, i.e. a potential operand for the combinator, this network will be inspected. If it is compatible to the currently deployed operand, the replacement combinator reads and removes the network from the record. The combinator replaces its current operand by the new network and outputs the old operand as result. If the record contains an incompatible network or no network at all, the record is sent to the operand of the replacement combinator. In this case, the record is processed as usual.

The routing behaviour of the replacement combinator is similar to that of a choice combinator, which inspects inbound records and dispatches it depending on the contents of the record. We use this similarity in the formalisation of the operational semantics of the replacement combinator (Fig. 10.1).

The UnWrap rule decomposes the combinator into three fundamental tasks: replacement, standard record processing and routing. The replacement task (Replace rule) expects each inbound record to carry a new operand network. The network is removed from the inbound record, deployed as new operand, and the previous operand output as result. If an inbound record does not carry a compatible network, the record is routed to the current operand and undergoes standard record processing. The routing is managed by the choice combinator, that the UnWrap rule inserted to connect replacement and standard record processing.

The Wrap rule combines the three tasks back into the replacement combinator. Depending on which branch of the choice combination was taken, the argument of the replacement combinator is chosen. If a replacement took place, the replacement task outputs its old operand. In this case, the outbound stream of the replacement task is not empty and the new network will be the operand of the reassembled combinator. Otherwise, the stream will be empty as no replacement was carried out. The result of the operand network is output and the operand is not changed.

The replacement combinator allows us to refine networks from the outside. The next step is now, to enable networks to trigger reconfiguration from the inside. The goal is to be able to define self-adaptive networks, that react to certain situation and reconfigure themselves. For a network to be able to do this, the network needs to send a reconfiguration message to *itself*. The reconfiguration message, of course, is a record containing a new network. But how do we define the message-to-self mechanism? The star combinator with its strict feed-forward semantics is not optimal: A new version of a network can only be sent to the *next* instance, which always leaves the previous network active in the current instance. For this reason, we design a feedback combinator that does not unfold but consists of one instance only.

$$\textsc{Mu} \quad : \quad \frac{(p, N) \to (N', \vec{q'}) \quad \vec{l}, \vec{r} = \mathrm{lrpsplit}(\vec{q'}, \sigma, \{\}) \qquad (\vec{l}, N'\mu\{\sigma\}) \to (N''\mu\{\sigma\}, \vec{q})}{(p, N\mu\{\sigma\}) \to (N''\mu\{\sigma\}, \vec{r} +\!+ \vec{q})}$$

Figure 10.2: The feedback combinator sends back records to the input of its operand if the pattern σ is matched. In this case, all results are output before any other record is processed.

Not unlike the star combinator, the feedback combinator μ (Fig. 10.2) 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 μ 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 ρ and μ, i.e. a feedback combinator applied to a replacement combinator applied to a network N, is very powerful. It allows us to define a network that emits a network and sends it to itself. More specifically, by defining the pattern of the feedback combinator accordingly, a new network that is output by N is sent back, where it is picked up by the replacement combinator. The replacement combinator receives the new network and replaces its operand.

Chapter 11

Examples

11.1 Streams of Factorial Numbers: a Functional Perspective

The purpose of our first example is to illustrate similarities and differences between concepts found in S-Net and mainstream functional programming languages. We employ a fairly simple and very well-known example, computing factorial numbers, and show how an implementation of factorial in the functional programming language Standard ML can can be broken down into atomic parts and then step by step transformed into an equivalent SNet. We will investigate the same problem starting with an imperative solution in Section 11.2. The sole purpose of this example is the illustration of S-Net language features. The toy character of the problem is by no means representative for our intended appliction domain. In real-world S-Net applications we assume boxes to represent substantial amounts of computations and the data exchanged between them to be of significant size.

```
fun fac n =
  let fun facit (x,r) = let val p = x<=1
                        in if p
                           then r
                           else let val rr = x*r
                                    val xx = x-1
                                in facit (xx,rr)
                                end
      val m = facit (n,1)
  in (n,m)
  end
```

Figure 11.1: Factorial function in Standard ML

Fig. 11.1 shows a definition of the factorial function in Standard ML syntax. The function `fac` takes one (integer) argument `n` and yields a pair of integers `n` and `m`, where `n` is the original argument and `m` is the factorial of `n`. Our implementation of factorial employs an iterative (tail-end recursive) scheme. Therefore, we need a local auxiliary function `facit` that takes two arguments, `x` and `r`. While the first parameter holds the number of which we compute the factorial of, the second is used to accumulate the result value and, hence, is set to 1 in the initial application of `facit`.

In the definition of the auxiliary function `facit` we first compute the termination predicate `p`.

If p holds, we simply return the parameter r, which in the given case is known to hold the correct factorial result. Otherwise, we multiply the current value of x to the intermediate result r and decrement x by one. Finally, we recursively apply the function facit to the updated values xx and rr.

```
net fac ({n} -> {n,m}) {
  net facit ({x,r} -> {r}) {
    box leq ((x) -> (x,p));
    box if ((p) -> (<T>) | (<F>));
    box dec ((xx) -> (xx));
    box mult ((x,r) -> (rr));
  }
  connect (leq..if..([{<T>}->{<stop>}]
                     || [{<F>,x,r}->{x,r};{xx=x}]
                         .. (dec|mult)
                          .. [|{xx},{rr}|]*{xx,rr}
                           .. [{xx,rr}->{x=xx,r=rr}]) ** {<stop>})
                   ..[{<stop>,x}->{}];

  box one (() -> (one));
}
connect one .. [{n,one}->{n,x=n,r=one}] .. facit .. [{r}->{m=r}];
```

Figure 11.2: Computing streams of factorial numbers in S-Net

The purpose of our case study is to carry over the functional implementation of the factorial function as directly as possible into an equivalent SNet. The result of our exercise is shown in Fig. 11.2. The nested definitions of the functions fac and facit is carried over one-to-one to the world of S-Net by having two nested network definitions of the same names. For documentary reasons we have added type signatures to the network definitions: The outer network fac accepts records with a field n and yields records with fields n and m; the auxiliary network facit takes records with fields x and r and yields records with field r only. The choice of field names is inspired by the use of identifiers in Fig. 11.1. Likewise, the introduction of many local identifiers in the Standard ML implementation of the factorial function is motivated by our wish to illustrate the equivalences between the two approaches.

In the definition of the network facit we find four box definitions. They reflect the basic building blocks of the functional implementation of factorial: The box leq computes the termination condition; the result is stored in field p. The box if makes the Boolean value of the field p visible to S-Net by turning it either into a tag <T> or a tag <F>. Last but not least, the boxes dec and mult do the required arithmetic.

Fig. 11.3 shows a graphical representation of the network topology of facit that is equivalent to the textual specification following the key word connect in Fig. 11.2. We start with a serial combination of leq and if. Note here the use of the concept of flow inheritance. The box leq computes the termination condition p solely on the basis of x. The other field r, which we know to be present in all records due to the type signature of facit is flow inherited around the box. Likewise, the box if only inspects the field p to create the tags <T> or <F>; the other fields x and r are flow inherited.

The boxes leq and if are connected in serial with a parallel choice combinator. Any record containing the tag <T> are directed to the first (upper) alternative because in that branch we have a filter box that requires records to have such a tag. Likewise, any record with a tag <F> is directed into the alternative branch due to the presence of another filter box in that branch which requires this tag.

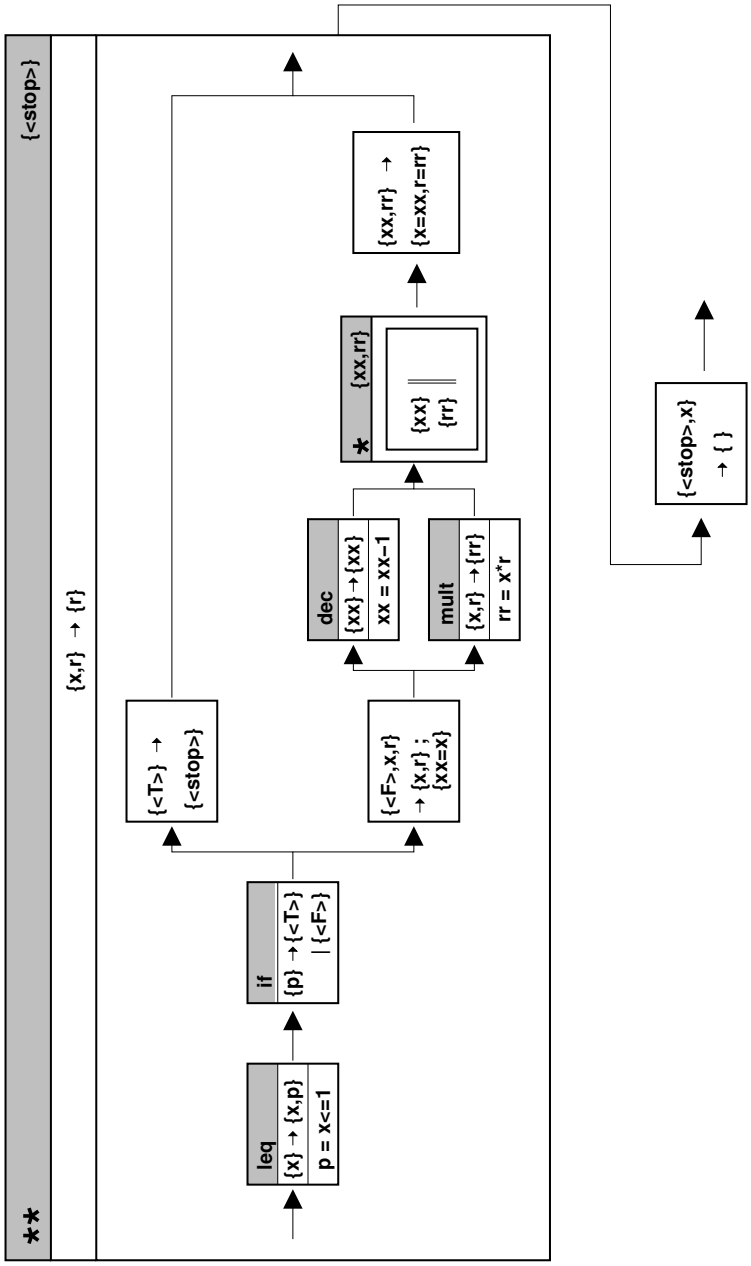

Figure 11.3: Graphical representation of network `facit`

The filter in line 8 simply turns the `T` tag into a `<stop>` tag. In contrast the filter in line 9 produces two records for any incoming records: while the `<F>` tag is stripped in both cases, the field `x` is duplicated. Since there is no data dependency between the decrementation and the multiplication, we can arrange the two boxes in parallel. Due to the best match rule, the choice is deterministic. Any record {`x,r`} is directed towards the `mult` box while {`xx`} records are directed to the `dec` box. The two arithmetic boxes produce records {`xx`} and {`rr`}, respectively. Note that we deliberately rename `x` to `xx` in the filter preceding the choice combinator to circumvent the covariance restriction (cf. Section 3.4) of parallel composition.

As we need to combine them again into a single record for further processing, we feed both into a synchrocell with pattern {`xx`},{`rr`}. Upon successful synchronisation, the synchrocell produces single {`xx,rr`} records. A subsequent filter box renames the fields back to `x` and `r`. Keeping in mind from Section 3.5 that a synchrocell dies after the first synchronisation, we must embed our synchrocell within a star combinator in order to be able to repeatedly compute factorial numbers for a stream of input records. The termination pattern of the star combinator is the synchronised record {`xx,rr`}.

This combination of synchrocell and star combinator is a very common design pattern in S-Net. It implements synchronisation across an unbounded number of records: For example, an incoming {`xx`} record is stored in the first synchrocell. If the following record is again of type {`xx`} it is forwarded by the first synchrocell (which now waits for {`rr`} records), but since an {`xx`} record does not match the termination pattern of the star combinator, a new synchrocell is created dynamically (replication of the operand SNet of the star combinator). This new synchrocell then captures the {`xx`} record. Supposed the following record is of type {`rr`}, it is captured by the first synchrocell, which synchronises the {`rr`} record with the stored {`xx`} record and produces a joint {`xx`},{`rr`} record. This combined record does match the termination pattern of the star combinator and, therefore, leaves the sync-star subnetwork. The first synchrocell dies after synchronisation with the effect that any subsequently incoming records are directly sent to the second synchrocell incarnation.

The entire network described so far is itself embedded within another star combinator. The operand network only computes a single iteration of the functional specification of factorial. The star combinator realises the tail-end recursive application of the function `facit` in the Standard ML implementation. With respect to the operational behaviour of the SNet, the star combinator repeatedly replicates the operand network until records are marked by the `<stop>` tag. If, for example, we compute the factorial of two, we end up with two replicas of the operand network. If we subsequently compute the factorial of three, the record travels through the two existing incarnations and then requires an additional replication. No further network replication occurs as long as we compute only factorials of numbers less than four.

The effective length of the path a record takes through the SNet is proportional to the argument value. Therefore, we use the deterministic variant of the star combinator to still preserve the sequence of records, i.e., a sequence of incoming records with values 3, 2 and 1 would yield a sequence of outgoing records with values (3,6), (2,2) and (1,1). Had we used the non-deterministic variant of the star combinator instead, we could have ended up with any permutation in the sequence of outgoing records.

A final filter box in the definition of `facit` discards the `<stop>` tag and the `x` field from outgoing records, i.e., any outgoing record solely has an `r` field. This complies with the Standard ML specification of `facit`, which also yields only a single value `r`. Given the notes on optional type signatures for networks in Section 3.2, this filter box could be left out. The given type signature for `facit` would equivalently result in discarding the additional record fields.

Very much like in the Standard ML implementation of factorial the implementation of the network `fac` itself is rather simple and is predominantly concerned with housekeeping tasks. It

has one local box named `one`. This box is required to complement each incoming argument value with the numeric value one as initial value for the result accumulation field `r` in `facit`. We need a box and a box language implementation for this rather simple task because as a pure coordination layer S-Net is unaware of the data associated with record fields. There is no notion of record field type in S-Net. Although in the given scenario all field values are integer numbers, this fact is simply unknown to S-Net. Hence, we need a box even for a simple task as creating a constant value.

From a purely technical perspective, of course, we could turn all record fields into tags. As tags carry integer values, this would allow us to express all required computations entirely on the level of S-Net. However, this is only works for integer numbers and clearly constitutes a misuse of tags, which are exclusively intended to be used for control purposes.

The network topology of `fac` is a simple pipeline. Firstly, we add a field `one` to each incoming record. Then, we rename `n` to `x` and `one` to `r` to meet the interface of `facit`. Note that in addition we keep a copy of `n` to produce pairs of `n` and `n!` in the end. An instance of `facit` implements the computational aspects of factorial, while a final filter box renames `r` to `m`.

This case study shows how concepts of functional programming (e.g. nested function definitions, function applications, tail-end recursion) can be expressed in the framework of S-Net in a systematic way. The example in particular serves as blueprint for expressing linear recursive functions in S-Net. In the factorial example the box language code is extremely simple, i.e. one instruction each. We chose this level of granularity in order to demonstrate the concepts of S-Net. However, without changing the principles of the SNet we could replace the box inscriptions by complex computations with record fields referring to large data structures. As long as the algorithmic pattern remains the same, we can easily turn a toy example like factorial into a real application. Leaving the concrete example behind, our case study sketches out a methodology to convert functional programs into SNets in order to express and to exploit concurrency.

11.2 Streams of Factorial Numbers: an Imperative Perspective

Our second example sticks to the pseudo application of computing a stream of factorial numbers. However, this time we start with an imperative solution of the problem. Fig. 11.4 shows a straightforward implementation in C. The C function `factorial` only computes a single factorial number given a suitable argument. We leave it to the imagination of the reader how this function could be mapped to an entire stream of arguments in order to produce a stream of pairs of argument and the argument's factorial number.

```
int factorial( int n)
{
  int r, x;
  r = 1;
  x = n;
  while (x>1) {
    r = x*r;
    x = x-1;
  }
  return( r);
}
```

Figure 11.4: Computing a single factorial number in C

The S-Net network `factorial` shown in Fig. 11.5 indeed transforms a stream of natural numbers into a stream of pairs, as reflected by its type signature {n}->{n,fac}. Although type signatures in S-Net are typically inferred by the compiler, we have typed all networks in Fig. 11.5 for the purpose of illustration.

```
net factorial ({n} -> {n,fac}) {
  box one (() -> (one));
  box leq ((x) -> (x,p));
  box if ((p) -> (<T>) | (<F>));
  box dec ((x) -> (x));
  box mult ((xx,r) -> (r));

  net init ({n} -> {n,r,x})
  connect one .. [{n,one}->{n,x=n,r=one}];

  net loop ({r,x} -> {r,x,<stop>}) {
    net pred ({x} -> {x,<T>} | {x,<F>})
    connect leq .. if;

    net then_branch ({<T>,x,r} -> {x,r})
    connect [{<T>}->{}]
             .. [{x,r}->{xx=x,r};{x}]
              .. (dec|mult)
               .. [|{x},{r}|]*{x,r};

    net else_branch ({<F>} -> {<stop>})
    connect [{<F>}->{}] .. [{}->{<stop>}];
  }
  connect (pred .. (then_branch || else_branch)) ** {<stop>});

  net exit ({<stop>,x,r} -> {fac})
  connect [{<stop>,x}->{}] .. [{r}->{fac=r}]
}
connect init .. loop .. exit;
```

Figure 11.5: Computing a stream of factorial numbers in S-Net

Since the true purpose of our example is to demonstrate as many language features of S-Net as feasible, we break down the problem into its atomic building blocks first. The five boxes only perform the most simple tasks like producing a box language representation of the number one or doing simple arithmetic computations. The topology of the network `factorial` is fairly simple: a pipeline consisting of an initialisation step, the main loop and a postprocessing step. This structure exactly coincides with the C implementation where the postprocessing step is somewhat hidden in the `return` statement.

The network `init`, very much like the first few lines of the C implementation, initialises new record fields `r` and `x` for the actual computation while the original argument `n` is preserved for the global output. Whereas the renaming of `one` to `r` and the copying of `n` to `x` can easily be done on the S-Net level using a filter box, we employ a user-defined box to create a proper box language representation of the number one.

From a purely technical perspective, of course, we could turn all record fields into tags. As tags carry integer values, this would allow us to express all required computations entirely on the level of S-Net. However, this only works for integer numbers and clearly constitutes a misuse of tags, which are exclusively intended for control purposes.

The `while`-loop of the C function directly carries over to a star combinator in S-Net. Since we want to preserve the original sequence when transforming a stream of numbers into a stream

of pairs of these numbers and their factorials, we use the deterministic variant of the combinator. The loop itself turns into the natural pipeline of evaluating the loop predicate and then either executing the consequence or the alternative. Note that the loop predicate (network `pred`) is entirely evaluated in the domain of a box language. Hence, the boolean result is hidden in an opaque record field `p` and can only be made accessible to S-Net by means of another box `if`, that takes field `p` and depending on its boolean interpretation either yields a tag `T` or a tag `F`.

These tags are used to route records either into the network `then_branch` or into the network `else_branch` as in either of them a filter box requires one or the other tag to be present in any incoming record. In the case of a loop the consequence of the predicate not holding is termination of the loop. Therefore, network `else_branch` just strips off tag `F`, which has fulfilled its purpose, and adds a new tag `stop`, which makes the record leave the `loop` network.

Likewise the network `then_branch`, which roughly implements the loop body of the C function `factorial`, starts with stripping off the tag `T` from each incoming record. Then, it uses another filter box to duplicate each incoming record into one that is identical and one that only contains field `x`. These two records contain the relatively free variables of the two expressions found in the loop body of the C function `factorial`. In the S-Net solution, these expressions are evaluated concurrently (`dec|mult`). Note that the best match rule of the parallel composition combinator plays a crucial role here in routing the {`xx,r`} record to the box `mult` and the {`x`} record to the box `dec`. Note further that we need to rename field `x` into `xx` in order to circumvent the covariance restriction (cf. Section 3.4) of parallel composition.

A subsequent synchrocell recombines records {`x`} and {`r`} into a joint record {`x,r`}. Note that the synchrocell is embedded within another serial replication. This combination of synchrocell and star combinator is a very common design pattern in S-Net. It implements synchronisation across an unbounded number of records: For example, an incoming {`x`} record is stored in the first synchrocell. If the following record is again of type {`x`}, it is forwarded by the first synchrocell (which now waits for {`r`} records), but since an {`x`} record does not match the termination pattern of the star combinator, a new synchrocell is created dynamically. This new synchrocell then captures the {`x`} record. Supposed the following record is of type {`r`}, it is captured by the first synchrocell, which synchronises the {`r`} record with the stored {`x`} record and produces a joint {`x,r`} record. This combined record does match the termination pattern of the star combinator and, therefore, leaves the sync-star network. The first synchrocell dies after synchronisation with the effect that any subsequent records are directly sent to the second synchrocell instance.

Last but not least, the `exit` network strips off field `x` and tag `stop` from any record since they are only used internally by the `factorial` network. Eventually, field `r`, as it is used internally in `factorial`, is renamed into `fac` before a record leaves the whole network.

Throughout the `factorial` network flow inheritance plays a crucial role for the composition of boxes and subnetworks. Take as a simple example the creation of a box language representation of the number one by box `one`. Thanks to flow inheritance we can specify this box in a way that adds the field `one` to any incoming record regardless of its existing fields and tags. This allows us to realise this box language component entirely independent of our application context in the implementation of `factorial` and create a fine opportunity for code reuse.

As pointed out in the beginning, the sole purpose of our example is to illustrate the use of the various S-Net language features and their relationship to constructs known from conventional programming languages. It is definitely not intended as an exercise in finding the most suitable description of how to compute factorial numbers. This task would hardly benefit from the degree of concurrency introduced by the S-Net in Fig. 11.5. Using boxes only for the most rudimentary computations and expressing anything else in S-Net is by no means representative for real world S-Net applications. Here, we expect boxes to represent substantial amounts of computational work and the S-Net layer to control only coarse-grained coordination aspects. However, such

a real world example would be not very useful for the purpose of illustrating S-Net features because it would require a fair amount of knowledge about the box language components as well as familiarity with the chosen application domain. We refer the reader interested into the interplay between box language and S-Net to [19] for a more elaborate case study.

11.3 Solving Sudokus with S-Net and SaC

This example investigates the interplay between S-Net and the box language SaC [16, 20]. We first develop a SaC program for solving sudokus and later on refine it to a distributed S-Net-SaC solution. This example also appeared in [19].

Sudokus are played on a 9 by 9 board of numbers. Starting out from a board with several given numbers, the overall aim is to fill all empty positions with numbers so that the following conditions hold: (i) each row contains the numbers 1 to 9 exactly once, (ii) each column contains the numbers 1 to 9 exactly once, and (iii) each of the nine 3 by 3 sub-boards contains the numbers 1 to 9 exactly once. Although in general we may have an arbitrary number of solutions or no solution at all, all well-constructed sudokus have a unique solution.

SaC (Single Assignment C) is a purely functional, data parallel array programming language. Core 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. This language kernel is extended by multi-dimensional state-less arrays: Any expression may evaluate 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. For a more thorough introduction to SaC we refer to [16, 20].

Fig. 11.6 shows an excerpt from the SaC sudoku solver. The central idea is to keep a 9 by 9 matrix of 9-element boolean vectors that represent the possible choices for each given position. We start out from an array containing `true` values only. Whenever we add a new number to the board, we eliminate all those options that are affected due to the 3 rules above, i.e., we set all corresponding positions in the same column, row and sub-matrix to `false`. This is essentially what the function `addNumber`, as shown in Fig. 11.6, does. It takes the following arguments:

1. a position in the board specified by two integer parameters `i` and `j`,
2. a number `k` to be placed at that position,
3. a two-dimensional board `board` holding all numbers set so far, and
4. a three-dimensional boolean array `opts` of options.

As a result, our function `addNumber` yields modified versions of the board and the options which reflect the insertion of the number `k` at position `i,j`.

While the modification of the board requires only the manipulation of a single element of the board (cf. line 4), the modification of the options is expressed by a with-loop which spans over the lines 6 to 11. The generator in line 7 sets all options in position `i,j` to `false`, line 8 falsifies the option for the given number `k` in row `i`, and line 9 falsifies the option for the given number `k` in column `j`. Line 10 eliminates the option in the 3 by 3 sub-matrix where `i,j` is located in. Note here that the decrement of `k` is due to the fact that array indexing in SaC always starts with 0, whereas the numbers to be placed in the sudoku start with 1.

With this function at hand, after an initialisation phase, which adds the pre-determined numbers, solving sudokus boils down to a search algorithm which successively adds numbers to all

```
int[*], bool[*] addNumber( int i, int j, int k,
                           int[*] board, bool[*] opts)
{
  board[i,j] = k;
  k = k-1; is = (i/3)*3; js = (j/3)*3;
  opts = with {
           ([i,j,0] <= iv <= [i,j,8]) : false;
           ([i,0,k] <= iv <= [i,8,k]) : false;
           ([0,j,k] <= iv <= [8,j,k]) : false;
           ([is,js,k] <= iv <= [is+2,js+2,k]) : false;
         } : modarray( opts);

  return( board, opts);
}

int[*], bool[*] solve( int[*] board, bool[*] opts)
{
  if (! isStuck( board, opts) && ! isCompleted( board)) {
    i,j = findFirst( 0, board);
    mem_board = board;
    mem_opts = opts;
    for( k=1; (k<=9) && (!isCompleted( board)); k++) {
      if( mem_opts[i,j,k-1] ) {
        board, opts = addNumber( i, j, k, mem_board, mem_opts);
        board, opts = solve( board, opts);
      }
    }
  }
  return( board, opts);
}
```

Figure 11.6: Excerpt from SAC sudoku solver

positions not yet filled until it either gets stuck or is completed. This is implemented by the function `solve` in Fig. 11.6. It takes an actual board and an array of options as arguments and computes the first solution it finds or, if no solution exists, the board where the algorithm got stuck. At the core of this function we find a recursive call embedded into a FOR-loop which realises the back-tracking of the search. For each valid option at a given position `i,j` we successively try to solve the given board until it is completed.

Since this, in the worst case, can lead to a 9-fold recursion for each of the numbers to be filled in, the choice of `i` and `j` directly affects the breadth of the search tree and, thus, has a vast impact on the runtime performance of the overall program. So far, we simply select the first occurrence of a zero in the board, i.e., the first empty field. In order to keep the potential need for back-tracking as small as possible, we replace the call to `findFirst` by a call of `findMinTrues( opts)` which selects a free position with a minimum number of options left.

If we want to parallelise this application, we can directly spot two potential sources for concurrency: `addNumber` and `findMinTrues` can be executed in a data-parallel fashion, and the recursive calls in `solve` can be done concurrently effectively transforming our depth-first search into a breadth-first search.

Our first step towards a combined S-NET-SAC solution is to shift the recursion from the SAC level to the level of S-NET. This can be achieved by transforming the recursive calls into S-NET-records and by embedding the function `solve` into an S-NET box, which then serves as an argument to a serial replicator. Fig. 11.7 shows our initial S-NET solution. We replace the original SAC function `solve` by the S-NET box implementing SAC function `solveOneLevel` shown

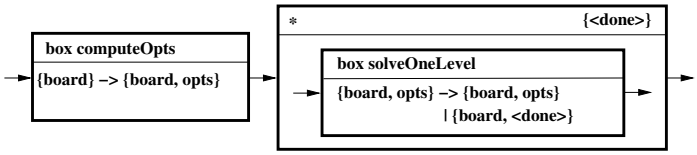

Figure 11.7: Initial S-Net sudoku solver

in Fig. 11.8 Instead of a recursive call, the new function `solveOneLevel` tries to place one further number at the selected position `i,j`. For each possible number at that position it outputs a record containing either the new board and its options or the final board and a tag `<done>`, which signals the completion of the puzzle.

```
void solveOneLevel( int[*] board, bool[*] opts)
{
  if ( !isStuck( board, opts) && !isCompleted( board)) {
    i,j = findMinTrues( opts);
    mem_board = board;
    mem_opts = opts;
    for( k=1; (k<=9) && !isCompleted(board); k++) {
      if( mem_opts[i,j,k-1] ) {
        board, opts = addNumber( i, j, k, mem_board, mem_opts);
        if ( isCompleted( board)) {
          snet_out( 1, board, opts);
        } else {
          snet_out( 2, board, 0);
        }
      }
    }
  }
}
```

Figure 11.8: SaC box code for S-Net sudoku solver

The SaC function `solveOneLevel` becomes an S-Net box, which itself is embedded into a serial replicator with the termination pattern specified in the upper right corner. The replicator dynamically unfolds into a pipeline of `solveOneLevel` boxes. As soon as one of these boxes produces a record containing the tag `<done>`, that record leaves the conceptually infinite pipeline.

It should be noted here that in our sudoku example this unfolding cannot lead to pipelines longer than 81 replicas of the `solveOneLevel` box. This is due to the fact that each `solveOneLevel` box only emits a record if it can add a number to the board. Otherwise, it either emits no record at all (search is stuck) or it emits a record that contains `<done>` (solution is found). Left to the serial replicator we have another box named `computeOpts`, which takes the incoming board and realises the initialisation of the options arrays by repeatedly calling the function `addNumber` from Figure 11.6.

If we assume that each S-Net box is actually run by an individual process/thread, the crucial question now is: to what extend do we exploit the possibility to concurrently examine different choices of the n^{th} number? If p is the number of pre-defined numbers, the n^{th} number is set by the $(n-p)^{th}$ replica of the `solveOneLevel` box. For each option k, it emits a record to the next replica. As a consequence, the $(n+1)^{th}$ number for each of these alternatives is placed sequentially. However, the placement of the $(n+2)^{th}$ number can happen concurrently with the placement of the $(n+1)^{th}$ number of the next alternative and so forth.

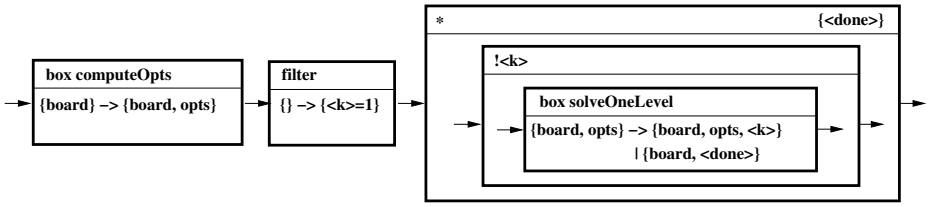

Figure 11.9: Refined S-Net with full unfolding

In order to be able to place the $(n+1)^{th}$ number concurrently we need to extend our network slightly. Effectively, we have to make sure that there are as many parallel replicas of the solveOneLevel box as we have options on each level, i.e., up to 9. This can be achieved by putting a parallel replicator around the `solveOneLevel` box within the serial replicator. Fig. 11.9 shows the modified S-Net. We use a new tag `<k>` for controlling the parallel splitting. Within the serial replicator, this tag can be conveniently generated by extending the output of the solveOneLevel box: whenever the board is not yet completed, we simply output the SaC-variable `k` along with the board and the options. Since `k` within each level represents the number that is being examined, this achieves the desired effect. However, we do not want to change our initialisation box `computeOpts`. Therefore, we need to insert a filter between the computeOpts box and the start operator which adds a tag `<k>` to each record. Note that the filter has the desired effect on records of the type `{board, opts}` although its fields do not occur in the filter. This is one of the benefits of the flow-inheritance of S-Net.

Another interesting feature of this network is that both replicators unfold dynamically. However, since the subsequent records with the same tag `<k>` are being processed by the same box, we know that on each stage no more than 9 replicas of the `solveOneLevel` box will be created as the value of `k` is always between 0 and 8. This guarantees a maximum of $9 \times 81 = 729$ of `solveOneLevel` boxes.

While 729 replicas of the `solveOneLevel` box might be acceptable, for bigger sudokus or in situations where we cannot derive proper upper limits for the unfoldings from the application itself, we usually want to control the unfolding of the replicators. This can be done by manipulating the control tags, in our case the tag `<k>` for the parallel unfolding and the tag `<done>` for the serial one. For example, we can control the number of parallel instances by a filter of the form

```
{<k>} -> {<k>=<k>\%4}
```

which we put into the serial replicator in front of the parallel replicator. By using the modulo operation represented by the `%` symbol, we reduce all potential values for `<k>` to the range 0 to 3, which implicitly limits the parallel unfolding to a maximum of 4 instances.

In order to be able to control the unfolding of the serial replicator, we need to communicate the current level of unfolding, i.e., the number of numbers placed already, rather than a boolean flag indicating completion. After changing the tag `<done>` to a tag `<level>` that contains this information, we can use a more elaborate predicate for leaving the serial replicator such as `{<level>} | level > 40`. This leads to a situation where non-completed sudokus exit the serial replicator. Therefore, we need to link up yet another box which calls the full solver function from Fig. 11.6 resulting in the S-Net shown in Fig. 11.10.

Last but not least, it should be mentioned that the presented SaC/S-Net implementation(s) typically solve 9×9 sudokus in far less than a second on standard desktop computers. However, as sudokus can be played on any board of size $n^2 \times n^2$ they are suitable candidates for parallelisation, indeed. Furthermore, we see solving sudoku puzzles as an algorithmic pattern for a broad range

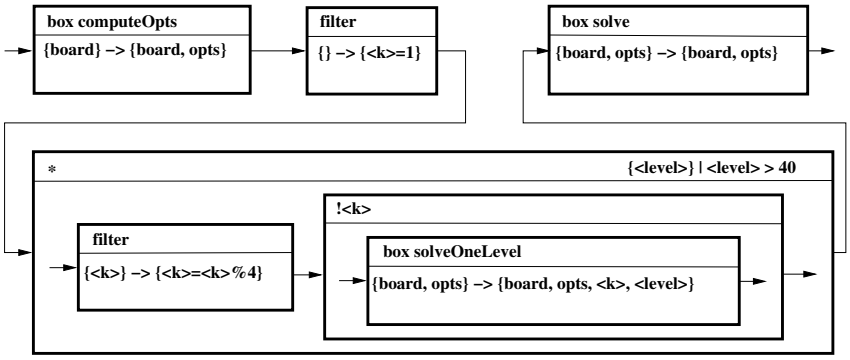

Figure 11.10: Sudoku S-Net with throttled unfolding

of search problems, including many with a higher relevance than sudokus themselves.

Chapter 12

Conclusion

We have presented the design of S-Net, a declarative language for describing streaming networks of asynchronous components. Several features distinguish S-Net from existing stream processing approaches.

- S-Net boxes are fully asynchronous components communicating over buffered streams.
- S-Net thoroughly separates coordination aspects from computations, which are described in a separate compute language.
- The restriction to SISO components allows us to describe complex streaming networks by algebraic formulae rather than error-prone wiring lists.
- We utilise the type system to guarantee basic integrity of streaming networks.
- Data items are routed through networks in a type-directed way making the concrete network topology a type system issue.
- Record subtyping and flow inheritance make S-Net components adapt to their environment, which facilitates composition of S-Net components developed in isolation.

Appendix A

Complete Syntax of S-Net

SNet	⇒	*[Definition]**
Definition	⇒	*TypeDef* | *TypeSigDef* | *NetDef* | *BoxDef*
TypeDef	⇒	**type** *TypeName* = *Type* ;
Type	⇒	*RecordType [| Type]*
	|	*TypeName [| Type]*
RecordType	⇒	{ *[RecordEntry [, RecordEntry]*]* }
RecordEntry	⇒	*Field* | *Tag*
Field	⇒	*FieldName*
Tag	⇒	*SimpleTag* | *BindingTag*
SimpleTag	⇒	< *SimpleTagName* >
BindingTag	⇒	< # *BindingTagName* >
TypeSigDef	⇒	**typesig** *TypeSigName* = *TypeSignature* ;
TypeSignature	⇒	*TypeMapping [, TypeSignature]**
	|	*TypeSigName [, TypeSignature]*
TypeMapping	⇒	*Type* -> *Type*

BoxDef ⇒ **box** *BoxName* (*BoxSignature*) ;

BoxSignature ⇒ ⌈*BoxType* ⌉ -> ⌈*BoxType* ⌈ | *BoxType* ⌉* ⌉

BoxType ⇒ (⌈*RecordEntry* ⌈ , *RecordEntry* ⌉* ⌉)

NetDef ⇒ **net** *NetName* ⌈ (*NetSignature*) ⌉ *NetBody*

NetSignature ⇒ *TypeSignature*
| *Type* -> ...

NetBody ⇒ ⌈ { ⌈*Definition* ⌉* } ⌉ **connect** *TopoExpr* ;

TopoExpr ⇒ *BoxName*
| *NetName*
| *Sync*
| *Filter*
| *Combination*
| (*TopoExpr*)

Filter ⇒ [*Pattern* -> ⌈*GuardedAction* ⌉* *Action*]
| [ ]

Pattern ⇒ { ⌈*RecordEntry* ⌈ , *RecordEntry* ⌉* ⌉ }

GuardedAction ⇒ **if** < *TagExpr* > **then** *Action* **else**

Action ⇒ ⌈*RecordOutput* ⌈ ; *RecordOutput* ⌉* ⌉

RecordOutput ⇒ { ⌈*OutputField* ⌈ , *OutputField* ⌉* ⌉ }

OutputField ⇒ *FieldName* ⌈ = *FieldName* ⌉
| < *TagName* ⌈ = *TagExpr* ⌉ >

TagName ⇒ *SimpleTagName* | *BindingTagName*

TagExpr ⇒ *TagName*
| *IntegerConst*
| `(` *TagExpr* `)`
| *UnaryOperator TagExpr*
| *TagExpr BinaryOperator TagExpr*
| *TagExpr* `?` *TagExpr* `:` *TagExpr*

UnaryOp ⇒ `!` | `abs`

BinaryOp ⇒ *ArithmeticOp*
| *ComparisonOp*
| *RelationalOp*
| *LogicalOp*

ArithmeticOp ⇒ `*` | `/` | `%` | `+` | `-`

RelationalOp ⇒ `==` | `!=` | `<` | `<=` | `>` | `>=`

LogicalOp ⇒ `&&` | `||`

ComparisonOp ⇒ `min` | `max`

Sync ⇒ `[|` *GuardPattern* [`,` *GuardPattern*]+ `|]`

GuardPattern ⇒ *Pattern* [**if** `<` *TagExpr* `>`]

Combination ⇒ *Serial* | *Star* | *Parallel* | *Split*

Serial ⇒ *TopoExpr SerialComb TopoExpr*

Star ⇒ *TopoExpr StarComb Terminator*

Terminator ⇒ *GuardPattern* [, *GuardPattern*]*

Parallel ⇒ *TopoExpr ParallelComb TopoExpr*

Split ⇒ *TopoExpr SplitComb Tag*

SerialComb ⇒ `..`

StarComb ⇒ `*` | `**`

ParallelComb ⇒ `|` | `||`

SplitComb ⇒ `!` | `!!`

Bibliography

[1] Kahn, G.: The semantics of a simple language for parallel programming. In Rosenfeld, L., ed.: Information Processing 74, Proc. IFIP Congress 74. August 5-10, Stockholm, Sweden, North-Holland (1974) 471–475

[2] Ashcroft, E.A., Wadge, W.W.: Lucid, a nonprocedural language with iteration. Communications of the ACM **20** (1977) 519–526

[3] Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data-flow programming language LUSTRE. Proceedings of the IEEE **79** (1991) 1305–1320

[4] Berry, G., Gonthier., G.: The Esterel synchronous programming language: Design, semantics, implementation. Science of Computer Programming **19** (1992) 87–152

[5] Binder, J.: Safety-critical software for aerospace systems. Aerospace America (2004) 26–27

[6] Caspi, P., Pouzet, M.: Synchronous Kahn networks. In Wexelblat, R.L., ed.: ICFP '96: Proceedings of the first ACM SIGPLAN international conference on Functional programming. (1996) 226–238

[7] Caspi, P., Pouzet, M.: A co-iterative characterization of synchronous stream functions. In Bart Jacobs, Larry Moss, H.R., Rutten, J., eds.: CMCS'98, First Workshop on Coalgebraic Methods in Computer Science Lisbon, Portugal, 28 - 29 March 1998. (1998) 1–21

[8] Michael I. Gordon *et al*: A stream compiler for communication-exposed architectures. In: Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA. October 2002. (2002)

[9] Stephens, R.: A survey of stream processing. Acta Informatica **34** (1997) 491–541

[10] Babcock, B., et al.: Models and issues in data stream systems (invited paper). In: Proc. of the 21st ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems (PODS 2002), Wisconsin, May 2002. (2002) 1–16

[11] Turner, D.A.: An approach to functional operating systems. In Turner, D.A., ed.: Research topics in Functional Programming. Addison-Wesley University Of Texas At Austin Year Of Programming Series. Addison-Wesley Publishing Company (1990) 199–217

[12] Stefanescu, G.: An algebraic theory of flowchart schemes. In Franchi-Zannettacci, P., ed.: Proceedings 11th Colloquium on Trees in Algebra and Programming, Nice, France, 1986. Volume LNCS 214., Springer-Verlag (1986) 60–73

[13] Broy, M., Stefanescu, G.: The algebra of stream processing functions. Theoretical Computer Science (2001) 99–129

[14] Stefanescu, G.: Network Algebra. Springer-Verlag (2000)

[15] Shafarenko, A.: Stream processing on the grid: an array stream transforming language. In: SNPD. (2003) 268–276

[16] Scholz, S.B.: Single Assignment C — efficient support for high-level array operations in a functional setting. Journal of Functional Programming **13** (2003) 1005–1059

[17] Grelck, C.: Shared memory multiprocessor support for functional array processing in SAC. Journal of Functional Programming **15** (2005) 353–401

[18] Grelck, C., Penczek, F.: Implementing S-Net: A Typed Stream Processing Language, Part I: Compilation, Code Generation and Deployment. Technical report, University of Hertfordshire, Department of Computer Science, Compiler Technology and Computer Architecture Group, Hatfield, England, United Kingdom (2007)

[19] Grelck, C., Scholz, S.B., Shafarenko, A.: Coordinating Data Parallel SAC Programs with S-Net. In: 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS'07), Long Beach, California, USA, IEEE Computer Society Press, Los Alamitos, California, USA (2007)

[20] Grelck, C., Scholz, S.B.: SAC: A functional array language for efficient multithreaded execution. International Journal of Parallel Programming **34** (2006) 383–427

Authors

Haoxuan Cai is a PhD student at Imperial College London. He works on the S-Net type system and has implemented the type inference system of the S-Net compiler.

Dr Clemens Grelck is a Principle Lecturer at the University of Hertfordshire and an Assistant Professor in Compiler Design at the University of Amsterdam. He works on the design of S-Net and leads the development activities.

Philip Hölzenspiess is a PhD student at the University of Twente. He has contributed the denotational semantics of S-Net.

Jukka Julku is an MSc student at Helsinki University of Technology and works for VTT Technical Research Centre of Finland. He has implemented large parts of the S-Net compiler and contributed sections on interfacing and metadata to this report.

Frank Penczek is a PhD student at the University of Hertfordshire. He implemented the multithreaded runtime system of S-Net and works on the operational semantics as well as on the extensions for reconfiguration and self-adaptivity.

Dr Sven-Bodo Scholz is a reader in Compilation Technology at the University of Hertfordshire. He contributes to the design of S-Net.

Prof Alex Shafarenko is a professor of Software Engineering at the University of Hertfordshire. He initiated the development of S-Net and works on language design and applications.