

Report on S-Net
A Typed Stream Processing Language

— **Part I** —

Foundations, Record Types and Networks

— **DRAFT** —

Version 0.7

June 03, 2008

Clemens Greck Alex Shafarenko

University of Hertfordshire
Department of Computer Science
Hatfield, Herts, AL10 9AB
United Kingdom

Abstract

This technical report gives a comprehensive account of the design of S-NET, a declarative coordination language and adaptive component technology based on stream processing and subtyping.

Document History

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
- Change in the syntax of filters

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 $\text{\textit{ÆTHER}}$ project consortium

Contents

1	Introduction	9
1.1	S-NET as a Component Technology	9
1.2	S-NET and Stream Processing	9
1.3	S-NET at a Glance	11
2	Types and Subtyping	13
2.1	Record Types	13
2.2	Record Subtyping	15
2.3	Type Signatures	16
2.4	Type Coercion	16
2.5	Flow Inheritance	17
2.6	Box Subtyping	18
2.7	Monotonicity	20
2.8	S-NET and Object-Orientation	21
3	Network Description Language	23
3.1	Overview	23
3.2	User-Defined Boxes	25
3.3	The Filter Box	26
3.4	The Synchrocell	28
3.5	Network Combinators	30
3.6	Deterministic Combinators	33
3.7	Combinator Associativity and Priority	34
3.8	S-NET Programming in the Large	34
4	Type Inference and Semantics	35
4.1	Foundations	35
4.2	Serial Composition	36
4.3	Serial Replication	38
4.4	Parallel Composition	42
4.5	Type Signature Completion	43
4.6	Parallel Replication	44
5	Interfaces	46
5.1	Atomic Box Implementation	46
5.2	Input/Output and the Outside World	48
5.3	Box Language Binding	48

6	Metadata	49
6.1	Purpose of Metadata	49
6.2	Metadata XML Elements	49
6.2.1	Network metadata	49
6.2.2	Box metadata	50
6.2.3	Graphical observers	51
6.3	Associating Metadata with S-NET Definitions	51
7	Examples	53
7.1	Factorial Numbers: a Functional Perspective	53
7.2	Factorial Numbers: an Imperative Perspective	57
7.3	Solving Sudokus with S-NET and SAC	60
7.4	Particles-in-cells simulation	64
8	Conclusion	66
A	Complete Syntax of S-Net	67

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 telecom 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 coarse 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 functions¹, 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

¹but they could have been any self-contained procedures rather than pure functions, as long as the only access they had to each other's state was via stream communication

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.

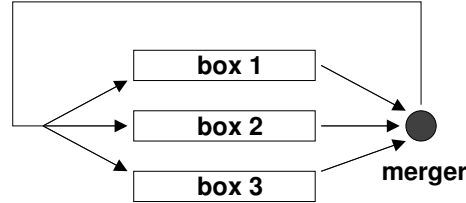


Figure 1.1: The Turner scheme

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 an atomic box produces none, one or more data items on its output stream. The functional properties of atomic boxes enable them to 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 synchronicity 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 an atomic 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.

<i>Type</i>	\Rightarrow	$RecordType [\mid Type]$ $ $ $TypeName [\mid Type]$
<i>RecordType</i>	\Rightarrow	$\{ [RecordEntry [, RecordEntry]^*] \}$
<i>RecordEntry</i>	\Rightarrow	$Field \mid Tag$
<i>Field</i>	\Rightarrow	$FieldName$
<i>Tag</i>	\Rightarrow	$SimpleTag \mid BindingTag$
<i>SimpleTag</i>	\Rightarrow	$\langle SimpleTagName \rangle$
<i>BindingTag</i>	\Rightarrow	$\langle \# BindingTagName \rangle$
<i>TypeDef</i>	\Rightarrow	type $TypeName = Type ;$

Figure 2.1: Syntax definition of S-NET types and type definitions

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

⁰The non-terminal symbols *TypeName*, *FieldName* and *TagName* uniformly refer to identifiers. We only distinguish them here for illustration.

```
{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 sent 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 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.

$$\begin{array}{lcl}
 \textit{TypeSignature} & \Rightarrow & \textit{TypeMapping} [, \textit{TypeSignature}]^* \\
 & & | \textit{TypeSigName} [, \textit{TypeSignature}] \\
 \textit{TypeMapping} & \Rightarrow & \textit{Type} \rightarrow \textit{Type} \\
 \textit{TypeSigDef} & \Rightarrow & \mathbf{typesig} \textit{TypeSigName} = \textit{TypeSignature} ;
 \end{array}$$

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

$$\{a,b\} \mid \{c,d\} \rightarrow \{x\} \mid \{y\}$$

is equivalent to the type signature

$$\begin{array}{l}
 \{a,b\} \rightarrow \{x\} \mid \{y\}, \\
 \{c,d\} \rightarrow \{x\} \mid \{y\}
 \end{array}$$

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 $(\{x, y\} \rightarrow \{x, y\})$. 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 $v^{[i]} \rightarrow \tau^{[i]}$, $i \in [1, \dots, n]$, be the type signature of a box X . Furthermore, let each output type $\tau^{[i]}$ have m_i variants $\tau^{[i]} = \{w_1^{[i]}, \dots, w_{m_i}^{[i]}\}$. 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 $\tau_* = \{V_1, \dots, V_{m_k}\}$ 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 name¹. 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 $\tau^{[i]} = \{w_1^{[i]}, \dots, w_{m_i}^{[i]}\}$, $i \in \{1, \dots, n\}$. 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 nonredundant, 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:

$$\frac{f : \tau_1 \rightarrow \tau_2, \tau_1 \sqsubseteq \tau'_1 \quad \tau'_2 \sqsubseteq \tau_2}{f : \tau'_1 \rightarrow \tau'_2}$$

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:

¹Obviously, an implementation is free to simply switch references.

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

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

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

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

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

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

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

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

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 well² Here

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

and μ is the number of variants in τ .

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

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

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

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

and

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

²Note 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.

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

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

and

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

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 $(n, m)v^{[i]} \rightarrow \tau^{[i]}$ is considered monotonic iff*

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

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 time³.

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

³To be precise, S-NET has a dynamic connectivity mechanism (the ! combinator); however, 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.

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 runtime 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. We call these networks S-Nets, as well.¹ 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).

<i>SNet</i>	\Rightarrow	$[\textit{Definition}]^*$
<i>Definition</i>	\Rightarrow	$\textit{TypeDef} \mid \textit{TypeSigDef} \mid \textit{NetDef} \mid \textit{BoxDef}$
<i>NetDef</i>	\Rightarrow	net <i>NetName</i> [(<i>NetSignature</i>)] <i>NetBody</i>
<i>NetSignature</i>	\Rightarrow	$\textit{TypeSignature}$ $\textit{Type} \rightarrow \dots$
<i>NetBody</i>	\Rightarrow	$[\{ [\textit{Definition}]^* \}] \textbf{connect} \textit{TopoExpr} ;$

Figure 3.1: Grammar of S-NET specifications

Fig. 3.1 provides a definition of core S-NET syntax. An S-NET specification 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; we do not need to repeat their explanation here.

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 (See Section 2.6 for details of box subtyping.). Otherwise, the S-NET compiler will raise a type error.

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, explain-

¹We use different fonts to distinguish between the language S-NET and the SNet networks it describes.

ing its extensional behaviour in a concise way. More importantly, subtyping on type signatures provides an opportunity to specialise or customise given S-Nets to particular needs. For example, a given SNet may support more input variants than are actually needed in a certain context. Restricting the input type of the SNet 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 SNet 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 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 S-NET-expressions are made up of instances of S-Nets in the current scope referred to be their names as well as instances of two built-in boxes: *filter boxes* for housekeeping tasks and *synchrocells* for synchronisation between records. They are explained in detail in the following two sections. We elaborate on the operators of our network topology expression language in Sections 3.5 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 right behind the end of the definition. This deliberately precludes any recursive network definition because a network may not 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 **/*.

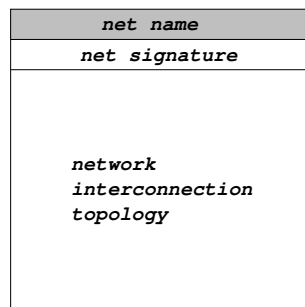


Figure 3.2: Graphical representation of S-NET networks

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 7.

3.2 User-Defined Boxes

S-NET essentially is a language for specifying hierarchical networks of boxes statically interconnected by typed streams. We call these networks S-Nets, as well² 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).

$$\begin{array}{ll}
 \text{BoxDef} & \Rightarrow \quad \mathbf{box} \text{ BoxName } (\text{BoxSignature}) \text{ BoxBody} \\
 \text{BoxSignature} & \Rightarrow \quad \text{BoxType} \rightarrow \text{BoxType} [\mid \text{BoxType}]^* \\
 \text{BoxType} & \Rightarrow \quad ([\text{RecordEntry} [, \text{RecordEntry}]^*]) \\
 \text{BoxBody} & \Rightarrow \quad \{ \lll \text{BoxLanguageName} \mid \text{Code} \ggg \} \\
 & \quad \mid \\
 & \quad ;
 \end{array}$$

Figure 3.3: Grammar of S-NET box declarations

Fig. 3.3 shows the S-NET syntax for declaring user-defined atomic 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 types. 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 of records it accepts and what kinds it emits in response they also describe the function call interface to the box language implementation. However, a general type signature may easily be derived from a box signature as is done by the type inference subsystem of the S-NET compiler.

The specification of the intensional behaviour of a box is outside the scope of S-NET. A box is implemented using a compute language (as opposed to S-NET as a coordination language), and its operational behaviour is opaque to S-NET. In general, the code that implements a box is to be found in a separate file. However, for convenience S-NET allows the programmer to inline foreign language code. This code is separated from S-NET code by the separator symbols `<<<` and `>>>`. S-NET does not process nor even parse the code in between these separator symbols. Instead, the code is passed on to the appropriate compiler. For S-NET to know which compiler to take, the foreign language code section starts with a language identification symbol, which is separated from the code by a bar symbol. Site-specific data like the exact compiler name, compiler flags, etc., are extracted from an S-NET configuration file using the language identifier.

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. inlined box language code.

²We use different fonts to distinguish between the language S-NET and the SNet networks it describes.

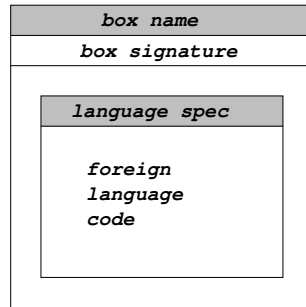


Figure 3.4: Graphical representation of user-defined boxes

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 an atomic box and a box language implementation. 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 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.

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

<i>Filter</i>	\Rightarrow	[<i>Pattern</i> -> [<i>GuardedAction</i>]* <i>Action</i>] []
<i>Pattern</i>	\Rightarrow	{ [<i>RecordEntry</i> [, <i>RecordEntry</i>]*] }
<i>GuardedAction</i>	\Rightarrow	if < <i>TagExpr</i> > then <i>Action</i> else
<i>Action</i>	\Rightarrow	[<i>RecordOutput</i> [; <i>RecordOutput</i>]*]
<i>RecordOutput</i>	\Rightarrow	{ [<i>OutputField</i> [, <i>OutputField</i>]*] }
<i>OutputField</i>	\Rightarrow	<i>FieldName</i> [= <i>FieldName</i>] < <i>TagName</i> [= <i>TagExpr</i>] >
<i>TagName</i>	\Rightarrow	<i>SimpleTagName</i> <i>BindingTagName</i>
<i>TagExpr</i>	\Rightarrow	<i>TagName</i> <i>IntegerConst</i> (<i>TagExpr</i>) <i>UnaryOperator</i> <i>TagExpr</i> <i>TagExpr</i> <i>BinaryOperator</i> <i>TagExpr</i> <i>TagExpr</i> ? <i>TagExpr</i> : <i>TagExpr</i>
<i>UnaryOp</i>	\Rightarrow	! abs
<i>BinaryOp</i>	\Rightarrow	<i>ArithmeticOp</i> <i>ComparisonOp</i> <i>RelationalOp</i> <i>LogicalOp</i>
<i>ArithmeticOp</i>	\Rightarrow	* / % + -
<i>RelationalOp</i>	\Rightarrow	== != < <= > >=
<i>LogicalOp</i>	\Rightarrow	&&
<i>ComparisonOp</i>	\Rightarrow	min max

Figure 3.5: Grammar of S-NET filter box

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.

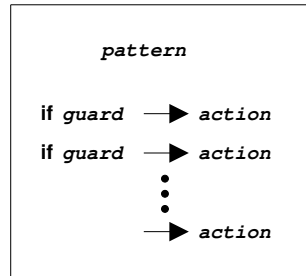


Figure 3.6: Graphical representation of S-NET filters

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 The Synchronocell

The synchronisation cell, or synchronocell for short, is the only “stateful” box in S-NET. Its concrete syntax is given in Fig. 3.7. 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 already familiar 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 synchronocell 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 synchronocell: a synchronocell only accepts records that match at least one of the patterns.

$$\begin{aligned}
 \text{Sync} &\Rightarrow [| \text{GuardPattern } [, \text{GuardPattern }]^+ |] \\
 \text{GuardPattern} &\Rightarrow \text{Pattern } [\text{if } < \text{TagExpr } >]
 \end{aligned}$$

Figure 3.7: Grammar of S-NET synchronocells

More precisely, a synchronocell has storage for exactly one record of each pattern. When a record arrives at a fresh synchronocell, it is kept in this storage and is associated with each pattern that it matches. Any record arriving thereafter is only kept in the synchronocell 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 synchronocell to be pairwise disjoint. Otherwise, we had indistinguishable fields in the output record. If an incoming record matches all patterns of a fresh synchronocell right away, it is immediately passed to the output stream without delay.

Once a synchronocell 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 synchronocell $[| v_1, \dots, v_n |]$ is

$$\begin{aligned}
 \{v_1\} &\rightarrow \{v_1\} | \{v_1, \dots, v_n\} \\
 &\dots \\
 \{v_n\} &\rightarrow \{v_n\} | \{v_1, \dots, v_n\}
 \end{aligned}$$

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 synchronocell shows the following behaviour with respect to flow inheritance. If a synchronocell stores a matching input record, it produces no output in response to this record. Hence, excess record fields, which would bypass the synchronocell otherwise, are discarded. Any record output after successful synchronisation is extended by the excess fields of the last incoming record because the synchronocell produces this output as a response to the input of this record. Last but not least, if a record is passed through the synchronocell in the case of overflow, there is output in response to input and, therefore, the excess fields bypass the synchronocell as usual.

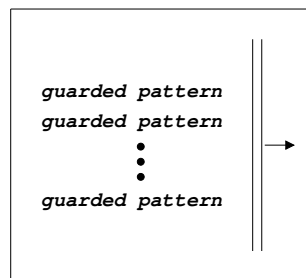


Figure 3.8: Graphical representation of S-NET synchronocells

Fig. 3.8 shows the graphical representation of a synchronocell. 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.

3.5 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 synchronocells. 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 as well as serial and parallel replication. We give a formal definition of the network topology expression language in Fig. 3.9.

<i>TopoExpr</i>	⇒	<i>BoxName</i> <i>NetName</i> <i>Sync</i> <i>Filter</i> <i>Combination</i> (<i>TopoExpr</i>)
<i>Combination</i>	⇒	<i>Serial</i> <i>Star</i> <i>Parallel</i> <i>Split</i>
<i>Serial</i>	⇒	<i>TopoExpr SerialComb TopoExpr</i>
<i>Parallel</i>	⇒	<i>TopoExpr ParallelComb TopoExpr</i>
<i>Star</i>	⇒	<i>TopoExpr StarComb Terminator</i>
<i>Terminator</i>	⇒	<i>GuardPattern</i> [, <i>GuardPattern</i>]*
<i>Split</i>	⇒	<i>TopoExpr SplitComb Range</i>
<i>Range</i>	⇒	<i>Tag</i> [: <i>Tag</i>]
<i>SerialComb</i>	⇒	..
<i>ParallelComb</i>	⇒	
<i>StarComb</i>	⇒	* **
<i>SplitComb</i>	⇒	! !!

Figure 3.9: 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. 3.10.



Figure 3.10: 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. 3.11 illustrates the parallel composition of two networks `foo` and `bar`, i.e. `foo|bar`.

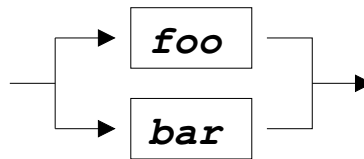


Figure 3.11: Illustration of parallel composition of networks: `foo|bar`

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 $\{a\} \rightarrow \{b\}$ and that of `bar` is $\{a, c\} \rightarrow \{b, d\}$. 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. 3.12 illustrates the operational behaviour of the star combinator for a network $\text{foo}*\{\text{stop}\}$: 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.

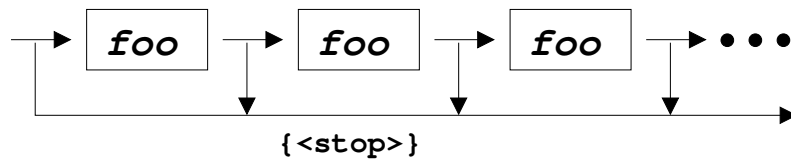


Figure 3.12: Illustration of serial replication of networks: $\text{foo}*\{\text{stop}\}$

Last but not least, the parallel replication combinator “!” takes a network or box as its left operand and either a single tag or a colon-separated pair of tags 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. 3.13 illustrates the operational behavior of the index split combinator for a network $\text{foo}!\text{tag}$. In analogy to serial replication, the instantiation of replicas of the operand network is demand-driven.

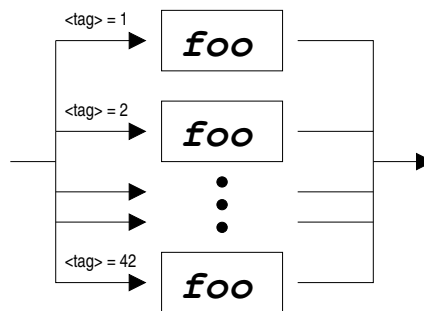


Figure 3.13: Illustration of indexed parallel replication of networks: $\text{foo}!\text{tag}$

If a pair of tags is given, the record is broadcast to all replicas with indices in the range between the value of the first tag and the value of the second tag (both inclusive). In analogy to the parallel composition, the output streams of the replicas are non-deterministically merged into the single output stream of the network.

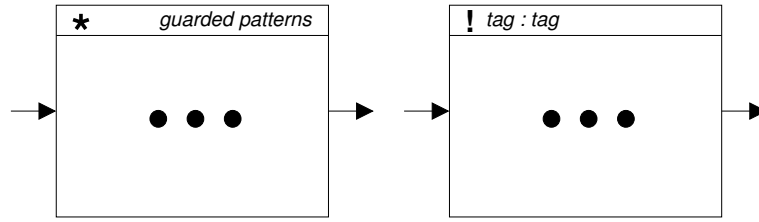


Figure 3.14: Graphical representations of serial (left) and parallel (right) replication

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. 3.11 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.

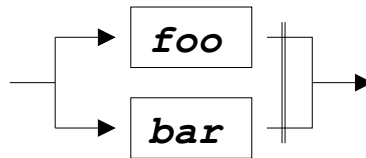


Figure 3.15: Graphical representation of deterministic parallel composition of networks: `foo | bar`

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. 3.14).

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:

$$“|” \prec “||” \prec “.” \prec “*” , “**” , “!” , “!!”$$

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 Inference and Semantics

The chapter on type inference and semantics has not yet been adapted to the recent changes in the language design of S-NET. This causes some inconsistency in terminology and language features between this chapter and the rest of the document. Nevertheless, we consider the following material useful and, therefore, leave it as is for the time being.

4.1 Foundations

Define the alphabet of a box $x : (n, m)v^{[i]} \rightarrow w_j^{[i]}$ as

$$\aleph(x) = \bigcup_{i=1}^n \left(v^{[i]} \cup \bigcup_{j=1}^{m_i} w_j^{[i]} \right),$$

and let Φ^∞ denote the (infinite) set of all possible field names. The semantics of a box \mathbf{x} , i.e. its action on any record $v \in V^0 = \aleph(x) \rightarrow D$, where D is the set of all possible field values, can be defined as a semantic function $\hat{x} : V^0 \xrightarrow{?} \alpha(V^0)$, where $\alpha(s)$ is the set of all sequences composed of members of s . We make \hat{x} total by including into V^0 a special null record (not to be confused with the empty record $\emptyset \rightarrow D$, which is the empty set of fields) ϵ , $V = V^0 \cup \epsilon$ and redefining $\hat{x} : V \rightarrow \alpha V$. Note that \hat{x} fully reflects record subtyping and flow inheritance, using the rules we have defined earlier. This function represents “raw” semantics, which is what S-NET sees when a box is operating in its environment. To be precise, \hat{x} is not necessarily a function; it is generally speaking a family of functions from which one is selected by nondeterministic choice, when the default action is taken in the absence of a specific subtype, as discussed in Section 4.5. To account for the nondeterminism we add an index to the semantic function \hat{x} . A representative of the family \hat{x}_q is a function that corresponds to a particular choice $q \in E(x)$ in nondeterministic variant matching. We will refer to set $E(x)$ as the *event set* of box \mathbf{x} , which is the set representing all available nondeterministic choices of the box. We assume all event sets to be finite and to contain elements of arbitrary nature. Those sets resulting from input variant matching are finite by construction; other event sets occur in the behavioural characteristics of combinators, which we shall discuss below. Those are also finite, albeit potentially large, sets.

Next we incorporate the infinite part of the field-name variety by generalising the domain V to $V^\infty = \Phi^\infty \rightarrow (D \cup \{\epsilon\})$, which results in the following semantic function $\bar{x}_q : V^\infty \rightarrow \alpha(V^\infty)$:

$$\bar{x}_q z = \text{map}(\chi z_2)(\hat{x}_q z_1),$$

where $z = z_1 \cup z_2$, $z_1 \in V$, $z_2 \in V^\infty \setminus V$, and

$$\chi a b = \begin{cases} \epsilon & \text{if } b = \epsilon \\ \text{map } (a \cup) b & \text{otherwise.} \end{cases}$$

Here map , as usual, applies its first argument to every member of the sequence represented by the second argument. The reader will recognise in the above formula the flow inheritance rule for fields that do not occur in the box signature. Note that since such fields do not cause additional nondeterminism, the event set remains the same as with \hat{x} .

Finally, semantic functions apply to a record as an argument, implying that the response of a box to an individual record does not depend on anything else (for a given nondeterministic choice). This is true for primitive S-NET boxes, but not for S-NET networks. The latter generally contain synchronisers and parallel combinators, whose output depends on the current as well as some previous records. The box semantics remains purely functional, except it is now a function from a set of *sequences* of records onto itself, which is the third form of semantic function (after \hat{x} and \bar{x}) that we intend to use. For a primitive box x for which \bar{x} is available, the third form is:

$$\check{x}_q = (\odot/) \circ (\text{map } \bar{x}_q).$$

Here \odot is a sequence concatenation operator, and $\odot/$ is applied to a sequence of sequences to concatenate it into a single sequence. Note that while the first and second form are fully equivalent, the third form is not generally reducible to them: given a third form semantic function, there may not exist a first/second form semantic function that defines it in terms of the above equation. Consequently, in defining the semantics of combinators we must employ exclusively the third form.

As a final observation, consider \check{x}_q for an arbitrary network. It is easy to see that if a_1 is a prefix of a , i.e. there exists a b such that $a = a_1 \odot b$, then also $\check{x}_q a_1$ is a prefix of $\check{x}_q a$, i.e. \check{x}_q is prefix-monotonic. Indeed, when a_1 has been received and responded to, the input sequence can continue to reach a but the output corresponding to a_1 has already been made. Prefix-monotonicity is analogous to causality in concurrency theory. Note, however, that this property only holds as long as \check{x}_q is taken at the same q for different input prefixes, and is immediately destroyed by nondeterminism.

Now we are ready for the discussion of S-NET operators.

4.2 Serial Composition

Consider two networks $a : (n, m)v^{[i]} \rightarrow w_j^{[i]}$ and $A : (N, M)V^{[i]} \rightarrow W_j^{[i]}$. The serial combinator $\mathbf{a}..A$ produces a network that responds to an incoming record ρ by putting it through network a first, and then feeding the output of a to network A . The output of A becomes the output of $\mathbf{a}..A$. Let us define the formal semantics of $\mathbf{a}..A$. Formally it is defined thus:

Definition 4.1 (serial combinator) *The serial combination $\mathbf{S} = \mathbf{a}..A$ of networks \mathbf{a} and A is a network whose behaviour is represented by the semantic family*

$$\check{S}_r = \check{A}_{q'} \circ \check{a}_{q''},$$

where $q' \in E(A)$, $q'' \in E(a)$, $E(S) = E(a) \times E(A)$, and $r = (q', q'') \in E(S)$.

To determine the type signature of $S = \mathbf{a}..A$, one needs to establish the minimum set of fields that ρ must have to be accepted by S . There can be more than one such set, each corresponding to an input variant. The acceptance of a record can be determined on the basis of which fields are

required by a and which additional fields are required to be flow inherited through a by its output record, so that A can accept that record. This results in the following type transformation, which we define in two stages.

First, introduce lexicographic flattening of the type signature whereby a single index k is introduced instead of i and j : $a :: (\nu)v^{[k]} \rightarrow w^{[k]}$, the double colon indicates that flattening has taken place. The new index k enumerates index pairs (i, j) in lexicographic order. For instance if there are two input variants producing three and four output variants, respectively, (i.e. $n = 2$, $m^{[1]} = 3$, $m^{[2]} = 4$) the correspondence between k , i and j is as follows:

k	1	2	3	4	5	6	7
i	1	1	1	2	2	2	2
j	1	2	3	1	2	3	4

Obviously $\nu = \sum_{i=1}^n m^{[i]}$, and the input variants $v^{[k]}$ are no longer pairwise distinct. Note that the flattened form of the signature contains exactly the same information as the standard form, and hence the transformation is reversible. The process of reversal consists in scanning the signature in the ascending order of k , noting the multiplicity of each $v^{[k]}$ and reconstructing $m^{[i]}$, n and $w_j^{[i]}$. Also note that the consistency rule that requires the signature to be sorted in a topological order of $v^{[i]}$ applies to $v^{[k]}$ just as much. The enumeration of $w_j^{[i]}$ in j has been arbitrary so far; in a flattened signature we demand that $w_j^{[i]}$ is topologically sorted in j in *increasing* order, i.e. for any i if $w_a^{[i]} \subseteq w_b^{[i]}$ then their indices in the flattened signature k_a and k_b must satisfy $k_a \leq k_b$ (in a way opposite to the sorting of $v^{[i]}$ in i). The reason for this arrangement will be given momentarily.

Now consider the flattened signatures $a :: (n)v^{[i]} \rightarrow w^{[i]}$ and $A :: (N)V^{[i]} \rightarrow W^{[i]}$. Define a (set-valued) $n \times N$ deficiency matrix

$$D_{ij} = V^{[j]} \setminus w^{[i]},$$

and a dual to it, but independent, excess matrix

$$X_{ij} = w^{[i]} \setminus V^{[j]}.$$

Each element of D_{ij} contains the set of fields that need to be flow inherited through a when the input matches variant $v^{[i]}$. The inheritance is only possible when none of the fields in the set is present in $v^{[i]}$. Provided that this condition is satisfied, an input record of type $v^{[i]} \cup D_{ij}$ is taken through both networks, resulting in the output type $X_{ij} \cup W^{[j]}$. The excess matrix defines the additional ‘‘baggage’’ due to the excess fields which will be flow inherited through network A . Now we can define the whole type transformation for the \dots operator:

$$a..A :: (|R|)\Theta(R),$$

where

$$R = \{v^{[i]} \cup (V^{[j]} \setminus w^{[i]}) \rightarrow (w^{[i]} \setminus V^{[j]}) \cup W^{[j]} \mid (V^{[j]} \setminus w^{[i]}) \cap v^{[i]} = \emptyset\}, \quad (4.1)$$

and $\Theta(X)$ is an indexed sequence of members $p \rightarrow q$ of set X sorted in a topological order of first p (decreasing) and then q (increasing). The set R is required to be nonempty; otherwise a type error is produced. Figure 4.1 depicts the set-theoretical relations between inputs and outputs as defined by Eq 4.1.

Now recall that the right-hand sides of the arrow in a flattened signature are sorted in an increasing topological order. Upon inspection of Eq. 4.1 it is immediately evident that since $v^{[i]}$ is sorted in a decreasing and $w^{[i]}$ in increasing order, the elements of set R are already sorted in the decreasing order of left-hand sides at any fixed j or at any fixed i . For similar reasons there is

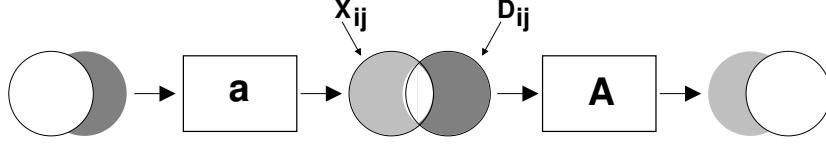


Figure 4.1: Flow inheritance through the .. combinator

increasing order of the right-hand sides at any fixed j (or, again, i). Hence the sorting that Θ is required to do can be made computationally quite efficient by employing merge-sort. The choice between indices i and j as a basis for merge-sort could depend on the overall length n vs N , which one provides the greater multiplicity, etc.

Finally observe that there is a potential for every output variant of \mathbf{a} to be combined with an input variant of A to produce an overall type transformation. This may or may not be the intention of the network designer in connecting the networks in series. It is quite reasonable to expect that certain output variants of \mathbf{a} are meant to correspond to perhaps only a single input variant of A . In general it is desirable to be able control the connection between networks when they are combined in series. This is achieved using already familiar binding tags. In their presence, Eq 4.1 is modified thus:

$$R = \{v^{[i]} \cup (V^{[j]} \setminus w^{[i]}) \rightarrow (w^{[i]} \setminus V^{[j]}) \cup W^{[j]} \mid (V^{[j]} \setminus w^{[i]}) \cap (v^{[i]} \cup B) = \emptyset\}, \quad (4.2)$$

where B is the set of all possible binding tags. The size of set R can be made as small as required by judiciously placing binding tags in the output of \mathbf{a} and the input of A .

4.3 Serial Replication

The closure combinator is denoted by the postfix asterisk. The result of its application is a network produced by infinite replication of the operand network with the replicas connected serially:

$$B..B..B.. \dots$$

The output from the infinite chain occurs at finite distances from its beginning, when a record falls within the fixed point of B .

First of all, we give the formal semantics. To start with, we define a *closure over a set* which is the core formalisation of the above description.

Definition 4.2 (closure over a set) *The closure of a network \mathbf{B} over a set F is a network \mathbf{B}° , whose semantic functions \check{B}_q° is as follows. First define a recurrence relation for an auxiliary third-form semantic function $c_q^{[k]}$. For any record sequence x :*

$$\begin{aligned} c^{[0]} x &= x; E(c^{[0]}) = \emptyset \\ c_q^{[k+1]} x &= \check{B}_{q'} \left(c_{q''}^{[k]} x \right), \text{ where } q = (q', q'') \\ E(c^{[k+1]}) &= E(c^{[k]}) \times E(\check{B}). \end{aligned}$$

Using the above, the closure of \mathbf{B} over a set F is

$$\check{B}_q^\circ = c_q^{[i_\infty]}, \quad (4.3)$$

where $i_\infty = \max_q i_q$, and $i_q = \min\{i \mid c_q^{[i]} \in F\}$. The event index r ranges over $E(\check{B}^\circ) = E(c^{[i_\infty]})$.

The closure over a set gives the semantics of a network chain in which a record sequence propagating along the chain is guaranteed to have fallen inside a certain set F along the way before it is extracted, irrespective of the nondeterministic choice. Now let $F(\check{B})$ be a set of sequences that go through the network B unchanged. In other words, F is a set of fixed points of the network B , i.e. solutions of the equation $(\forall q \in E(\check{B}))\check{B}_q x = x$. It is easy to see that the closure of B over $F(\check{B})$ corresponds to the natural intuition of the replica chain introduced at the beginning of the current section. Indeed a sequence of records which at some point reached a fixed point cannot change by going through the network anymore, hence can be “teleported” through the whole infinite chain to the output.

There is of course no guarantee that i_∞ , which is the position on the chain at which it is guaranteed that the fixed point is reached irrespective of nondeterminism, is finite, hence there is a possibility of a nonterminating closure. Also, the fixed-point set F cannot be produced algorithmically from the algorithm of \check{B} in the general case, hence the only general solution involves comparing the input and output sequences of each B replica in a chain to determine whether or not the fixed point has been reached. Even if it were practical, the fixed point observation for a given record sequence would need to have been done for every member of the large event set (which is selected by the environment with repetitions at run time) before a fixed point can be found. This makes the number of observations hard to limit *a priori*.

In search of a remedy, let us take a closer look at the recurrent process in Definition 4.2. It is easy to see that if for some $i < i_\infty$, and some q , some $a \in F$ is a prefix of $c_q^{[i]}$, then the eventual fixed point, if it is ever reached, will have a as a prefix, too, thanks to the prefix-monotonicity of semantic functions, which was noted earlier. Indeed, since a fixed point is not sensitive to nondeterminism, $c_q^{[i]}$ will have a as a prefix of the output even though q varies with i . This means that it is possible to output a out of the chain even *before* the fixed point is reached¹. In particular, a single-record fixed point can be dispatched immediately to the output without accumulating sequences if it is possible to establish that it always goes through the network B unchanged (if only followed by further output records). Unfortunately, the nondeterminism of B means that even after such a behaviour has been detected once, testing must continue in case any subsequent replica behaves differently.

A solution lies in the type system. Consider a part of the fixed-point set $T \subseteq F$ whose members are single-record sequences that have no value-bearing fields (while they may, and in most cases will, have tags). It is easy to see that such records are not prone to nondeterminism in B since the record type fully determines the record value. Due to flow inheritance, a record whose field-name set v matches the rule $t \rightarrow t$ for some $t \in T$, belongs to F . Consequently, any value bearing fields in v are flow-inherited, and thus left unchanged; this is true irrespective of the nondeterministic choice, since the value-bearing fields bypass the closure network completely. As a result, a sequence comprising a single record $r = v \rightarrow D$ is statically guaranteed to be a member of F . Let us denote the set of all sequences composed of records such as r as T^+ . We can now introduce the following

Definition 4.3 (hard closure) *A hard closure of a box B is a network B^* whose semantic function \check{B}^*_q is the closure of the box B over the set T^+ .*

The use of hard closure to define the effect of B^* is tantamount to allowing all records to propagate along the chain of replicas until each of them matches one of the fixed-point type rules, at which point the sequence can be assumed to have traversed the whole infinite chain.

Definition 4.3 serves as a formal basis of the closure combinator in S-NET and exhausts the issue of semantics. Next we must define the type of B^* given the type of B .

First let us introduce an equivalent form of the box type signature. For a box $B : (n, m)v^{[i]} \rightarrow w_j^{[i]}$ introduce a function $\phi : Q \times \mathbb{N} \rightarrow Q$, where $Q = \mathcal{P}(V \cup W) \cup \{\omega\}$, $V = \bigcup_{i=1}^n v^{[i]}$ and

¹this corresponds to “laziness” in functional semantics

$W = \bigcup_{i=1}^n \bigcup_{j=1}^{m_i} w_j^{[i]}$. Here ω is a special symbol that signifies invalid type, Q the set of all field names used in the type signature and \mathbb{N} is the set of natural numbers up to the maximum number of variants in any output. The function ϕ applied to a record (understood as a set of field-names) and a number k produces the output field-set corresponding to the k th variant of the output type in response to the given input type variant as per the type signature of B with flow inheritance taken into account:

$$\phi(x, k) = \begin{cases} \omega & \text{if } \mu(x) = 0 \vee k > m_{\mu(x)} \vee x = \omega \\ (x \setminus v^{[\mu(x)]}) \cup w_k^{[\mu(x)]}, & \text{otherwise} \end{cases}.$$

Here $\mu(x)$ is the index of the rule that matches x , or 0, if no match can be found. Now denote the mapping of the type signature $\Sigma = (n, m)v^{[i]} \rightarrow w_j^{[i]}$ onto its functional representation $\phi : Q \times \mathbb{N} \rightarrow Q$ as $\Psi(\Sigma)$.

Proposition 4.1 *Ψ is bijective modulo the variant and type orderings that are neutral to the type transformation defined by Σ .*

The proof is constructive. First create a signature $\Sigma^0 = (n, m)v^{[i]} \rightarrow w_j^{[i]}$, where $n = 2^{|A(Q)|}$, $v_i = T_{i, \subseteq} \mathcal{P}(A(Q))$, $m_i = \max_j (\{j \mid \phi(v_i, j) \neq \omega\} \cup \{0\})$ and $w_j^{[i]} = \phi(v_i, j)$ for all $j \leq m_i$. Here $T_{i, \subseteq} X$ is the i th member of set X in some topological order of \subseteq . Next we do a series of deletions from Σ^0 . First find all v^i for which $m_i = 0$ and delete the corresponding rules from the signature. Then for any pair of rules $v^{[i_1]} \rightarrow w_j^{[i_1]}$ and $v^{[i_2]} \rightarrow w_j^{[i_2]}$ such that $v^{[i_1]} \subset v^{[i_2]}$, $m_{i_1} = m_{i_2}$ and $\forall j_2 \exists j_1 w_{j_2}^{[i_2]} = w_{j_1}^{[i_1]} \cup (v^{[i_2]} \setminus v^{[i_1]})$, delete the second rule. It is clear that the first series of deletions removes all rules that denote the response to a type error, hence they were not in the original signature. The second series of deletions removed the rules that could be produced from other rules by flow inheritance. Since ϕ was produced from the type signature by making type mismatch and flow inheritance explicit, it is straightforward that the resulting signature must be the same as the original one, up to the ordering of the rules in a different topological order, and the arbitrary ordering of the variants in output types. Since we do not distinguish between type signatures that only differ in those two orderings, the proposition is proven.

Next we define the serial operator on functions $Q \times \mathbb{N} \rightarrow Q$.

Definition 4.4 *Consider two functions $\phi_{1,2} : Q \times \mathbb{N} \rightarrow Q$. For any $v \in Q$, let $\sigma_v(\phi_1, \phi_2)$ denote the lexicographically ordered series of all pairs $(n_1, n_2) \in \mathbb{N} \times \mathbb{N}$ that satisfy the condition*

$$\phi_2(\phi_1(v, n_1), n_2) \neq \omega,$$

and $\sigma_v^n(\phi_1, \phi_2)$ the n th member of the series. Then the serial combination $\phi_1.. \phi_2$ is a function $\phi_s : Q \times \mathbb{N} \rightarrow Q$ defined thus:

$$\phi_s(v, n) = \phi_2(\phi_1(v, n_1^{[n]}), n_2^{[n]}),$$

where $(n_1^{[n]}, n_2^{[n]}) = \sigma_v^n(\phi_1, \phi_2)$, or if n exceeds the length of the sequence then $(n_1, n_2) = (N, N)$ where N is a large enough number so that $(\forall x \in Q) \phi_{1,2}(x, N) = \omega$.

Now we can strengthen Proposition 4.1 to the following

Proposition 4.2 *Ψ is an isomorphism between the algebras $(\Sigma, ..)$ and $(Q \times \mathbb{N} \rightarrow Q, ..)$ modulo the variant and type orderings that are neutral to the type transformation defined by Σ .*

The proof is obtained by comparing Eq 4.1 with Definition 4.4. The serial combination of type functions is similar, but not identical to the semantic set of the serial combinator. The former describes the *possible* types that are produced in response to an input record type, whereas the latter describes the actual record values produced in response to a given record value. The algebra $(Q \times \mathbb{N} \rightarrow Q, ..)$ is in fact a semigroup:

Proposition 4.3 *The operation \dots as defined by Definition 4.4 is associative.*

To prove this, we must prove that for all $\phi_{1-3} : Q \times \mathbb{N} \rightarrow Q$, $(\phi_1 \dots \phi_2) \dots \phi_3 = \phi_1 \dots (\phi_2 \dots \phi_3)$. By applying both sides to some record v and number n we obtain:

$$\phi_3(\phi_2(\phi_1(v, n_1^L), n_2^L), n_3^L) = \phi_3(\phi_2(\phi_1(v, n_1^R), n_2^R), n_3^R),$$

where on the left hand side $(m, n_3^L) = \sigma_v^n(\phi_1 \dots \phi_2, \phi_3)$, and $(n_1^L, n_2^L) = \sigma_v^m(\phi_1, \phi_2)$. Clearly as n increases, so does first n_3^L as far as possible on the first σ -list, then m will start to increase. As m increases, it causes n_2^L to increase first as far as possible according to the second σ -list, then n_1^L will begin to increase. We conclude that, as n increases, it enumerates triplets (n_1^L, n_2^L, n_3^L) in lexicographic order.

On the right-hand side, $(n_1^R, k) = \sigma_v^n(\phi_1, \phi_2)$; $(n_2^R, n_3^R) = \sigma_{\phi_1(v, n_1)}^k(\phi_1, \phi_2 \dots \phi_3)$. Here similarly, as n increases, first k will rise according to the first σ -list, and so first n_3^R and then n_2^R will rise on the second sigma list, and finally n_1 according to the first σ -list. We conclude that as n increases, it enumerates triplets (n_1^R, n_2^R, n_3^R) in lexicographic order.

Finally, it is easy to see that the left-hand side and the right hand side are each a list (indexed by n) of all non- ω values of $\phi_3(\phi_2(\phi_1(v, n_1), n_2), n_3)$ for a given v and any n_{1-3} , and since we have shown that these lists are sorted in the same way with regard to triplets (n_1, n_2, n_3) , they are equal.†

Now let us return to the issue of type. Since we are interested in hard closure B^* , let us define the projector box for $B : v^{[i]} \rightarrow \tau^{[i]}$ as $B^\rightarrow : v^{[i]} \rightarrow \tau_*^{[i]}$, where $\tau_*^{[i]} = v^{[i]}$ if $v^{[i]}$ matches a member of set T and \emptyset otherwise, where T , as before, contains non-value bearing records from the B fixed-point. The projector box disposes of any input records that do not match the fixed point and passes through those that do.

Observe that

$$B^* \equiv \underbrace{B \dots B, \dots, B \dots B}^{\text{L times}} \tag{4.4}$$

for sufficiently large L Here \equiv denotes the equality of type signatures. Indeed, once a certain power (with respect to the \dots operator) of B yields a record that will be captured by the projector box, any further application of B is ineffectual, hence the signature for $L + 1$ must be a superset of the signature for L . On the other hand, since the alphabet of the box B is finite, there is only a finite variety of rules to include into B^* and a finite capacity not to produce relevant rules for a number of iterations. The latter stems from the finiteness of the whole signature (both relevant and irrelevant parts). Consequently a finite chain must exist that captures the whole type transformation of B^* .

The length of the chain, albeit finite, is hard to limit. One can construct examples where rules collude to transform a record from one type to the next a large number of times until types start to repeat (and hence a whole variety of rules become irrelevant to the fixed point). We have not been able to obtain chain length bounds weaker than exponential in the signature size, which is unsatisfactory for practical purposes.

There is, however, a way to bound the complexity of the type calculation once we take into consideration the fact that in any practical network the size of the type signature of the *result* would be expected to be small. Indeed, it is likely that a large type formula is caused by a design error when an unintended match occurs between some input and output types. Such errors can always be prevented by employing binding tags, but only at the expense of flexibility. Next we will show that the complexity of type calculation is linear in the size of the resulting signature and will propose an appropriate algorithm.

Recall that Propositions 4.2 and 4.3 establish associativity of the serial combinator viewed as a type constructor. Let us change the evaluation order in Eq 4.4 to achieve back chaining, i.e.

$$B^{[0]} = B^\rightarrow; B^{[n+1]} = B..B^{[n]}$$

The advantage of the back chaining is that at each iteration a subset of the eventual type signature is produced. Most importantly though, each iteration must yield at least one new type rule for the process to continue. Indeed, if for some n , $B^{[n+1]} \equiv B^{[n]}$, then

$$B^{[n+2]} \equiv B..B^{[n+1]} \equiv B..B^{[n]} \equiv B^{[n+1]} \equiv B^{[n]},$$

and so $B^* = B^{[n]}$. We conclude that the number of iterations does not exceed the size of the resulting signature, which gives the aforementioned linear complexity bound. Finally observe that the type calculation process can be limited to a reasonable number of output rules, say one hundred: signatures of this size are so unwieldy that they are unlikely to be the result of a deliberate design. Upon reaching the critical size the algorithm could produce appropriate diagnostics (a listing of the rules obtained thus far) and abort.

4.4 Parallel Composition

Consider boxes $A: v_a^{[i]} \rightarrow \tau_a^{[i]}$ and $B: v_b^{[i]} \rightarrow \tau_b^{[i]}$. The choice combinator $A|B$ produces a box C that works as follows. A record appearing at the input of C is compared in type with v_a ; if it matches, then the record is directed to A ; if it matches v_b , the record is directed to B ; if the record matches both v_a and v_b , then the maximum match is used; if the maximum match is ambiguous, then a nondeterministic choice is made between A and B in determining the destination of the record. The outputs of A and B are merged into one stream arbitrarily. Here is a formal definition:

Definition 4.5 (choice) *The choice combination $C = A|B$ of networks A and B is a network whose behaviour is represented by the following semantic family \check{C}_q . For every input stream x , the action of the semantic function is*

$$\check{C}_q x = \mathbf{merge}_{q''''}(\check{A}_{q'} x_A, \check{B}_{q''} x_B),$$

where

$$(x_A, x_B) = \mathbf{split}_{q''''}$$

and

$$E(C) = E(A) \times E(B) \times U^\infty \times U^\infty; q = (q', q'', q''', q'''').$$

Here the two auxiliary functions $\mathbf{merge} : (\alpha(V \rightarrow D), \alpha(V \rightarrow D)) \rightarrow \alpha(V \rightarrow D)$ and $\mathbf{split}_q : \alpha(V \rightarrow D) \rightarrow (\alpha(V \rightarrow D), \alpha(V \rightarrow D))$ are defined as follows. The \mathbf{split}_q function splits the stream into two according to the input types of A and B using maximum match; where the choice is ambiguous, the splitting is done according to the event $q \in U^\infty$, the latter being the set of binary strings of unlimited length. Each bit of q represents one instance of choice. The function \mathbf{msort}_q is the merge of two record streams into a single stream under the control of $q \in U^\infty$.

The set of typing rules for a choice combination is straightforward. For convenience we use rule-sets for boxes A and B , Σ_A and Σ_B , rather than lists of rules (i.e. signatures) as before. Given a rule-set the corresponding signature is obtained immediately by topological sorting.

Let us further partition Σ_A and Σ_B into equivalence classes on the basis of bounding tags. Specifically, introduce a partition

$$P_A = \{p \mid p \subset \Sigma_A \wedge (\forall v_1 \rightarrow \tau_1, v_2 \rightarrow \tau_2 \in p) BT(v_1) = BT(v_2)\},$$

and similar partitions P_B and $P_{A \cup B}$. Also introduce a function $\Pi_G : G \rightarrow P_G$ such that $\Pi_G(r) = p^r$ where p^r is an element of P_G that contains r .

For each element of a partition $p \in P$ introduce $BT(p) = BT(v \rightarrow \tau)$, with any $(v \rightarrow \tau) \in p$. Also denote as $BT(P)$ the set of sets of binding tags:

$$BT(P) = \{s \mid s = BT(p) \wedge p \in P\}.$$

then the result rule set has three parts: $\Sigma_C = \Sigma'_A \cup \Sigma'_B \cup \Sigma_*$, where $\Sigma'_{A,B} \subseteq \Sigma_{A,B}$ are defined as

$$\begin{aligned} \Sigma'_A &= \{r \mid \Pi_A(r) = \Pi_{A \cup B}(r)\} \\ \Sigma'_B &= \{r \mid \Pi_B(r) = \Pi_{A \cup B}(r)\} \end{aligned}$$

The sets $\Sigma'_{A,B}$ are subsets of $\Sigma_{A,B}$ that contain only rules from Σ_A (or Σ_B) that have no common input subtype with a rule from Σ_B (or Σ_A), respectively. Those rules pass straight through to the result set. The rest of the rules have to be combined at the input and flow inheritance should be taken into account.

For the set Σ_* we have:

$$\Sigma_* = \left\{ v \rightarrow \tau_*(v) \mid \tau_* = \left(\bigcup_{(v \rightarrow \tau) \in \Sigma_+} \tau \right) \neq \emptyset \right\},$$

where

$$\begin{aligned} \Sigma_+ = \{v \rightarrow \tau \mid & (\exists (v_A \rightarrow \tau_A, v_B \rightarrow \tau_B) \in (\Sigma_A \setminus \Sigma'_A) \times (\Sigma_B \setminus \Sigma'_B)) \\ & v = v_A \sqcup v_B \wedge \tau = (\tau_A \dashv (v_B \setminus v_A)) \cup (\tau_B \dashv (v_A \setminus v_B))\}, \end{aligned}$$

and the infix operator \dashv denotes extension by flow inheritance:

$$\tau \dashv x = \{v \cup x \mid v \in \tau\}.$$

Note that the least upper bound $v_A \sqcup v_B$ exists only when $BT(v_A) = BT(v_B)$. Also note that the resulting type signature is complete and monotonic by construction.

4.5 Type Signature Completion

Finally, recall that the input types of boxes are required to be complete (see Section 2.3). Our approach to completeness is slightly different from the way we treat monotonicity. Since in the absence of binding tags any two input variants produce a common subtype, which would require a certain kind of output type in response, it is convenient to rely on a default action rather than painstakingly defining all possible responses. Indeed in most cases the input will not deliver such records, and so no matter what the default solution is it will not be used. In those rare cases when a record does match two variants, it is logical to choose a match nondeterministically, which is how S-NET behaves. Indeed, the record matches both variants and there is no reason to prefer one to the other, but it is useful in implementation to be able to choose an alternative that is ready to proceed (as opposed to an alternative that is busy processing some previous record). In S-NET, it is possible to create a network that would profit from such non-determinism, since flow inheritance makes it possible to preserve the unmatched fields no matter which variant has been selected. One can easily imagine an arrangement under which the output of such a nondeterministic box is fed to a further box, which uses the unmatched fields to do some further processing.

```

repeat
  for all pairs of rules  $(i_1, i_2)$  in  $x$  such that  $BT(v^{[i_1]}) = BT(v^{[i_2]})$  do:
    if  $(\exists k < \min(i_1, i_2))v^{[k]} = v^{[i_1]} \cup v^{[i_2]}$ 
      add rule  $(v^{[i_1]} \cup v^{[i_2]}) \rightarrow (\tau_*^{[i_1]} \cup \tau_*^{[i_2]})$ ,
      where  $\tau_*^{[i_1, 2]} = \{w \cup (v^{[k]} \setminus v^{[i_2, 1]}) \mid w \in \tau^{[i_1, 2]}\}$ 
    else if  $\tau^{[k]} \sqsubseteq (\tau^{[i_1]} \cup \tau^{[i_2]})$ 
      continue
    else
      fail
  end
until the signature is monotonic

```

Figure 4.2: Algorithm to make type signature complete

The type signature of a box with the default action taken into account becomes complete in the sense of Definition 2.5. To calculate it, use the following algorithm:

The correctness proof is straightforward, since we only add rules whose absence violates monotonicity and since we can always add those rules if they are not contradicted by the signature, in which case we fail, and finally since there is only a finite number of rules to add to any finite signature before it becomes monotonic.

If the above process fails, it yields a triplet of rules that violate Definition 2.5. Those rules could be either primary, i.e. coming from the original type signature, or secondary, i.e. added by the process, but the latter can eventually be traced back to primary rules. Thus the programmer gets a complete diagnostic. The maximum number of added rules is exponential in the number of intersecting variants for a given set of binding tags, but the latter number in any real design would be very small. Nevertheless, in an implementation the compiler can refuse to complete the signature if it is too large and when, at the same time, no cut-off information is derivable from the environment. Also, any nondeterminism is easily detected by the compiler when boxes are connected into a network, and the type transformation in each box is made certain. A warning then could be issued in case such behaviour is not intentional.

Hereinafter we assume that all type signatures are completed using the above algorithm and will freely use incomplete input types.

4.6 Parallel Replication

This operator is written in the form $B!k$, where B is a network and k is a tag, called the *index* hereinafter. Informally, it creates an array of replicas of network B and assigns each replica a unique integer value of tag k . The input stream must contain only records that have tag k (among others). Each input record is directed to the replica of B associated with the value of k . The output of all replicas is merged into a single stream. The array of replica is assumed to be infinite, which is safe since SNet boxes and networks cannot produce output without input, hence the replicas that have received no input are semantically non-existent. This means that at any given time the significant part of the replica array is finite.

Next we give formal definitions. First, the type signature. Let $\Sigma = v_i \rightarrow \tau_i$ be the signature of B . The signature may have pairs of rules whose input types only differ by the presence of k . Obviously, the rule without k will not be effective since k will definitely be present in any input

record and the second rule of the pair is a better match. Dropping all ineffective rules, and adding the tag k to the input types (applying flow inheritance whenever the tag is not already present there), we get the type signature of $B!k$ as a combination of altered rules which originally lack k at the input and the original rules that require k : $\Sigma_l = \Sigma_1 \cup \Sigma_2$, where

$$\Sigma_1 = \{v \rightarrow (\tau \dashv k) \mid (v \rightarrow \tau) \in \Sigma \wedge k \notin v \wedge (\forall(v' \rightarrow \tau') : \Sigma)v' \neq v \cup \{k\}\}$$

and

$$\Sigma_2 = \{v \rightarrow \tau \mid (v \rightarrow \tau) \in \Sigma \wedge k \in v\}.$$

The index tag takes integer values from an interval $[a, b]$, which we assume to be finite, corresponding to the box language integer data type. Now construct set $T = \{t_i\}$ of $b - a + 1$ unique binding tags; we enumerate this set with subscript i ranging in the same interval. Now construct a third set, a set of replicas of network B , as follows:

$$B_i = \theta(B, t_i),$$

where $\theta(B, t)$ is the same network as B , except each of the input variants v is augmented with the binding tag t_i , $i = a, a + 1, \dots, b$. Now we are ready to define the semantics.

Definition 4.6 *The network $B!k$ is semantically equivalent to the following*

$$\text{isplit}..(B_a|B_{a+1}|\dots|B_b),$$

where

$$\text{isplit} : \{k\} \rightarrow \{t_a, k\}|\{t_{a+1}, k\}|\dots|\{t_b, k\}$$

is a box that expects a singleton record $\{k\}$ and produces records $\{t_k, k\}$ based on the value of k .

Chapter 5

Interfaces

5.1 Atomic Box Implementation

As pointed out in Section 3.2, S-NET is a pure coordination language. As such it provides no means whatsoever to specify computations. For that S-NET relies on an external compute language to implement atomic boxes. S-NET is not fixed to a specific box language and may well combine boxes implemented in different box language in a single network.

However, proper interaction between S-NET and box languages requires that box languages provide a certain infrastructure and code written in box languages 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. Even if a specification contains inlined box language code, S-NET is unable to take advantage of the additional information because syntax and semantics of box languages are unknown to S-NET. This is the price for a clear separation between coordination and computation language and the box language independent design of S-NET. Another consequence of this is that 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. Hence, S-NET only transfers pointer 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. Both binding and non-binding tags are represented by unboxed integers. Values associated with tags are visible to both S-NET and the box language(s). This silently assumes a common representation of integer values across diverse box languages and S-NET. However, in practice any reasonable hardware architecture provides some uniform format to store integer numbers, and any reasonable programming language in our targeted application domain provides access to the genuine hardware-supported data types.

Fig. 5.1 demonstrates the interplay between the S-NET box signature and a C-binding box implementation. The first line declares an atomic box named `example`. It 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`. For the purpose of illustration we use inlined box language code in Fig. 5.1; the same code may alternatively reside in a different file.

```

box example ((a,b,<t>) -> (c,<t>) | (x,y,z))
{
  <<< C | 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 5.1: Example of a C implementation of an atomic box

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 language binding. 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` returns the S-NET handle. Eventually, the box language function returns the given handle.

5.2 Input/Output and the Outside World

5.3 Box Language Binding

Chapter 6

Metadata

6.1 Purpose of Metadata

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.feis.herts.ac.uk
```

Section 6.2 describes the form of the metadata and the currently supported XML metadata elements in more detail.

Any 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. Section 6.3 discusses how exactly metadata is associated with S-NET definitions in the source code.

6.2 Metadata XML Elements

Metadata must contain the XML element `metadata` as its root element. This element may only be omitted if the metadata only contains one net or one box element. If the metadata does not contain an XML-declaration, it is assumed to be XML version 1.0. The sole purpose of the `metadata` element is to act as a container for net and box elements.

6.2.1 Network metadata

The XML element `net` is a container for metadata attached to a specific net definition. Each net element is associated with one corresponding 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 6.3. Net elements themselves may contain one or more net, box or observer elements as children.

6.2.2 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 file or by the element's **name** attribute. Box element's children may contain one or more observer elements or up to one of each of the elements discussed below.

Interface element

The **interface** element describes the box language interface used with box language related to certain box. The box code interface may be declared as name attribute of the interface element or as character data inserted as a child of the interface element.

Boxsource element

The **boxsource** element describes the name of the box language code file related to the box. The box source code file name may be declared as name attribute of the **boxsource** element or as character data as a child of the boxsource element.

Boxfunction element

The **boxfunction** element describes the name of the box function used in the box. The box function name may be declared as name attribute of the boxfunction element or as character data as a child of the element.

Compiler element

The **compiler** element describes the name of compiler that should be used to compile the box language code. The compiler name may be declared as name attribute of the compiler element or as character data as a child of compiler the element.

Compilerflags element

The **compilerflags** element describes the flags that should be used in the compilation command when compiling the box language code. The compiler flags may be declared as character data inserted as a child of the compilerflags element.

Ldflags element

The **ldflags** element describes the linker flags that should be used in the compilation command when compiling the box language code. The flags may be declared as character data as a child of the ldflags element.

6.2.3 Graphical observers

The `observer` element allows a graphical observer to be attached to an S-NET network definition or box declaration. An observer is an optional S-NET entity that can be used to observe the contents of records as they move through streaming network. Observers may have a `type` attribute, which specifies what traffic the observer observes. The attribute may have the following values: “before”, “after” and “both”. They specify whether records passing the network or box are examined before, after or at both ends of the net or box. If the `type` attribute is omitted, a default value “both” is used instead.

Observer elements may have one or more `field`, `tag` and `btag` elements as children. Each of these elements specifies one S-NET field, tag or binding tag that the observer should report. If no specific fields, tags or binding tags are defined, then all passing data is examined.

Each `field`, `tag` or `btag` element must either have name attribute that contains the label of the entity to be observed, or have the label as child element in form of character data.

6.3 Associating Metadata with S-Net Definitions

This section outlines the basic ideas how the metadata is associated to S-NET definitions in the S-NET code files.

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. Value that does not contain any slashes is considered to mean S-NET definition in the current scope. Element with empty name attribute or no name attribute at all is considered to be in error.

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 the name does not point to any S-NET definitions then the metadata element and all elements under it 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. 6.1 gives an example of how metadata is associated to the S-NET definitions. The metadata inside the net C in file A.snet is associated with the box B and contains metadata used to compile the box language code correctly. Metadata from another file metadata.md could be used to attach a graphical observer in to the same box B.

```
file A.snet:

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

    <metadata xmlns="snet.feis.herts.ac.uk">
      <box name="B">
        <interface name="C2SNet">
          <boxsource name="boxfuns.c">
            <boxfunction name="funB">
              <compiler name="gcc">
                <compilerflags>-Wall -O2 -g -c</compilerflags>
                <ldflags>-L/path/to/libs -lmylib1 -lmylib2</ldflags>
              </box>
            </metadata>
          </connect B;
        </connect C;

file metadata.md:

<box name="A/C/B" xmlns="snet.feis.herts.ac.uk">
  <observer type="after">
    <tag name="T" />
  </observer>
</box>
```

Figure 6.1: Example: S-NET with metadata both inlined and located in a separate file

Chapter 7

Examples

7.1 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 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 7.2. 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 application 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. See Section 7.4 for a more representative application of this kind.

```
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
                        end
      val m = facit (n,1)
  in (n,m)
  end
```

Figure 7.1: Factorial function in Standard ML

Fig. 7.1 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 7.2: 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. 7.2. 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. 7.1. 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. 7.3 shows a graphical representation of the network topology of `facit` that is equivalent to the textual specification following the key word `connect` in Fig. 7.2. 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.

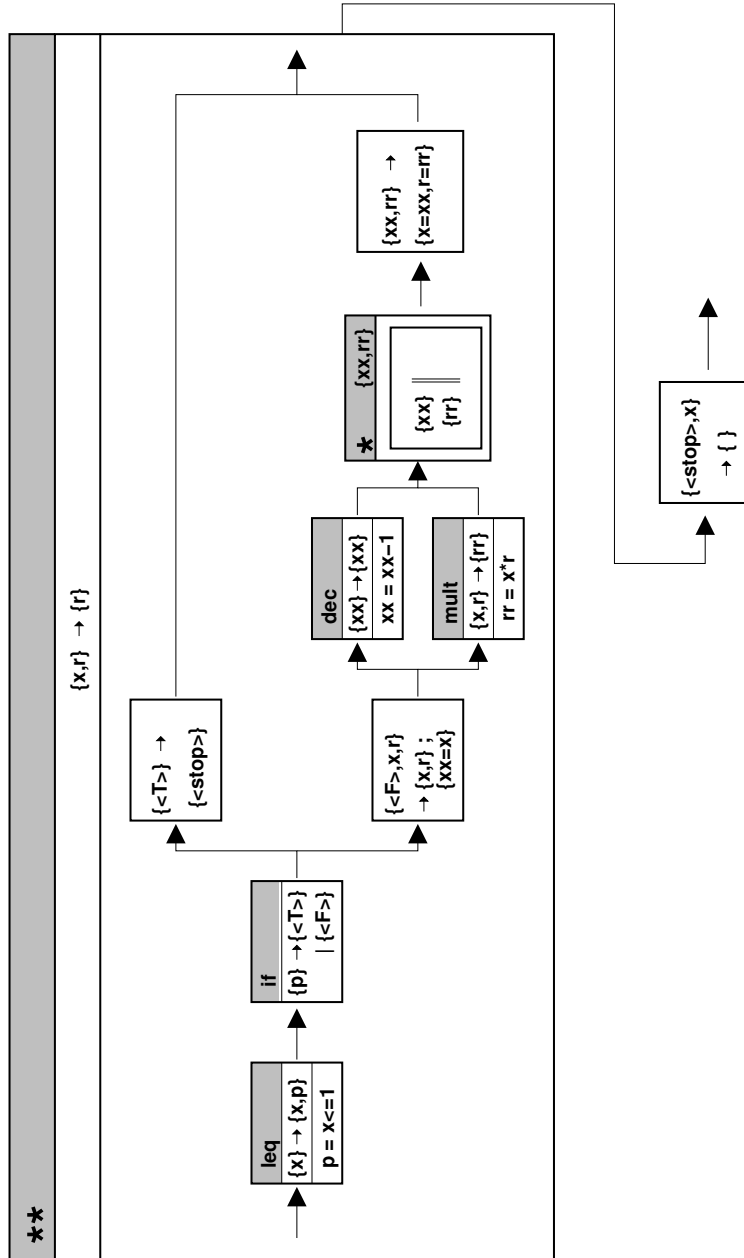


Figure 7.3: Graphical representation of network *facit*

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 3.5) 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 3.4 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 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, one atomic 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 S-Nets in order to express and to exploit concurrency.

7.2 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. 7.4 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 7.4: Computing a single factorial number in C

The S-NET network `factorial` shown in Fig. 7.5 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. 7.5 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 7.5: 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 3.5) 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. 7.5. 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 [18] for a more elaborate case study.

7.3 Solving Sudokus with S-Net and SaC

This example investigates the interplay between S-NET and the box language SAC [16, 19]. 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 [18].

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, 19].

Fig. 7.6 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. 7.6, 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 7.6: 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. 7.6. 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 application¹, we can directly spot 2 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

¹It should be mentioned here that this algorithm leads to code that typically solves 9 by 9 sudokus in far less than a second. However, as sudokus can be played on any board of size $n^2 \times n^2$ parallelisation becomes essential for bigger puzzles.

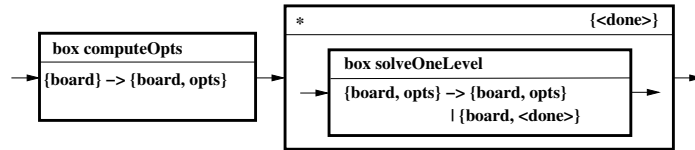


Figure 7.7: Initial S-NET sudoku solver

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. 7.7 shows our initial S-NET solution. We replace the original SAC function `solve` by the S-NET box implementing SAC function `solveOneLevel` shown in Fig. 7.8. 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 7.8: 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 7.6.

If we assume that each S-NET box is actually run by an individual process/thread, the crucial question now is: to what extent 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 $(n - p)^{\text{th}}$ 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.

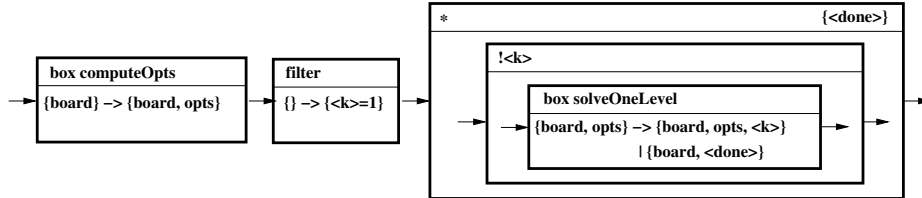


Figure 7.9: Refined S-NET with full unfolding

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. 7.9 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. 7.6 resulting in the S-NET shown in Fig. 7.10.

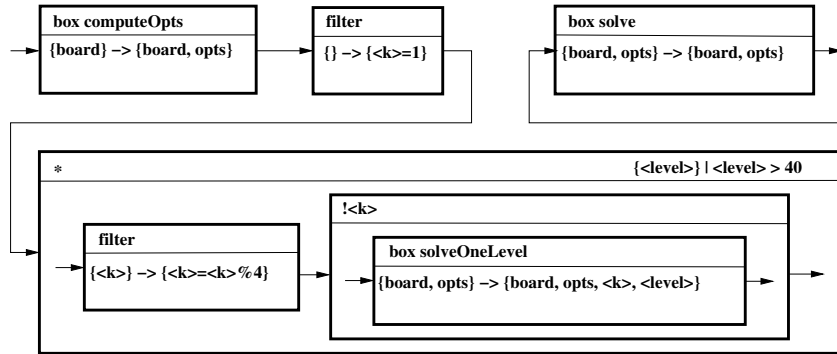


Figure 7.10: Sudoku S-NET with throttled unfolding

7.4 Particles-in-cells simulation

This section will introduce a large concurrent application which we shall design top-down using S-NET. The application is in the area of plasma physics modelling. It will serve as a computational science example of S-NET applicability to large distributed problems. The reader may be taken aback by our choosing a problem that clearly requires application area knowledge to engage with, but we would like to offer reassurances that this example is uniquely amenable to abstraction. In fact, in the sequel no knowledge of physics would be required at all, just commonsense ideas about cause-and-effect relationships in a world where objects create force fields and the latter act on the former. The details of both object and fields and the peculiarities of the computational methods relevant to these will be left out completely. It is the coordination and concurrency aspect of the application that we will dwell upon, and also the amazing fact that the solution can be designed almost entirely in the abstract, defining concurrent communicating structures and the required application-specific components.

We are not seeking to demonstrate the state of the art in computational physics, nor even a fully realistic working simulation. Nevertheless, our example is not very far from it, and the inevitable simplifications do not change either the nature of the example or its specific challenges, only make the latter more tractable in a short section in an otherwise computer-science publication.

Our starting point was the report [], where the following problem was presented. Imagine a large number of particles, each characterised by its position and velocity. The particles have the property that they create a forcefield around them which can be quantified by solving certain equations that depend on the particle density and current. The force field is calculated for an array of spacial positions; each particle located near one of those position will be affected with the corresponding force. There is another set of equations that determine the next position of a particle, given the force acting on it at a certain time. The time in this simulation is discrete (which is normally the case with most computational physics) and is measured in “ticks”. Consequently the general solution algorithm is as follows.

For each tick: assume the availability of location/velocity data for all the particles of the simulation and a static constant-strided grid of spatial positions; perform the following steps.

1. determine the particle density and current distribution over a spatial grid using the available positions and velocities of the particles. Use interpolation and assign parts of a particle to the nodes of the grid nearest to its position. Each particle carries a unit of density and a unit of current (which should be multiplied by its velocity to determine the contribution of the particle to the current at its location). Consequently two grids are created at the end of

this step: the density grid and the current grid.

2. based on the current/density grids obtained at the previous step, solve the field equations and obtain the force field distribution over the spatial grid.
3. using the force field data from the previous step, find by interpolation the force acting on each particle using its position. Solve the motion equations on the basis of the forces obtained, and find the positions of all the particles at the end of the tick

Put simply, the simulation calculates forces and new particle positions at every tick. Although we cannot care less how these calculations are done, we concern ourselves with the flow of data across the computational boxes, its partitioning and synchronisation.

For the sake of simplicity let us radically reduce the number of dimensions in our space grid to just one, so that the force field depends on only one coordinate, the particles can only move along one axis and both the velocity and the position of each particle are scalars. It is known from physics (and the reader is advised to take our word for it) that in a one-dimensional world, force fields only depend on the density of particles and not on their current.

Next, let us assume that the field and the particle motion computations take commensurable amounts of time, so that neither can be neglected in the analysis of computational complexity. The challenge then will be to define a concurrent approach that allows the work to be split between n processors. Naturally S-NET has no notion of processor, but it is reasonable to expect that explicit parallel solutions can be presented in a form such as `Slave!(<p>)` where the tag `<p>` carries a virtual processor number. The tag name could then be communicated to an S-NET run-time system so that a corresponding task distribution may be achieved. In this section we will assume that this is the case, and reserve the tag `<p>` for such purposes. With this in mind, here is a simple solution to the particle problem, see fig 7.11.

```
net sim({x,v, <d>} -> {<out>,rho,phi}) {
...
...
}
connect initially..(solve!!<d>)*(<out>)
```

Figure 7.11: A skeleton solution

The ellipses in the figure are place holders for additional net definitions that describe the solution in detail; however the connect formula is already quite informative: the whole application is an unfolding pipeline, from which the answer is extracted in-order, based on the tag `<out>`. Each stage of the pipeline is a set of replicas of a solution network `solve` that takes a portion of the particle set and a portion of the space grid to the next time tick. The `solve` net itself relies on application boxes and various S-NET elements for its functionality; defining it is our next task. Note that the application net `sim` only accepts one kind of record: namely a record having an array of particle locations x , a conforming array of particle velocities v , and the processor tag. This record is supplied by the environment, with x and v reflecting the initial locations of the particles. Based on the record, the box `initially` prepares some starting values for the computational process described by the rest of the formula. The net `sim` produces output records which contain the particle density ρ and the field strength ϕ spatial arrays from the tick whose number is contained in `<out>`.

Chapter 8

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

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

<i>BoxDef</i>	⇒	box <i>BoxName</i> (<i>BoxSignature</i>) <i>BoxBody</i>
<i>BoxSignature</i>	⇒	<i>BoxType</i> -> <i>BoxType</i> [<i>BoxType</i>]*
<i>BoxType</i>	⇒	([<i>RecordEntry</i> [, <i>RecordEntry</i>]*])
<i>BoxBody</i>	⇒	{ <<< <i>BoxLanguageName</i> <i>Code</i> >>> }
		;
<i>NetDef</i>	⇒	net <i>NetName</i> [(<i>NetSignature</i>)] <i>NetBody</i>
<i>NetSignature</i>	⇒	<i>TypeSignature</i>
		<i>Type</i> -> ...
<i>NetBody</i>	⇒	[{ [<i>Definition</i>]* }] connect <i>TopoExpr</i> ;
<i>TopoExpr</i>	⇒	<i>BoxName</i>
		<i>NetName</i>
		<i>Sync</i>
		<i>Filter</i>
		<i>Combination</i>
		(<i>TopoExpr</i>)
<i>Filter</i>	⇒	[<i>Pattern</i> -> [<i>GuardedAction</i>]* <i>Action</i>]
		[]
<i>Pattern</i>	⇒	{ [<i>RecordEntry</i> [, <i>RecordEntry</i>]*] }
<i>GuardedAction</i>	⇒	if < <i>TagExpr</i> > then <i>Action</i> else
<i>Action</i>	⇒	[<i>RecordOutput</i> [; <i>RecordOutput</i>]*]
<i>RecordOutput</i>	⇒	{ [<i>OutputField</i> [, <i>OutputField</i>]*] }
<i>OutputField</i>	⇒	<i>FieldName</i> [= <i>FieldName</i>]
		< <i>TagName</i> [= <i>TagExpr</i>] >
<i>TagName</i>	⇒	<i>SimpleTagName</i> <i>BindingTagName</i>

```

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 > ]

```

<i>Combination</i>	\Rightarrow	<i>Serial</i> <i>Star</i> <i>Parallel</i> <i>Split</i>
<i>Serial</i>	\Rightarrow	<i>TopoExpr</i> <i>SerialComb</i> <i>TopoExpr</i>
<i>Star</i>	\Rightarrow	<i>TopoExpr</i> <i>StarComb</i> <i>Terminator</i>
<i>Terminator</i>	\Rightarrow	<i>GuardPattern</i> [, <i>GuardPattern</i>]*
<i>Parallel</i>	\Rightarrow	<i>TopoExpr</i> <i>ParallelComb</i> <i>TopoExpr</i>
<i>Split</i>	\Rightarrow	<i>TopoExpr</i> <i>SplitComb</i> <i>Range</i>
<i>Range</i>	\Rightarrow	<i>Tag</i> [: <i>Tag</i>]
<i>SerialComb</i>	\Rightarrow	..
<i>StarComb</i>	\Rightarrow	* **
<i>ParallelComb</i>	\Rightarrow	
<i>SplitComb</i>	\Rightarrow	! !!

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