

Report on S-Net
A Typed Stream Processing Language

— **Part I** —

Foundations, Record Types and Networks

— **DRAFT** —

Version 0.4

November 28, 2006

Clemens Grellck and Alex Shafarenko

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

Abstract

We propose a view on a data-processing application as a typed streaming network. The arcs of the network represent record-valued data streams and the nodes encapsulate recurrence relations on them. We propose a type system in which both the arcs and the nodes are statically subtyped, with the overall subtyping consistency of the network assured by type reconciliation algorithms. The proposed type system makes extensive use of a homomorphically-restricted subtyping, which, on the one hand, provides for generic node specification and, on the other, supports efficient type inference and type reconciliation.

Contents

1	Introduction	3
1.1	Background and Motivation	3
1.2	Proposed Approach	3
1.3	Record Types	4
1.4	Subtyping	5
1.5	Memory	6
2	Related Work	8
2.1	Stream Processing	8
2.2	Subtyping	10
3	Types and Subtyping	12
3.1	Record Types	12
3.2	Record Subtyping	14
3.3	Type Signatures	15
3.4	Type Coercion	16
3.5	Flow Inheritance	16
3.6	Box Subtyping	18
3.7	Monotonicity	20
3.8	S-NET and Traditional OOP	21
4	Network Description Language	23
4.1	Defining SNETs	23
4.2	The Synchronocell	26
4.3	The Filter Box	28
4.4	Network Combinators	30
5	Type Inference and Semantics	34
5.1	Foundations	34
5.2	Serial Combinator	36
5.3	Closure	38
5.4	Choice operator	43
5.5	Type signature completion	44
5.6	Index splitter	45

6 Interfaces	47
6.1 Atomic Box Implementation	47
6.2 Input/Output and the Outside World	49
6.3 Box Language Binding	49
7 Examples	50
7.1 Defining factorial in S-NET	50
8 Conclusions and Future Work	56
A Complete Syntax of S-Net	57

Chapter 1

Introduction

1.1 Background and Motivation

Component technology is crucial to implementing large systems on chip (SoCs). Indeed the complexity of on-chip solutions can only be overcome by decomposition and abstraction. The former requires application-specific building blocks, the latter demands that formal interfaces be set up between the blocks and the connecting infrastructure. This, of course, is not unique to systems on chip; any object-orientated technology would be based on similar ideas. What is typical of the SoCs is the static nature of the connection between the blocks and the increased role of provable properties in assuring the required functionality of the whole system. One way of viewing a static component network is associated with the concept of stream processing.

1.2 Proposed Approach

This paper proposes a component technology for stream processing, which we have named S-NET. An S-NET is a recursively nested single-input, single-output (SISO) streaming network that combines SISO processing boxes and coordinates their interaction. Processing boxes are atomic: they do not expose any internal content and, hence, are completely “black”. These atomic 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.

S-NET is in fact a coordination language: It provides means to describe the orderly behaviour among boxes and the streaming network used for communication between boxes. Atomic 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.

Atomic boxes communicate with each other and with the execution environment solely by means of data received and sent via typed streams. S-NET allows atomic boxes to be composed into SISO networks, which recursively form composite boxes in further network layers. Composition of boxes involves splitting and merging communication streams depending on their types. It is described using *network combinators*, that are inspired by Stefanescu’s network algebra [1].

The restriction to a single input and a single output stream allows us to use variant types to capture data provenance under a type system. This makes the network topology a type issue alongside all the standard type issues, and opens up an avenue for comprehensive subtyping, which is what usually makes a component technology so effective in the object-oriented world. Still, the multiplicity of input and output streams as often found in stream processing can easily be mimicked: one may think of an SISO entity as one with all the input streams interleaved into a single stream and all the output streams similarly de-interleaved.

S-NET networks are asynchronous: an entity’s output is assumed to be buffered. When processing is done by several components whose results must be combined, a synchronisation facility is generally required. 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 atomic boxes have no state, but can compute.

Finally, we propose genericity and specialisation mechanisms on the basis of static record subtyping. These mechanisms make it possible to statically optimise streaming networks with generic components. They also enable the component designer to provide several versions of a box depending on a subtype. Crucially, S-NET does not require explicit subtype declarations; a subtype inference algorithm is applied to determine the most appropriate subtype.

1.3 Record Types

Data items sent via streams are organised as non-recursive, tagged variant records with arbitrary non-record fields. Consequently, the types associated with streams in an S-NET network are non-recursive, tagged variant record types. 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 knows about how to interpret the data. As far as S-NET is concerned atomic record fields and the corresponding data travelling along the streams are as opaque as the atomic boxes themselves.

Tagged variant records allow a single record to alternatively store different fields. For instance, a geometric body type can include a sphere record with

the fields *centre* and *radius*, as well as an ellipsoid record with the *centre* and three axes, and a cone with a *centre*, *height* and an *apex* angle. A box may be capable of processing all these shapes, in which case a union type is required. To distinguish different members of the union type, which we shall call *variants* hereinafter, S-NET uses pattern-matching and tags.

Recursive record types are not supported in S-NET for the following reasons. A nonrecursive 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. It is important to understand that these path names are static and so all leaves are accessible at once in no particular order¹.

By contrast, a recursive record type can be thought of as a set of nonrecursive 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 (nonrecursive) records at a time by following references. When a recursive data structure is to be communicated, it is common to stream only relevant parts of it, under the control of a client-server protocol, rather than the whole data structure at once. Even when the latter is unavoidable, such data structures are not send in their natural form, but rather in a serialised, ‘marshalled’ stream, which is a stream of nonrecursive records with abstract label fields.

1.4 Subtyping

Subtyping is an important adaptation mechanism of a component technology. An S-NET box may be capable of processing more variants than there are in the incoming typed stream, which should not cause trouble. Likewise, if a box receives a valid variant extended with additional fields, these fields should simply be ignored and passed on to the output so that a further box may process them. These common sense considerations provide the motivation for subtyping. The first two of them constitute conventional record subtyping, whereas the third one is, to our knowledge, a new concept, which we call *flow inheritance*. It is fundamental to S-NET and is used extensively.

Record subtyping is a somewhat controversial issue. On the one hand, it is already part of the mainstream, which is demonstrated by its full adoption by the language Python [2]. Python is strongly (though dynamically) typed, but its strong typing is due to software engineering concerns rather than the pursuit of efficiency.

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 does not restrict subtyping this way, and

¹Since there are no operations on whole records in S-NET, the subrecords are not important, hence it is assumed that all nonrecursive records are flat.

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 with array processing. 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 time².

As records are consumed and produced by S-NET boxes, the boxes themselves must have properties with respect to record subtyping. In particular, since a subset of variants constitutes a subtype, it is useful to know the box reaction to each variant, rather than to the whole type that consists of them. This knowledge can be used to statically determine a lower output type when a lower type is offered to the input type, which is a form of box specialisation. The type signatures of S-NET boxes are formulated accordingly: they list an output type versus an input variant, which we call a *detailed* type signature. To give an example of a similar phenomenon (albeit not subtyping as such) consider type `List` as defined normally with two variants, `Nil` and `Cons`. For a function `List` \rightarrow `List`, such as `map`, the type information available from its semantics is more detailed than the above signature. Assuming *empty* and *nonempty* being subtypes of `List`, the signature of `map` is, in fact,

$$\text{map} : \text{empty} \rightarrow \text{empty}, \text{nonempty} \rightarrow \text{nonempty}$$

A function does not have to respond with a single-variant type to each variant, so the most general detailed signature is in the form $\{v \rightarrow \tau\}$, where v denotes a variant, τ a type (i.e. a set of variants) and the braces denote a set. For instance, a function `tail` : `List` \rightarrow `List` which returns `Nil` when applied to an empty list, has a detailed signature

$$\text{tail} : \text{nonempty} \rightarrow \{\text{empty}, \text{nonempty}\}, \text{empty} \rightarrow \text{empty}.$$

Programming languages tend to treat such properties as dynamic (by only accepting the general signature such as `List` \rightarrow `List`), while S-NET allows them to be statically verified when a detailed signature is present whether explicitly, or implicitly, by the provision of several implementation modules for the same box, differentiated by subtype.

1.5 Memory

It was mentioned before that S-NET processing boxes are stateless. They produce zero, one or more records in response to a single record at the input in a receive-process-send cycle. A cycle has no memory of the previous cycles. However the input and output records can have common fields. These are called ‘recurrent variables’ and could be used for holding persistent data if at least part

²To 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.

of the output is diverted back to input. To function as a box ‘state’, the recurrent variables must be synchronised with any input that the box receives, which is one of the many situations that involve S-NET *synchrocells*. In this example a synchrocell is introduced in a feedback loop of a processing box, but they are equally useful for ‘zipping’ two or more box outputs into a single stream, for matching pairs of records on the basis of common index, etc. However, even in the simple case of recurrent variables, the general solution with a synchrocell allows for box multi-threading (several synchrocells in the feedback loop) and process farming (several cells with several boxes) — all purely by network configuration without touching the ins or outs of the participating boxes. It is also crucial to S-NET that subtyping allows such configurations to be typed and checked automatically, and that generic boxes continue to be specialised correctly as the network topology changes.

It should be noted that the S-NET categorisation of memory as synchrocells outside, and data memory inside, boxes clearly separates two memory aspects which are otherwise combined in conventional programming: memory as work storage for computations and memory as a means of inter-process communication. It is that separation that promotes flexible specification, which assists generic parallel and distributed computing.

Chapter 2

Related Work

2.1 Stream Processing

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 [3] and the language Lucid [4]. 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 80s, a whole host of synchronous dataflow languages sprouted, notably the languages Lustre [5] and Esterel[6], 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 [7]. The authors of Lustre broadened their work towards what they termed synchronous Kahn’s networks[8, 9], 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 [10]. The latest comprehensive survey of stream processing and the underlying theory for it can be found in [11]. There is also a growing activity in *database stream processing* [12] , 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[13] 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 2.1. Note that each processing box has a single input and a single output. This does not lead to a loss of generality due to the fact that a function requiring multiple input streams can be represented as a function of a single stream argument where the elements of the multiple streams are somehow merged into a single sequence of records. Similarly, a single output stream can be split into any given number of secondary output streams by picking out records for each of the output sequences. The issue of how exactly the inputs are merged is a delicate one; an efficient solution would depend on the properties of the function in question. The merging usually benefits from being nondeterministic, as this accommodates the delays incurred in receiving the contributing streams by allowing the first message that arrives to be passed on to the processing function without waiting for its turn. On the other hand, the processing block can be required to be deterministic, in which case it may not be ready to accept a given input at an arbitrary time.

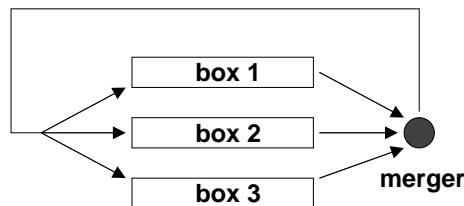


Figure 2.1: The Turner scheme

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

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

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 [14] and recently including models for nondeterministic stream processing developed collaboratively with Broy [1]. 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 [15]). It is interesting to note that apparently the StreamIt team [10] as well as ourselves [16] were unaware of those and had to re-invent them, albeit for purely pragmatic reasons.

2.2 Subtyping

The issue of type inference with atomic subtyping has a long history, too. We cite papers [17, 18, 19, 20, 21] as ones where foundation work was done. Of these, paper [21] is probably the most relevant as it tackles the issue of decidability of general type inference in the presence of subtyping, but it does not bound its cost. The main thrust of our work is towards homomorphism of types and effective constraint-satisfaction algorithms that make type inference possible. This issue was not approached systematically until a simplified theory was given in [22]. Our concept of type homomorphism is consonant to Lievant’s idea of “discrete polymorphism” proposed in [23], where it was suggested that overloadings should be treated as models of a single theory. We believe that h-overloading is less restrictive as it allows higher instances to “expand” the functionality of the lower ones without destroying the consistency between them.

Technically, the most relevant to our work could be the paper by Rehof and Mogensen [24], where a method is described for what they termed a “definite constraint satisfaction problem”. Here all constraints are presented in a form

similar to ours: $v_0 \geq f(v_1, \dots, v_k)$, where f is a nondecreasing function. Then an algorithm is presented, with a complexity linear in the number of constraints, (i.e. quadratic in the number of variables n) which finds the least solution. The main difference is that in [24] the system of constraints is assumed to be *closed*, i.e. all variables are subject to type minimisation within the constraints. In the present paper, we approach a more general problem of constraint satisfaction with unknown external parameters, which are types of the external variables that are *not* subject to minimisation. In our case, the solution is a function of those types. The algorithm from [24] does not apply to such situations. We have proposed a slightly more costly solution, with the cost $O(n^3)$, but which allows external types to be parameters in the type assignment.

Chapter 3

Types and Subtyping

3.1 Record Types

The type system of S-NET supports non-recursive variant records with *record subtyping*. As defined in Fig. 3.1, 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	$TypeName [Type]$ $ RecordType [Type]$
<i>RecordType</i>	\Rightarrow	$\{ [RecordEntry [, RecordEntry]^*] \}$
<i>RecordEntry</i>	\Rightarrow	$Field Tag$
<i>Field</i>	\Rightarrow	$FieldName$
<i>Tag</i>	\Rightarrow	$SimpleTag BindingTag$
<i>SimpleTag</i>	\Rightarrow	$\langle TagName \rangle$
<i>BindingTag</i>	\Rightarrow	$\langle \# TagName \rangle$
<i>TypeDef</i>	\Rightarrow	type $TypeName = Type ;$

Figure 3.1: Syntax definition of S-NET types and type definitions

We distinguish between two different kinds of record entries: *fields* and *tags*. A field is characterised by its *field name*; at runtime it is associated with a value, but this value is opaque to S-NET and may only be generated, inspected or manipulated by using an appropriate box language. A tag is also given by its name, enclosed in angular brackets. However, at runtime it is associated with an integer value that is visible to both: box language code and S-NET. The rationale of tags lies in controlling the flow of records through a network. They

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

should not be misused to hold information that by chance can be represented by integer values. Furthermore, we distinguish between *simple tags* and *binding tags*, which are marked by the hash character (“#”). They show slightly different behaviour with respect to subtyping, as explained in Section 3.2.

We illustrate S-NET types by a simple example from 2-dimensional geometry: For example, we may represent a rectangular by the S-NET type

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

providing fields for the coordinates of the reference point and edge lengths in both dimensions. Likewise, we may represent a circle by the center point coordinates and its radius:

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

Using the S-NET support for variant records we may easily define a type for geometric bodies in general, encompassing both rectangles and circles:

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

Often it is convenient to implicitly name anonymous variants by introducing tags:

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

We refer to types that consist of a single variant only as *record types* because each record at runtime has an exact type description without variants.

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 proper type at any syntactical position that requires a type. For example, we may first define type abstractions for rectangle types and circle types:

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

and use them later on to define the more general type `body` to represent geometric bodies of either kind:

```
type body = rectangle | circle;
```

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

```
body
```

and

```
rectangle | circle
```

and

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

are identical.

3.2 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 3.1 (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 additional 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 body2 = triangle | rectangle | circle
```

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

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 `body2` is again more general than its subtype `body` as it encompasses all three different geometric bodies, whereas `body2` is more specific in ruling out triangles from the set of geometric bodies 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 only is a subtype of another one 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 less record entries. However, this is contrary to the intuition. 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

$$\begin{aligned} \textit{TypeSignature} &\Rightarrow \textit{TypeMapping} [, \textit{TypeMapping}]^* \\ \textit{TypeMapping} &\Rightarrow \textit{Type} \rightarrow \textit{Type} \end{aligned}$$

Figure 3.2: Grammar for S-NET type signatures

position. Changing our definitions of types representing individual geometric bodies to

```
type rectangle = {<#rectangle>, x1, y1, dx2, dy2};
type circle   = {<#circle>, x1, y1, r};
type triangle = {<#triangle>, x, y, dx1, dy1, dx2, dy2};
type body3    = triangle | rectangle | circle
```

using binding tags prevents this and allows us to model our geometric bodies in a more useful way.

Unlike many object-oriented languages like C++ or Java our definition of record subtyping allows any type to have multiple supertypes (which are not in subtype relationship themselves). Without the use of binding tags the type `{}` (i.e. the empty record) is the most common supertype. Otherwise, for each set of binding tags `BT`, `BT` itself is the most common supertype.

3.3 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. 3.2, 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 a 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.

In 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

$$\begin{aligned} &(\{a,b\} \rightarrow \{x\} \mid \{y\}, \\ &\quad \{c,d\} \rightarrow \{x\} \mid \{y\}) \end{aligned}$$

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.

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.

3.4 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 3.2 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 3.1 (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 3.2 (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 3.2, $BT(x)$ 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.

3.5 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 $(\{\mathbf{x}, \mathbf{y}\} \rightarrow \{\mathbf{x}, \mathbf{y}\})$. 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 $\{\mathbf{x}, \mathbf{y}\}$ 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 3.3 (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

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

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.

3.6 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:

Definition 3.4 (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

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

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

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

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

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

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

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.

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.

3.7 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 3.5 (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 $v_j \supseteq v_i$ 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.

3.8 S-Net and Traditional OOP

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 example of type `body`, conventional subtyping would preclude the construction

of a generic box that alters the body position expressed in terms of fields `x1` and `y1` 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 `x1` and `y1` is the same for all variants, for instance, a coordinate shift `x1->x1+a`, `y1->y1+b`. The conventional OOP approach to this would be via a base class with fields `x1` and `y1` and a method `shift` to be inherited by all subclasses. This restricts the design in that there can be more than one set of common fields (which would require multiple inheritance), but more importantly since the significance of a common group of fields may become apparent only when an entirely new processing box is introduced into a streaming network, and in that case a re-design of the class hierarchy may become necessary.

Subtyping by subsetting as introduced in the beginning of this section does allow *a-posteriori* introduction of a supertype (equivalent to a base class), which obviates the re-design. The price to pay in implementation is the price of a runtime coercion (i.e. a selective copy of fields, or an extra level of indirection to avoid the need to copy), since it can no longer be assumed that the fields to be processed are necessarily a prefix of the field list.

Chapter 4

Network Description Language

4.1 Defining S-Nets

S-NET essentially is a language for specifying hierarchical networks of boxes statically interconnected by typed streams. We call these networks S-Nets, as well¹ 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 [25, 26]. Likewise the concrete data travelling along the typed streams is opaque to S-NET and is only meaningful to the box language(s).

Fig. 4.1 provides a definition of core S-NET syntax. An SNet consists of a sequence of definitions of types, boxes (black boxes) and compound network (white boxes). Types and type definitions have already been described in detail in Chapter 3. Hence, we do not repeat the explanation of the type language of S-NET. 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 3.3). 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

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

<i>SNet</i>	\Rightarrow	$[Definition]^*$
<i>Definition</i>	\Rightarrow	<i>BoxDef</i> <i>NetDef</i> <i>TypeDef</i>
<i>BoxDef</i>	\Rightarrow	box <i>BoxName</i> (<i>BoxSignature</i>) <i>BoxBody</i>
<i>BoxSignature</i>	\Rightarrow	<i>BoxType</i> \rightarrow <i>BoxType</i> [<i>BoxType</i>]*
<i>BoxType</i>	\Rightarrow	([<i>RecordEntry</i> [, <i>RecordEntry</i>]*])
<i>BoxBody</i>	\Rightarrow	{ <<< <i>BoxLanguageName</i> <i>Code</i> >>> }
		;
<i>NetDef</i>	\Rightarrow	net <i>NetName</i> [(<i>NetSignature</i>)] [<i>NetBody</i>] <i>Connect</i>
<i>NetSignature</i>	\Rightarrow	<i>TypeSignature</i> <i>Type</i>
<i>NetBody</i>	\Rightarrow	{ [<i>Definition</i>]* }
<i>Connect</i>	\Rightarrow	connect <i>SNetExpr</i> ;

Figure 4.1: Grammar of SNet specifications

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.

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 specification. 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 3.6 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 box, explaining its extensional behaviour in a concise way. More importantly, the subtyping relationship 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 content SNet does not process these entries, flow inheritance ensures that they are added to each outgoing record before leaving the content 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.

Instead of providing a full type signature, the programmer may also choose to solely provide an input type of a network. In this case it is still up to the S-NET compiler to infer the proper output type. From a technical point of view, it would even be feasible to mix pure input types and full type signatures on the level of individual input type variants, i.e. providing output types to some input type variants but omitting them for others. However, we consider this specification freedom rather confusing and, hence, require either a pure input type or a full type signature per box type declaration.

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 SNETs in the current scope referred to be their names as well as instances of two primitive boxes: `filter` and `sync`.

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. The scope of a definition also ends with the completion of a new definition bearing the same name if the new definition is on the same nesting level of network definitions as the original definition. If the re-definition is located in a nested network definition, the scope of the outer definition is suspended for the scope of the inner definition.

A network identifier in a connect expression that is unbound with respect to the above scoping rules is considered to refer to an external network definition. In fact, each S-NET file itself defines a scope within the set of S-NET files in a local file system. More precisely, compiler parameters and environment variables define a sequence of directories to look for S-NET definitions within the file system. Whereas the scoping rules sketched out above apply within each individual file, at most a single network definition within an S-NET file is actually visible outside: the one that bears the same name as the file it is stored in. So, any locally unbound network identifier is supposed to be bound in the global file system level scope. Based on the identity of network name and file name the S-NET compiler is able to identify the corresponding network definition in the local file system (if it exists).

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 `*/`.

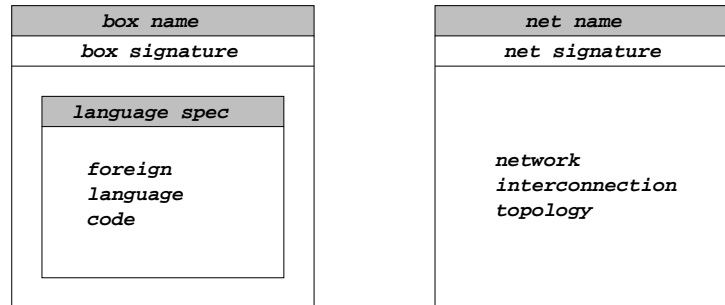


Figure 4.2: Graphical representation of S-NET boxes and networks

Fig. 4.2 sketches out graphical representations of boxes (black boxes) and networks (white boxes). Both contain the box name in a head line, followed by their type information. Given the fact that providing a type signature is optional for networks, this field may well be empty. Graphical representations of partially compiled S-NET programs may feature the inferred type signature here.

In the case of a box the remaining field may remain blank or may contain the inlined box language implementation along with some information on the box language used. In the case of a network the remaining field shows a graphical representation of the interconnection topology. Their building blocks, both textual and graphical, are subject to the following sections.

4.2 The Synchronocell

The synchronisation cell, or synchronocell for short, is the only “stateful” box in S-NET. Its concrete syntax is given in Fig. 4.3. Embedded within `[|` and `|]` parentheses, we find an at least 2-element list of patterns. Syntactically, a pattern merely is a record type. 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 type of the record is a subtype of the pattern. The pattern acts as an input type specification of the synchronocell: a synchronocell only accepts records that match at least one of the patterns.

$$\begin{aligned} \textit{Sync} &\Rightarrow [| \textit{Pattern} [, \textit{Pattern}]^+ |] \\ \textit{Pattern} &\Rightarrow \textit{RecordType} \end{aligned}$$

Figure 4.3: Grammar of S-NET synchronocells

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

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

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

$$\begin{array}{lcl} \{v_1\} & \rightarrow & \{v_1\} \mid \{v_1, \dots, v_n\} \\ & & \dots \\ \{v_n\} & \rightarrow & \{v_n\} \mid \{v_1, \dots, v_n\} \end{array}$$

It reflects the fact that any incoming record may either be passed through in case of an overflow or it may trigger synchronisation, in which case the output record contains fields from all patterns.

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

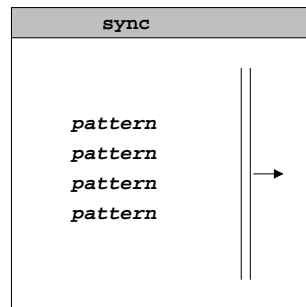


Figure 4.4: Graphical representation of S-NET synchrocells

Fig. 4.4 shows the graphical representation of a synchrocell. Following the name “sync” in the status line, the main field contains the patterns line by line. The double line to the right of the patterns insinuates the synchronisation.

4.3 The Filter Box

The primitive filter box in S-NET is devoted to all kinds of housekeeping operation. 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 with certain values,
- duplicating record fields,
- splitting records,
- ...

<i>Filter</i>	⇒	[<i>Pattern</i>	->	[<i>FilterOut</i>	[;	<i>FilterOut</i>]*]]		
							[]		
<i>FilterOut</i>	⇒		{		[<i>FilterOutField</i>		[,		<i>FilterOutField</i>]*]	}
<i>FilterOutField</i>	⇒		<i>Field</i>		[<-		<i>FieldInit</i>]				
						<i>Tag</i>		[<-		<i>TagInit</i>]		
<i>FieldInit</i>	⇒		<i>Field</i>											
<i>TagInit</i>	⇒		<i>Tag</i>											
						<i>IntegerConst</i>								

Figure 4.5: Grammar of S-NET filter box

Fig. 4.5 shows the syntax of S-NET filter boxes. Enclosed in square brackets their main syntactic constituents are a pattern and a potentially empty semicolon-separated list of actions. Like the input type of any other SNet, the pattern, which syntactically coincides with a record type, acts as a specification for the records the filter box accepts as input. More precisely, the filter box only accepts records as input whose type is a supertype of the pattern.

The list of actions defines how the filter box reacts on an incoming record that fits the input specification. First of all, the list of actions may be empty expressing the fact that the filter consumes any incoming record. Otherwise,

the length of the list determines the number of records produced in response to any incoming record.

Each filter output specification is of a set of record entries 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 filter output specification, it is discarded (at least for this filter output; it may of course still be used in other filter outputs). If a record entry occurs in the filter output 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 an existing field. This may, for instance, be used to rename fields without changing their values. In the case of a tag, we have more choices for initialisation: We may take the value of an existing tag, we may initialise a new tag by an integer constant or we may simply leave out any explicit initialisation in which case the tag value defaults to zero. Future versions of S-NET may even support simple arithmetic operations on tag values.

For example, the following filter box

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

accepts only records that contain fields **a** and **b**. It renames **a** to **c**, discards **b** and adds a new tag **<d>** that is set to 42.

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.

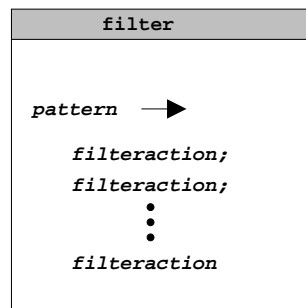


Figure 4.6: Graphical representation of S-NET filters

Type inference for filter boxes is rather straightforward. Essentially, the pattern acts as an input type while the filter action sequence makes up the

output type by simply stripping of record entry initialisations as necessary.

Fig. 4.6 sketches out a graphical representation of S-NET filter boxes.

4.4 Network Combinators

Network topologies are defined using an expression language featuring four network combinators: *serial*, *star*, *choice* and *splitter*. The grammar of network topology specifications is defined in Fig. 4.7 while Fig. 4.8 sketches out the corresponding graphical representations.

<i>SNetExpr</i>	⇒	<i>BoxName</i> <i>NetName</i> <i>Sync</i> <i>Filter</i> <i>Combination</i> (<i>SNetExpr</i>)
<i>Combination</i>	⇒	<i>Serial</i> <i>Star</i> <i>Choice</i> <i>Split</i>
<i>Serial</i>	⇒	<i>SNetExpr</i> <i>SerialCombinator</i> <i>SNetExpr</i>
<i>Star</i>	⇒	<i>SNetExpr</i> <i>StarCombinator</i> <i>Terminator</i>
<i>Terminator</i>	⇒	(<i>Pattern</i> [, <i>Pattern</i>]*)
<i>Choice</i>	⇒	<i>SNetExpr</i> <i>ChoiceCombinator</i> <i>SNetExpr</i>
<i>Split</i>	⇒	<i>SNetExpr</i> <i>SplitCombinator</i> <i>Range</i>
<i>Range</i>	⇒	(<i>Tag</i> [: <i>Tag</i>])
<i>SerialCombinator</i>	⇒	. .
<i>StarCombinator</i>	⇒	* **
<i>ChoiceCombinator</i>	⇒	
<i>SplitCombinator</i>	⇒	! !!

Figure 4.7: Grammar of S-NET network combinators

The binary serial combinator “. .” connects the output of the left operand to the input of the right operand. The input of the left operand and the output of the right one become those of the resulting network. The serial combinator establishes computational pipelines. Graphically, it is represented by an arrow connecting the two argument S-Nets.

The binary choice combinator “||” or “|” combines its operands in parallel. If an incoming record matches the type signature of one of the operand S-Nets, it is sent to that SNet. Should an incoming record match the type signatures of both operand S-Nets, it is non-deterministically sent to one of them. An implementation is free to choose an appropriate scheduling technique in this case. For example, it may send the record to the less loaded operand SNet for proper

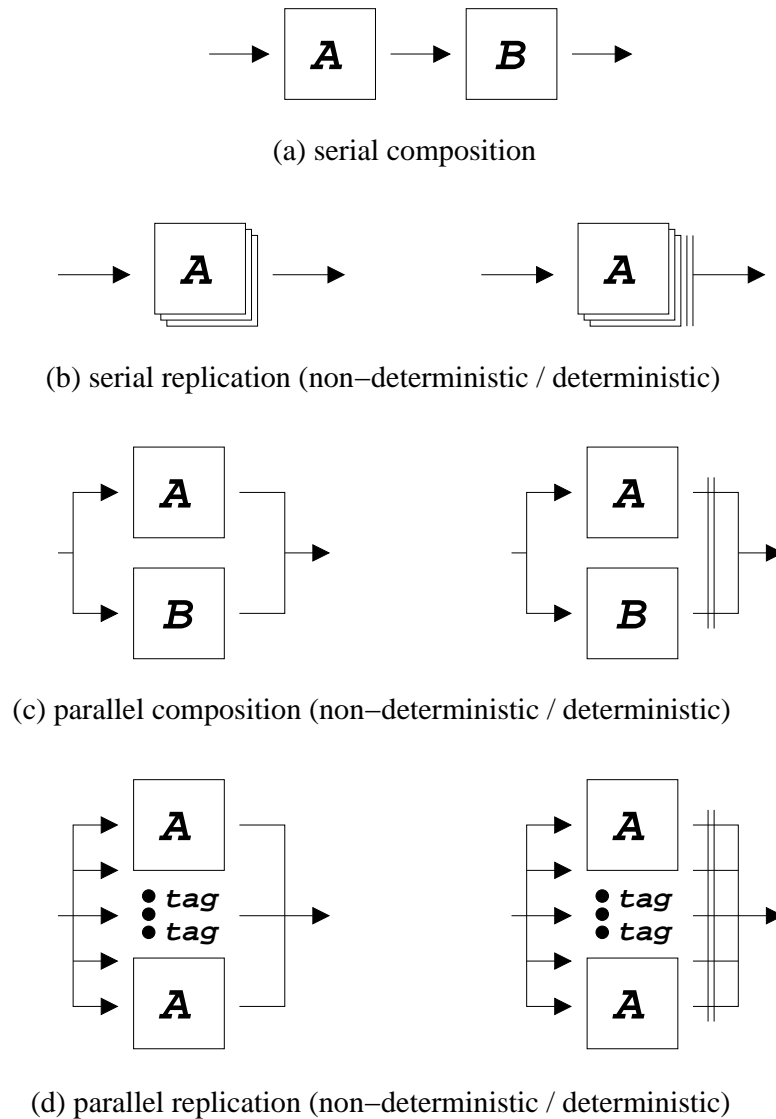


Figure 4.8: Graphical representation of S-NET network combinators

workload balancing. The graphical representation of the choice combinator employs a split and a merged arrow to visualise the choice operator.

The output streams of the operand networks are merged into a single stream, which becomes the output stream of the combined network. Here, the two variants of the choice combinator, “|” and “|” behave differently. Whereas the “|” variant merges the two output streams non-deterministically, the “|”

variant preserves the sequence of records. More precisely, any output generated by one of the operand SNETs in response to an incoming record on the joint input stream is sent to the joint output stream before any records produced by any of the operand SNETs in response to a subsequent input record. In the graphical representation an additional vertical double line symbolises the additional synchronisation required to achieve this deterministic behaviour.

Providing these two variants of the choice 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.

The star combinator “*” or “**” replicates the operand SNET (the left operand) infinitely many times and connects the replicas using serial combination. Actual replication of the operand network is demand-driven or lazy. The right operand of the star combinator defines a set of patterns. As soon as a record matches one of these patterns, it is released from the network and sent to the 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 SNET. In this sense the star combinator resembles a while-loop rather than a repeat-until-loop. This coincidence with the meaning of star in regular expressions particularly motivates our choice of the star symbol.

Similar to the parallel choice combinator we provide two versions of the star combinator: “*” and “**”. The latter sends records to the output stream as soon as they match the terminator pattern. However, this rather simple and efficient behaviour does not preserve the sequence of records: an earlier record may simply travel through more incarnations of the operand SNET than a subsequent record with the result of being sent to the output stream first. Whenever the sequence of records matters, the “*” version of the star combinator preserves it at the expense of additional runtime overhead. In the graphical representation an additional vertical double line symbolises the additional synchronisation required to achieve this deterministic behaviour.

The binary index split combinator “!|” or “!” takes an SNET as its left operand and either a single tag or a pair of tags as its right operand. Like the star combinator, the index split combinator replicates the operand SNET, but connects the replicas using the choice operator. 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 named tag, i.e., all records that have the same tag value will be processed by the same replica of the operand SNET. 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).

The graphical representation of the index split combinator, as shown in

Fig. 4.8, symbolically replicates the operand SNetS with three vertical dots representing the dynamic number of replicas. The name(s) of the tag(s) that control replication are annotated at the vertical dots.

In analogy to the parallel choice combinator, the output streams of the replicas are merged into the single output stream of the network either non-deterministically (“!!”) or under preservation of causality with respect to the sequence of records on the input stream (“!”). As in the graphical representation of the choice combinator an additional vertical double line visualises the necessary synchronisation in order to achieve this deterministic behaviour.

The definition of an expression language based on unary and binary infix combinators immediately raises the questions of associativity and priorities of combinators. All 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, i.e., 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:

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

Chapter 5

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. Nevertheless, we consider the following material useful to read and, therefore, leave it as is for the time being.

5.1 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 5.5. 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.

5.2 Serial Combinator

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 5.1 (serial combinator) *The serial combination $S = \mathbf{a}..A$ 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 $S = a..A$, 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

$$R = \{v^{[i]} \cup (V^{[j]} \setminus w^{[i]}) \rightarrow (w^{[i]} \setminus V^{[j]}) \cup W^{[j]} \mid (V^{[j]} \setminus w^{[i]}) \cap v^{[i]} = \emptyset\}, \quad (5.1)$$

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 5.1 depicts the set-theoretical relations between inputs and outputs as defined by Eq 5.1.

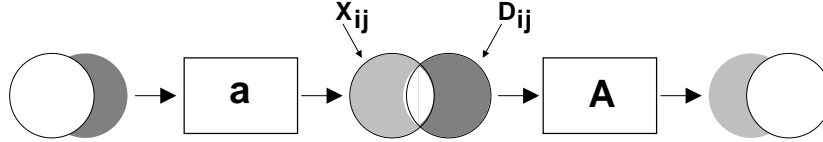


Figure 5.1: Flow inheritance through the \dots combinator

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. 5.1 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 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 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 5.1 is modified thus:

$$R = \{v^{[i]} \cup (V^{[j]} \setminus w^{[i]}) \rightarrow (w^{[i]} \setminus V^{[j]}) \cup W^{[j]} \mid (V^{[j]} \setminus w^{[i]}) \cap (v^{[i]} \cup B) = \emptyset\}, \quad (5.2)$$

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 \mathbf{A} .

5.3 Closure

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 5.2 (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

$$\check{B}_q^\circ = c_q^{[i_\infty]}, \quad (5.3)$$

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 \mathbf{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 5.2. 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

¹this corresponds to “laziness” in functional semantics

Definition 5.3 (hard closure) *A hard closure of a box B is a network B^* whose semantic function \tilde{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 5.3 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 $W = \bigcup_{i=1}^n \bigcup_{j=1}^{m_i} w_j^{[i]}$. 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 5.1 *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 5.4 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 5.1 to the following

Proposition 5.2 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 5.1 with Definition 5.4. 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 5.3 The operation $..$ as defined by Definition 5.4 is associative.

To prove this, we must prove that for all $\phi_{1-3} : Q \times \mathbb{N} \rightarrow Q$, $(\phi_1.. \phi_2).. \phi_3 = \phi_1.. (\phi_2.. \phi_3)$. 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 $(m, n_3^L) = \sigma_v^n(\phi_1.. \phi_2, \phi_3)$, 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, $(n_1^R, k) = \sigma_v^n(\phi_1, \phi_2)$; $(n_2^R, n_3^R) = \sigma_{\phi_1(v, n_1)}^k(\phi_1, \phi_2.. \phi_3)$. 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

$$B^* \equiv \underbrace{B..B, \dots, ..B}_{L \text{ times}} ..B^\rightarrow \quad (5.4)$$

for sufficiently large L . Here \equiv denotes the equality of type signatures. Indeed, once a certain power (with respect to the $..$ 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 5.2 and 5.3 establish associativity of the serial combinator viewed as a type constructor. Let us change the evaluation order in Eq 5.4 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.

5.4 Choice operator

Consider boxes \mathbf{A} : $v_a^{[i]} \rightarrow \tau_a^{[i]}$ and \mathbf{B} : $v_b^{[i]} \rightarrow \tau_b^{[i]}$. The choice combinator $\mathbf{A|B}$ 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 \mathbf{A} and \mathbf{B} are merged into one stream arbitrarily. Here is a formal definition:

Definition 5.5 (choice) *The choice combination $C = \mathbf{A|B}$ of networks \mathbf{A} and \mathbf{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

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

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 \mathbf{A} and \mathbf{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 \mathbf{A} and \mathbf{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. The rule set of the choice combination, Σ_C is as follows:

$$\Sigma_C = \left\{ v \rightarrow \tau_* \mid (\exists \tau)(v \rightarrow \tau) \in \Sigma_+ \wedge \tau_* = \bigsqcup_{(v \rightarrow \tau) \in \Sigma_+} \tau \right\},$$

where

$$\Sigma_+ = \{v \rightarrow \tau \mid \exists(v_A \rightarrow \tau_A \in \Sigma_A, v_B \rightarrow \tau_B \in \Sigma_B)v = v_A \sqcup v_B \wedge \tau = \tau_A \sqcup \tau_B\} .$$

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.

5.5 Type signature completion

Finally, recall that the input types of boxes are required to be complete (see Section 3.3). 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.

The type signature of a box with the default action taken into account becomes complete in the sense of Definition 3.5. 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 3.5. 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

```

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 5.2: Algorithm to make type signature complete

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.

5.6 Index splitter

This operator is written in the form $B!k$, where B is a network and k is a field name, called the *index* hereinafter. Informally, it creates an array of replicas of network B and assigns each replica a unique value from the field type of k . The input stream must contain only records that have field k (among others). Each input record is directed to the replica of B assigned its value of k . The output of all replicas is merged into a single stream. Note that the array is conceptually infinite since SNet has no access to field types, and that the specific replica is selected on the basis of value identity at run time, the more so that in any practical input stream, the variety of k values would be a small set compared to the full type. The assumption of the infinite array is safe since SNet boxes and networks cannot produce output without input, hence the replicas that receive no input are semantically non-existent.

Next we give formal definitions. First the type signature. Let $v_i \rightarrow \tau_i$ be the signature of B . The signature of $B!k$ is then $(v_i \cup \{k\}) \rightarrow \delta(k, v_i, \tau_i)$, where

$$\delta(k, v, \tau) = \begin{cases} (k \cup \tau) & \text{if } k \notin v, \\ \tau & \text{otherwise} \end{cases} .$$

Assume the index takes values from a set V , and, for simplicity, that the set is finite. Now construct set $T = \{t_i\}$ of unique binding tags of the same size. Let $f : V \rightarrow T$ be a bijection between the sets: $T = \{t_i = f i \mid i \in V\}$ Construct

a third set, a set of replicas R of network B , as follows:

$$R = \{B_i = \theta(B, t_i) \mid i \in V\},$$

where $\theta(B, t)$ is the same network as B , except each of the input variants v is augmented with the binding tag t . Now the semantics of $B!k$ is given by the following

Definition 5.6 *The network $B!k$ is semantically equivalent to the following*

$$\text{isplit}..(B_{i_1}|B_{i_2}|\dots|B_{|V|}),$$

where

$$\text{isplit} : \{k\} \rightarrow t_{i_1}\{k\}|t_{i_2}\{k\}|\dots|t_{i_{|V|}}\{k\}$$

is a box that expects single-field records $\{k\}$ and produces single-field records $\{t, k\}$, where $t = f k$. Here $i_1 \dots i_{|V|}$ is some enumeration of set V .

Chapter 6

Interfaces

6.1 Atomic Box Implementation

As pointed out in Section 4.1, 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.

```

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 6.1: Example of a C implementation of an atomic box

Fig. 6.1 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. 6.1; the same code may alternatively reside in a different file.

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.

6.2 Input/Output and the Outside World

6.3 Box Language Binding

Chapter 7

Examples

7.1 Defining factorial in S-Net

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.

```
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 term-

nation 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 ((x) -> (xx));
    box mult ((x,r) -> (rr));
  }
  connect (leq..if..([<T>-><stop>]
    || [<F>,x,r]->{x,r};{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 relatively 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 $\langle T \rangle$ or a tag $\langle F \rangle$. 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

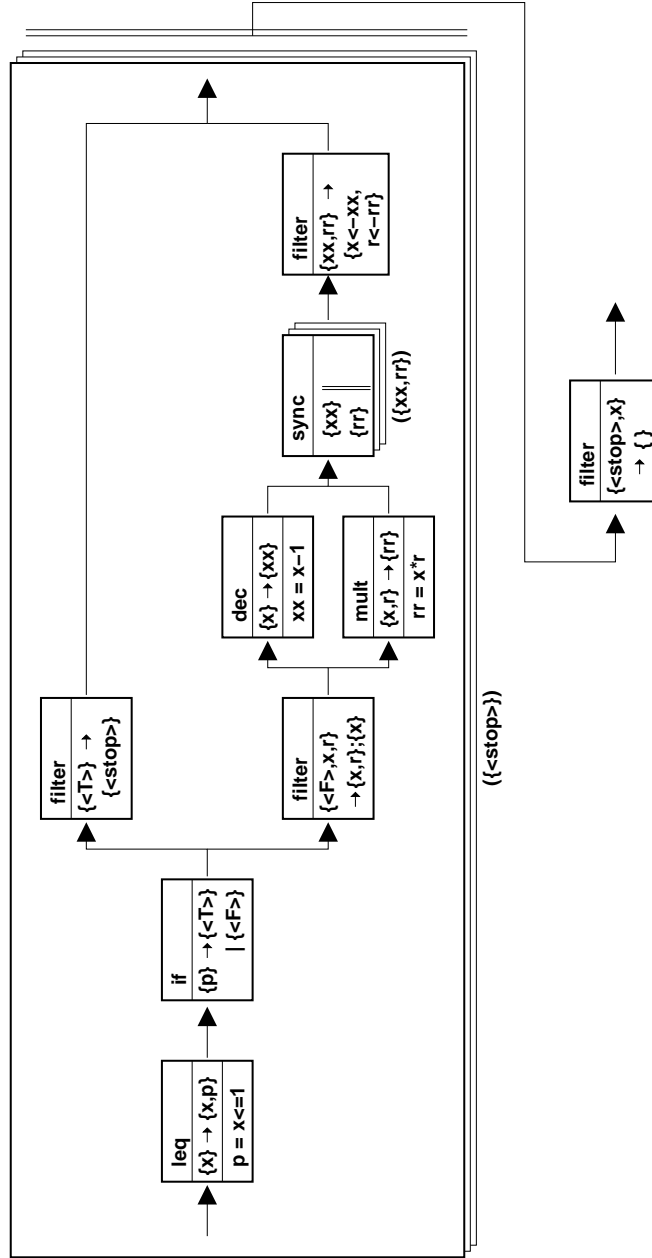


Figure 7.3: Graphical representation of network facit

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.

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 decrement and the multiplication, we can arrange the two boxes in parallel. Due to the best match rule, the choice turns out to be deterministic again. Any record `{x,r}` is directed towards the `mult` box while `{x}` records are directed to the `dec` box. The two arithmetic boxes produce records `{xx}` and `{rr}`, respectively.

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 4.2 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 4.1, 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.

Chapter 8

Conclusions and Future Work

Appendix A

Complete Syntax of S-Net

<i>SNet</i>	⇒	<i>[Definition]*</i>
<i>Definition</i>	⇒	<i>BoxDef NetDef TypeDef</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>Connect</i>
<i>NetSignature</i>	⇒	<i>TypeSignature</i> <i>Type</i>
<i>TypeSignature</i>	⇒	<i>TypeMapping</i> [, <i>TypeMapping</i>]*
<i>TypeMapping</i>	⇒	<i>Type</i> -> <i>Type</i>
<i>Type</i>	⇒	<i>TypeName</i> [<i>Type</i>]
		<i>RecordType</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>TagName</i> >
<i>BindingTag</i>	⇒	< # <i>TagName</i> >
<i>TypeDef</i>	⇒	type <i>TypeName</i> = <i>Type</i> ;
<i>NetBody</i>	⇒	{ <i>[Definition]*</i> }

<i>Connect</i>	⇒	connect <i>SNetExpr</i> ;
<i>SNetExpr</i>	⇒	<i>BoxName</i>
		<i>NetName</i>
		<i>Sync</i>
		<i>Filter</i>
		<i>Combination</i>
		(<i>SNetExpr</i>)
<i>Filter</i>	⇒	[<i>Pattern</i> -> [<i>FilterOut</i> [; <i>FilterOut</i>]*]]
		[]
<i>FilterOut</i>	⇒	{ [<i>FilterOutField</i> [, <i>FilterOutField</i>]*] }
<i>FilterOutField</i>	⇒	<i>Field</i> [<- <i>FieldInit</i>]
		<i>Tag</i> [<- <i>TagInit</i>]
<i>FieldInit</i>	⇒	<i>Field</i>
<i>TagInit</i>	⇒	<i>Tag</i>
		<i>IntegerConst</i>
<i>Sync</i>	⇒	[<i>Pattern</i> [, <i>Pattern</i>]+]
<i>Pattern</i>	⇒	<i>RecordType</i>
<i>Combination</i>	⇒	<i>Serial</i> <i>Star</i> <i>Choice</i> <i>Split</i>
<i>Serial</i>	⇒	<i>SNetExpr</i> <i>SerialCombinator</i> <i>SNetExpr</i>
<i>Star</i>	⇒	<i>SNetExpr</i> <i>StarCombinator</i> <i>Terminator</i>
<i>Terminator</i>	⇒	(<i>Pattern</i> [, <i>Pattern</i>]*)
<i>Choice</i>	⇒	<i>SNetExpr</i> <i>ChoiceCombinator</i> <i>SNetExpr</i>
<i>Split</i>	⇒	<i>SNetExpr</i> <i>SplitCombinator</i> <i>Range</i>
<i>Range</i>	⇒	(<i>Tag</i> [: <i>Tag</i>])
<i>SerialCombinator</i>	⇒	..
<i>StarCombinator</i>	⇒	* **
<i>ChoiceCombinator</i>	⇒	
<i>SplitCombinator</i>	⇒	! !!

Bibliography

- [1] Broy, M., Stefanescu, G.: The algebra of stream processing functions. *Theoretical Computer Science* (2001) 99–129
- [2] van Rossum, G.: The Python Language Reference Manual. Network Theory Ltd (2003)
- [3] 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
- [4] Ashcroft, E.A., Wadge, W.W.: Lucid, a nonprocedural language with iteration. *Communications of the ACM* **20** (1977) 519–526
- [5] Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE* **79** (1991) 1305–1320
- [6] Berry, G., Gonthier, G.: The estereel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming* **19** (1992) 87–152
- [7] Binder, J.: Safety-critical software for aerospace systems. *Aerospace America* (2004) 26–27
- [8] 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
- [9] 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
- [10] 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)

-
- [11] Stephens, R.: A survey of stream processing. *Acta Informatica* **34** (1997) 491–541
- [12] 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
- [13] 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
- [14] 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
- [15] Stefanescu, G.: *Network Algebra*. Springer-Verlag (2000)
- [16] Shafarenko, A.: Stream processing on the grid: an array stream transforming language. In: *SNPD*. (2003) 268–276
- [17] Cardelli, L., Wegner, P.: On understanding types, data abstraction, and polymorphism. *Computing Surveys* **4** (1985) 471–522
- [18] Mitchell, J.: Type inference with simple subtypes. *Journal of Functional Programming* **1** (1991) 245–285
- [19] Reynolds, J.C.: Using category theory to design implicit conversions and generic operators. In Jones, N.D., ed.: *Semantics-Directed Compiler Generation*, volume 94 of *Lecture Notes in Computer Science*, Springer-Verlag (1980) 211–258
- [20] Fuh, Y.C.C., Mishra, P.: Type inference with subtypes. *Theoretical Computer Science* **73** (1990) 155–175
- [21] Kaes, S.: Type inference in the presence of overloading, subtyping and recursive types. In: *LFP '92: Proceedings of the 1992 ACM conference on LISP and functional programming*, New York, NY, USA, ACM Press (1992) 193–204
- [22] Shafarenko, A.: Coercion as homomorphism: type inference in a system with subtyping and overloading. In: *PPDP '02: Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming*. (2002) 14–25
- [23] Lievant, D.: Discrete polymorphism. In: *Proc. 1990 ACM Conference on LISP and Functional Programming*, June 27-29, 1990, Nice, France. (1990.) 288–297

-
- [24] Rehof, J., Mogensen, T.: Tractable constraints in finite semilattices. *Science of Computer Programming* **35** (1999) 191–221
 - [25] 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
 - [26] Grelck, C.: Shared memory multiprocessor support for functional array processing in SAC. *Journal of Functional Programming* **15** (2005) 353–401