Asynchronous Stream Processing with S-Net

Clemens Grelck[1,2], Sven-Bodo Scholz[1], and Alex Shafarenko[1]

[1] University of Hertfordshire
Department of Computer Science
College Lane, Hatfield, AL10 9AB, United Kingdom
{c.grelck,s.scholz,a.shafarenko}@herts.ac.uk
[2] University of Amsterdam, Institute of Informatics
Science Park 107, 1098 XG Amsterdam, Netherlands
c.grelck@uva.nl

Abstract. We present the rationale and design of S-Net, a coordination language for asynchronous stream processing. The language achieves a near-complete separation between the application code, written in any conventional programming language, and the coordination/communication code written in S-Net. Our approach supports a component technology with flexible software reuse. No extension of the conventional language is required. The interface between S-Net and the application code is in terms of one additional library function.
The application code is componentised and presented to S-Net as a set of components, called boxes, each encapsulating a single tuple-to-tuple function. Apart from the boxes defined using an external compute language, S-Net features two built-in boxes: one for network housekeeping and one for data-flow style synchronisation. Streaming network composition under S-Net is based on four network combinators, which have both deterministic and nondeterministic versions. Flexible software reuse is comprehensive, with the box interfaces and even the network structure being subject to subtyping. We propose an inheritance mechanism, named flow inheritance, that is specifically geared towards stream processing. The paper summarises the essential language constructs and type concepts and gives a short application example.

1 Introduction

This paper will introduce a coordination language for asynchronous stream processing. The concept of coordination language arises wherever an application has to be presented as a set of concurrent communicating activities, each defined in application-specific terms as a meaningful program unit, while all together representing a concurrently executing, parallel (and potentially distributed) application. The application program units are presented in an appropriate fully-fledged programming language, such as C, Java, etc., while the aspects of communication, concurrency and synchronisation (referred to by the term *coordination*) are captured by a separate, coordination, language. The whole idea of coordination hinges on the principle that the integration between the coordination and

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

The approach developed in this paper is targeted at stream processing. This is a well-established area, which is very important in a time when distributed computing, multimedia and signal processing permeate the computing and telecommunication sectors. This paper focuses on asynchronous stream processing, which on the one hand, enables the philosophy of data-flow synchronisation developed in the 1980s to be taken on board (thanks to the coordination aspect, which assumes course granularity), whilst on the other hand, develop a whole host of analysis techniques thanks to the regular nature of stream communication (as opposed to general message-passing). The result is a very compact and powerful coordination language, called S-Net which reflects the modern notions of subtyping, encapsulation and inheritance, while completely separating all communication and concurrency concerns from the application code.

S-Net provides means to describe the orderly behaviour among components, named *boxes* and the streaming network used for communication between them. Boxes are Single Input Single Output (SISO) entities implemented externally using an appropriate *box language*. Functional languages are particularly suitable for this purpose as they naturally adhere to the restrictions imposed by the interface (i.e. no side-effects and no state sharing). Nevertheless, imperative box languages may be used as well, but require some discipline by the programmer.

Boxes communicate with each other and with the execution environment solely by means of data received and sent via their input and output streams, respectively. S-Net allows boxes to be composed into SISO networks. The input and output streams of a box or network are typed. Composition of boxes involves merging their streams and also splitting them depending on types. It is described using *network combinators*, that are inspired by Stefanescu's network algebra [1].

S-Net networks are asynchronous by definition: an entity's output is assumed to be buffered. When processing is done by several components whose results must be combined, generally a synchronisation facility is 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 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.

The remainder of this paper is organised as follows. We will commence with a brief overview of stream processing in Section 2. The type concepts inherent to S-Net are presented in Section 3. Sections 4 and 5 introduce the S-Net approach to box and network definition, respectively. The important issue of synchronisation in streaming networks is discussed in Section 6. We illustrate our approach by a small example in Section 7. Section 8 discusses some related work, and we conclude in Section 10.

2 Background: 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 [2] and the language Lucid [3]. Kahn introduced the model of infinite-capacity, deterministic process networks 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 sequences. A variable would represent such a sequence, acting as a stream of values of that variable in time. Ordinary operators in Lucid acted on variables point-wise, by effectively synchronising streams and applying the operation across pairs of corresponding stream elements. Additionally there were also some "temporal" operators, which were intended for altering the order of elements in a sequence.

Somewhat later, in the 1980s, a whole host of synchronous dataflow languages sprouted, notably the languages Lustre [4] and Esterel[5], 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 [6]. The authors of Lustre broadened their work towards what they termed synchronous Kahn's networks [7, 8], 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 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 [9]. The latest comprehensive survey of stream processing and the underlying theory for it can be found in [10]. There is also a growing activity in *database stream processing* [11], 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[12] remarked that streams could be used as software glue for complex parallel software systems, even operating systems. In his interpretation, streams were lazy lists, which were produced on demand for their consumers. The lists were seen as an interface between the deterministic parts of a parallel system, which were pure stream-processing functions[3], and the external interleavers/mergers that realise the inter-process communication and capture its nondeterministic behaviour.

This arrangement is sketched out in Fig. 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.

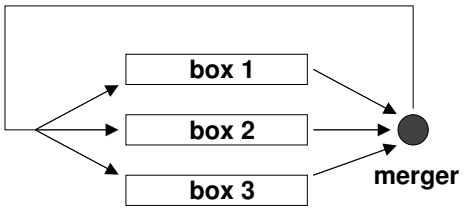

Fig. 1. The Turner scheme

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

[3] In fact, 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.

that order to be recovered from the merged stream, the provenance information can be preserved by, for example, tagging the ordered records by the same tag.

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

Besides these pragmatic considerations, we must mention here equally important theoretical advances in streaming networks. The key work in this area has been done by Stefanescu, who has developed several semantic models for streaming networks starting from flowcharts [13] 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 [14]). It is interesting to note that apparently the StreamIt team [9] as well as ourselves [15] were unaware of those and re-invented them for network construction.

3 The type system of S-Net

3.1 Record types

The type system of S-Net is based on non-recursive variant records with *record subtyping*. As defined in Fig. 2, 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. We distinguish two different kinds of record entries: *fields* and *tags*. A field is characterised by its *field name* (label); it is associated with an opaque value at runtime, i.e., fields can only be generated, inspected or manipulated by using an appropriate box language. A tag is represented by a name enclosed in angular brackets. At runtime tags are associated with a single integer value each. This value is visible to both box language code and S-Net. Furthermore, we distinguish between *simple tags* and *binding tags*, the latter being marked with the hash character ("#"). The

rationale of tags lies in controling the flow of records through a network. They should not be misused to hold box language data that by chance can be represented as integer values. Binding tags behave differently from fields and simple tags with respect to subtyping and provide explicit means to control subtyping where some restriction is useful. We explain this in detail in Section 3.2.

$$
\begin{array}{lll}
\textit{Type} & \Rightarrow & \textit{RecordType} \left[\texttt{|} \ \textit{RecordType} \right]^* \\
\textit{RecordType} & \Rightarrow & \texttt{\{} \left[\textit{RecordEntry} \left[\texttt{,} \ \textit{RecordEntry} \right]^* \right] \texttt{\}} \\
\textit{RecordEntry} & \Rightarrow & \textit{FieldName} \mid \textit{Tag} \\
\textit{Tag} & \Rightarrow & \texttt{<} \ \textit{TagName} \ \texttt{>} \mid \texttt{<} \ \texttt{\#} \ \textit{BindingTagName} \ \texttt{>} \\
\textit{TypeSignature} & \Rightarrow & \textit{TypeMapping} \left[\texttt{,} \ \textit{TypeMapping} \right]^* \\
\textit{TypeMapping} & \Rightarrow & \textit{Type} \ \texttt{->} \ \textit{Type}
\end{array}
$$

Fig. 2. Syntax definition of S-Net types and type definitions. The non-terminal symbols *FieldName*, *TagName* and *BindingTagName* uniformly refer to identifiers. We only distinguish them here for the purpose of illustration.

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

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

providing fields for the coordinates of a reference point (`x` and `y`) and edge lengths in both dimensions (`dx` and `dy`). 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 give anonymous variants a name. In S-Net this may be achieved using tags:

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

or binding 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, but coverage of this topic would exceed the space available. We refer the interested reader to [16] for a complete treatment of the subject.

3.2 Record subtyping

S-Net supports structural subtyping on record types. Subtyping essentially is based on the subset relationship between sets of record entries. Informally, a 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>, x, y, radius}
```

Likewise, we may add another type to represent triangles:

```
  {<rectangle>, x, y, dx, dy}
| {<circle>, x, y, radius}
| {<triangle>, x, y, dx1, dy1, dx2, dy2};
```

which again is a supertype of

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

and

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

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 our second example, the supertype is again more general than its subtype as it encompasses all three different geometric bodies. Subtype {`<circle>,x,y,radius,colour`} is more specific than its supertypes because it rules out triangles and rectangles from the set of geometric bodies covered. Let us give a formal definition of record subtyping.

Definition 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*

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

Subtype relationship requires both subtype and supertype to have exactly the same binding tags. This explains our motivation to distinguish between simple and binding tags: Binding tags provide a means to excercise explicit control over record subtyping. For instance, the type {`x,y`} defining the position of a geometric body is a supertype of all previous types. 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 position. Changing our type to

```
   {<#rectangle>, x, y, dx, dy}
 | {<#circle>, x, y, radius}
 | {<#triangle>, x, y, dx1, dy1, dx2, dy2};
```

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

Type signatures describe the stream-to-stream transformation performed by a box or a network. They are similar to function types. As defined in Fig. 2, a 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 or network accepts for processing; the output type characterises the records that the box or network may produce as as response. For example, the type signature

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

describes a box or network that accepts records that either contain fields `a` and `b` or fields `c` and `d`. In response, the box or network produces records that either contain tag `x` or tag `y`.

An input type that consists of multiple variants like in the previous example 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 above is equivalent to the type signature

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

Therefore, we assume (single variant) record types as input types from here on, we call these type signatures *normalised*. A multi-variant output type means that a box or network may produce any of the records specified in response to receiving an input record that fits the associated input type. However, it is important to note that S-Net boxes may produce as many output records in response to a single input record as they like, including none at all. Multiple output records may follow the same output variant or be all different from each other. In analogy to types, S-Net supports abstractions on type signatures; see [16] for details.

3.4 Type coercion

As explained earlier, an S-Net box or network accepts any record whose type is a subtype of the type signature's input type. In general, this requires an up-coercion to the most appropriate supertype. As an example, let us assume

```
  {<#rectangle>, x, y, dx, dy}
| {<#circle>, x, y, radius}
| {<#triangle>, x, y, dx1, dy1, dx2, dy2};
```

as input type of some network. 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 our input type by an additional variant for coloured circles as in

```
  {<#rectangle>, x, y, dx, dy}
| {<#circle>, x, y, radius}
| {<#circle>, x, y, radius, colour}
| {<#triangle>, x, y, dx1, dy1, dx2, dy2};
```

we would choose that more specific mapping for whenever we deal with coloured circles.

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>, x, y, dx, dy}
```

would match both variants equally well. Only some targets for coercion can cause such ambiguities; the following definition introduces a uniqueness condition for type coercions:

Definition 2 (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, $BT(x)$ 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

Up-coercion of records upon entry to a certain box or network creates a subtle problem in the stream-processing context of S-Net. In an object-oriented setting the control flow eventually returns from a method invocation that causes an up-coercion. While during the execution of the specific method the object is treated as being one of the respective superclass, it always retains its former state in the calling context. In a stream-processing network, however, records enter a box or network through its input stream and leave it through its output stream, which are both connected to different parts of the whole network. If an up-coercion results in a loss of record entries, this loss is not temporary but permanent.

Unfortunately, the permanent loss of record entries is hardly useful or desirable. For example, we may have a box that manipulates the position of a geometric body which could be a rectangle {`x, y, dx, dy`}, a circle {`x, y, radius`} or a ray {`x, y, phi`}. The associated type signature of such a box could be just {`x, y`}`->`{`x, y`}. Using simple tags instead of binding tags for variant identification, this box would accept circles, rectangles and rays focussing on their common data (i.e. the position) and ignoring their specific record entries.

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

Definition 3 (flow inheritance).
Let $v^{[i]} \to \tau^{[i]}$, $i \in [1, \ldots, n]$, be the type signature of a box X. Furthermore, let each output type $\tau^{[i]}$ have m_i variants $\tau^{[i]} = \{w_1^{[i]}, \ldots, w_{m_i}^{[i]}\}$. Then for any $k \leq n$ and any field or non-binding tag $\phi \notin v^{[k]}$ such that

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

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

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

and

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

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

Informally, an input variant can be extended with a new field or simple tag (but not binding tag) ϕ, if it does not clash with any other variant. The output type associated with this input variant is extended with the field named ϕ in each of its variants unless it is present there already. Any number of flow inheritance extensions can be applied to a box, resulting in several fields being added. Value-wise, the extension is in terms of copying the value of the input record field ϕ over to the output record field with the same name. If the output already contains an identically named field, then that field's value supersedes the inherited one.

4 Boxes

4.1 User-defined boxes

From the perspective of S-Net boxes are the atomic building blocks of streaming networks. The boxes themselves are implemented using a box language different from S-Net. A single S-Net network may well combine boxes implemented

using different box languages. Interoperability between different languages requires a careful interface design whose proper description goes well beyond the scope and size of this paper. Therefore, we restrict ourselves to sketch out the principles.

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

BoxSignature ⇒ *BoxType* -> *BoxType* [| *BoxType*]*

BoxType ⇒ ([*RecordEntry* [, *RecordEntry*]*])

Fig. 3. Grammar of S-Net box declarations

Fig. 3 shows the S-Net syntax for declaring user-defined boxes. Boxes are declared in S-Net using the key word `box` followed by a box name as unique identifier and a box signature enclosed in round brackets. The box signature very much resembles a type signature with two exceptions: we use round brackets instead of curly brackets and we have exactly one type mapping that has a single-variant input type. For example, the following line of code

```
box foo ((a,b,<t>) -> (a,b) | (<t>));
```

declares a box named `foo`, which accepts records containing (at least) fields `a` and `b` plus a tag `t` and in response produces records that either contain fields `a` and `b` or a tag `t`. It is entirely up to the box implementation to decide how many output records it actually emits and of which of the output variants they are. This may well depend on the values of the input record entries and, hence, can only be determined at runtime.

```
snet_handle_t *foo( snet_handle_t *handle, int *a, mytype_t *b, int t)
{
  /* some computation on a, b and t */

  snetout( handle, 1, a, b);

  /* some computation */

  snetout( handle, 2, t);

  return( handle);
}
```

Fig. 4. Example box function implementation in C

As mentioned earlier, box signatures differ from regular type signatures in the restriction to a single type mapping and the use of round brackets instead

of curly brackets. The latter emphasises the fact that in box signatures sequence does matter, whereas type signatures are true sets of mappings between true sets of record entries. Sequence is essential to support a mapping to function parameters of some box language implementation rather than using inefficient means such as string matching of field and tag names. For example, we may want to associate the above box declaration `foo` with a C language implementation in the form of the C function `foo` shown in Fig. 4.

The entries of the input record type are effectively mapped to the function parameters in their order of appearance in the box signature. We implement record fields as opaque pointers to some data structure and tags as integer parameters. In addition to the box-specific parameters the box function implementation always receives an opaque S-Net handle, which provides access to S-Net internal data. Since boxes in S-Net generally produce a variable number of output records in response to a single input record, we cannot exploit the function's return value to determine the output record. Instead, we provide a special function `snetout` that allows us to produce and send output records dynamically during the execution of the box function. The first argument to `snetout` again is the internal handle that establishes the necessary link to the execution environment. The second argument to `snetout` is a number that determines the output type variant used. So, the first call to `snetout` in the above example refers to the first output type variant. Consequently, the following arguments are two pointers. The second call to `snetout` refers to the second output type variant and, hence, a single integer value follows. Eventually, the box function implementation must return the internal handle to signal completion to the S-Net context.

This is just a raw sketch of box language interfacing. Concrete interface implementations may look differently to accommodate characteristics of certain box languages, and even the same box language may actually feature several interface implementations with varying properties. For a detailed description of available box language interface implementations see [16].

4.2 The filter box

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

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

Syntactically, a filter box is enclosed in square brackets and consists of a type pattern to the left of an arrow symbol and a semicolon-separated sequence of filter actions to the right of the arrow symbol, for example:

```
[{a,b,<t>} -> {a} ; {c=b,<u=42>} ; {b,<t=t+1>}]
```

This filter box accepts records that contain fields `a` and `b` as well as tag `t`. In general, the type-like notation to the left of the arrow symbol acts as a pattern on records; any incoming record's type must be a subtype of the pattern type.

As a response to each incoming record, the filter box produces three records on its output stream. The specifications of these three records are separated by semicolons to the right of the arrow symbol. Outgoing records are defined in terms of the identifiers used in the pattern. In the example, the first record produced only contains the field `a` adopted from the incoming record (plus all flow-inherited record entries). The second record produced contains again the field `b` from the input record, but it is renamed to `c`. In addition there is a tag `u` set to the integer value 42. The last of the three records produced contains the field `b` and the tag `t` from the input record, where the value associated with tag `t` is incremented by one.

S-Net supports a simple expression language on tag values that essentially consists of arithmetic, relational and logical operators as well as a conditional expression. However, it lacks any means of abstraction of values or functions. It is intended for simple computations like the one in the example. As soon as the numerical relationship between tag values exceeds a certain complexity, it is recommended to use a user-defined box and a fully-fledged box language instead. For full coverage of filter box syntax see [16].

5 Networks

5.1 Network definitions

User-defined and built-in boxes form the atomic building blocks for complex stream processing networks; their hierarchical definition is at the core of S-Net. As a simple example of a network definition take the following:

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

Following the key word `net` we have the network name, in this case `example` and an optional block of local definitions enclosed in curly brackets. This block may contain box declarations, as in the above example, but, likewise, further network definitions or the type and type signature definitions, that we have briefly mentioned in Section 3. Hierarchical network definitions incur nested scopes, but in the absence of relatively free variables the scoping rules are rather straightforward. Fig. 5 gives the corresponding syntax rules.

A distinctive feature of S-Net is the fact that complex network topologies are not defined by some form of wire list, but instead by an expression language. The topology is static in the main: the only part of it that is variable is based on replicating a subnetwork in a serial or parallel connection a statically unknown number of times. This means that any type relationships that exist between

NetDef ⇒ **net** *NetName* ⌈ (*TypeSignature*) ⌋ *NetBody*

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

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

Combination ⇒ *Serial* | *Parallel* | *Star* | *IndexSplit*
| *ParallelDet* | *StarDet* | *IndexSplitDet*

Serial ⇒ *TopoExpr* `..` *TopoExpr*

Parallel ⇒ *TopoExpr* `|` *TopoExpr*

Star ⇒ *TopoExpr* `*` *Pattern*

IndexSplit ⇒ *TopoExpr* `!` *Tag*

ParallelDet ⇒ *TopoExpr* `||` *TopoExpr*

StarDet ⇒ *TopoExpr* `**` *Pattern*

IndexSplitDet ⇒ *TopoExpr* `!!` *Tag*

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

Fig. 5. Grammar of S-Net network definitions

component inputs and outputs cannot be invalidated by topology alterations: all replica networks created dynamically are identical to the parent and their number does not affect any type assertions provable in S-Net.

Each network definition contains such a topology expression following the key word `connect`. Atomic expressions are made up of box and network names defined in the current scope as well as of built-in filter boxes and synchrocells. Complex expressions are inductively defined using a set of network combinators that represent the four essential construction principles in S-Net: serial and parallel composition as well as serial and parallel replication.

5.2 Serial composition

The binary serial combinator "`..`" connects the output stream of the left operand to the input stream of the right operand. The input stream of the left operand and the output stream of the right operand become those of the combined net-

work. The serial combinator establishes computational pipelines, as illustrated in Fig. 6.

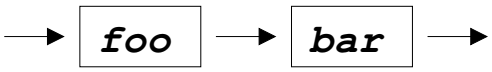

Fig. 6. Illustration of serial composition of networks: `foo..bar`

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

In contrast to box declarations, type signatures for network defintions are generally inferred by the compiler rather than annotated by the programmer. For example the inferred type signature of the network `example` is

```
{a,b} -> {d,e}
```

However, a type signature may actually be annotated, enclosed in round brackets following the network name, very much similar to the syntax of box declarations. Using this feature proves useful for various different purposes, e.g. documentation of network properties in the source code or specialisation of the inferred type signature.

5.3 Parallel composition

The binary parallel combinator "`|`" combines its operand networks or boxes in parallel. Any incoming record is sent to exactly one operand depending on its type and the type signatures of the operand networks or boxes. Fig. 7 illustrates the parallel composition of two networks `foo` and `bar`, i.e. `foo|bar`.

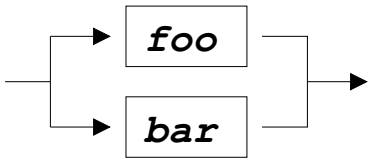

Fig. 7. Illustration of parallel composition of networks: `foo|bar`

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

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

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

For cases in which this efficient but non-deterministic behaviour is undesired, S-Net offers a deterministic variant of parallel composition: Using "`||`" rather than "`|`" as combinator, any output generated by one of the operand networks 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 networks in response to a subsequent input record.

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.

5.4 Serial replication

The serial replication combinator "`*`" replicates the operand network (the left operand) infinitely many times and connects the replicas by serial composition. The right operand of the combinator defines a set of type patterns. As soon as a record matches one of them, i.e., the record's type is subtype of the type pattern, the record is released and sent to the global output stream. In fact, an incoming record that matches one of the termination patterns right away is immediately

passed to the output stream without being processed by the operand network. This coincidence with the meaning of star in regular expressions particularly motivates our choice of the star symbol, and we sometimes refer to the serial replication combinator as the *star combinator*. Fig. 8 illustrates the operational behaviour of the star combinator for a network `foo*{<stop>}`: Records travel through serially combined replicas of `foo` until they contain a tag `<stop>`. Actual replication of the operand network is demand-driven. Hence, networks in S-Net are not static, but generally evolve dynamically.

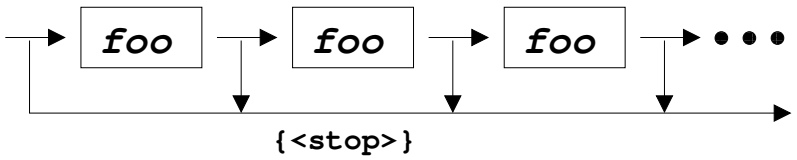

Fig. 8. Illustration of serial replication of networks: `foo*{<stop>}`

Similar to the parallel composition combinator we actually provide two versions of the star combinator: "`*`" and "`**`". The former sends records to the output stream as soon as they match the termination 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 network 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.

5.5 Indexed parallel replication

Last but not least, the parallel replication combinator "`!`" takes a network or box as its left operand and a tag as its right operand. Like the star combinator, it replicates the operand, but connects the replicas using parallel rather than serial composition. The number of replicas is conceptually infinite. Each replica is identified by an integer index. Any incoming record goes to the replica identified by the value associated with the given tag, i.e., all records that have the same tag value will be processed by the same replica of the operand network. Since parallel replication actually splits a stream of records depending on a certain tag, we also refer to "`!`" as the *index split combinator*. Fig. 9 illustrates the operational behaviour of the index split combinator for a network `foo!<tag>`. In analogy to serial replication, the instantiation of replicas of the operand network is demand-driven.

In analogy to the parallel composition, 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 ("`!!`").

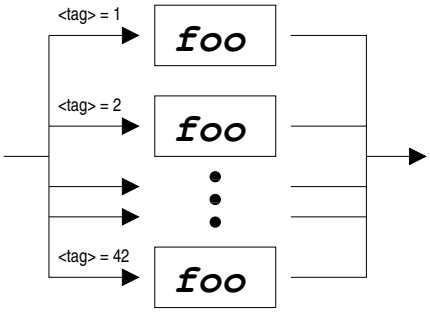

Fig. 9. Illustration of indexed parallel replication of networks: `foo!<tag>`

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

5.6 Combinator associativities and priorities

The definition of an expression language based on unary and binary infix combinators immediately raises the question 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:

"`||`" $\prec$ "`|`" $\prec$ "`..`" $\prec$ "`*`", "`**`", "`!`", "`!!`"

6 Synchronisation

What does synchronisation mean in the streaming network context of S-Net? Network combinators inspect records for routing purposes, but never manipulate individual records. This is the privilege of boxes and filters. They both may split a record into several ones, but because they process one record after the other in a stateless manner, we have not yet seen any way to join records. Joining records is the essence of synchronisation in the context of S-Net, and we have a special construct for this purposes: the *synchrocell*.

The synchrocell is the only "stateful" box in S-Net. Embedded within `[|` and `|]` parentheses, we find an at least 2-element list of patterns, for example:

```
[|{a, b}, {c, d}|]
```

The concept of a pattern, syntactically resembling a record type, is already familiar from the introduction of filter boxes in Section 4. A guarded pattern is associated with a guard expression defined using our simple expression language introduced for filter boxes. The principle idea behind the synchrocell is that it keeps incoming records which match one of the patterns until all patterns have been matched. Only then the records are merged into a single one that is released to the output stream. Matching here means that the type of the record is a subtype of the pattern. The pattern also acts as an input type specification of the synchrocell: a synchrocell only accepts records that match at least one of the patterns.

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

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

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

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

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

Synchrocells require a special treatment with respect to flow inheritance: At first glance, one may say that if a synchrocell stores a matching input record, it produces no output in response to this record. Hence, excess record fields, which would bypass the synchrocell otherwise, should be discarded. Any record output after successful synchronisation should be extended by the excess fields of the last incoming record because the synchrocell produces this output as a response to the input of this record. Last but not least, if a record is passed through the synchrocell in the case of overflow, there is output in response to input

and, therefore, the excess fields bypass the synchrocell as usual. However, this behaviour leads to very irregular and difficult to control behaviour of synchrocells where the sequence of arrival of records to be matched is non-deterministic (and so for good reason). In this case, flow inheritance would keep excess fields in a non-deterministic way as well, and that makes orderly processing of synchronised records in the subsequent network extremely difficult. Hence, it is not the record that triggers synchronisation which keeps its excess fields, but the record that matched the (lexically) first pattern in the definition of the synchrocell. Excess fields of all records that match other than the lexically first pattern are discarded immediately.

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

7 Example: stream of factorial numbers

The purpose of our example is to illustrate similarities and differences between concepts found in S-Net and mainstream programming languages. We employ a fairly simple and very well-known example, computing factorial numbers, and show how the text book C implementation shown in Fig. 10 can be carried over to S-Net.

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

Fig. 10. Computing a single factorial number in C

The C function `factorial` in Fig. 10 only computes a single factorial number given a suitable argument. We leave it to the imagination of the reader how this function could be mapped to an entire stream of arguments in order to produce a stream of pairs of argument and the argument's factorial number. Being geared towards stream processing the S-Net network `factorial` shown in Fig. 11 does exactly this, as is reflected by its type signature `{n}->{n,fac}`. Although type

signatures in S-Net are typically inferred by the compiler, we have typed all networks in Fig. 11 for the purpose of illustration.

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

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

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

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

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

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

Fig. 11. Computing a stream of factorial numbers in S-Net

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

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

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

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

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

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

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

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

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

As pointed out in the beginning, the sole purpose of our example is to illustrate the use of the various S-Net language features and their relationship to constructs known from conventional programming languages. It is definitely not intended as an exersise in finding the most suitable description of how to compute factorial numbers. This task would hardly benefit from the degree of concurrency introduced by the S-Net in Fig. 11. Using boxes only for the most rudimentary computations and expressing anything else in S-Net is by no means representative for real world S-Net applications. Here, we expect boxes to represent substantial amounts of computational work and the S-Net layer to control only coarse-grained coordination aspects. However, such a real world example would not be very useful for the purpose of illustrating S-Net features because it would require a fair amount of knowledge about the box language components as well as familiarity with the chosen application domain. We refer the reader interested into the interplay between box language and S-Net to [17] for a more elaborate case study.

8 Related work

The coordination aspect of the proposed stream processing language is related to a large body of work in so-called data-driven coordination, see [18]. Unlike most data-driven coordination languages, we have a *complete* separation of coordination and computation. This is achieved by using opaque SISO stream transformers implemented in a separate box language chosen by the programmer.

The earliest related proposal, to our knowledge, is the coordination language HOPLa from the Utrecht University's Ariadne project [19]. It is a Linda-like coordination language, which uses record subtyping (which they call "flexible records") in a manner similar to S-Net, but does not handle variants as we do, and has no concept of flow inheritance. Also, HOPLa has no static "wiring" and does not use type to establish a stream configuration.

Another early source to mention is the language Sisal [20], which pioneered high-performance functional array processing with stream communication. Sisal was not intended as a coordination language, though, and no attempt at the separation of communication and computation was made in it. Still it is important

to acknowledge the stream variables of SISAL as an early example of task decomposition using streams.

Among more recent papers, we cite the work on the language Eden [21] as related to our effort, since it is based on the concept of stream communication. Here streams are lazy lists produced by processes defined in Haskell using a process abstraction and explicitly instantiated, which are coordinated using a functional-style coordination language. Also, like S-NET, Eden defines a connection topology for the processing entities; it however deploys the processes completely dynamically and even allows completely dynamic channels. Eden has no provision for subtyping and does not integrate topology with types.

Another recent advancement in coordination technology is Reo [22]. The focus of the language Reo is on streams but it concerns itself primarily with issues of channel and component mobility, and it does not exploit static connectivity and type-theoretical tools for network analysis.

9 Discussion

Having briefly presented S-NET we now turn to the issue that naturally arises in discussion of any programming language, but especially a coordination language based on a small set of language constructs: does this language have sufficient power to support any kind of concurrent applications or is it limited to only a certain type of algorithm or a certain concurrency class? In this section we shall discuss these issues, attempting to sketch an argument, if only informally, to demonstrate that the language in question is fairly general purpose.

First of all, for a stream-processing language the variety of network topologies and the need to encode them in a program in a clear and expressive way already constitute a major design challenge. Works by Stefanescu [14] provided a classification of communication graphs for stream processing, and suggested some structuring primitives. However, the formalisation remains quite complex and not suitable for much more than semantics research. There are two main causes for this complexity. The one presenting the easier challenge is the multiplicity of input/output streams incident to a component. To describe the connection of such components in a programming language in a structured fashion would require a nomenclature of primitives, for example the network algebra [13], augmented with some type theory for stream types and processing functions. The algebra includes so-called branching constants, which are primitives that describe the connection patterns of a pair of multi-stream components. It is hard to visualise how exactly components are connected when faced with a sizeable formula saturated with branching constants.

The other cause, which goes deeper than any concerns of expressivity, is the fact that a general streaming network contains cycles. The network algebra offers a primitive to support cycles, but to reason about cyclic processing is as hard as it is to reason about a large set of functions recursively calling one another on a non-recursive data structure (a stream sequence in this case). Besides that,

a cyclic network is prone to deadlock unless measures are taken to schedule the activities in a safe way, which is not always straightforward.

9.1 Acyclic networks

Let us consider the first complication identified above. To start with, is the multiplicity of input/output streams to a component strictly necessary? Could there be several *logical* input or output streams sharing the same sequence of messages? Answers to these questions depend entirely on the buffering mechanism underlying the communication abstraction. Clearly if the buffering space is "sufficient" (and the meaning of this should be defined rigorously), messages can be transparently floated into the same output stream and then demultiplexed at the other end according to their destinations. If, on the other hand, streams have their allocated buffer pools and the allocation is fixed, there is a marked difference between a singly and multiply connected parts: in the former case the transmission on an individual stream can be blocked when its buffer space is exhausted, while in the latter, this applies to the actual connection as a whole and not individual virtual streams. As a result, one virtual stream can be active enough to exhaust the buffer space of the connection and thus prevent other virtual streams from carrying messages for a long time, see Fig. 12. Similarly to any case of spurious dependencies that are due to the finiteness of the resources, this one can cause a deadlock if not controlled properly.

In non-real-time, high-performance streaming applications, the programmer's intuitive view of network processing includes the notion of back pressure, i.e. the blocking of message producers when the consumer is busy. Back pressure tends to propagate back in a pipeline until it reaches the network inputs. It effectively ensures that each cross-section of a pipeline operates with the speed of the slowest stage, and, given static per-stream buffer allocation in the absence of cycles, programming with back pressure guarantees progress. In managing the concurrency of asynchronous networks, back pressure alone is not usually sufficient. Even in an acyclic graph (which is similar to a pipeline) there are multiple streams flowing across the network, and also any synchronisation points depend on more than one stream for progress. For that reason, the coordination programmer must introduce problem-specific mechanisms for throttling concurrency in order to avoid hazards such as starvation and deadlock. But then it should be possible to use the same mechanism to ensure that the top and bottom cases in Fig. 12 have similar behaviours. The bottom case, though, is much easier to manage in terms of network composition, as we shall see below, leading to the SISO design principle we have employed in S-Net.

For all its deceptive simplicity, the stream aggregation exemplified in Fig. 12 is nontrivial. Indeed the form of coordination that it requires is known as a stream merger. Even though this is a service provided to a component by the coordination layer, it is not necessarily transparent to the recipient. Indeed the situation in the figure can be refined by showing the part on the left as being a composition of two chunks and assigning each virtual stream to either chunk. The operation of the chunks could be completely independent, yet the single actual

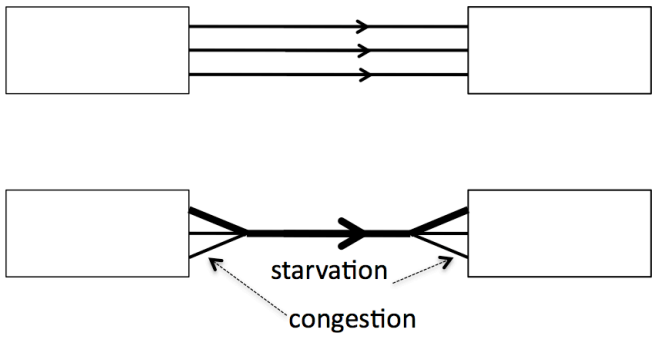

Fig. 12. Network transformation from multiply (top) to singly (bottom) connected parts

stream out of the left-hand part combines messages from the chunks in some order. The right-hand part may require messages from both chunks for progress, and it may need them in a different order. In the multiply connected case, the correct order could be achieved easily, by blocking a stream until messages from it are required. By contrast, if we follow the SISO principle, we must also provide a reordering mechanism so that a consumer may consume in the order of need, rather than in the arbitrary order of the merger.

The trick here is to avoid having to specify either order in the language, and to rely on the adaptivity of the implementation. S-Net enables such adaptivity by allowing nondeterminism. As we have seen before, the confluence of streams can be in no particular order; when that is the case, (i) it is clearly indicated in the coordination program by occurrence of the nondeterministic version of a combinator and (ii) the implementation is in a position to monitor the recipient of the joint stream and change the priorities of the stream merger to save buffer space and reduce the processing latency. Factor (ii) refers to the use of blocking and back pressure not dissimilar to the multiply connected case of fig. 12, which makes the combination of SISO and nondeterminism almost the replacement of multiply connected networks.

Staying within the confines of an acyclic network, the most general topology of a fully connected network is a directed acyclic graph (DAG). The in- and out-degrees of each vertex correspond to the number of input and output streams of the component located at the vertex. To satisfy the SISO requirement, first of all let us augment the graph with a global input node *In*, into which the confluence of all input streams flows as a single combined stream α. The node *In* is (multiply) connected to the nodes that must receive an input stream and its function is to select the relevant portion of α and deliver it to the relevant node. Also augment the graph with a node *Out* that takes all the global output streams and flows then into a single combined output stream

Next, for each component c_i compute $s_i = \max \pi_{i,0}$, which is the maximum path length between c_i and $c_0 = In$. Then arrange the components in a serial-

parallel composition as shown in Fig. 13 in the ascending order of their s_i, from 0 to some $N = s_k$, where $c_k = \mathit{Out}$. Here a triple line at the bottom of a parallel group represents the bypass stream that carries messages not addressed to the other components of the group. Finally drop the *In/Out* nodes as they have become redundant. The SISO version of the graph (shown at the bottom of the diagram) would be incomplete without some routing rules that define which member of a parallel group should receive which incoming message. Since routing is essentially matching the beginning of a path with its end, it is profitable to use a type system for that; this way, other type constraints that component interfaces may export could be captured by the same mechanism. The fact that type systems ordinarily deal with value properties of data rather than the topological matching of a path should not discourage us: as evidenced by algebraic data types, type systems have the ability to introduce abstract labels, which can be used for targeting specific components' inputs.

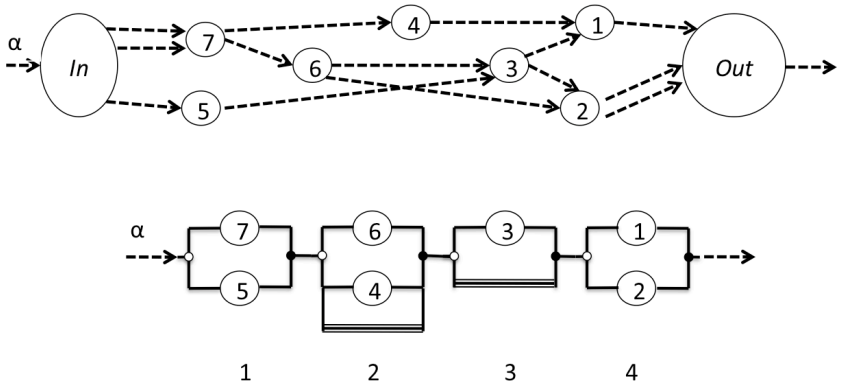

Fig. 13. The transformation of the network DAG (top) to a SISO pipeline(bottom)

The example in Fig. 13 serves as an illustration of the following design principle promoted by S-Net acyclic segments of the network should be coordinated as groups of subnetworks under serial and parallel composition; this provides skeletal routing, and the precise routing information is to be captured by message and component types. The principle, if understood literally, could be construed as the directive to cascade messages through groups of components arranged in parallel. That is not necessarily the case. Indeed the structuring is intended for the purposes of a programming (coordination) language and is there to represent, with the assistance of an appropriate type system, the topological properties of the original multiply-connected network. The implementation can easily reconstruct that network and determine for every message type its destination for a direct dispatch. On the other hand, certain types of hardware (e.g. massively parallel multicore processors) do not allow arbitrary connectivity

anyway, so having to cascade messages through a chain of routers may not be an extra burden.

9.2 Cyclicity

Practical networks tend to be cyclic. Indeed any network solution that involves iteration must apply the same algorithm to data several times, and in an acyclic network that would result in node duplication along with the undesirable duplication of the components placed at the nodes. Yet, for reasons mentioned earlier, it would be beneficial to avoid cyclic configurations in a coordination language. Under normal circumstances these requirements would seem irreconcilable; however for streaming networks there is at least a compromise solution, which we will consider next.

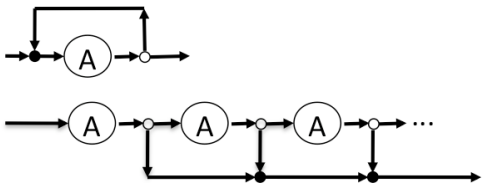

Fig. 14. Unrolling a cyclic SISO network (top) into an infinite regular graph (bottom). Circles ∘ represent splitters by message type and bullets • represent mergers.

It is true that a cyclic network is not equivalent to any finite acyclic network. However, if we allow for infinite networks then cyclicity is quite avoidable. Indeed, a cyclic graph can be unrolled by repeatedly following the edges that form a cycle and duplicating the vertices that have already been visited ad infinitum. Doing this for every cycle that occurs in the graph will convert it to an infinite *regular, acyclic* graph. Informally, a feedback loop is being replaced by a feed-forward infinite pipeline, see Fig. 14. Vertex duplication is, of course, predicated on the fact that the components located at the original and copy vertices can be made identical. This, in turn, requires them to be stateless, since otherwise it would be possible to find the original component and its copy in different states and detect the difference between the cyclic and unrolled configurations. Feed-forward networks are a useful abstraction in its own right: they can represent finite, repetitive, pipelined computations even of a stateful network, if the amount of unrolling is limited (cf. loop unrolling in code optimisation) and if the state information can be decoupled from the component and communicated over the pipeline alongside other data. If a feed-forward structure is used to represent cyclicity, the key difference between them, as made clear in Fig. 14, is the delivery of the input stream. In the cyclic configuration the input messages and the feedback stream arrive at the input of a single subnet A, while in the unrolled version the input stream has to be forwarded to the kth generation

replica, with ever increasing k. The forwarding should be the responsibility of A; however, to avoid the potentially inefficient cascade it is best to use the coordination language facilities that are required already for bypassing messages in an acyclic network, as shown in Fig. 13. The coordination language compiler will then have a chance to recognise cascaded forwarding and to generate management code that eliminates it. Another optimisation the compiler or the run-time system may need to support is the management of the chain length. Indeed, as new messages enter the chain, the replicas of A will generally produce records that are diverted down to the output stream and records that continue to the next replica. It is reasonable to assume that at some point $k = k_t$ the replica c_{k_t} will not produce any output for the next one and so the chain will stop expanding. For each new message entering the chain the value of t will generally be different, but when t decreases, ut may be expedient to collect the tail replicas as garbage (assuming that any persistent state that they may have accumulated has been used up and destroyed[4].)

Consequently, to represent network cycles and repeatable computations, S-Net introduces a feed-forward combinator $A*$ whereby a single subnet A is replicated conceptually infinitely, with only a finite part being used at any given time. Output is achieved by flowing messages of the output type of A into a single stream as shown in Fig. 14 and the input can either be consumed by the first replica or cascaded by replicas together with other continuation data. The coordination compiler and its runtime system must strive to recognise and eliminate cascades and inactive replicas to make this efficient in the general case, and it has all the information it needs to be able to do so.

The reader will now see that the S-Net coordination solution is fairly general. The constraints that S-Net imposes on applications can be summarised as follows:

1. Either the environment or the application code itself must ensure that streams flowing through parallel compositions of networks are reasonably balanced, i.e. the record rates should be similar enough for any instantaneous imbalances to be mitigated by the available buffer space. For static dataflow networks these rates are also statically known, which makes the balancing feasible statically; in a more dynamic case, characteristic of a typical S-Net environment, care must be taken not to overload the buffer space. The compiler/runtime system can and should introduce back pressure to block overactive producers.
2. The application should require a limited degree of loop unrolling. This means that any $*$-networks must have a limited depth, which can be achieved by either the run-time system (via back pressure) or the programmer, by controlling the split between component-level and network level iteration. Generally speaking, this is the old problem of throttling concurrency in a possibly more pleasing guise.

[4] It should be noted that although application components in our approach have no persistent state, coordination objects generally do, but that state is visible to the coordination layer.

3. Last, but not least, the efficiency of the S-Net coordination crucially depends on the adaptivity of its implementation. If the latter is capable of compiling subnetworks into a single conventional program (or a multithreaded program as the case may be) on the fly, then the design principle for the programmer would be aggressive decomposition down to very light and compact components. Any excessive concurrency could be absorbed dynamically by switching to the co-compiled version of a hot spot. If an implementation of S-Net has no such adaptivity, the feasibility of coordination would critically depend on the granularity of component algorithms.

10 Conclusions and future work

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

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

S-Net has been fully implemented and is now available for download from the project homepage at `http://www.snet-home.org/`. The implementation consists of a compiler including a type inference system [23], a multithreaded runtime system for shared memory architectures [24] and on top of that an MPI-based runtime system extension for distributed and hybrid memory architectures [25].

We are currently working on an application suite to demonstrate the suitability of S-Net to coordinate concurrent activities on a representative scale. These applications are drawn from a variety of domains including plasma physics and radar imaging. A smaller scale case study on the interplay between S-Net and the functional array language SaC [26] as component implementation language can be found in [17]. The theme here is the concurrent solving of Sudoku puzzles, which we deem representative for a relevant class of search problems.

References

1. Broy, M., Stefanescu, G.: The algebra of stream processing functions. Theoretical Computer Science (2001) 99–129
2. 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
3. Ashcroft, E.A., Wadge, W.W.: Lucid, a nonprocedural language with iteration. Communications of the ACM **20** (1977) 519–526
4. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data-flow programming language LUSTRE. Proceedings of the IEEE **79** (1991) 1305–1320
5. Berry, G., Gonthier., G.: The Esterel synchronous programming language: Design, semantics, implementation. Science of Computer Programming **19** (1992) 87–152
6. Binder, J.: Safety-critical software for aerospace systems. Aerospace America (2004) 26–27
7. 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
8. 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
9. 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)
10. Stephens, R.: A survey of stream processing. Acta Informatica **34** (1997) 491–541
11. 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
12. 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
13. Stefanescu, G.: An algebraic theory of flowchart schemes. In Franchi-Zannettacci, P., ed.: Proceedings 11th Colloquium on Trees in Algebra and Programming, Nice, France, 1986. Volume LNCS 214., Springer-Verlag (1986) 60–73
14. Stefanescu, G.: Network Algebra. Springer-Verlag (2000)
15. Shafarenko, A.: Stream processing on the grid: an array stream transforming language. In: SNPD. (2003) 268–276
16. Grelck, C., Shafarenko, A. (eds):, Penczek, F., Grelck, C., Cai, H., Julku, J., Hölzenspies, P., Scholz, S.B., Shafarenko, A.: S-Net Language Report 1.0. Technical Report 487, University of Hertfordshire, School of Computer Science, Hatfield, England, United Kingdom (2009)
17. Grelck, C., Scholz, S.B., Shafarenko, A.: Coordinating Data Parallel SAC Programs with S-Net. In: 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS'07), Long Beach, California, USA, IEEE Computer Society Press, Los Alamitos, California, USA (2007)
18. Papadopoulos, G.A., Arbab., F.: Coordination models and languages. In: Advances in Computers. Volume 46. Academic Press (1998)

19. Florijn, G., Bessamusca, T., Greefhorst, D.: Ariadne and HOPLa: flexible coordination of collaborative processes. In Ciancarini, P., Hankin, C., eds.: First International Conference on Coordination Models, Languages and Applications (Coordination'96), Cesena, Italy, 15-17 April, 1996. LNCS 1061. (1996) 197–214
20. Feo, J.T., Cann, D.C., Oldehoeft, R.R.: A report on the sisal language project. J. Parallel Distrib. Comput. **10** (1990) 349–366
21. Loogen, R., Ortega-Mallén, Y., Peña-Marí, R.: Parallel functional programming in Eden. Journal of Functional Programming **15** (2005) 431–475
22. Arbab, F.: Reo: a channel-based coordination model for component composition. Mathematical. Structures in Comp. Sci. **14** (2004) 329–366
23. Cai, H., Eisenbach, S., Grelck, C., Penczek, F., Scholz, S.B., Shafarenko, A.: S-Net Type System and Operational Semantics. In: Proceedings of the Æther-Morpheus Workshop From Reconfigurable to Self-Adaptive Computing (AMWAS'08), Lugano, Switzerland. (2008)
24. Grelck, C., Penczek, F.: Implementation Architecture and Multithreaded Runtime System of S-Net. In Scholz, S., Chitil, O., eds.: Implementation and Application of Functional Languages, 20th International Symposium, IFL'08, Hatfield, United Kingdom, Revised Selected Papers. Lecture Notes in Computer Science, Springer-Verlag (2009) to appear.
25. Grelck, C., Julku, J., Penczek, F.: Distributed S-Net. In Morazan, M., ed.: Implementation and Application of Functional Languages, 21st International Symposium, IFL'09, South Orange, NJ, USA, Seton Hall University (2009) 39–54
26. Grelck, C., Scholz, S.B.: SAC: A functional array language for efficient multithreaded execution. International Journal of Parallel Programming **34** (2006) 383–427