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.

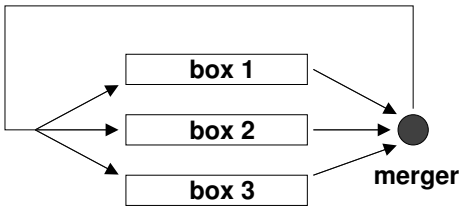

Fig. 1. The Turner scheme

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

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.

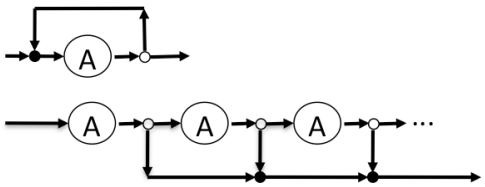

Fig. 14. Unrolling a cyclic SISO network (top) into an infinite regular graph (bottom). Circles ∘ represent splitters by message type and bullets • represent mergers.

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