An Operational Semantics for S-Net

Frank PENCZEK [a,1] Clemens GRELCK [a,b] Sven-Bodo SCHOLZ [a]
[a] *University of Hertfordshire, School of Computer Science, United Kingdom*
[b] *University of Amsterdam, Institute of Informatics, The Netherlands*

Abstract. We present the formal operational semantics of S-Net, a coordination language and component technology based on stream processing. S-Net turns conventional (sequential) functions/procedures into asynchronous components interacting with each other through a streaming network; it defines network topologies inductively by a small combinator language that captures essential forms of concurrency. Our formal semantics allows us to reason about program properties and defines the design space for alternative implementation strategies.

1. Introduction

Today's hardware trend towards multi-core/many-core chip architectures [1,2] places immense pressure on software manufacturers: For the first time in history software does not automatically benefit from new generations of hardware. Today, software must become parallel in order to benefit from future processor generations! However, existing software is predominantly sequential, and writing parallel software is notoriously difficult. So far, parallel computing has been confined to supercomputing. Now, it must go mainstream. This step requires new tools and techniques that radically facilitate parallel programming.

S-Net [3] is such a novel technology. Our key design principle is the separation of concerns between *application engineering* and *concurrency engineering*. The former applies domain-specific knowledge to provide application building blocks of suitable granularity using a familiar sequential programming environment. The latter applies expert knowledge of target architectures and concurrency in general to orchestrate sequential building blocks into a parallel application.

S-Net turns conventional functions/procedures into asynchronous, state-less components named *boxes*. Each box has a single input and a single output stream. This restriction is motivated by our guiding principle: The concern of a box is mapping input values into output values, whereas its purpose within a streaming network is unknown to the box itself. Concurrency concerns like synchronisation and routing that immediately arise if a box had multiple input streams or multiple output streams are thus kept away from boxes.

It is a distinguishing feature of S-Net that streams are no explicit objects. Instead, we use algebraic formulae to define the connectivity of boxes. We have

[1]Corresponding Author: Frank Penczek, University of Hertfordshire, Science and Technology Research Institute, Hatfield, Herts, AL10 9AB, United Kingdom, e-mail: f.penczek@herts.ac.uk

identified four fundamental construction principles for streaming networks: *serial composition* of two components, *parallel composition* of two components where some routing oracle determines the branch to take, *serial replication* of one component where data is streamed through consecutive instances of the component, and *indexed parallel replication* where an index attached to the data determines which branch (i.e. which replica) to take. Network construction preserves the SISO property: any network, regardless of its complexity, again is a SISO component. We build S-Net on these construction principles because they are pairwise orthogonal, each represents a fundamental principle of composition beyond the concrete application to streaming networks (i.e. serialisation, branching, recursion, indexing), and they naturally express the prevailing models of parallelism (i.e. task parallelism, pipeline parallelism, data parallelism).

The contribution of this paper is a formal operational semantics of S-Net. Defining the meaning of language constructs in a rigorous formal nomenclature will allow us to reason about properties of programs. Equally important, it precisely defines the design space of implementations. This is of particular interest as different implementations of S-Net with orthogonal design philosophies are currently under development [4,5].

The remainder of this paper is structured as follows: Section 2 introduces S-Net in greater detail. In Section 3 we present our operational semantics. We discuss some related work in Section 4 and conclude in Section 5.

2. S-Net at a glance

S-Net structures messages between components as non-recursive records, i.e. sets of label-value pairs. Labels are subdivided into *fields* and *tags*. Fields are associated with values from the box language domain that are opaque to S-Net. Tags are associated with integer numbers that are accessible both on the S-Net and the box language level. Tag labels are distinguished from field labels by angular brackets. For example, `{a,b,<t>}` is the record type (set of labels) of messages containing fields `a` and `b` and tag `t`.

In S-Net, we define a box using the key word `box` followed by the box name and a *type signature*, i.e. a mapping from an *input type* to a disjunction of *output types*. For example,

```
box foo ({a,<b>} -> {c} | {c,d,<e>})
```

declares a box named `foo` that expects records with a field labelled `a` and a tag labelled `b`. The box responds with a number of records that either have just a field `c` or fields `c` and `d` as well as tag `e`. Both the number of output records and the choice of variants are at the discretion of the box implementation alone. As soon as a record is available on the input stream, the box consumes that record, applies the associated function to the record elements and emits the resulting records on its output stream. S-Net boxes are stateless by definition, i.e., the mapping of an input record to a stream of output records is free of side-effects.

In fact, the above type signature makes box `foo` accept *any* input record that has *at least* field `a` and tag `b`, but may well contain further fields and tags. The formal foundation of this behaviour is *structural subtyping* on records: Any record

type t_1 is a subtype of t_2 ($t_1 \preccurlyeq t_2$) iff $t_2 \subseteq t_1$. This subtyping relationship extends nicely to multivariant types, e.g. the output type of box `foo`: A multivariant type x is a subtype of y if every variant $v \in x$ is a subtype of some variant $w \in y$. Subtyping on the input type of a box means that a box may receive input records that contain more fields and tags than the box is supposed to process. Such fields and tags are retrieved from the record before the box starts processing and are added to each record emitted by the box in response to this input record, unless the output record already contains a field or tag of the same name. We call this behaviour *flow inheritance*. In conjunction, record subtyping and flow inheritance prove to be indispensable when it comes to making boxes that were developed in isolation to cooperate with each other in a streaming network.

In S-Net, we define streaming networks using the key word `net` followed by the network name. For example,

`net bar connect` *expr*

defines the network `bar`. Following the key word `connect`, an expression built out of previously defined box and network names and four network combinators determines the network topology.

Let `A` and `B` be the names of two S-Net networks or boxes. Serial composition (denoted `A..B`) constructs a new network where the output stream of `A` becomes the input stream of `B` while the input stream of `A` and the output stream of `B` become the input and output streams of the compound network. Parallel composition (denoted `(A|B)`) constructs a network where all incoming records are either sent to `A` or to `B` and the resulting record streams are merged to the compound output stream. Type inference [6] associates each network with a type signature similar to the type signatures of boxes. Any incoming record is directed towards the branch whose input type matches the type of the record best. Serial replication (denoted `A*`*`type`*) constructs an unbounded chain of serially composed instances of `A` with exit pattern *`type`*. At the input stream of each instance of `A`, we compare the type of an incoming record (i.e. the set of labels) with *`type`*. If the record's type is a subtype of the specified type (we say, it matches the exit pattern), the record is routed to the compound output stream, otherwise into the instance of `A`. Indexed parallel replication (denoted `A!<`*`tag`*`>`) replicates instances of `A` in parallel. Unlike in static parallel composition routing is value dependant on a tag specified as right operand. All incoming records must feature this tag; its value determines the instance of the left operand the record is sent to. Output records are non-deterministically merged into a single output stream.

Parallel composition (`|`), serial replication (`*`) and indexed parallel replication (`!`) involve non-deterministic merging of output streams. While S-Net guarantees the order of records communicated over individual streams, very few guarantees on record ordering can be given in these three cases. For many applications the concrete record order is irrelevant, but for others we need stronger guarantees. Therefore, S-Net features deterministic variants of these combinators, written `||`, `**` and `!!`, respectively. Deterministic combinators maintain the *causal order* of records between their input and their output stream, i.e. any record that is an offspring of an earlier record on the input stream is guaranteed to leave the network before any offspring of a later record on the input stream.

Last but not least, S-Net features a synchronisation component that we call *synchrocell*; it takes the syntactic form `[|type,type|]`. Types again act as patterns for incoming records. A record that matches one of the patterns is kept in the synchrocell. As soon as a record arrives that matches the other pattern, the two records are combined into one, which is forwarded to the output stream. Incoming records that only match previously matched patterns are immediately sent to the output stream. Indeed, after successful synchronisation a synchrocell becomes an identity component and may be removed by a runtime system. This extremely simple behaviour of synchrocells captures the essential notion of synchronisation in the context of streaming networks. More complex synchronisation behaviours, e.g. continuous synchronisation of matching pairs in the input stream, can easily be achieved using synchrocells and network combinators. See [7] for more details on this and on the S-Net language in general.

3. Operational Semantics

In order to precisely define the meaning of a program, we provide a formal operational semantics for all entities of S-Net in the form of deduction rules. These rules allow us to construct inductive proofs about the result of a given program and given input that is valid wrt. the program's type. Rules of the form

$$(p, M) \rightarrow (M', q)$$

describe state transitions of our abstract machine: The program M is evaluated on input p and produces output q. As the program M potentially changes during this evaluation, the right-hand-side of the above rule contains M' rather than M. We use this notation in the following formalisation to capture input-dependant transformations, i.e. synchronisation and dynamic unfolding, of the original program. Furthermore, we use the following notation: A single italic letter, as for example p, denotes a single record. A single letter with an arrow on top denotes a stream of records, e.g. $\vec{p}$. The concatenation of two streams is denoted by $\vec{p} ++ \vec{q}$ (appending $\vec{q}$ to $\vec{p}$). An n-fold concatenation, denoted by $++_{i=1}^{n}(\vec{p_i})$, expands to $\vec{p_1} ++ \vec{p_2} ++ ... ++ \vec{p_n}$. As some of the presented rules are defined on single input records, we provide the following map rule in order to apply these rules to streams:

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

The box construct in S-Net allows us to execute user-defined functions, denoted f in the box rule. Note here that the semantics of f are not captured in this framework.

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

The serial combination is defined in rule Ser. The combinator's behaviour is broken down into two independant evaluation steps of the operands:

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

The non-deterministic choice combinator analyses inbound records in order to dispatch these to the most appropriate operand network. To do this,

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

the inbound stream $\vec{p}$ is divided into two sub-streams $\vec{p_l}, \vec{p_r}$ using algorithm `lrsplit` which is shown at the end of this section. The algorithm splits the inbound stream up such that records on $\vec{p_l}$ match the left operand of the combinator and the records on $\vec{p_r}$ match the right operand. These result streams are non-deterministically merged, i.e. both streams may be arbitrarily interleaved.

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

In order to maintain record order for the deterministic variant of the choice combinator (DCHOICE), the inbound stream $\vec{p}$ is divided into pairs of sub-streams $(\vec{p_{l_i}}, \vec{p_{r_i}})$ such that each (potentially empty stream) $\vec{p_{l_i}}$ (resp. $\vec{p_{r_i}}$) contains records matching the input type of the left (resp. right) operand network. The algorithm to do this is shown at the end of this section. The pairs are processed by the operand networks and produce streams $\vec{q_{l_i}}$ and $\vec{q_{r_i}}$ as result. The output streams are concatenated to a result stream and represent the result for one input pair. The overall outbound stream is constructed by concatenating result streams of all input pairs.

The feed-forward semantics of the star combinator is made prominent by the STAR rules. An unrolling of the operand network is reduced to a serial composition and a recursive application of the rules. The non-deterministic variant of this combinator is defined by NDCHOICE, the order preserving variant by DCHOICE. As can be seen from DSTAR, record order in the deterministic case is only preserved with respect to the outermost level. The $[\sigma\texttt{->}\sigma]$ component that appears in the rules is a typed ID component. Its type σ ensures that records matching the exit pattern are attracted to the branch of the ID component.

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

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

The split combinator uses a special form of the choice combinator, syntactically distinguishable from the standard choice combinator by two parameters on top of the bar, which we introduce only informally: This combinator uses value

dependant routing rather than purely type directed routing. It examines the value of a given tag name κ and checks it against given value j. If the values match, the combinator routes the record to its left operand and to its right operand otherwise.

$$\textsc{NDsplit} \quad : \quad \frac{(p, N \overset{\kappa=j}{|} N!\kappa) \to (N', \vec{q})}{(p, N!\kappa) \to (N', \vec{q})} \qquad \textsc{Dsplit} \quad : \quad \frac{(p, N \overset{\kappa=j}{||} N!!\kappa) \to (N', \vec{q})}{(p, N!!\kappa) \to (N', \vec{q})}$$

We use multiple rules to capture synchro cell semantics: If a record matches a previously unmatched pattern, i.e. the type τ_p of a record p is subtype of pattern σ, the pattern is marked as matched (rule Syncs). An index at the lower-right corner of the synchro cell indicates this. The Syncn rule captures the case for a matched first pattern. We will ommit symmetric cases concerning the second pattern. Merging and conversion to an identity component is described by the Syncm rule. If a record immediately matches both patterns, the record passes unmodified and the synchro cell again will act as an identity component (rule Id), as shown by the Synci rule.

$$\textsc{Syncs} \quad : \quad \frac{\tau_{p_a} \preccurlyeq \sigma_a \wedge \neg(\tau_{p_a} \preccurlyeq \sigma_b)}{(p_a, [\![\sigma_a, \sigma_b]\!]) \to ([\![\sigma_a, \sigma_b]\!]_{p_a}, \epsilon)}$$

$$\textsc{Syncn} \quad : \quad \frac{\tau_p \preccurlyeq \sigma_a \wedge \neg(\tau_p \preccurlyeq \sigma_b)}{(p, [\![\sigma_a, \sigma_b]\!]_{p_a}) \to ([\![\sigma_a, \sigma_b]\!]_{p_a}, p)}$$

$$\textsc{Synci} \quad : \quad \frac{\tau_p \preccurlyeq \sigma_a \wedge \tau_p \preccurlyeq \sigma_b}{(p, [\![\sigma_a, \sigma_b]\!]) \to (\mathtt{id}, p)} \qquad \textsc{Id} \quad : \quad \frac{}{(p, \mathtt{id}) \to (\mathtt{id}, p)}$$

The following code samples show the implementation of some algorithms that are used within the inference rules. Due to space limitations we have chosen to present only those algorithms here which we deem to be the most interesting. In the code below, `ndsel` denotes non-deterministic selection and `bestmatch` choses those patterns that have the most labels in common with a given record.

```
prefix :: stream → pattern → pattern → (stream, stream)
prefix p⃗ σ_l σ_r =
case p⃗ of
  ϵ → (ϵ,ϵ)
  p : p⃗s | ndsel (bestmatch p σ_l σ_r) == σ_l → (p:q⃗, r⃗)
  p : p⃗s | otherwise → (ϵ, p⃗)
where (q⃗, r⃗) = prefix p⃗s σ_l σ_r

splitOnePair :: stream → pattern → pattern → (stream, stream, stream)
splitOnePair p⃗ σ_l σ_r =
case p⃗ of
  ϵ → (ϵ, ϵ, ϵ)
  p : p⃗s → (q⃗_l, q⃗_r, r⃗)
where (q⃗_l, t⃗) = prefix p⃗ σ_l σ_r
      (q⃗_r, r⃗) = prefix t⃗ σ_r σ_l

lrpsplit :: stream → pattern → pattern → (streams, streams)
lrpsplit p⃗ σ_l σ_r =
```

```
case p⃗ of
  ϵ → (ϵ⃗, ϵ⃗)
  p : p⃗s → (l⃗_p : l⃗⃗_pl, r⃗_p : r⃗⃗_pl)
where (l⃗_p, r⃗_p, q⃗) = splitOnePair p⃗ σ_l σ_r
      (l⃗⃗_pl, r⃗⃗_pl) = lrpsplit q⃗ σ_l σ_r

lrsplit :: stream → pattern → pattern → (stream, stream)
lrsplit p⃗ σ_l σ_r = let (p⃗⃗_l, p⃗⃗_r) = lrpsplit p⃗ σ_l σ_r
                    in (map concat p⃗⃗_l, map concat p⃗⃗_r)

merge :: record → record → record
merge p q = p ∪ (q \ (p ∩ q))
```

4. Related Work

Due to space limitations we only give a very brief account of selected formalisation approaches for stream processing languages. For a survey on these languages in general see [8], for a thorough formal treatment of the subject [9] and [10]).

Some early works in the area of stream processing did not propose concrete languages as such, but specified computational models instead. In [11] the authors set out to provide a model for queue-based parallel computations and formalised their approach on graph-theoretical foundations. A function-based approach is presented in [12], where stream functions are introduced as a means for structured programming. A concrete language developed in the 1970s is Lucid [13]. Its semantics is rooted in temporal logic.

A more recent development is the language Eden [14], that extends Haskell with process abstraction and instantiation facilities. The formalisation of its semantics [15] uses a layered approach and distinguishes between the computational layer and coordination layer, but still describes both in the same framework.

Another functionally-based language is Hume, which combines a Haskell-like box language with synchronous data flow. The language provides detailed cost and space analysis, and so its operational semantics focus strongly on these issues. A formalisation may be found in [16].

Formalisation of the semantics for Streamit [17] mainly focuses on the implemented messaging system and captures constraints on the execution schedule for a program. The aim here is to be able to reason about guarantees on message delivery and latency of Streamit programs.

5. Conclusion

We have developed a high-level coordination language and presented its formal semantics. This formal specification is the foundation for any implementation of the language. In fact, we have implemented a complete, portable tool chain [4] that enables users to harness the computational power of modern multi-core architectures while at the same time they can stick to their familiar (sequential) programming environment for the bulk of an application.

In the setting of S-Net it is rather natural to separate the formalisation of operational semantics of the coordination layer from the actual computation inside the boxes. Nevertheless, it is an intriguing objective to provide a comprehensive formal system that would allow simultaneous reasoning about the embedded box language and the coordination layer. In [18], Matthews *et.al.* present approaches for such a system. We leave this as interesting future work.

Acknowledgments

We would like to thank Alex Shafarenko for many fruitful discussions on S-Net and the anonymous reviewers for their valuable comments.

References

[1] Sutter, H.: The free lunch is over: A fundamental turn towards concurrency in software. Dr. Dobb's Journal **30** (2005)

[2] Held, J., Bautista, J., Koehl, S.: From a few cores to many: a Tera-scale computing research overview. Technical report, Intel Corporation (2006)

[3] Grelck, C., Scholz, S.B., Shafarenko, A.: A Gentle Introduction to S-Net: Typed Stream Processing and Declarative Coordination of Asynchronous Components. Parallel Processing Letters **18** (2008) 221–237

[4] Grelck, C., Penczek, F.: Implementation Architecture and Multithreaded Runtime System of S-Net. In Scholz, S., Chitil, O., eds.: IFL'08, Hatfield, United Kingdom. LNCS, Springer-Verlag (2009)

[5] Bousias, K., Jesshope, C., Thiyagalingam, J., Scholz, S.B., Shafarenko, A.: Graph Walker: Implementing S-Net on the Self-adaptive Virtual Processor. In: Proceedings of AMWAS'08, Lugano, Switzerland. (2008)

[6] Cai, H., Eisenbach, S., Grelck, C., et.al., F.P.: S-Net Type System and Operational Semantics. In: Proceedings of AMWAS'08, Lugano, Switzerland. (2008)

[7] 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, United Kingdom (2009)

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

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

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

[11] Karp, R.M., Miller, R.E.: Properties of a model for parallel computations: Determinancy, termination, queueing. SIAM Journal on Applied Mathematics **14** (1966) 1390–1411

[12] Burge, W.H.: Stream processing functions. j-IBM-JRD **19** (1975) 12–25

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

[14] Loogen, R., Ortega-Mallén, Y., Peña-Marí, R.: Parallel functional programming in Eden. Journal of Functional Programming **15** (2005) 431–475

[15] Hidalgo-Herrero, M., Ortega-Mallén, Y.: An operational semantics for the parallel language eden. Parallel Processing Letters **12** (2002) 211–228

[16] Hammond, K.: The dynamic properties of hume: a functionally-based concurrent language with bounded time and space behaviour. In: IFL 2000 Aachen, Germany, September 4-7, 2000, Selected Papers. Volume 2011 of LNCS., Springer (2001) 122–139

[17] Thies, W., Karczmarek, M., Amarasinghe, S.P.: Streamit: A language for streaming applications. In: Computational Complexity. (2002) 179–196

[18] Matthews, J., Findler, R.B.: Operational semantics for multi-language programs. SIGPLAN Not. **42** (2007) 3–10