Towards reconfiguration and self-adaptivity in S-Net

Penczek, F.; Scholz, S.-B.; Grelck, C.

Publication date
2008

Document Version
Submitted manuscript

Published in
20th International Symposium on the Implementation and Application of Functional Languages (IFL 2008)

Towards Reconfiguration and Self-Adaptivity in S-Net

— Draft —

Frank Penczek[1], Sven-Bodo Scholz[1] and Clemens Grelck[1,2]

[1] University of Hertfordshire
Department of Computer Science
Hatfield, Herts, AL10 9AB, United Kingdom
{f.penczek,s.scholz,c.grelck}@herts.ac.uk
[2] University of Amsterdam
Institute of Informatics
Kruislaan 403, 1098 SJ Amsterdam, Netherlands
grelck@science.uva.nl

Abstract. Stream processing is a well-suited model for parallel programming, as it allows the programmer to design parallel algorithms intuitively by arranging computational tasks into a data-flow graph and consecutively constructing a streaming network from it. However, a network that was designed with a specific workload in mind will not work optimally if anticipated parameters, as for example data rates or computational costs per data item, change over time. To nonetheless achieve optimal performance, a restructuring of the network and re-implementation of computational components is inevitable.

As the deployment of a revised network usually causes service disruption, we present a system that supports reconfiguration of streaming networks at runtime. The reconfiguration of networks can either be triggered externally, i.e. initiated by the user, or from within the network itself (self-adaptation) by, for example, monitoring certain runtime parameters.

Our system is based on S-Net, a coordination language for asynchronous stream processing systems. S-Net supports the simultaneous use of computational components implemented in a range of programming languages and it offers network combinators to construct streaming networks from these components. We will introduce S-Net and extensions of the language that allow for reconfiguration and self-adaptation of networks at runtime. The extensions are designed as network combinators and integrate seamlessly into the existing language.

1 Introduction

Parallel programming, which has traditionally been used for high-performance computing on large-scale parallel architectures, is going mainstream. The broad

availability of inexpensive multi-core CPUs drives the demand for general purpose programming languages that enable programmers to harness the full computational power of these novel architectures. For languages as C, C++, Java, Fortran etc. several APIs are available that extend these languages by facilities for parallel programming. The APIs allow programmers to use their language of choice and to re-use legacy code but are rather low-level: MPI and PVM require explicit communication management via send- and receive calls. APIs as PThreads, which implement communication via shared memory, require explicit synchronisation and locking to ensure correct and deadlock-free execution. These APIs also require manual assignment of work to available processes or threads. OpenMP offers an higher-level approach through directives, but requires special compiler support. All these approaches intertwine management code and computational code which makes programming an error-prone process and obfuscates the resulting code.

On the other end of the spectrum, there are languages with (partially) implicit parallelism. This high-level approach, which is taken by for example languages as SAC or Data Parallel Haskell, takes away the burden of implementing low-level and often quite complex tasks of work distribution, communication and synchronisation. Being able to entirely focus on the design of an algorithm, turns programming into a much more efficient process and leads to better readable and maintainable code. The only drawback is that programmers need to learn how to use these languages; legacy code cannot be reused and needs to be reimplemented.

To combine the advantages of the above approaches but not their disadvantages, we propose a coordination language that strictly separates code for computation from code for coordination of computation. The model we have developed distinguishes between computational components that interact with each other and the design of a communication network over which the interaction is carries out. The computational aspect is kept as a separate concern – in fact, we allow arbitrary languages for the implementation of the computational components. The focus of the proposed language lies on the coordination aspect. It provides combinators to construct a streaming network which contains computational components as nodes. This approach especially supports reuse of legacy code without cluttering it up with code for the traditional APIs, as all communication is defined outside the computational components. It also allows for a very lean language design, as computational aspects are left to a programming language of the user's choice.

As indicated above, our system is in fact a coordination language for stream processing networks. Stream processing is a well-suited model for parallel programming, as it allows the programmer to design parallel algorithms intuitively by arranging computational tasks into a data-flow graph and consecutively constructing a streaming network from it. The amount of exposed concurrency is influenced by the design decisions made for decomposition of the problem into separate computational components, as each component may potentially be assigned to different computing resources. The topology of the network, which is

constructed with the provided combinators, defines the communication scheme of the program. This allows development of parallel programs on a high level of abstraction and in an intuitive way: The program design is representable as an box-and-arrow diagram which hides the low-level and often complex requirements of communication and synchronisation issues.

Ideally, any streaming network is designed with a target system in mind. The granularity of the decomposition is chosen to match the computing resources of the target system to generate an optimal workload. This, of course, requires knowledge about the target. If this knowledge is not available, a programmer can only design a very generic model. However, a network that was designed with a specific workload in mind will not work optimally if anticipated parameters, as for example data rates or computational costs per data item, change over time. To nonetheless achieve optimal performance, a restructuring of the network and re-implementation of computational components is inevitable. Once this re-engineering has taken place, the generic model is superseded by a more specialised implementation which then is deployed. As the deployment of a revised network usually causes service disruption, our system supports reconfiguration of streaming networks at runtime. The reconfiguration of networks can either be triggered externally, i.e. initiated by the user, or from within the network itself (self-adaptation) by, for example, monitoring certain runtime parameters.

Our approach to provide these facilities is twofold. Firstly, we promote networks to first class citizens of the language. That is, data items that are sent across the network may be networks itself. Secondly, we introduce a replacement combinator. This combinator marks (sub-)networks as replaceable by new versions of networks which it receives over the stream. The combinator ensures that replacement of networks is only carried out if a received network is compatible, i.e. we guarantee that the replacement process will not break the specified semantics of the overall system. This approach enables a user to react to changes of the environment, e.g. increase or decrease of resources, bandwidth changes, etc., and reconfigure the system accordingly by sending revised versions of sub-networks over the existing network infrastructure.

As networks are first class citizens, revised versions of networks may also be created and emitted from withing networks itself. Combining this with a feedback combinator, our design is powerful enough to implement self-adaptive networks that dynamically reconfigure themselves based on runtime measurements of, for example, throughput or execution times.

2 Introducing S-Net

to be included in the final version of this paper

3 Introducing S-Net$^{\Omega}$

to be included in the final version of this paper

4 S-Net Language Components and Semantics

Throughout the following chapters we shall 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 $\vec{p}$), concatenation of an element and a stream by $p \triangleleft \vec{p}$ (prefixing $\vec{q}$ by p) and $\vec{q} \triangleright p$ (appending p to $\vec{q}$). An n-fold concatenation, denoted by $+{+}_{i=1}^{n}(\vec{p}_i)$, expands to $\vec{p}_1 {+}{+}\, \vec{p}_2 {+}{+} \ldots {+}{+}\, \vec{p}_n$.

4.1 Boxes and Primitive Boxes

Boxes are the only components in S-Net that are able to process and modify data of record fields. On input of a single record, a box may produce an arbitrary number (including zero) of result records. A distinctive feature of S-Net is the fact that function f of rule Box may be implemented in an arbitrary programming language.

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

A synchro cell, the only stateful component in S-Net, is a facility to store and merge records that match a user-defined pattern. For this, synchro cells are parametrised over two patterns. Any inbound record is matched against these patterns. If a record matches a previously unmatched pattern, it is stored by the synchro cell, the pattern is marked as matched and no output is produced (rule Syncs). An index at the lower-right corner of the synchro cell indicates this. If a pattern was matched before, the record is simply output again, as described by the Syncn rule. A record that matches the last remaining unmatched pattern triggers a merge of stored and inbound record. After merging and outputting the resulting record, the synchro cell will act as an identity component (Syncm rule). If a record immediately matches both patterns, the record passes unmodified and the synchro cell again will act as an identity component, as shown by the Synci rule.

$$\textsc{Syncs} \quad : \quad \frac{\text{ismatch}(\sigma_a, p_a) \wedge \neg\text{ismatch}(\sigma_b, p_a)}{(p_a, [\![\sigma_a, \sigma_b]\!]) \rightarrow ([\![\sigma_a, \sigma_b]\!]_{p_a}, \epsilon)}$$

$$\textsc{Syncn} \quad : \quad \frac{\text{ismatch}(\sigma_a, p) \wedge \neg\text{ismatch}(\sigma_b, p)}{(p, [\![\sigma_a, \sigma_b]\!]_{p_a}) \rightarrow ([\![\sigma_a, \sigma_b]\!]_{p_a}, p)}$$

$$\textsc{Synci} \quad : \quad \frac{\text{ismatch}(\sigma_a, p) \wedge \text{ismatch}(\sigma_b, p)}{(p, [\![\sigma_a, \sigma_b]\!]) \rightarrow (\texttt{id}, p)}$$

$$\textsc{Syncm} \quad : \quad \frac{\text{ismatch}(\sigma_b, p)}{(p, [\![\sigma_a, \sigma_b]\!]_{p_a}) \rightarrow (\texttt{id}, \text{merge}(p_a, p))}$$

The filter (see rule FILTER) is a versatile instrument to modify the structure of records. Filters can split records, rename, copy or discard fields and tags and even insert new tags and modify their value. The behaviour of each filter is controlled by a pattern σ and by a user-defined list α of aforementioned actions. The pattern defines which constituents of inbound records are accessible by the filter actions. Only fields and tags that are present in the pattern may be used in the action list. On arrival of a record, the actions are applied to the inbound record and all resulting records are output. The S-NET$^{\Omega}$ filter is also equipped with pattern matching facilities for network functions. With this, it is possible to assign identifiers to sub-expressions of a function on the left-hand side of the filter and use these identifiers on the right-hand side to construct new network functions. The filter also features runtime variables that contain values of the current system clock tick and the average throughput seen by the filter. These values may be assigned to tags for later use.

$$\textsc{Filter} \quad : \quad \frac{\vec{q} = \mathrm{apply}(\alpha, p)}{(p, [\sigma \texttt{ -> } \alpha]) \rightarrow ([\sigma \texttt{ -> } \alpha], \vec{q})}$$

4.2 S-Net Combinators

The four S-NET combinators, which are used to construct networks from boxes, primitive boxes and networks, are very briefly introduced here.

The serial combinator connects the output of its left operand to the input of its right operand. The output of the right operand thereby forms the output of the newly constructed network.

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

The choice combinator creates a new network by connecting its two operands in parallel. Records are routed to either of the operands depending on which operand is a more specific match. The specificity of the match is determined by analysing the type of the inbound record and the input type of the operands. If both operands match equally well, one is chosen non-deterministically. The choice combinator comes in two variants, deterministic and non-deterministic. The deterministic variant preserves the order of inbound and resulting outbound records. If record order is not a concern, the non-deterministic variant may be used. The DCHOICE rule formalises the behaviour of the deterministic choice combinator.

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

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. These pairs are processed by the operand networks, whose internal state may change due to synchro cells, 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 behaviour of the non-deterministic choice combinator is expressed by the NDCHOICE rule.

$$\text{NDCHOICE} \quad : \quad \frac{\vec{p_l}, \vec{p_r} = \text{lrsplit}(\vec{p}, \tau_M, \tau_N) \quad (\vec{p_l}, M) \to (M', \vec{q_l}) \quad (\vec{p_r}, N) \to (N', \vec{q_r})}{(\vec{p}, M|N) \to (M'|N', \text{ndzip}(\vec{q_l}, \vec{q_r}))}$$

Any inbound stream of records $\vec{p}$ is divided into two separate sub-streams $\vec{p_l}$ and $\vec{p_r}$, such that each record in $\vec{p_l}$ (resp. $\vec{p_r}$) matches the inbound type of the left (resp. right) operand of the choice combinator. The operand networks, whose internal state may change due to synchro cells, produce $\vec{q_l}$ and $\vec{q_r}$ as results. These result streams are non-deterministically merged, i.e. both streams may be arbitrarily interleaved.

The star combinator requires an operand and a pattern for operation. Any inbound record that matches the pattern is immediately output. If the record does not match the pattern, it is sent to the operand. At the output of the operand, the same process repeats. If the result matches the patten, it is output, otherwise a new instance of the operand is spawned and the record is sent to it. The star combinator is available as deterministic and non-deterministic combinator. The deterministic variant guarantees that any result of an earlier input will exit the network before any other result of a later input, i.e. outbound streams of multiple inbound records are not interleaved and in the same order of their corresponding inbound records.

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

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

The split combinator routes inbound records to different instances of its operand depending on the value of a specified tag. The name κ of the tag that determines the instance of the operand is given as parameter to the split combinator. Each instance of the operand stays associated with the combinator and is reused whenever a record is processed that holds the appropriate instance value. [todo: The DSPLIT is too restrictive and has to be defined on streams to refelct the fact that records of different branches can be processed concurrently. Add subscript or different font for N to annotate associated set of instances.]

$$\textsc{Dsplit} \quad : \quad \frac{\text{value}(\kappa, p) = j \quad (p, N_j) \to (N'_j, \vec{q})}{(p, N!!\kappa) \to (N!!\kappa, \vec{q})}$$

$$\textsc{NDsplit} \quad : \quad \frac{\forall_{i=1}^{n} p_i \in \vec{p} : \quad \text{value}(\kappa, p_i) = v_i \quad (p_i, N_{v_i}) \to (N'_{v_i}, \vec{q_i})}{(\vec{p}, N!\kappa) \to (N!\kappa, \text{ndzip}(\vec{q_1}, ..., \vec{q_n}))}$$

4.3 Id, Empty and Map Rule

The following rules are merely 'helper-rules'. The Id rule allows records to be passed on without any alteration. An empty stream has no effect on any component, as shown by the Empty rule. The Map rule allows to derive a semantics for streams if component rules are defined on single records.

$$\textsc{Id} \quad : \quad \overline{(p, \texttt{id}) \to (\texttt{id}, p)}$$

$$\textsc{Empty} \quad : \quad \overline{(\epsilon, M) \to (M, \epsilon)}$$

$$\textsc{Map} \quad : \quad \frac{\forall_{i=1}^{n} : (p_i, M_i) \to (M_{i+1}, \vec{q_i}) \quad \vec{s} = +\!+_{i=1}^{n}(\vec{q_i})}{(\vec{p}, M_1) \to (M_{n+1}, \vec{s})}$$

4.4 Algorithms

eval

eval :: pattern → record → Bool
eval σ p =
...

ismatch

ismatch :: pattern → record → Bool
ismatch σ p =
$p \preccurlyeq \sigma \wedge$ eval σ r

score

score :: pattern → record → $\mathbb{N}_0 \cup \{-1\}$
score σ p =
...

bestmatch

bestmatch :: record → pattern → pattern → set of pattern
bestmatch r σ_l σ_r =
$\{\sigma_i \mid i, j \in \{l, r\} \wedge \text{score } \sigma_i \ r \geq \text{score } \sigma_j \ r\}$

prefix

prefix :: stream → pattern → pattern → (stream, stream)
prefix $\vec{p}\ \sigma_l\ \sigma_r$ =
  case $\vec{p}$ of
    $\epsilon \rightarrow (\epsilon,\epsilon)$
    $p : \vec{ps} \mid$ ndsel (bestmatch $p\ \sigma_l\ \sigma_r$) == $\sigma_l \rightarrow (p{:}\vec{q}, \vec{r})$
    $p : \vec{ps} \mid$ *otherwise* $\rightarrow (\epsilon, \vec{p})$
    where
      $(\vec{q}, \vec{r})$ = prefix $\vec{ps}\ \sigma_l\ \sigma_r$

splitOnePair

splitOnePair :: stream → pattern → pattern → (stream, stream, stream)
splitOnePair $\vec{p}\ \sigma_l\ \sigma_r$ =
  case $\vec{p}$ of
    $\epsilon \rightarrow (\epsilon, \epsilon, \epsilon)$
    $p : \vec{ps} \rightarrow (\vec{q_l}, \vec{q_r}, \vec{r})$
    where
      $(\vec{q_l}, \vec{t})$ = prefix $\vec{p}\ \sigma_l$
      $(\vec{q_r}, \vec{r})$ = prefix $\vec{t}\ \sigma_r$

lrpsplit

lrpsplit :: stream → pattern → pattern → (stream of streams, stream of streams)
lrpsplit $\vec{p}\ \sigma_l\ \sigma_r$ =
  case $\vec{p}$ of
    $\epsilon \rightarrow (\vec{\epsilon},\vec{\epsilon})$
    $p : \vec{ps} \rightarrow (\vec{l_p} : \vec{l_{pl}}, \vec{r_p} : \vec{r_{pl}})$
    where
      $(\vec{l_p}, \vec{r_p}, \vec{q})$ = splitOnePair $\vec{p}\ \sigma_l\ \sigma_r$
      $(\vec{l_{pl}}, \vec{r_{pl}})$ = lrpsplit $\vec{q}\ \sigma_l\ \sigma_r$

lrsplit

lrsplit :: stream → pattern → pattern → (stream, stream)
lrsplit $\vec{p}\ \sigma_l\ \sigma_r$ =
  let $(\vec{p_l}, \vec{p_r})$ = lrpsplit $\vec{p}\ \sigma_l\ \sigma_r$ in
    (flatten $\vec{p_l}$, flatten $\vec{p_r}$)

merge

merge :: record → record → record
record $\vec{p}\ \vec{q}$ =
  $\vec{p} \cup ((\vec{q} \setminus BT(\vec{q})) \setminus (\vec{p} \cap \vec{q}))$

apply

apply :: record → action → stream
apply p α =
...

5 Components and Semantics of the Extended Language

We extend the S-Net core language by two combinators which will be powerful enough to allow for reconfiguration as well as self-adaptation scenarios. As S-Net$^{\Omega}$ is an extension to S-Net, it includes all combinators of the original language. In this paper, all components that are shared between S-Net and S-Net$^{\Omega}$ will be referred to as S-Net components, whereas components that are not part of core language will be referred to as S-Net$^{\Omega}$ components.

5.1 Special Records and Tags

to be included in the final version of the paper

5.2 Network Combinators

The ρ-combinator replaces its current network function if an inbound record contains a compatible replacement function. A network function is considered compatible if it is a subtype of the current function. Any record that does not carry a suitable replacement function is processed by the current network function.

$$\textsc{Rho} \quad : \quad \frac{\begin{array}{c}\exists \mathbb{F}_p \ni f \preccurlyeq M(p,f) \rightarrow (X,\vec{q}) \vee \\ \forall \mathbb{F}_p \ni f \not\preccurlyeq M(p,M) \rightarrow (X,\vec{q})\end{array}}{(p,\rho[M]) \rightarrow (\rho(X),\vec{q})}$$

The μ-combinator constitutes a feedback combinator. Similar to the star combinator, the behaviour of this combinator is influenced by a user-defined pattern. The pattern is matched against inbound records at the output of the network the combinator is applied to. Records that do not match the specified pattern are output. In this case, the combinator has no observable effect. All records that match the pattern are re-inserted to the operand network. As the Mu shows, the feedback path of the combinator has priority over the inbound stream. Records are only read from the inbound stream when all records of the feedback path have been processed. A non-deterministic variant of this combinator, which relaxes this strict behaviour and allows for interleaved feedback and inbound record streams, is currently under investigation and remains as future work.

$$\textsc{Mu} \quad : \quad \frac{(p, M) \rightarrow (M', \vec{q'}) \quad \vec{l}, \vec{r} = \text{lrpsplit}(\vec{q'}, \sigma, \{\}) \quad (\vec{l}, M'\mu\{\sigma\}) \rightarrow (M''\mu\{\sigma\}, \vec{q})}{(p, M\mu\{\sigma\}) \rightarrow (M''\mu\{\sigma\}, \vec{q} +\!+ \vec{q'})}$$

6 Example: Reconfiguration and Adaptation Based on Throughput Monitoring

to be included in the final version of the paper

7 Related Work

to be included in the final version of the paper

8 Conclusion and Future Work

We have presented a coordination language for stream processing that natively supports reconfiguration and restructuring of the network topology and its components. This was achieved by promoting networks to first class citizens of the language. Due to this design, networks may not only be sent across the streaming network, they may also be emitted by components from within networks itself. This further extended our system by adaptation abilities, which render it possible to implement self-modifying networks. The self-adaptation process can be triggered by various sources, as for example an increase or decrease of available computing resources and fluctuating execution times of computational components for inbound data.

Rather than providing reconfiguration and adaptivity facilities implicitly and hiding it from the user, e.g. by implementing these as opaque mechanisms of a runtime system, we designed these as a native part of the language. By extending the core S-Net language by only two combinators, we empower users to explicitly implement reconfiguration and self-adaptation scenarios. A distinct feature of our approach is the possibility to replace networks by new versions without having to specify alternative implementations at deploy or even design time of the original network.

Although our approach enters the domain of self-modifying code, the system guarantees static properties: For any data item that enters the network, it is guaranteed that a path exists for it through the entire network. Replacement of networks is only carried out if a newly received network is of compatible type and thus does not break the specified semantics. The main enabling factor for this is the strict separation of computational languages and the coordination language as this allows for a very concise type system.

What we have presented in this paper is work in progress. A non-deterministic variant of the replacement and feedback operators are under development. The

record-discarding behaviour of synchro-cells at replacement events is not fully satisfactory and may need refinement. Extended semantics for this and a formal definition of the type system for S-Net$^{\Omega}$ remain as future work.