SNet: type-directed generic stream-processing concept

Prof Alex Shafarenko
University of Hertfordshire
U.K.
SNet box

• Single Input – Single Output (SISO) boxes – and networks!
  – output records in response to input records
  – input records with variants

• Type signature:

  priority

  | output options |

  pat1 → pat11, pat12, …
  pat2 → pat21, pat22, …
  ...

where

  pat ::= { field1, field2, … } | {}  
  field ::= id | tag | binding-tag
What is SNet?

- Computational scheme as a network of boxes
- Asynchronous communication, black boxes
- Typed connections, type-directed topology, type is static
- Messages are non-recursive records:
  - Simultaneous (synchronous) state of data items (fields)
  - Not merely, and not necessarily an ADT
- Major role for subtyping and static subtype inference
  - Design flexibility (OOP flavour) + automatic specialization
- Explicit separation of synchronising memory from nonsynchronising (data) memory
- SNet Language:
  - 4 operators (combinators) + 1 memory construct
  - An application is a formula using boxes and combinators
Box behaviour

• Box cycle:

1. Input record
2. Determine which variant (by best-fit pattern matching).
3. Save unmatched (extra) fields if any
4. Produce output records, if any
5. Append the saved fields to each out-record unless identically named fields present already
6. go to 1

N.B. two or more independent variants could be matched at a time:
{a,b,c} matches {a,b} and {a,c}, both will fire
To prevent that, we use binding tags: <X> a,b will only match a pattern with <X> (no flow inheritance), while <x> a,b matches anything that requires <x> a,b or less, e.g. {a}. 

subtyping: “flow inheritance”
Box’s behaviour: example

Signature:

\[
\begin{align*}
\{a, b, c\} & \rightarrow \{d, e\}, \{d\} \\
\{a, b\} & \rightarrow \{<X> a, d\}, \{a, d, g\}, \{w\} \\
\{x\} & \rightarrow \{w\} \\
\{} & \rightarrow \{}
\end{align*}
\]

Input: \{a,r\}, \{a,b,c\}, \{a,b,w\}, \{x\}

Output: \{a,r\}, \{d,e\}, \{d\}, \{w\}, \{<X> a, d, w\}, \{w\}
SISO $\rightarrow$ SISO combinators

A.B

A \rightarrow B

A \rightarrow A \rightarrow \ldots \rightarrow A

A[].B

A \rightarrow A \rightarrow A \rightarrow A

A^*

A!i

○ type-directed splitter

• merger
A splitter is a type-directed multiplexor-replicator. It is **NOT** a combinator in its own right.

**Splitter cycle:**

**Can split more than 2 ways!**

1. input record
2. match A? pass to A
3. match B? pass to B
4. go to 1
Merger

- Standard non-deterministic merger
- In any implementation must ensure fairness
- Could be prioritised if real-time behaviour desirable
Box State - theory

- Boxes do not interact other than by streams
- Hence B: [item] → [item], a pure, lazy function
- Can be factorised:
  \[ B x = foldl \ (.) \ (map \ f \ x) \ init \]
- where \( f: \text{item} \rightarrow \text{s} \rightarrow \text{s} \) is a strict function, \( \text{init}: \text{s} \) some initial data and \( \ (.) \) is a function composition, \( \text{foldl} \) is a lazy folder with some strict output \( \text{out} \):
  \[ \text{foldl} \ \text{op} \ h++t \ \text{init} = (\text{out} \ \text{it})++\text{foldl} \ \text{op} \ t \ \text{item} \]
  where \( \text{item} = \text{op} \ \text{init} \ \text{head} \)

Observe that one can always define

\[ f \text{ item x} = ((\text{first} \ x)++[\text{item}]),(B ((\text{first} \ x)++[\text{item}]) - B (\text{first} \ x))) \]

\[ \text{init} = ([], B []) , \]

but in reality the amount of state carried around is \( O(1) \).
Box State: plumbing

- Implement B using a ‘hard’ zip synchroniser…

- … or even better, without feedback …

\[(s \cdot ((f \cdot sb)) \cdot ob)\]
One stage of *

- Box $f$ is completely stateless as is every other white box; re-enterable, copy-safe, etc.
- The only stateful box is $synch$
- $Synch$ algorithm:
  - Accepts **two** record types
  - Waits for one than the other
  - Outputs a joint record
  - Then goes into overflow mode
  - All records received in overflow go through with an overflow tag.
- $Synch$ is the only **stateful** construct.
Output

• \( B^* = B \ldots B \ldots B \ldots \) so what’s the output?
• The only thing that emerges out of an infinite chain is what does not get processed at all
  – Fixed point: \( B x = x \)
  – Type-directed fixed point \( B: \{\text{Tag}\} \rightarrow \{\text{Tag}\} \) has zero cost
    e.g. if \( x = \{\text{Tag } z,w\} \), the output is \( \{\text{Tag } z,w\} \).
• Multiple fixed points possible
Implementation (feasibility)

- Feedback:
The !i combinator

- A!i is a []-composition of A replicas

- All input patterns of A must include i as a field, which serves as a replica selector. i can be an array.
- Provides for dynamic **homogeneous** connections
Typing A.B

scan the rules in signatures A, B in order;
for each A: \( x \rightarrow y \)
   for every B: \( z \rightarrow w \) produce
   \( x \cup (z\setminus y) \rightarrow w \cup (y\setminus z), \)
   iff \( x \cap (z\setminus y) = \emptyset \)
Then sort rules in topological order and unite left hand sides

Example:
A:
   \{a,c\} \rightarrow \{d\}
   \{a\} \rightarrow \{b\}
B:
   \{b,x\} \rightarrow \{z\}
   \{a\} \rightarrow \{e\}
A.B:
   \{a,c,b,x\} \rightarrow \{z,d\}
   \{a,x\} \rightarrow \{z\}
Multiple output

• A rule can have more than one output pattern
• Split into several rules:

\[
\text{pat1} \rightarrow \text{pat11, pat12, ...} \\
\text{becomes} \\
\text{pat1} \rightarrow \text{pat11} \\
\text{pat1} \rightarrow \text{pat12} \\
... \\
\text{The priority rule need not change since all matching patterns are followed.}
Typing A[]B

1. Compute
   \[ L = \{ v \mid v = x \cup z, A: x \rightarrow y, B: z \rightarrow w \} \]
   and topologically sort it by cardinality of \( v \) in descending order;

2. Scan \( L \). For every \( v \) in \( L \):
   2.1. find \( A: x \rightarrow y \) by scanning down A signature until \( v \) matches \( x \);
   2.2. find \( B: z \rightarrow w \) by scanning down B signature until \( v \) matches \( z \);
   2.3. produce the rule
       \[ v \rightarrow y \cup (v \setminus x), w \cup (v \setminus z) \]

3. copy the rules from signatures A and B, honouring the signature order, but otherwise interleaving arbitrarily.

4. Clean up by removing ineffectual rules.

Example:
\[
\begin{align*}
A: & \quad \{a,c\} \rightarrow \{d\} \\
& \quad \{a\} \rightarrow \{b\} \\
B: & \quad \{b,x\} \rightarrow \{z\} \\
& \quad \{a\} \rightarrow \{e\} \\
A[]B: & \quad \{a,c,b,x\} \rightarrow \{d\}, \{z\} \\
& \quad \{a,b,x\} \rightarrow \{b\}, \{z\} \\
& \quad \{a,c\} \rightarrow \{d\}, \{c,e\} \\
& \quad \{a\} \rightarrow \{b\}, \{e\} \\
& \quad \{b,x\} \rightarrow \{z\} \\
& \quad \{a\} \rightarrow \{b\} \\
& \quad \{a\} \rightarrow \{e\}
\end{align*}
\]
Binding tags

• Using the bar combinator with several boxes leads to combinatorial explosion.
• Similar explosion with the dot combinator.
• To control the explosion use binding tags:
  Subtyping rule: \( x =\{<\text{Tag}> \text{ field1} \ldots \} \) matches pattern \( x \) only if pattern \( x \) contains the same \( <\text{Tag}> \).
  In other words, \( <\text{Tag}> \) cannot be inherited and must be matched.
• Easy to see rule 2 fails if A and B use different tags.
  – indeed \( \{<\text{TagA}>, <\text{TagB}>, \ldots\} \) matches neither A nor B
  – hence only the rules from the signatures A and B come through
• For the dot, modify rule
  1.1. for every B: \( z \rightarrow w \) produce \( x \cup (z \setminus y) \rightarrow w \cup (y \setminus z) \), iff \( x \cap (z \setminus y) \cup BT(z \setminus y) = \emptyset \) and
  for each \( z' \) encountered earlier \( z' \) does not match \( (y \cup z) \)
  where \( BT(a) \) is the set of binding tags in \( a \)
Typing $A^*$

$A^* = A.A.A.A....$

Taking literally, this leads to type situations the programmer would not likely be able to foresee.

Example:

A:

\[
\begin{align*}
\{a\} &\rightarrow \{b\} \\
\{b,d\} &\rightarrow \{c\} \\
\{b\} &\rightarrow \{a\}, \{z\} \\
\{c\} &\rightarrow \{b,d\}, \{w\}
\end{align*}
\]

Intention: generate a stream of type \{z\} records from one or more \{a\} records; generate a stream of type \{w\} records from one or more \{b,d\} records;

Unintended effect: \{a,d\} produces \{w\} records, instead of \{z,d\} records.
Solution for A*

Frontal freeze:
for every input record to A*, take the part that matches an A input pattern (using priority if not unique), and save the rest. Give that part to A* for processing and append the saved portion to the result.

Thus version of inheritance would not interfere with the feedback connections.
Gives the designer of A* a completely predictable environment to set record types.

Type calculation:
Take each l.h.s. pattern of A in turn through the chain A.A.A… until fixed-point is reached.
While doing this keep stripping any records that do not match A, since they will pass through A*; add them to result list straight away.
Also keep record of patterns produced on r.h.s. to detect duplicates.
The fixed point is reached when only duplicates are produced by .A . When this happens
yield the result list.
Example again

A:

\{a\} \rightarrow \{b\}
\{b,d\} \rightarrow \{c\}
\{b\} \rightarrow \{a\}, \{z\}
\{c\} \rightarrow \{b,d\},\{w\}

Signature of A*:

\{a\} \rightarrow \{z\}
\{b,d\} \rightarrow \{w\}
\{b\} \rightarrow \{z\}
\{c\} \rightarrow \{w\}

i.e. as intended.
Pass operator

Things like frontal freeze are useful elsewhere, hence the construct:

\[ X = \texttt{pass \ pat1; pat2…; patN to BOX} \]

Effect: X is a box with the following behaviour:
- input matches any pat1…patN?
  - yes: strip and save excess fields, pass it to BOX, append the excess fields to result
  - no: type error
Object Orientation for Boxes?

• SNet sees it as a design discipline

Subclassing A to $A' = f(A[e]).b$
Customisation

- Three-phase model:
  - compile check syntax/semantics, deduce type sigs.
  - deploy connect, configure and produce binary code
  - run
- deploy time parameters ("compile-time constants") and expressions
  - denote and represent values and expressions over values at deployment
  - have no presence at run time
Type inference results

- Inference of SNet network expressions’ signatures from the signatures of the boxes by direct calculation in Kleene’s algebra (to be published).
- Inference of field types under the constraint that the boxes have *homomorphic type signatures* (Shafarenko,Scholz 2004).
- Hence complete knowledge of the connection topology and channel datatypes before run time, but… also the provision of homogenous (hence type-safe) dynamic connections.
Box arrays

• Need an array version of [] for array processing

```plaintext
[] [] T[ 0 ≤ k ≤ N, 0 ≤ m ≤ M ] box-instance;
```

“deploy time”

Here T is an indexed binding-tag,

```plaintext
box-instance ::= box | case(exp1) box1; case(exp2) box2;… end
```

a box-selection is a choice of a particular box type depending on a deploy-time expression. This provides for heterogeneous arrays.

`[] []` creates an instance of box-selection for each multi-index and combines them all with a `[]`.

`[] []` expects a single pattern `{<T[x, y]>}`, which it converts to `{<T>, k=x, m=y}` and then passes them down to the `(x, y)` box-instance

an indexed tag is matched by tag name and index values, not index names!

e.g. `<T[1,3]>` matches `<T[k,m]>` if k=1 and m=3,
but also any `<T[n,r]>` if n=1 and r=3
Typical use

- Stream processing in signal processing & multimedia
- Evolution computing
  - state of the computation is distributed between an array of boxes
  - each box advances the state by one step based on its own state and the state of
    - neighbouring boxes (regular problems)
    - distant boxes (irregular problems)
  - The new state becomes the current state when all boxes complete
  - There is an optional output of the solution from each step
  - Examples abound: CFD, plasma simulation, computational biology, quantum chemistry, grand-challenge problems, etc.
SNet Template for E.C.

solution =
    decompose .
    (  
        [[]] Tag[ 0 ≤ k ≤ K, 0 ≤ m ≤ M ]
        case (k*m=0 | k=K | m=M) borderBox
            else regularBox
    )*
    . recompose . visualize
contd.

regularBox

\[
\begin{align*}
\{\text{<initcond>, k, m, data}\} & \rightarrow \{\text{T[k,m]}, \text{data}\}, \\
\{\text{T}, k, m, \text{exchange}\} & \rightarrow \text{nil} \\
\{\text{T}, k, m, \text{data}\} & \rightarrow \{\text{T[k,m]}, \text{data}\}, \\
& \quad \{\text{T[ ]}, \text{exchange}\}, \\
& \quad \{\text{output} \ k, m, \text{step}, \text{summary}\} 
\end{align*}
\]

borderBox

\[
\begin{align*}
\{\text{<initcond>, k, m, conditions}\} & \rightarrow \text{nil} \\
\{\text{T}, k, m, \text{data}\} & \rightarrow \{\text{T[ ]}, \text{exchange}\} 
\end{align*}
\]
Synchronisation

• A matching store approach:

  \[
  \text{match[size]} <T1>, <T2> \text{ on Field1, Field2 , … , FieldN}
  \]
  \[
  \text{yielding} <T3> \text{ overflow} <T4>;
  \]

Function: match records

\[
\{<T1>, \text{ Field1, Field2 , … , FieldN, other-fields1}\}
\]
\[
\{<T2>, \text{ Field1, Field2 , … , FieldN, other-fields2}\}
\]

on values of Field1, Field2 , … , FieldN. Inherit other fields, to produce:

\[
\{<T3>, \text{ Field1, Field2 , … , FieldN, other-fields1, other-fields2 }\}
\]

Problem: Requires multiple inheritance. Solution: specify other-fields1
in match.
Conclusions

- SISO box/network paradigm is feature-rich and nontrivial
- Record subtyping is very potent, capable of capturing hierarchies and topologies
- Type inference through 4 combinators
- A form of OOP
- A form of functional programming
- A form of distributed parallel programming
- Have hand-coded examples for evolution computations
- Need “real” implementation