Semantics and type theory of S-Net
—DRAFT—

Alex Shafarenko, Clemens Grelck and Sven-Bodo Scholz

Department of Computer Science, University of Hertfordshire, AL10 9AB, U.K.

Abstract. The paper briefly introduces the language S-Net and discusses in detail its concept of type and subtyping. Novel subtyping features are described and analysied: tag-controlled record subtyping and flow inheritance. A simple semantic formalism is proposed and the semantics of S-Net closure defined. We prove that type inference for S-Net is computationally efficient provided that the resulting type structures are of reasonable size.

1 Introduction

Our other paper presented concurrently at this conference introduces the language S-Net, its functionality and design principles. This current paper focuses on the issues of semantics and type theory for the language. We begin with a brief reminder of the relevant features of S-Net.

The language S-Net is a coordination language for defining the connectivity and interaction between stream-transforming black boxes combined into networks using combinators.

Boxes A *box* is a piece of software (or hardware) with a Single Input and a Single Output (SISO), which operates as follows. The box is connected to an input stream, from which it reads one record, processes it and produces some output records, after which it terminates ready for the next record. The *atomic* boxes are written in a *box language* of which S-Net has no knowledge. Atomic boxes respond to a single record sort[1], produce zero or more records of one or more statically known sorts. The atomic boxes have no persistent state; after processing a record, the box can be assumed to have no memory of previous operation, hence a clone of an atomic box behaves the same as itself.

Combinators SISO boxes and networks are combined into SISO networks using combinators, of which there are only four.

1. A **serial combinator** connects two boxes in a pipelined fashion. The input of the first box becomes the input of the result, the output of the second box becomes the output of the result. The other input/output pair are connected together.

[1] we shall define record types precisely in the following sections; here we use the term 'sort' loosely to refer to some type information

2. A **choice combinator** connects two boxes in parallel so that the common input of the resulting network is spit between the boxes (according to their input sort) and the two outputs are merged. There are versions of the choice operator for a deterministic and a nondeterministic merger.
3. A **closure combinator** (conceptually) makes an infinite chain of replicas of the operand box and connects them using the series combinator. The only stream members that escape the chain are those corresponding to the fixed point of the operand box. S-Net relies on static fix points, that do not depend on data values (those are external to S-Net, anyway), which are determined solely by sort relationships. Those are inferred by using sort inference algorithms discussed here.
4. The **index splitter** is an infinite version of the choice combinator similar to the closure combinator being an infinite version of the series combinator. Conceptually, an infinite number of replicas of a single box are connected in parallel, as if by using the choice combinator, but the specific box is selected on the basis of some index rather than sort. In fact, the index value is present in the type system as an S-Net as well as the box language artefact. There is a version of the index splitter with both deterministic and nondeterministic merger at the back, just like the situation with the splitter.

Data type concept Boxes are connected by streams that carry data items. Each item is a nonrecursive record consisting of named fields. A field is atomic to S-Net, although field names could be used to express nested reocrds, but not recursive record types: all records in S-Net are essentially flat. Another important consideration is that records are field *sets* not field lists in that two records that only differ in the order of their fields are considered identical. For the sake of simplicity, we will not focus on field types in this talk, assuming that they are either known (encoded in the field name, for instance) or dynamic. Therefore we define the record type as a set of field names. In addition we support special fields that carry no data, call them *tags*. In fact a record has one or more possible *variants*, which are sets of field-names/tags, so the record type can be considered a set of variants. The variants must be pair-wise distinct, but they have no names, unlike fields, and so can only be distinguished by their field sets, although tags can always be included to make disambiguation easier. The reason for such a data concept is that we wish to promote subtyping, which will be discussed in the next section. The form of subtyping we use is quite standard: we form subtypes by adding fields to a variant or dropping variants from the record type. Sections 1.1 to 1.7 below focus on type issues in detail.

Semantics Our approach to semantics is based on the fact that all S-Net primary boxes are in fact pure functions from stream to stream (that applies to both boxes written in the box language and synchrocells) , so their semantics is best described by a stream transforming function. Since the combinators have the ability to arbitrarily interleave streams (the choice and index splitter have a nondeteministic version each), nondeterminism needs to be taken into account. We achieve that by defining an event set by which the semantic function is indexed. The semantics of the combinators is expressed in terms of the result semantic function and the result semantic event set. We focus on semantics in section **??** and subsequent section.

Related work Due to the lack of space we shall mention only briefly some key papers that support S-Net structural and type-theoretical concepts. The concept of streaming networks and some basic ideas of stream processing can be found in [1–6] The work on streaming networks pertaining to semantics has been reported in [7–11]. Type theoretical approaches relevant to this project can be found in [12, 13].

We acknowledge support from the EU Advanced Computer Architecture initiative under the Integrated Project AETHER.

1.1 Record types

The type system of S-Net supports nonrecursive variant records with *record subtyping*. As formally defined in Fig. 1, a *record type* in S-Net is a possibly empty set of anonymous *variants*. Each variant again is a possibly empty set of named *record fields*. We distinguish two different kinds of record fields: *value fields* and *tags*. A value field is characterised by its *field name* and, as the name suggests, is associated with some value at runtime. This value, however, is opaque to S-Net and may only be generated, inspected or manipulated by using an appropriate box language. Likewise, a tag carries a name, but is associated with an integer value, that is visible to both: the box language code and S-Net.

$$
\begin{array}{lll}
\mathit{Type} & \Rightarrow & \mathit{TypeName} \mid \left\{ \left[\mathit{Variant} \left[\;,\; \mathit{Variant} \right]^* \right] \right\} \\
\mathit{Variant} & \Rightarrow & \mathit{VariantName} \mid \left\{ \left[\mathit{Field} \left[\;,\; \mathit{Field} \right]^* \right] \right\} \\
\mathit{Field} & \Rightarrow & \mathit{FieldName} \mid \texttt{<} \; \mathit{TagName} \; \texttt{>} \\
\mathit{TypeDef} & \Rightarrow & \textbf{type} \; \mathit{TypeName} \; \texttt{:=} \; \mathit{Type} \; \texttt{;} \\
\mathit{VariantDef} & \Rightarrow & \textbf{variant} \; \mathit{VariantName} \; \texttt{:=} \; \mathit{Variant} \; \texttt{;}
\end{array}
$$

Fig. 1. Syntax definition of S-Net types and type definitions

S-Net supports non-recursive abstractions on types. Using the key word `type` an identifier may be bound to a type specification. As an example for a record type

```
type body := { {<Triangle>, col, x1, y1, dx2, dy2, dx3, dy3},
               {<rectangle>, col, x1, y1, dx2, dy2},
               {<circle>, col, x1, y1, r} };
```

[1] The non-terminal symbols *TypeName*, *FieldName* and *TagName* uniformly refer to identifiers. We only distinguish them here for illustration.

defines a type `body` for the representation of geometric bodies, which are either triangles, rectangles or circles. Each body has a color (`col`), coordinates of a reference point (`x1` and `y1`) and varying numbers of further coordinates, e.g. edge lengths of a rectangle (`dx2` and `dy2`) or the radius (`r`) of a circle.

Likewise, we may define abstractions on variants using the key word `variant`. For example the above type definition may equivalently be written as

```
variant triangle  := {<Triangle>, col, x1, y1, dx2, dy2, dx3, dy3};
variant rectangle := {<rectangle>, col, x1, y1, dx2, dy2};
variant circle    := {<circle>, col, x1, y1, r};
type body         := { triangle, rectangle, circle};
```

We distinguish two different kinds of tags: *binding tags* and *non-binding tags*. Whereas the names of the former start with a capital letter, the names of the latter start with a small letter. So, in the above example `Triangle` in fact is a binding tag while `rectangle` and `circle` are non-binding tags. Binding tags and non-binding tags behave differently with respect to record subtyping, as defined in the following section.

1.2 Record subtyping

As mentioned earlier, S-Net supports subtyping on record types. Record subtyping is based essentially on the subset relationship between collections of record fields.

Definition 1 (record subtyping).
Record subtyping is defined by the following rules:

1. *Let $BT(x)$ denote the set of binding tags in a variant x.*
2. *A variant v_1 is a subtype of a variant v_2, $v_1 \sqsubseteq v_2$, if*

$$v_1 \supseteq v_2 \wedge BT(v_1) = BT(v_2)\,.$$

3. *A record type t_1 is a subtype of a record type t_2, $t_1 \sqsubseteq t_2$, if*

$$(\forall v_1 \in t_1 \exists v_2 \in t_2) v_1 \sqsubseteq v_2\,.$$

With the above definition of record subtyping, the purpose of binding tags becomes apparent: They provide a means to explicitly control record subtyping. One record variant can only be a subtype of another if the two have the same set of binding tags. In contrast, non-binding tags with respect to record subtyping behave just like ordinary value fields. For instance, the variant

```
{<Triangle>, <colored>, x1, y1, dx2, dy2, dx3, dy3, color}
```

is a subtype of the variant tagged by `<Triangle>` of the definition of type `body` above. It contains exactly the same binding tags (`<Triangle>`) and all other record fields of the previous variant plus an additional non-binding tag `colored`. Note that the sequence of fields in the definition is irrelevant as variants are effectively sets of fields. Here is a subtype of the type `body` defined above:

```
type body1 := { {<circle>, color, x1, y1, r, shading},
                {<rectangle>, color, x1, y1, dx2, dy2, shading} }
```

Each variant is in subtype relationship to the corresponding variant of the supertype `body` identified by the same binding tag. To both variants we have added an additional value field `shading`.

With respect to record subtyping there are two special record types in S-NET: $\bot$ or $\{\}$, the record type with no variants, and $\top$ or $\{\{\}\}$, the record type with a single variant having no fields. Any type t is a supertype of $\bot$: $(\forall t)\bot \sqsubseteq t$. Furthermore, any type t free from binding tags is a subtype of $\top$: $(\forall t)BT(t) = \emptyset \implies t \sqsubseteq \top$.

1.3 Record type coercion

Now let us consider the issue of *record type coercion*. Coercion is achieved by throwing away the fields that are absent in the target type. This potentially causes an ambiguity whenever there is more than one suitable variant in the target type and, consequently, a choice of fields to dispose of. The above type `body` is a subtype of the type `anchored` defined as follows:

```
type anchored := { {<Triangle>, color},
                   {x1, y1},
                   {x1, y1, dx2, dy2} }
```

Indeed, each of the three variants of the (sub-)type `body` defined before can be shortened to one of the variants of (super-)type `anchored`. However, due to the special role of binding tags (here `<Triangle>`), the first variant of type `body` can only be coerced to the first variant of type `anchored`. In contrast, the `<rectangle>` variant of `body` can be coerced to either the second or the third variants of `anchored`. We resolve this ambiguity by defining coercion to always take the most specific candidate variant, i.e., if a variant v may be coerced to variants v_1 or v_2 and $v_1 \sqsubseteq v_2$, then v is effectively coerced to v_1. Hence, in the above example the `<rectangle>` variant of `body` is coerced to the third variant of `anchored` while `<circle>` is coerced to the second variant.

Still some of the ambiguity remains in that there can be two mutually incoercible targets for a given variant. For instance, in the type `confused`, defined as

```
type confused := { {x1,y1},
                   {dx2,dy2} }
```

there is a choice of variant to use for a `<rectangle>`. 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

$$\forall v, w \in \tau : BT(v) = BT(w) \implies v \cup w \in \tau\,.$$

$$\textit{TypeSignature} \Rightarrow \Big\{\ \textit{Mapping}\ \Big[\ \texttt{,}\ \ \textit{Mapping}\ \Big]^*\ \Big\}$$

$$\textit{Mapping} \quad \Rightarrow \Big[\textit{Variant}\Big]\ \texttt{->}\ \ \textit{Type}$$

Fig. 2. Grammar for S-Net type signatures

As in the definition of record subtyping above, $BT(x)$ denotes the set of binding tags of a variant 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. Note that all single-variant types are automatically complete.

1.4 Type signatures

Now, we are ready to define the concept of a *type signature*, i.e. the type associated with an S-Net box. Essentially, a type signature consists of an input record type and an output record type. The input record type $T_{in} = \bigcup_{i=1}^{n}\{v_i\}$ with variants v_i specifies the records a box accepts for processing. The output record type $T_{out} = \bigcup_{i=1}^{n} \tau_i$ is in fact a collection of types τ_i with each τ_i being the type of records potentially created in response to receiving a record of variant v_i. In a conventional programming language would look very much like $T_{in} \to T_{out}$, but instead we use the following syntax to provide a subtyping structure:

$$\Sigma = v^{[1]} \to \tau^{[1]};\ v^{[2]} \to \tau^{[2]};\ldots v^{[n]} \to \tau^{[n]}\,,$$

where each $v^{[i]}$ is an input variant specification and each $\tau^{[i]}$ is a full type specifications of the output:

$$v^{[i]} = \{\phi_0^{[i]},\ldots,\phi_{m_i}^{[i]}\};\ \ \tau^{[i]} = \{V_1^{[i]},\ldots,V_{k_i}^{[i]}\};\ V_j^{[i]} = \{\Phi_{j,0}^{[i]},\ldots,\Phi_{j,r_j}^{[i]}\}\,,$$

with ϕ and Φ representing fields and V variants. When a box B is given an input stream x of type T, every record of x is coerced up from type T to the input type T_{in} of B as described above. The concrete S-Net syntax for type specifications is given in Fig. 2.

Variant records in conventional languages have named variants and, hence, identification of an individual variant is by its names. In contrast, records in S-Net have anonymous variants containing named fields. Thus, variant identification is done primarily by analysis of the field set. As a consequence, input types of type signatures must be complete to ensure proper variant identification. Completeness may, for example, be achieved by the use of binding tags, which effectively mimick the named variants of conventional languages.

Whilst the conventional approach completely avoids the variant ambiguity, it also precludes subtyping on the basis of field names only. For instance, in our example of type `body`, conventional subtyping would preclude the construction of a generic box that alters the body position expressed in terms of fields `x1` and `y1` that are common to all variants. As a result a box would require variants to be identified individually and specifically, even when the processing of fields `x1` and `y1` is the same for all variants, for instance, a coordinate shift `x1->x1+a`, `y1->y1+b`. The conventional OOP approach to this would be via a base class with fields `x1` and `y1` and a method `shift` to be inherited by all subclasses. This restricts the design in that there can be more than one set of common fields (which would require multiple inheritance), but more importantly since the significance of a common group of fields may become apparent only when an entirely new processing box is introduced into a streaming network, and in that case a re-design of the class hierarchy may become necessary.

Subtyping by subsetting as introduced in the beginning of this section does allow *a-posteriori* introduction of a supertype (equivalent to a base class), which obviates the re-design. The price to pay in implementation is the price of a run-time coercion (i.e. a selective copy of fields, or an extra level of indirection to avoid the need to copy), since it can no longer be assumed that the fields to be processed are necessarily a prefix of the field list.

1.5 Flow inheritence

Streaming networks promote pipelining whereby a record travels along a chain of boxes that apply various processing algorithms to its content. Since a box can legally be fed with a subtype of the input type, this would result in the loss of all fields that are not required by the input type, but these fields could possibly be required by another box further down the pipeline. To remedy that, the following type rule is introduced:

Definition 3 (flow inheritance).
Let $v^{[i]} \to \tau^{[i]}$, $i \in [1,\dots,n]$, be the type signature of a box X. Furthermore, let each output type $\tau^{[i]}$ have m_i variants $\tau^{[i]} = \{w_1^{[i]},\dots,w_{m_i}^{[i]}\}$. Then for any $k \le n$ 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*

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

and

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

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

Informally, an input variant can be extended with a new field ϕ (which can be a non-binding tag but not a binding tag), 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[2].

[2] Obviously, an implementation is free to simply switch references.

If the output already contains an identically named field, then that field's value supersedes the inherited one. For convenience, we shall write box signatures in the form $(n, m)v^{[i]} \to w_j^{[i]}$, which signifies a box with input variants $v^{[i]}$ and the corresponding output types $\tau^{[i]} = \{w_1^{[i]}, \ldots, w_{m_i}^{[i]}\}$, $i \in \{1, \ldots, n\}$. Note that n is a scalar integer that describes the number of input variants, whereas m denotes a vector of n integers describing the number of variants in each output type associated with one input variant.

Flow inheritance creates a subtyping hierarchy for boxes. For example, a box that accepts records with a single field named x and which produces records with a single field name y is a supertype of a box that accepts $\{x, z\}$ and returns $\{y, z\}$. As a side effect, flow inheritance can be a source of redundancy in type signatures. Indeed, in the above example if the signature of the *same* box contains the rules $\{x\} \to \{y\}$ and $\{x, a\} \to \{y, a\}$, then clearly the second rule can be deleted without changing the effective box type. Value-wise, the second rule carries additional information, namely that a record $\{x, a\}$ if presented to the input, will cause a record $\{y, a\}$ to appear with a potentially *different value* of a, while, assuming that b does not occur anywhere in the signature, if $\{x, b\}$ is presented at the input it would cause the output of $\{y, b\}$ with the output value of b being exactly the same as its input value. Still, as far as types are concerned, we can always assume that the signature is nonredundant, since the redundant rules change nothing in the type transformation defined by it.

1.6 Box subtyping

Other forms of subtyping come from the conventional subtyping rules for a function:

$$\frac{f : \tau_1 \to \tau_2,\ \tau_1 \sqsubseteq \tau_1'\ \tau_2' \sqsubseteq \tau_2}{f : \tau_1' \to \tau_2'}$$

and our concept of records that allows a subtype to have fewer variants and more fields in each variant. Accordingly, we state four subtyping rules. The rules may violate the topological order of the left-hand sides as fields and variants are inserted at arbitrary positions. To restore the order we use the topological permutation T_S defined on any set of variants S=$\{v_i\}$ as a permutation of the index range $[1, |S|]$ such that $T_S(j)$ enumerates the indices of the variants $v_{T_S(j)}$ in topological order as j traverses the index range in ascending order. Here is the summary of the subtyping rules:

Definition 4 (box subtyping).
Let box X have the type signature $(n, m)v^{[i]} \to w_j^{[i]}$. Then for any $k \leq n$, the following are subtypes of X:

input field: *the type $(n, m)V^{[T_V(i)]} \to W_j^{[T_V(i)]}$, where for some field name $\phi \in v^{[k]}$*

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

provided that $\phi \neq v^{[k]} \setminus v^{[l]}$ for all $l > k$; otherwise, for any $l > k$ such that $\phi = v^{[k]} \setminus v^{[l]}$

$$V^{[i]} = \begin{cases} v^{[i]} & \text{if } i \neq k \wedge i < l, \\ v^{[k]} \setminus \{\phi\} & \text{if } i = k, \\ v^{[i-1]} & \text{if } i \geq l \end{cases}, \quad W_j^{[i]} = \begin{cases} w_j^{[i]} & \text{if } i \neq k \wedge i < l, \\ w_j^{[k]} \cup w_j^{[l]} & \text{if } i = k, \\ w_j^{[i-1]} & \text{if } i \geq l \end{cases},$$

input variant: *for any variant $\pi \notin \{v^{[i]}\}$, the type $(n+1, M)V^{[i]} \rightarrow W_j^{[i]}$, where*

$$V^{[i]} = \begin{cases} v^{[i]} & \text{if } i < k, \\ v^{[i-1]} & \text{if } i > k, \\ \pi & \text{otherwise} \end{cases},$$

$$W_j^{[i]} = \begin{cases} w_j^{[i]} & \text{if } i < k, \\ w_j^{[i-1]} & \text{if } i > k, \\ \tau & \text{otherwise} \end{cases},$$

provided that $(\forall i > k) v^{[i]} \not\sqsubseteq \pi$ and τ is such that for all i for which $\pi \sqsubseteq v^{[i]}$, the relation $\tau \sqsubseteq \{w_j^{[i]} \cup (\pi \setminus v^{[i]})\}$ holds as well[3] Here

$$M^{[i]} = \begin{cases} m^{[i]} & \text{if } i < k, \\ m^{[i-1]} & \text{if } i > k, \\ \mu & \text{otherwise} \end{cases},$$

and μ is the number of variants in τ.

output field: $(n, m)v^{[i]} \rightarrow W_j^{[i]}$, *where for all $j \leq n$, $r \leq m^{[j]}$, some $l \leq m^{[k]}$ and a field name ϕ*

$$W_r^{[j]} = \begin{cases} w_r^{[j]} & \text{if } j \neq k \text{ or } r \neq l, \\ w_l^{[k]} \cup \{\phi\} & \text{otherwise;} \end{cases}$$

output variant: $(n, M)v^{[i]} \rightarrow W_j^{[i]}$, *where for some $l \leq m^{[k]}$*

$$W_j^{[i]} = \begin{cases} w_j^{[i]} & \text{if } i \neq k, \\ w_j^{[k]} & \text{if } i = k \text{ and } j < l, \\ w_{j+1}^{[k]} & \text{if } i = k \text{ and } l \leq j \leq m^{[k]} - 1 \end{cases},$$

and

$$M^{[i]} = \begin{cases} m^{[i]} & \text{if } i \neq k, \\ m^{[k]} - 1 & \text{otherwise} \end{cases},$$

[3] Note that this rule accounts for a newly introduced variant having extra fields over an existing one, so that these fields would have been flow inherited given the original type.

flow inheritance: *for any field name $\phi \notin v^{[k]}$, $(n,m)V^{[i]} \to W_j^{[i]}$, where*

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

and

$$W_j^{[i]} = \begin{cases} w_j^{[i]} & \text{if } i \neq k, \\ w_j^{[k]} \cup \{\phi\} & \text{if } i = k \text{ and } j \leq m^{[k]} \end{cases},$$

provided that $V^{[i]}$ is in topological order.

The above subtyping rules are general and consequently quite complex, although their meaning and application in most situations would be straightforward. Still two problems with subtyping remain at this point. Firstly, suppose that when a record of type v is sent to a box, the box responds with a certain output type τ; if the record is of a subtype $v' \sqsubseteq v$, the output type τ' can be completely unrelated to τ, even though the intention could have been to just use the fields common with v and ignore any fields in $v' \setminus v$. One could argue that the type signature of the box is quite clear about what the response is to any given type, and so if the additional fields are 'caught' by one of the alternatives, this would be deliberate and the user of the box would know about it. However, the second-order version of this problem causes a serious difficulty: in a network of boxes, how does the type signature of the network change if a box is replaced by its subtype? The answer potentially depends on every box in the network and cannot be abstracted easily. Even if we could obtain it, the 'second-order' signature of the network with respect to one participating box would, in general, be quite unwieldy, as it would have the type signature of the box in question as parameters. It is unlikely that such a device would be practical. To avoid problems of this kind, we constrain all box signatures to be monotonic, a property that we illustrate in the following.

1.7 Monotonicity

Informally, monotonicity means that one can use a subtype in place of a supertype and still assume that the output type is the same or coercible to the same. When there are more than one possible supertypes at the input, e.g. when the variant $\{a, b\}$ is present alongside the variants $\{a\}$ and $\{b\}$, the output type in response to each supertype must be included. In other words, a type signature is monotonic provided that for each input variant $v^{[i]}$ that is a subtype of some other input variant $v^{[j]}$, its associated output type $\tau^{[i]}$ is also a subtype of the output type $\tau^{[j]}$.

Definition 5 (monotonicity).
The type signature $(n,m)v^{[i]} \to \tau^{[i]}$ is considered monotonic iff

$$(\forall i \in \{1, \dots, n\})\, \tau^{[i]} \sqsubseteq \bigcup_{j \in \sigma v_i} \tau^{[j]}$$

where σv_i denotes the set of indices j of all supertypes $v_j \sqsupseteq v_i$ of v_i in the given type signature.

There is of course no guarantee of value consistency. For instance, a monotonic type signature that takes a single-field record x to a single-field record y can catch $\{x, z\}$ and produce a different value y as well as further fields in the output record. This, however, is not a problem since the input record with field z carries more information which can be expected to affect the output value. The only thing that monotonicity guarantees is that the field y will not disappear merely because one has additionally supplied z at the input.

Monotonicity appears to be a useful property, but it does not come without a price. Consider an output type τ as a response to input v. If $v' \sqsubseteq v$ causes the box to yield output of type $\tau' \sqsubseteq \tau$, it follows that τ' cannot have variants essentially different from those that τ is made up of, in particular, one cannot introduce a nonempty variant that has no common fields with any variant of τ. However, imagine that the processing of v' sometimes raises certain exceptions that never arise when processing v, and so a variant is required to encode those. Then τ must include that variant (or a subset thereof) even though it will never be used at run-time as a response to v. Adding a variant to τ, would raise the box type, so such an alteration may cause a complete re-design of the network. Hence, some account should be taken of possible extensions already when designing the initial version of a box, which is undesirable as it prevents extensibility of the network. The solution is in exploiting the multiplicity of supertypes. One could, for example, add a rule such as $\langle x \rangle \to \tau''$ to the box signature and then include tag $\langle x \rangle$ into v'. Then τ' would be allowed to "inherit" any variants from τ'' and extend them as appropriate. Direct use of the $\langle x \rangle$ input can be guarded against by including a unique binding tag into one of the variants of τ'' which is not used by τ'. If the environment supplies $\langle x \rangle$, then it will not be able to match the unique tag appearing at the output and the resulting type error will alert the user. Such schemes could get as complex and secure as necessary and desirable, and the basic type infrastructure of S-Net will provide the required type guarantee.

It is interesting to note that flow inheritance is itself a form of monotonic subtyping. Indeed, it adds the same field to the input record and to each of the output records, thus replacing every record by its subtype. It is therefore obvious that if a signature is monotonic, applying flow inheritance to it will keep it monotonic. The same is true of subtyping by input variant (see above); the rest of the subtyping rules: input/output field and the output variant should be further constrained by the condition that the resulting signature is monotonic. It is possible to state such constraints explicitly as a restriction on the choice of ϕ, and where appropriate output τ, but since the modifications required are straightforward we shall leave them out to save space.

2 Semantics

Define the alphabet of a box $x : (n, m)v^{[i]} \to w_j^{[i]}$ as

$$\aleph(x) = \bigcup_{i=1}^{n} \left(v^{[i]} \cup \bigcup_{j=1}^{m_i} w_j^{[i]} \right) ,$$

and let Φ^∞ denote the (infinite) set of all possible field names. The semantics of a box `x`, i.e. its action on any record $v \in V^0 = \aleph(x) \to D$, where D is the set of all possible field values, can be defined as a semantic function $\hat{x} : V^0 \overset{?}{\to} \alpha(V^0)$, where $\alpha(s)$ is the set of all sequences composed of members of s. We make $\hat{x}$ total by including into V^0 a special null record (not to be confused with the empty record $\emptyset \to D$, which is the empty set of fields) ϵ, $V = V^0 \cup \epsilon$ and redefining $\hat{x} : V \to \alpha V$. Note that $\hat{x}$ fully reflects record subtyping and flow inheritance, using the rules we have defined earlier. This function represents "raw" semantics, which is what S-Net sees when a box is operating in its environment. To be precise, $\hat{x}$ is not necessarily a function; it is generally speaking a family of functions from which one is selected by nondeterministic choice, when the default action is taken in the absence of a specific subtype, as discussed in section **??**. To account for the nondeterminism we add an index to the semantic function $\hat{x}$. A representative of the family $\hat{x}_q$ is a function that corresponds to a particular choice $q \in E(x)$ in nondeterministic variant matching. We will refer to set $E(x)$ as the *event set* of box `x`, which is the set representing all available nondeterministic choices of the box. We assume all event sets to be finite and to contain elements of arbitrary nature. Those sets resulting from input variant matching are finite by construction; other event sets occur in the behavioural characteristics of combinators, which we shall discuss below. Those are also finite, albeit potentially large, sets.

Next we incorporate the infinite part of the field-name variety by generalising the domain V to $V^\infty = \Phi^\infty \to (D \cup \{\epsilon\})$, which results in the following semantic function $\bar{x}_q : V^\infty \to \alpha(V^\infty)$:

$$\bar{x}_q\, z = \mathrm{map}\,(\chi\, z_2)(\hat{x}_q\, z_1)\,,$$

where $z = z_1 \cup z_2$, $z_1 \in V$, $z_2 \in V^\infty \setminus V$, and

$$\chi\, a\, b = \begin{cases} \epsilon & \text{if}\, b = \epsilon \\ \mathrm{map}\ (a\cup)\ b & \text{otherwise}\,. \end{cases}$$

Here `map`, as usual, applies its first argument to every member of the sequence represented by the second argument. The reader will recognise in the above formula the flow inheritance rule for fields that do not occur in the box signature. Note that since such fields do not cause additional nondeterminism, the event set remains the same as with $\hat{x}$.

Finally, semantic functions apply to a record as an argument, implying that the response of a box to an individual record does not depend on anything else (for a given nondeterministic choice). This is true for primitive S-NET boxes, but not for S-NET networks. The latter generally contain synchronisers and parallel combinators, whose output depends on the current as well as some previous records. The box semantics remains purely functional, except it is now a function from a set of *sequences* of records onto itself, which is the third form of semantic function (after $\hat{x}$ and $\bar{x}$) that we intend to use. For a primitive box x for which $\bar{x}$ is available, the third form is:

$$\check{x}_q = (\odot/) \circ (\mathrm{map}\, \bar{x}_q)\,.$$

Here $\odot$ is a sequence concatenation operator, and $\odot/$ is applied to a sequence of sequences to concatenate it into a single sequence. Note that while the first and second form are fully equivalent, the third form is not generally reducible to them: given a third form semantic function, there may not exist a first/second from semantic function that defines it in terms of the above equation. Consequently, in defining the semantics of combinators we must employ exclusively the third form.

As a final observation, consider $\check{x}_q$ for an arbitrary network. It is easy to see that if a_1 is a prefix of a, i.e. there exists a b such that $a = a_1 \odot b$, then also $\check{x}_q\, a_1$ is a prefix of $\check{x}_q\, a$, i.e. $\check{x}_q$ is prefix-monotonic. Indeed, when a_1 has been received and responded to, the input sequence can continue to reach a but the output corresponding to a_1 has already been made. Prefix-monotonicity is analogous to causality in concurrency theory. Note, however, that this property only holds as long as $\check{x}_q$ is taken at the same q for different input prefices, and is immediately destroyed by nondeterminism.

Now we are ready for the discussion of S-NET operators.

3 Serial Combinator

Consider two networks $a : (n, m)v^{[i]} \to w_j^{[i]}$ and $A : (N, M)V^{[i]} \to W_j^{[i]}$. The serial combinator `a..A` produces a network that responds to an incoming record ρ by putting it through network a first, and then feeding the output of a to network A. The output of A becomes the output of `a..A` . Let us define the formal semantics of `a..A` . Formally it is defined thus:

Definition 6 (serial combinator). *The serial combination* `S = a..A` *of networks* `a` *and* `A` *is a network whose behaviour is represented by the semantic family*

$$\check{S}_r = \check{A}_{q'} \circ \check{a}_{q''}\,,$$

where $q' \in E(A)$, $q'' \in E(a)$, $E(S) = E(a) \times E(A)$, and $r = (q', q'') \in E(S)$.

To determine the type signature of $S = a..A$, one needs to establish the minimum set of fields that ρ must have to be accepted by S. There can be more than one such set, each corresponding to an input variant. The acceptance of a record can be determined on the basis of which fields are required by a and which additional fields are required to be flow inherited through a by its output record, so that A can accept that record. This results in the following type transformation, which we define in two stages.

First, introduce lexicographic flattening of the type signature whereby a single index k is introduced instead of i and j: $a :: (\nu)v^{[k]} \to w^{[k]}$, the double colon indicates that flattening has taken place. The new index k enumerates index pairs (i, j) in lexicographic order. For instance if there are two input variants producing three and four output variants, respectively, (i.e. $n = 2$, $m^{[1]} = 3$, $m^{[2]} = 4$) the correspondence between k, i and j is as follows:

k	1 2 3 4 5 6 7
i	1 1 1 2 2 2 2
j	1 2 3 1 2 3 4

Obviously $\nu = \sum_{i=1}^{n} m^{[i]}$, and the input variants $v^{[k]}$ are no longer pairwise distinct. Note that the flattened form of the signature contains exactly the same information as the standard form, and hence the transformation is reversible. The process of reversal consists in scanning the signature in the ascending order of k, noting the multiplicity of each $v^{[k]}$ and reconstructing $m^{[i]}$, n and $w_j^{[i]}$. Also note that the consistency rule that requires the signature to be sorted in a topological order of $v^{[i]}$ applies to $v^{[k]}$ just as much. The enumeration of $w_j^{[i]}$ in j has been arbitrary so far; in a flattened signature we demand that $w_j^{[i]}$ is topologically sorted in j in *increasing* order, i.e. for any i if $w_a^{[i]} \subseteq w_b^{[i]}$ then their indices in the flattened signature k_a and k_b must satisfy $k_a \leq k_b$ (in a way opposite to the sorting of $v^{[i]}$ in i). The reason for this arrangement will be given momentarily.

Now consider the flattened signatures $a :: (n)v^{[i]} \to w^{[i]}$ and $A :: (N)V^{[i]} \to W^{[i]}$. Define a (set-valued) $n \times N$ deficiency matrix

$$D_{ij} = V^{[j]} \setminus w^{[i]},$$

and a dual to it, but independent, excess matrix

$$X_{ij} = w^{[i]} \setminus V^{[j]}.$$

Each element of D_{ij} contains the set of fields that need to be flow inherited through a when the input matches variant $v^{[i]}$. The inheritance is only possible when none of the fields in the set is present in $v^{[i]}$. Provided that this condition is satisfied, an input record of type $v^{[i]} \cup D_{ij}$ is taken through both networks, resulting in the output type $X_{ij} \cup W^{[J]}$. The excess matrix defines the additional "baggage" due to the excess fields which will be flow inherited through network A. Now we can define the whole type transformation for the `..` operator:

$$\text{a..A} :: (|R|)\Theta(R),$$

where

$$R = \{v^{[i]} \cup (V^{[j]} \setminus w^{[i]}) \to (w^{[i]} \setminus V^{[j]}) \cup W^{[j]} \mid (V^{[j]} \setminus w^{[i]}) \cap v^{[i]} = \emptyset\}, \qquad (1)$$

and $\Theta(X)$ is an indexed sequence of members $p \to q$ of set X sorted in a topological order of first p (decreasing) and then q (increasing). The set R is required to be nonempty; otherwise a type error is produced. Figure 3 depicts the set-theoretical relations between inputs and outputs as defined by Eq 1.

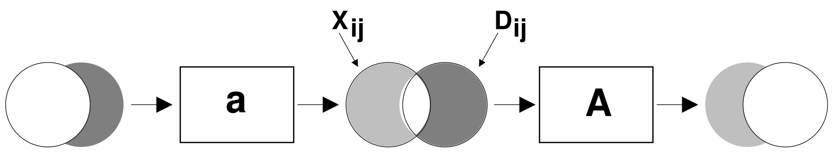

Fig. 3. Flow inheritance through the `..` combinator

Now recall that the right-hand sides of the arrow in a flattened signature are sorted in an increasing topological order. Upon inspection of Eq. 1 it is immediately evident that since $v[i]$ is sorted in a decreasing and $w^{[i]}$ in increasing order, the elements of set R are already sorted in the decreasing order of left-hand sides at any fixed j or at any fixed i. For similar reasons there is increasing order of the right-hand sides at any fixed j (or, again, i). Hence the sorting that Θ is required to do can be made computationally quite efficient by employing merge-sort. The choice between indices i and j as a basis for merge-sort could depend on the overall length n vs N, which one provides the greater multiplicity, etc.

Finally observe that there is a potential for every output variant of `a` to be combined with an input variant of A to produce an overall type transformation. This may or may not be the intention of the network designer in connecting the networks in series. It is quite reasonable to expect that certain output variants of `a` are meant to correspond to perhaps only a single input variant of A. In general it is desirable to be able control the connection between networks when they are combined in series. This is achieved using already familiar binding tags. In their presence, Eq 1 is modified thus:

$$R = \{v^{[i]} \cup (V^{[j]} \setminus w^{[i]}) \to (w^{[i]} \setminus V^{[j]}) \cup W^{[j]} \mid (V^{[j]} \setminus w^{[i]}) \cap (v^{[i]} \cup B) = \emptyset\}, \quad (2)$$

where B is the set of all possible binding tags. The size of set R can be made as small as required by judicially placing binding tags in the output of `a` and the input of `A`.

4 Closure

The closure combinator is denoted by the postfix asterisk. The result of its application is a network produced by infinite replication of the operand network with the replicas connected serially:

$$B..B..B..\ \ldots$$

The output from the infinite chain occurs at finite distances from its beginning, when a record falls within the fixed point of B.

First of all, we give the formal semantics. To start with, we define a *closure over a set* which is the core formalisation of the above description.

Definition 7 (closure over a set). *The closure of a network* `B` *over a set* F *is a network* $\mathtt{B}^\circ$*, whose semantic functions* $\check{B}^\circ_q$ *is as follows. First define a recurrence relation for an auxiliary third-form semantic function* $c^{[k]}_q$*. For any record sequence* x*:*

$$\begin{aligned} c^{[0]}\,x &= x;\;\; E(c^{[0]}) = \emptyset \\ c^{[k+1]}_q\,x &= \check{B}_{q'}\left(c^{[k]}_{q''}\,x\right),\; \textit{where}\; q = (q', q'') \\ E(c^{[k+1]}) &= E(c^{[k]}) \times E(\check{B})\,. \end{aligned}$$

Using the above, the closure of `B` *over a set* F *is*

$$\check{B}^\circ{}_q = c^{[i_\infty]}_q\,, \tag{3}$$

where $i_\infty = \max_q i_q$*, and* $i_q = \min\{i \mid c^{[i]}_q \in F\}$*. The event index* r *ranges over* $E(\check{B}^\circ) = E(c^{[i_\infty]})$*.*

The closure over a set gives the semantics of a network chain in which a record sequence propagating along the chain is guaranteed to have fallen inside a certain set F along the way before it is extracted, irrespective of the nondeterministic choice. Now let $F(\check{B})$ be a set of sequences that go through the network `B` unchanged. In other words, F is a set of fixed points of the network `B`, i.e. solutions of the equation $(\forall q \in E(\check{B}))\check{B}_q\,x = x$. It is easy to see that the closure of `B` over $F(\check{B})$ corresponds to the natural intuition of the replica chain introduced at the beginning of the current section. Indeed a sequence of records which at some point reached a fixed point cannot change by going through the network anymore, hence can be "teleported" through the whole infinite chain to the output.

There is of course no guarantee that i_∞, which is the position on the chain at which it is guaranteed that the fixed point is reached irrespective of nondeterminism, is finite, hence there is a possibility of a nonterminating closure. Also, the fixed-point set F cannot be produced algorithmically from the algorithm of $\check{B}$ in the general case, hence the only general solution involves comparing the input and output sequences of each `B` replica in a chain to determine whether or not the fixed point has been reached. Even if it were practical, the fixed point observation for a given record sequence would need to have been done for every member of the large event set (which is selected by the environment with repetitions at run time) before a fixed point can be found. This makes the number of observations hard to limit *a priori*.

In search of a rememdy, let us take a closer look at the recurrent process in Definition 7. It is easy to see that if for some $i < i_\infty$, and some q, some $a \in F$

is a prefix of $c_q^{[i]}$, then the eventual fixed point, if it is ever reached, will have a as a prefix, too, thanks to the prefix-monotonicity of semantic functions, which was noted earlier. Indeed, since a fixed point is not sensitive to nondeterminism, $c_q^{[i]}$ will have a as a prefix of the output even though q varies with i. This means that it is possible to output a out of the chain even *before* the fixed point is reached[4]. In particular, a single-record fixed point can be dispatched immediately to the output without accumulating sequences if it is possible to establish that it always goes through the network B unchanged (if only followed by further output records). Unfortunately, the nondeterminism of B means that even after such a behaviour has been detected once, testing must continue in case any subsequent replica behaves differently.

A solution lies in the type system. Consider a part of the fixed-point set $T \subseteq F$ whose members are single-record sequences that have no value-bearing fields (while they may, and in most cases will, have tags). It is easy to see that such records are not prone to nondeterminism in B since the record type fully determines the record value. Due to flow inheritance, a record whose field-name set v matches the rule $t \to t$ for some $t \in T$, belongs to F. Consequently, any value bearing fields in v are flow-inherited, and thus left unchanged; this is true irrespective of the nondeterministic choice, since the value-bearing fields bypass the closure network completely. As a result, a sequence comprising a single record $r = v \to D$ is statically guaranteed to be a member of F. Let us denote the set of all sequences composed of records such as r as T^+. We can now introduce the following

Definition 8 (hard closure). *A hard closure of a box* `B` *is a network* `B*` *whose semantic function $\check{B}^*{}_q$ is the closure of the box B over the set T^+.*

The use of hard closure to define the effect of B^* is tantamount to allowing all records to propagate along the chain of replicas until each of them matches one of the fixed-point type rules, at which point the sequence can be assumed to have traversed the whole infinite chain.

Definition 8 serves as a formal basis of the closure combinator in S-NET and exhausts the issue of semantics. Next we must define the type of B^* given the type of B.

First let us introduce an equivalent form of the box type signature. For a box $B : (n, m)v^{[i]} \to w_j^{[i]}$ introduce a function $\phi : Q \times \mathbb{N} \to Q$, where $Q = \mathcal{P}(V \cup W) \cup \{\omega\}$, $V = \bigcup_{i=1}^{n} v^{[i]}$ and $W = \bigcup_{i=1}^{n} \bigcup_{j=1}^{m_i} w_j^{[i]}$. Here ω is a special symbol that signifies invalid type, Q the set of all field names used in the type signature and $\mathbb{N}$ is the set of natural numbers up to the maximum number of variants in any output. The function ϕ applied to a record (understood as a set of field-names) and a number k produces the output field-set corresponding to the kth variant of the output type in response to the given input type variant

[4] this corresponds to "laziness" in functional semantics

as per the type signature of B with flow inheritance taken into account:

$$\phi(x,k) = \begin{cases} \omega & \text{if } \mu(x)=0 \vee k > m_{\mu(x)} \vee x = \omega \\ (x \setminus v^{[\mu(x)]}) \cup w_k^{[\mu(x)]}, & \text{otherwise} \end{cases} .$$

Here $\mu(x)$ is the index of the rule that matches x, or 0, if no match can be found. Now denote the mapping of the type signature $\Sigma = (n,m)v^{[i]} \to w_j^{[i]}$ onto its functional representation $\phi : Q \times \mathbb{N} \to Q$ as $\Psi(\Sigma)$.

Proposition 1. *Ψ is bijective modulo the variant and type orderings that are neutral to the type transformation defined by Σ.*

The proof is constructive. First create a signature $\Sigma^0 = (n,m)v^{[i]} \to w_j^{[i]}$, where $n = 2^{|A(Q)|}$, $v_i = T_{i,\subseteq}\mathcal{P}(A(Q))$, $m_i = max_j\,(\{j \mid \phi(v_i,j) \neq \omega\} \cup \{0\})$ and $w_j^{[i]} = \phi(v_i,j)$ for all $j \leq m_i$. Here $T_{i,\subset}X$ is the ith member of set X in some topological order of $\subseteq$. Next we do a series of deletions from Σ^0. First find all v^i for which $m_i = 0$ and delete the corresponding rules from the signature. Then for any pair of rules $v^{[i_1]} \to w_j^{[i_1]}$ and $v^{[i_2]} \to w_j^{[i_2]}$ such that $v^{[i_1]} \subset v^{[i_2]}$, $m_{i_1} = m_{i_2}$ and $\forall j_2 \exists j_1 w_{j_2}^{[i_2]} = w_{j_1}^{[i_1]} \cup (v^{[i_2]} \setminus v^{[i_2]})$, delete the second rule. It is clear that the first series of deletions removes all rules that denote the response to a type error, hence they were not in the original signature. The second series of deletions removed the rules that could be produced from other rules by flow inheritance. Since ϕ was produced from the type signature by making type mismatch and flow inheritance explicit, it is straightforward that the resulting signature must be the same as the original one, up to the ordering of the rules in a different topological order, and the arbitrary ordering of the variants in output types. Since we do not distinguish between type signatures that only differ in those two orderings, the proposition is proven.

Next we define the serial operator on functions $Q \times \mathbb{N} \to Q$.

Definition 9. *Consider two functions $\phi_{1,2} : Q \times \mathbb{N} \to Q$. For any $v \in Q$, let $\sigma_v(\phi_1,\phi_2)$ denote the lexicographically ordered series of all pairs $(n_1,n_2) \in \mathbb{N} \times \mathbb{N}$ that satisfy the condition*

$$\phi_2(\phi_1(v,n_1),n_2) \neq \omega\,,$$

and $\sigma_v^n(\phi_1,\phi_2)$ the nth member of the series. Then the serial combination $\phi_1..\phi_2$ is a function $\phi_s : Q \times \mathbb{N} \to Q$ defined thus:

$$\phi_s(v,n) = \phi_2(\phi_1(v,n_1^{[n]}),n_2^{[n]})\,,$$

where $(n_1^{[n]},n_2^{[n]}) = \sigma_v^n(\phi_1,\phi_2)$, or if n exceeds the length of the sequence then $(n_1,n_2) = (N,N)$ where N is a large enough number so that $(\forall x \in Q)\phi_{1,2}(x,N) = \omega$.

Now we can strengthen Proposition 1 to the following

Proposition 2. *Ψ is an isomorphism between the algebras $(\Sigma, ..)$ and $(Q \times \mathbb{N} \to Q, ..)$ modulo the variant and type orderings that are neutral to the type transformation defined by Σ.*

The proof is obtained by comparing Eq 1 with Definition 9. The serial combination of type functions is similar, but not identical to the semantic set of the serial combinator. The former describes the *possible* types that are produced in response to an input record type, whereas the latter describes the actual record values produced in response to a given record value. The algebra $(Q \times \mathbb{N} \to Q, ..)$ is in fact a semigroup:

Proposition 3. *The operation .. as defined by Definition 9 is associative.*

To prove this, we must prove that for all $\phi_{1\text{-}3} : Q \times \mathbb{N} \to Q$, $(\phi_1..\phi_2)..\phi_3 = \phi_1..(\phi_2..\phi_3)$. By applying both sides to some record record v and number n we obtain:

$$\phi_3(\phi_2(\phi_1(v, n_1^L), n_2^L), n_3^L) = \phi_3(\phi_2(\phi_1(v, n_1^R), n_2^R), n_3^R)\,,$$

where on the left hand side $(m, n_3^L) = \sigma_v^n(\phi_1..\phi_2, \phi_3)$, and $(n_1^L, n_2^L) = \sigma_v^m(\phi_1, \phi_2)$. Clearly as n increases, so does first n_3^L as far as possible on the first σ-list, then m will start to increase. As m increases, it causes n_2^L to increase first as far as possible according to the second σ-list, then n_1^L will begin to increase. We conclude that, as n increases, it enumerates triplets (n_1^L, n_2^L, n_3^L) in lexicographic order.

On the right-hand side, $(n_1^R, k) = \sigma_v^n(\phi_1, \phi_2)$; $(n_2^R, n_3^R) = \sigma^k_{\phi_1(v,n_1)}(\phi_1, \phi_2..\phi_3)$. Here similarly, as n increases, first k will rise according to the first σ-list, and so first n_3^R and then n_2^R will rise on the second sigma list, and finally n_1 according to the first σ-list. We conclude that as n increases, it enumerates triplets (n_1^R, n_2^R, n_3^R) in lexicographic order.

Finally, it is easy to see that the left-hand side and the right hand side are each a list (indexed by n) of all non-ω values of $\phi_3(\phi_2(\phi_1(v, n_1), n_2), n_3)$ for a given v and any $n_{1\text{-}3}$, and since we have shown that these lists are sorted in the same way with regard to triplets (n_1, n_2, n_3), they are equal.†

Now let us return to the issue of type. Since we are in interested in hard closure $B*$, let us define the projector box for $B : v^{[i]} \to \tau^{[i]}$ as $B^{\to} : v^{[i]} \to \tau_*^{[i]}$, where $\tau_*^{[i]} = v^{[i]}$ if $v^{[i]}$ matches a member of set T and $\emptyset$ otherwise, where T, as before, contains non-value bearing records from the B fixed-point. The projector box disposes of any input records that do not match the fixed point and passes through those that do.

Observe that

$$B* \equiv \underbrace{B..B, \ldots, ..B}_{L\,\text{times}}..B^{\to} \tag{4}$$

for sufficiently large L Here $\equiv$ denotes the equality of type signatures. Indeed, once a certain power (with respect to the .. operator) of B yields a record that will be captured by the projector box, any further application of B is ineffectual, hence the signature for $L+1$ must be a superset of the signature for L. On the other hand, since the alphabet of the box B is finite, there is only a finite variety of rules to include into $B*$ and a finite capacity not to produce relevant rules for a number of iterations. The latter stems from the finiteness of the whole signature (both relevant and irrelevant parts). Consequently a finite chain must exist that captures the whole type transformation of $B*$.

The length of the chain, albeit finite, is hard to limit. One can construct examples where rules collude to transform a record from one type to the next a large number of times until types start to repeat (and hence a whole variety of rules become irrelevant to the fixed point). We have not been able to obtain chain length bounds weaker than exponential in the signature size, which is unsatisfactory for practical purposes.

There is, however, a way to bound the complexity of the type calculation once we take into consideration the fact that in any practical network the size of the type signature of the *result* would be expected to be small. Indeed, it is likely that a large type formula is caused by a design error when an unintended match occurs between some input and output types. Such errors can always be prevented by employing binding tags, but only at the expense of flexibility. Next we will show that the complexity of type calculation is linear in the size of the resulting signature and will propose an appropriate algorithm.

Recall that Propositions 2 and 3 establish associativity of the serial combinator viewed as a type constructor. Let us change the evaluation order in Eq 4 to achieve back chaining, i.e.

$$B^{[0]} = B^{\rightarrow};\ B^{[n+1]} = B..B^{[n]}$$

The advantage of the back chaining is that at each iteration a subset of the eventual type signature is produced. Most importantly though, each iteration must yield at least one new type rule for the process to continue. Indeed, if for some n, $B^{[n+1]} \equiv B^{[n]}$, then

$$B^{[n+2]} \equiv B..B^{[n+1]} \equiv B..B^{[n]} \equiv B^{[n+1]} \equiv B^{[n]}\ ,$$

and so $B^* = B^{[n]}$. We conclude that the number of iterations does not exceed the size of the resulting signature, which gives the aforementioned linear complexity bound. Finally observe that the type calculation process can be limited to a reasonable number of output rules, say one hundred: signatures of this size are so unwieldy that they are unlikely to be the result of a deliberate design. Upon reaching the critical size the algorithm could produce appropriate diagnostics (a listing of the rules obtained thus far) and abort.

Conclusions

This paper presents an overview of type-theoretical and semantic issues of the language S-NET. We have defined the type system of the language and described the semantics of the most important combinators. We have shown that the closure combinator of S-NET is amenable to type inference with complexity proportional to the size of the result type signature.

This paper reports work in progress on the EU project AETHER. The immediate next steps include implementing the S-Net compiler using the type inference algorithms reported here. The semantic definitions presented above will be complemented by the rest of S-Net semantics and a model implementation will be achieved within the term of the AETHER project.

References

1. Berry, G., Gonthier., G.: The esterel synchronous programming language: Design, semantics, implementation. Science of Computer Programming **19** (1992) 87–152
2. 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
3. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data-flow programming language LUSTRE. Proceedings of the IEEE **79** (1991) 1305–1320
4. A.Shafarenko.: Retran: a recurrent paradigm for data-parallel computing. Computer Systems Science and Engineering **11** (1996) 201–209
5. Stephens, R.: A survey of stream processing. Acta Informatica **34** (1997) 491–541
6. 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)
7. 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
8. 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
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. 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
12. Shafarenko, A.: Coercion as homomorphism: type inference in a system with subtyping and overloading. In: PPDP '02: Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming. (2002) 14–25
13. Shafarenko, A., Scholz, S.B.: General homomorphic overloading. In: Implementation and Application of Functional Languages. 16th International Workshop, IFL 2004, Lübeck, Germany, September 2004. Revised Selected Papers. LNCS 3474, Springer Verlag (2004) 195–210