

Report on S-Net
A Typed Stream Processing Language

— **Part I** —

Foundations, Record Types and Networks

Alex Shafarenko and Clemens Greck

University of Hertfordshire
Department of Computer Science
Hatfield, Herts, AL10 9AB
United Kingdom

August 25, 2006

— **DRAFT** —

Abstract

We propose a view on a data-processing application as a typed streaming network. The arcs of the network represent record-valued data streams and the nodes encapsulate recurrence relations on them. We propose a type system in which both the arcs and the nodes are statically subtyped, with the overall subtyping consistency of the network assured by type reconciliation algorithms. The proposed type system makes extensive use of a homomorphically-restricted subtyping, which, on the one hand, provides for generic node specification and, on the other, supports efficient type inference and type reconciliation.

Chapter 1

Introduction

1.1 Background and Motivation

Component technology is crucial to implementing large systems on chip (SoCs). Indeed the complexity of on-chip solutions can only be overcome by decomposition and abstraction. The former requires application-specific building blocks, the latter demands that formal interfaces be set up between the blocks and the connecting infrastructure. This, of course, is not unique to systems on chip; any object-orientated technology would be based on similar ideas. What is typical of the SoCs is the static nature of the connection between the blocks and the increased role of provable properties in assuring the required functionality of the whole system. One way of viewing a static component network is associated with the concept of stream processing.

1.2 Proposed Approach

This paper proposes a component technology for stream processing, which we have named S-NET. An S-NET is a recursively nested single-input, single-output (SISO) streaming network that combines SISO processing boxes and coordinates their interaction. Processing boxes are atomic: they do not expose any internal content and, hence, are completely “black”. These atomic boxes are entirely stateless and strictly operate in an input-process-output work cycle. Upon receiving a data item on its input stream an atomic box produces none, one or more data items on its output stream. The functional properties of atomic boxes enable them to be deployed cheaply and moved and replicated at will, without giving rise to data integrity concerns.

S-NET is in fact a coordination language: It provides means to describe the orderly behaviour among boxes and the streaming network used for communication between boxes. Atomic boxes are implemented externally using an appropriate *box language*. Functional languages are particularly suitable for this purpose as they inherently adhere to the restrictions imposed by the interface.

Nevertheless, imperative box languages may be used as well, but require some discipline by the programmer.

Atomic boxes communicate with each other and with the execution environment solely by means of data received and sent via typed streams. S-NET allows atomic boxes to be composed into SISO networks, which recursively form composite boxes in further network layers. Composition of boxes involves splitting and merging communication streams depending on their types. It is described using *network combinators*, that are inspired by Stefanescu’s network algebra [1].

The restriction to a single input and a single output stream allows us to use variant types to capture data provenance under a type system. This makes the network topology a type issue alongside all the standard type issues, and opens up an avenue for comprehensive subtyping, which is what usually makes a component technology so effective in the object-oriented world. Still, the multiplicity of input and output streams as often found in stream processing can easily be mimicked: one may think of an SISO entity as one with all the input streams interleaved into a single stream and all the output streams similarly de-interleaved.

S-NET networks are asynchronous: an entity’s output is assumed to be buffered. When processing is done by several components whose results must be combined, a synchronisation facility is generally required. It is introduced in the form of a SISO synchrocell, which is the only kind of “stateful” box in an S-NET. A synchrocell expects records of several types to appear at its input; it combines them into a joint record and outputs the result. The internal state of a synchro-cell is made up by the records waiting to be synchronised. Note that synchro-cells, though “stateful”, have no computation to perform, whereas atomic boxes have no state, but can compute.

Finally, we propose genericity and specialisation mechanisms on the basis of static record subtyping. These mechanisms make it possible to statically optimise streaming networks with generic components. They also enable the component designer to provide several versions of a box depending on a subtype. Crucially, S-NET does not require explicit subtype declarations; a subtype inference algorithm is applied to determine the most appropriate subtype.

1.3 Record Types

Data items sent via streams are organised as non-recursive, tagged variant records with arbitrary non-record fields. Consequently, the types associated with streams in an S-NET network are non-recursive, tagged variant record types. Elementary types are effectively opaque to S-NET. Since all actual data is produced and consumed by box language programs, only the box language code knows about how to interpret the data. As far as S-NET is concerned atomic record fields and the corresponding data travelling along the streams are as opaque as the atomic boxes themselves.

Tagged variant records allow a single record to alternatively store different fields. For instance, a geometric body type can include a sphere record with

the fields *centre* and *radius*, as well as an ellipsoid record with the *centre* and three axes, and a cone with a *centre*, *hight* and an *apex* angle. A box may be capable of processing all these shapes, in which case a union type is required. To distinguish different members of the union type, which we shall call *variants* hereinafter, S-NET uses pattern-matching and tags.

Recursive record types are not supported in S-NET for the following reasons. A nonrecursive record is a mere collection of fields accessible at once, in no particular order. The fields may themselves be records, so in fact a non recursive record can be thought of as a finite tree, whose leaves are named by the (unique) path from root to leaf. It is important to understand that these path names are static and so all leaves are accessible at once in no particular order¹.

By contrast, a recursive record type can be thought of as a set of nonrecursive records in which some fields represent cross-references, and where each record has a special statically-unknown label for use in cross referencing. The data structure as a whole is characterised by (partial) access order, so it cannot be accessed at once, but rather one group of (nonrecursive) records at a time by following references. When a recursive data structure is to be communicated, it is common to stream only relevant parts of it, under the control of a client-server protocol, rather than the whole data structure at once. Even when the latter is unavoidable, such data structures are not send in their natural form, but rather in a serialised, ‘marshalled’ stream, which is a stream of nonrecursive records with abstract label fields.

1.4 Subtyping

Subtyping is an important adaptation mechanism of a component technology. An S-NET box may be capable of processing more variants than there are in the incoming typed stream, which should not cause trouble. Likewise, if a box receives a valid variant extended with additional fields, these fields should simply be ignored and passed on to the output so that a further box may process them. These commonsense considerations provide the motivation for subtyping. The first two of them constitute conventional record subtyping, whereas the third one is, to our knowledge, a new concept, which we call *flow inheritance*. It is fundamental to S-NET and is used extensively.

Record subtyping is a somewhat controversial issue. On the one hand, it is already part of the mainstream, which is demonstrated by its full adoption by the language Python [2]. Python is strongly (though dynamically) typed, but its strong typing is due to software engineering concerns rather than the pursuit of efficiency.

Object-oriented languages have a more restrictive concept of subtyping whereby fields are sequentially ordered and only the tail fields can be ignored to produce a supertype. The motivation here is to preserve static field offset and thus to efficiently compile field access. S-NET does not restrict subtyping this way, and

¹Since there are no operations on whole records in S-NET, the subrecords are not important, hence it is assumed that all nonrecursive records are flat.

could pay the penalty of up to a single additional reference per field. The penalty would have been payable even without the liberal subtyping, since we accept that fields can have statically unknown size, which is the case with array processing. Still, with the added reference the record structure is fully static, due to the static topology of S-NET networks, which ensures that the type relationship between the producer and the consumer is resolved at compile time².

As records are consumed and produced by S-NET boxes, the boxes themselves must have properties with respect to record subtyping. In particular, since a subset of variants constitutes a subtype, it is useful to know the box reaction to each variant, rather than to the whole type that consists of them. This knowledge can be used to statically determine a lower output type when a lower type is offered to the input type, which is a form of box specialisation. The type signatures of S-NET boxes are formulated accordingly: they list an output type versus an input variant, which we call a *detailed* type signature. To give an example of a similar phenomenon (albeit not subtyping as such) consider type `List` as defined normally with two variants, `Nil` and `Cons`. For a function `List` \rightarrow `List`, such as `map`, the type information available from its semantics is more detailed than the above signature. Assuming *empty* and *nonempty* being subtypes of `List`, the signature of `map` is, in fact,

$$\text{map} : \text{empty} \rightarrow \text{empty}, \text{nonempty} \rightarrow \text{nonempty}$$

A function does not have to respond with a single-variant type to each variant, so the most general detailed signature is in the form $\{v \rightarrow \tau\}$, where v denotes a variant, τ a type (i.e. a set of variants) and the braces denote a set. For instance, a function `tail` : `List` \rightarrow `List` which returns `Nil` when applied to an empty list, has a detailed signature

$$\text{tail} : \text{nonempty} \rightarrow \{\text{empty}, \text{nonempty}\}, \text{empty} \rightarrow \text{empty}.$$

Programming languages tend to treat such properties as dynamic (by only accepting the general signature such as `List` \rightarrow `List`), while S-NET allows them to be statically verified when a detailed signature is present whether explicitly, or implicitly, by the provision of several implementation modules for the same box, differentiated by subtype.

1.5 Memory

It was mentioned before that S-NET processing boxes are stateless. They produce zero, one or more records in response to a single record at the input in a receive-process-send cycle. A cycle has no memory of the previous cycles. However the input and output records can have common fields. These are called ‘recurrent variables’ and could be used for holding persistent data if at least

²To be precise, S-NET has a dynamic connectivity mechanism (the `!` combinator); however, the dynamic connection is always with a member of a type-homogeneous box collection. Hence, the type relationship between the producer and consumer is always statically known.

part of the output is diverted back to input. To function as a box ‘state’, the recurrent variables must be synchronised with any input that the box receives, which is one of the many situations that involve S-NET *synchro-cells*. In this example a synchro-cell is introduced in a feedback loop of a processing box, but they are equally useful for ‘zipping’ two or more box outputs into a single stream, for matching pairs of records on the basis of common index, etc. However, even in the simple case of recurrent variables, the general solution with a synchro-cell allows for box multi-threading (several synchro-cells in the feedback loop) and process farming (several cells with several boxes) — all purely by network configuration without touching the ins or outs of the participating boxes. It is also crucial to S-NET that subtyping allows such configurations to be typed and checked automatically, and that generic boxes continue to be specialised correctly as the network topology changes.

It should be noted that the S-NET categorisation of memory as synchro-cells outside, and data memory inside, boxes clearly separates two memory aspects which are otherwise combined in conventional programming: memory as work storage for computations and memory as a means of inter-process communication. It is that separation that promotes flexible specification, which assists generic parallel and distributed computing.

Chapter 2

Related Work

2.1 Stream Processing

The concept of stream processing has a long history. The view of a program as a set of processing blocks connected by a static network of channels goes back at least as far as Kahn’s seminal work [3] and the language Lucid [4]. Kahn introduced the model of infinite-capacity, deterministic process network and proved that it had properties useful for parallel processing. Lucid was apparently the first language to introduce the basic idea of a block that transforms input sequences into output ones. A variable would represent such a sequence, acting as a stream of values of that variable in time. Ordinary operators in Lucid acted on variables point-wise, by effectively synchronising streams and applying the operation across pairs of corresponding stream elements. Additionally there were also some “temporal” operators, which were intended for altering the order of elements in a sequence.

Somewhat later, in the 80s, a whole host of synchronous dataflow languages sprouted, notably the languages Lustre [5] and Esterel[6], which introduced explicit recurrence relations over streams and further developed the concept of synchronous networks. These languages are still being used for programming reactive systems and signal processing algorithms today, including industrial applications such as the recent Airbus flight control system and various other aerospace applications [7]. The authors of Lustre broadened their work towards what they termed synchronous Kahn’s networks[8, 9], i.e functional programs where the connection between functions, although expressed as lists, is in fact ‘listless’: as soon as a list element is produced, the consumer of the list is ready to process it, so that there is no queue and no memory management is required.

A nonfunctional interpretation of Kahn’s networks is also receiving attention, the latest stream processing language of this category being, to the best of our knowledge, the MIT’s StreamIt [10]. The latest comprehensive survey of stream processing and the underlying theory for it can be found in [11]. There is also a growing activity in *database stream processing* [12] , which concerns

itself with the problem of responding to a database query "on the fly", using the same limited-memory, sliding-window view of processing blocks that started with Lucid and continued through the aforementioned stream-processing languages. Still, despite much work having been done in various niche areas, stream processing has yet to be recognised as a general-purpose paradigm in the same sense as, for instance, object-oriented or functional programming.

Around the time that Lustre was introduced, David Turner[13] remarked that streams could be used as software glue for complex parallel software systems, even operating systems. In his interpretation, streams were lazy lists, which were produced on demand for their consumers. The lists were seen as an interface between the deterministic parts of a parallel system, which were pure stream-processing functions¹, and the external interleavers/mergers that realise the inter-process communication and capture its nondeterministic behaviour.

This arrangement is sketched in fig 2.1. Note that each processing box has a single input and a single output. This does not lead to a loss of generality due to the fact that a function requiring multiple input streams can be represented as a function of a single stream argument where the elements of the multiple streams are somehow merged into a single sequence of records. Similarly, a single output stream can be split into any given number of secondary output streams by picking out records for each of the output sequences. The issue of how exactly the inputs are merged is a delicate one; an efficient solution would depend on the properties of the function in question. The merging usually benefits from being nondeterministic, as this accommodates the delays incurred in receiving the contributing streams by allowing the first message that arrives to be passed on to the processing function without waiting for its turn. On the other hand, the processing block can be required to be deterministic, in which case it may not be ready to accept a given input at an arbitrary time.

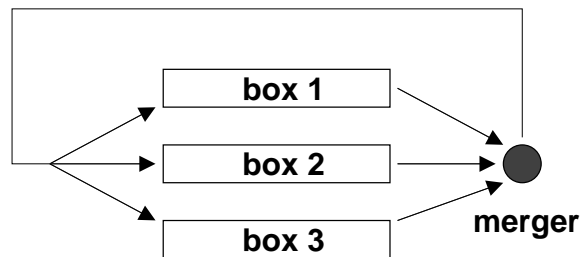


Figure 2.1: The Turner scheme

Note that a merged stream has no overall order: only records belonging to a single tributary stream have a precedence relation defined on them. To allow that order to be recovered from the merged stream, the provenance information can be preserved by, for example, tagging the ordered records by the same tag.

¹but they could have been any self-contained procedures rather than pure functions, as long as the only access they had to each other's state was via stream communication

Overall, the Turner scheme seems very attractive as it neatly separates the computational aspect of stream processing from the communication aspect; it confines non-determinism to the part of the system where no value processing takes place (since merging, filtering and splitting only re-package streams without computing new values of basic types); and it uniformly represents an application as a set of interconnected, side-effect-free, single-input, single-output stream functions. The only quality that it seems to lack is satisfactory support for modularity. The problem is that streams in complex systems tend to be record-based, and the processing functions expect a certain set of fields to be present in the records. Moreover, rather than streams having a single record layout, variant records are often required, so that a number of different algorithms can be carried out by a single block. In addition, certain “control” records can be used for exception handling, load balancing, etc. The boxes can be usefully *extended* by adding more variants and passing the unused fields downstream to further, perhaps newly inserted, boxes which provide additional functionality. Those are examples of network structuring, subtyping and inheritance that one would expect to find in a practical stream-processing paradigm.

Besides these pragmatic considerations, we must mention here equally important theoretical advances in streaming networks. The key work in this area appears to have been done by Stefanescu, who has developed several semantic models for streaming networks starting from flowcharts [14] and recently including models for nondeterministic stream processing developed collaboratively with Broy [1]. This work aims to provide an algebraic language for denotational semantics of stream processing and as such is not focused on pragmatic issues. It nevertheless offers important structuring primitives, which are used as the basis for a network algebra (see [15]). It is interesting to note that apparently the StreamIt team [10] as well as ourselves [16] were unaware of those and had to re-invent them, albeit for purely pragmatic reasons.

2.2 Subtyping

The issue of type inference with atomic subtyping has a long history, too. We cite papers [17, 18, 19, 20, 21] as ones where foundation work was done. Of these, paper [21] is probably the most relevant as it tackles the issue of decidability of general type inference in the presence of subtyping, but it does not bound its cost. The main thrust of our work is towards homomorphism of types and effective constraint-satisfaction algorithms that make type inference possible. This issue was not approached systematically until a simplified theory was given in [22]. Our concept of type homomorphism is consonant to Lievant’s idea of “discrete polymorphism” proposed in [23], where it was suggested that overloadings should be treated as models of a single theory. We believe that h-overloading is less restrictive as it allows higher instances to “expand” the functionality of the lower ones without destroying the consistency between them.

Technically, the most relevant to our work could be the paper by Rehof and Mogensen [24], where a method is described for what they termed a “definite

constraint satisfaction problem”. Here all constraints are presented in a form similar to ours: $v_0 \geq f(v_1, \dots, v_k)$, where f is a nondecreasing function. Then an algorithm is presented, with a complexity linear in the number of constraints, (i.e. quadratic in the number of variables n) which finds the least solution. The main difference is that in [24] the system of constraints is assumed to be *closed*, i.e. all variables are subject to type minimisation within the constraints. In the present paper, we approach a more general problem of constraint satisfaction with unknown external parameters, which are types of the external variables that are *not* subject to minimisation. In our case, the solution is a function of those types. The algorithm from [24] does not apply to such situations. We have proposed a slightly more costly solution, with the cost $O(n^3)$, but which allows external types to be parameters in the type assignment.

Chapter 3

Types and Subtyping

3.1 Record Types

The type system of S-NET supports nonrecursive variant records with *record subtyping*. As formally defined in Fig. 3.1, 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.

<i>Type</i>	\Rightarrow	<i>TypeName</i> { [<i>Variant</i> [, <i>Variant</i>]*] }
<i>Variant</i>	\Rightarrow	<i>VariantName</i> { [<i>Field</i> [, <i>Field</i>]*] }
<i>Field</i>	\Rightarrow	<i>FieldName</i> < <i>TagName</i> >
<i>TypeDef</i>	\Rightarrow	type <i>TypeName</i> := <i>Type</i> ;
<i>VariantDef</i>	\Rightarrow	variant <i>VariantName</i> := <i>Variant</i> ;

Figure 3.1: 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} };
```

⁰The 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.

3.2 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 3.1 (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: \perp 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 \perp : $(\forall t)\perp \sqsubseteq t$. Furthermore, any type t free from binding tags is a subtype of \top : $(\forall t)BT(t) = \emptyset \implies t \sqsubseteq \top$.

3.3 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 3.2 (complete record type) *A record type τ is called complete iff*

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

$$\begin{aligned}
\textit{TypeSignature} &\Rightarrow \{ \textit{Mapping} [, \textit{Mapping}]^* \} \\
\textit{Mapping} &\Rightarrow [\textit{Variant}] \rightarrow \textit{Type}
\end{aligned}$$

Figure 3.2: 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.

3.4 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 $T_{in} \rightarrow T_{out}$, but instead we use the following syntax to provide a subtyping structure:

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

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

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

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. 3.2.

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 mimic 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 $x1 \rightarrow x1+a$, $y1 \rightarrow y1+b$. 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 runtime 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.

3.5 Flow Inheritance

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 3.3 (flow inheritance) *Let $v^{[i]} \rightarrow \tau^{[i]}$, $i \in [1, \dots, n]$, 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 $k \leq n$ and any field or non-binding tag $\phi \notin v^{[k]}$ such that*

$$(\forall i \neq k) BT(v^{[k]}) \neq BT(v^{[i]}) \vee v^{[k]} \cup \{\phi\} \not\subseteq v^{[i]},$$

the box X can be subtyped by flow inheritance to the type $X' : V^{[i]} \rightarrow T^{[i]}$, where

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

and

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

Here $\tau_* = \{V_1, \dots, V_{m_k}\}$ 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 name¹. 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 $(n, m)v^{[i]} \rightarrow w_j^{[i]}$, which signifies a box with input variants $v^{[i]}$ and the corresponding output types $\tau^{[i]} = \{w_1^{[i]}, \dots, w_{m_i}^{[i]}\}$, $i \in \{1, \dots, n\}$. 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 $\{x\} \rightarrow \{y\}$ and $\{x, a\} \rightarrow \{y, a\}$, 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.

3.6 Box Subtyping

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

$$\frac{f : \tau_1 \rightarrow \tau_2, \tau_1 \sqsubseteq \tau'_1 \quad \tau'_2 \sqsubseteq \tau_2}{f : \tau'_1 \rightarrow \tau'_2}$$

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

¹Obviously, an implementation is free to simply switch references.

input field: the type $(n, m)V^{[Tv^{(i)}]} \rightarrow W_j^{[Tv^{(i)}]}$, where for some field name

$$\phi \in v^{[k]}$$

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

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 well² 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 $l \leq m^{[k]}$

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

²Note 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.

and

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

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

3.7 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 3.5 (monotonicity) *The type signature $(n, m)v^{[i]} \rightarrow \tau^{[i]}$ 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 $\langle x \rangle \rightarrow \tau''$ 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.

Chapter 4

Network Description Language

4.1 Box Declarations

S-NET essentially is a language for specifying hierarchical networks of boxes statically interconnected by typed streams. As a pure coordination language S-NET does not provide any means for the specification of computations, i.e. the concrete behaviour of boxes. This is left to existing computation or box languages like C or SAC. Fig. 4.1 provides a definition of core S-NET syntax.

<i>SNetBox</i>	⇒	<i>AtomicBox</i> <i>CompoundBox</i>
<i>AtomicBox</i>	⇒	box <i>BoxName</i> (<i>BoxSignature</i>) <i>Body</i>
<i>BoxSignature</i>	⇒	<i>Variant</i> -> <i>Type</i>
<i>Body</i>	⇒	{ <<< <i>LanguageName</i> <i>LanguageCode</i> >>> }
		;
<i>CompoundBox</i>	⇒	box <i>BoxName</i> [{ [<i>SNetBox</i>]+ }] <i>Connect</i>
<i>Connect</i>	⇒	connect <i>SNetExpr</i>
<i>SNetExpr</i>	⇒	<i>BoxName</i> <i>Primitive</i> <i>Sync</i>
		<i>Combinator</i> <i>Operator</i>
		(<i>SNetExpr</i>)

Figure 4.1: Grammar of S-NET box specifications

A box in S-NET is declared by the key word **box** followed by the box name. We distinguish two different kinds of boxes: *atomic boxes* and *compound boxes*. An atomic box is implemented using a compute language (as opposed to S-NET as a coordination language) and its operational behaviour is opaque to S-NET. In contrast, compound boxes consist of recursively nested further box specifica-

tions (both atomic and compound) and a representation of their interconnection topology.

The extensional behaviour of an atomic box is specified in terms of a type signature that is restricted to a single mapping between input variant and output type. The specification of the intensional behaviour of a box is outside the scope of S-NET. In general, the code for an atomic box will be found in a separate file, but for convenience S-NET allows the programmer to inline foreign language code. This code is separated from S-NET code by the separator symbols <<< and >>>. S-NET does not process nor even parse the code in between these separator symbols. Instead, the code is passed on to the appropriate compiler. For S-NET to know which compiler to take, the foreign language code section starts with a language identification symbol, which is separated from the code by a bar symbol. Site-specific data like the exact compiler name, compiler flags, etc., are extracted from an S-NET configuration file using the language identifier.

Compound boxes consist of a potentially empty sequence of subbox specifications enclosed in braces. Unlike atomic boxes, compound boxes come without a type signature. Type signatures of compound boxes are inferred by the S-NET compiler. The specification of a compound box is completed by the definition of the interconnection topology of the local boxes following the key word **connect**. In S-NET we specify interconnection topologies by an expression language. These S-NET-expressions are made up of instances of the local boxes referred to by their name, a set of primitive boxes, a special synchronisation cell as well as a number of network combinators and network operators.

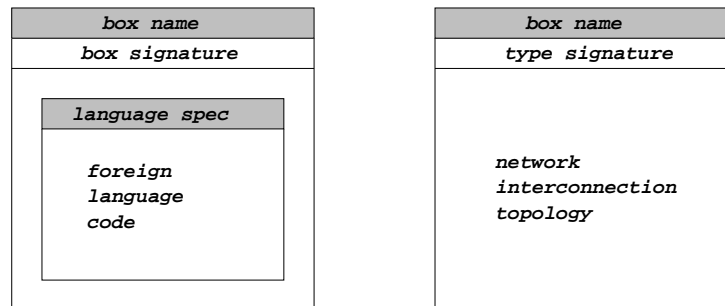


Figure 4.2: Graphical representation of S-NET boxes

Using the geometric term “box” for the constituents of S-NET programs already insinuates a graphical or visual representation of S-NET code equivalent to the textual representation, we have outlined so far. Fig. 4.2 sketches out graphical representations of both atomic (left) and compound (right) boxes. Both contain the box name in a kind of status line, followed by their type signatures. Given the fact that providing a type signature is optional for compound boxes, this field may well be empty. Graphical representations of partially compiled S-NET programs may feature the inferred type signature here.

In the case of an atomic box the remaining field may remain blank. Otherwise, it contains a subbox with the inlined foreign language code and the language specification in the status line of the subbox. In the case of a compound box the remaining field shows a graphical representation of the interconnection topology. Their building blocks, both textual and graphical, are subject to the following sections.

4.2 Primitive Boxes

Fig. 4.3 shows the concrete syntax of the three primitive boxes: *driver*, *link* and *plug*.

- The driver [- produces a stream of empty records. Unlike all other boxes in S-NET, it generates output without being triggered by an incoming record. The driver has the type signature $\{ \rightarrow \{\{\}\} \}$.
- The plug -] consumes any incoming record and produces absolutely nothing. The type signature of the plug is $\{\{\} \rightarrow \{\}\}$.
- The link -- realises a simple identity function: it forwards any incoming record to its output stream. The link has the type signature $\{\{\} \rightarrow \{\{\}\} \}$.

<i>Primitive</i>	\Rightarrow	<i>Driver</i> <i>Plug</i> <i>Link</i>
<i>Driver</i>	\Rightarrow	[[< <i>TagName</i> > [, < <i>TagName</i> >]*] -
<i>Plug</i>	\Rightarrow	- [< <i>TagName</i> > [, < <i>TagName</i> >]*]]
<i>Link</i>	\Rightarrow	- [< <i>TagName</i> > [, < <i>TagName</i> >]*] -

Figure 4.3: Grammar of S-NET primitive boxes

All three primitive boxes in S-NET can be refined by a set of tags *tags*. In the case of the driver this results in an output stream of records containing the given tags; the type signature becomes $\{ \rightarrow \{\{tags\}\} \}$. For both the plug and the link the specification of additional tags leads to a restriction of the input type; the type signatures become $\{\{tags\} \rightarrow \{\}\}$ for the plug and $\{\{tags\} \rightarrow \{\{tags\}\} \}$ for the link, respectively.

Fig. 4.4 sketches out graphical representations for driver, link and plug. Each can be represented by a box with the respective symbol in the status line. If further specifications, e.g. tags, are present, they appear in the main field. Otherwise, it remains blank.

4.3 The Synchronocell

The synchronisation cell, or synchronocell for short, is the only “stateful” box in S-NET; its concrete syntax is given in Fig. 4.5. A synchronocell starts with the key

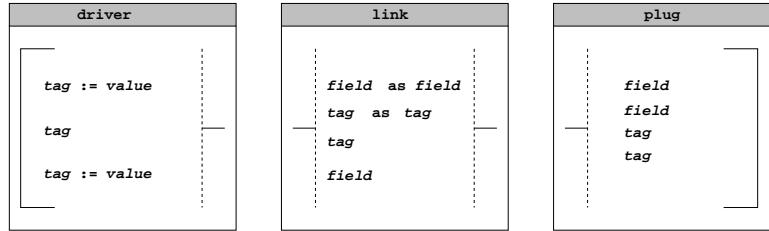


Figure 4.4: Graphical representation of S-NET primitive boxes

word **sync** followed by two or more variant specifications enclosed in brackets. Its behaviour can be refined by the use of tag specifications as explained in the sequel.

$Sync \Rightarrow \mathbf{sync} \ (\ VariantPattern \) \ SyncTags$
 $VariantPattern \Rightarrow Pattern \ [\ , \ Pattern \]^*$
 $Pattern \Rightarrow \{ \ PatternField \ [\ , \ PatternField \] \}$
 $PatternField \Rightarrow \begin{array}{l} \ FieldName \ [\ \mathbf{as} \ FieldName \] \\ | \\ \ < \ TagName \ > \ [\ \mathbf{as} \ < \ TagName \ > \] \end{array}$
 $SyncTags \Rightarrow [\ OutTag \] \ [\ OverflowTag \] \ [\ FixedPointTag \]$
 $OutTag \Rightarrow \mathbf{out} \ < \ TagName \ >$
 $OverflowTag \Rightarrow \mathbf{ofl} \ < \ TagName \ >$
 $FixedPointTag \Rightarrow \mathbf{fix} \ < \ TagName \ >$

Figure 4.5: Grammar of S-NET synchronocells

A synchronocell has storage for exactly one record of each given variant. When a record matching one of the variants arrives at the synchronocell, it is kept in this storage. Any record arriving thereafter that matches the same variant is immediately passed through the synchronocell. If the optional overflow flag is specified, that tag is added to each record passed through the synchronocell in this way. Let us assume all but one variants have been matched by an incoming record with the records properly stored in the synchronocell. As soon as a record arrives that matches the last remaining previously unmatched variant, all stored records are released. The output record is created by merging the fields of all stored records into the last matching record. This requires the various patterns to be pairwise disjoint. Otherwise, we had indistinguishable fields in the output record. If this prerequisite is not met right away, the key word **as** allows us to rename fields. As a consequence we may use patterns with overlapping field names as long as the fields are properly renamed to have unique names in the output stream. If the optional output tag is specified, that tag is also added to the output record.

If an incoming record matches more than one variant, all matching variants are marked as being matched and the incoming record is associated with all of them.¹ If an incoming record matches several variants, some of which are so far unmatched while others have already been matched by earlier records, the incoming record is only stored with the previously unmatched variants. However, given the fact that the eventual output record is constructed by merging the fields of all records involved, the sequence in which records arrive is irrelevant in this situation. Should an incoming record match all variants and the synchronocell is still in its initial state, i.e. all variants being unmatched, the record is immediately passed to the output stream. If present, the specified fixed point tag is added to the record in this case.

Once a synchronocell has received incoming records for each of its input variants, its purpose is fulfilled and the cell effectively dies. More precisely, all records received after a full match are immediately passed to the output channel with the overflow tag added where appropriate.

The type signature of a synchronocell

```
sync (v1, ..., vn) out out ofl ofl fix fix is
{v1}          ->  {{v1, ofl}, {v1, ..., vn, out}}
```

...

```
{vn}          ->  {{vn, ofl}, {v1, ..., vn, out}}
```

```
{v1, ..., vn} ->  {{v1, ..., vn, fix}, {v1, ..., vn, out}, {v1, ..., vn, ofl}}
```

It reflects the fact that any incoming record may either be passed through in case of an overflow or it may trigger synchronisation, in which case the output record contains fields from all variants. An incoming record that matches all variants may or may not be a fixed point as the synchronocell may or may not have previously received matching records for some of its input variants. Likewise, the cell may or may not have synchronised before. While in the former case even a fully matching input record is passed through as an overflow, in the latter case it triggers synchronisation. As a consequence, a fully matching record is always passed through the synchro cell, but any of the three tags may in fact be added.

The synchronocell has a certain behaviour under flow inheritance. If a synchronocell stores a matching input record, it produces no output in response to this record. Hence, excess record fields, which would bypass the synchronocell otherwise, are discarded. Any record output after successful synchronisation is extended by the excess fields of the last incoming record because the synchronocell produces this output as a response to the input of this record. Last but not least, if a record is passed through the synchronocell in the case of overflow, there is output in response to input and, therefore, the excess fields bypass the synchronocell as usual.

Fig. 4.6 shows the graphical representation of a synchronocell. Following the name “sync” in the status line, the main field contains the patterns line by line. The double line to the right of the patterns insinuates the synchronisation.

¹Of course, an implementation avoids copying the record.

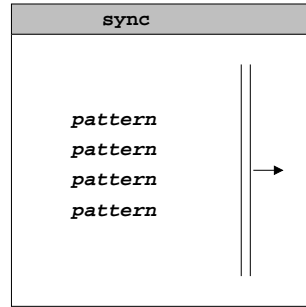


Figure 4.6: Graphical representation of S-NET synchronocells

4.4 Network Combinators

Complex network topologies are formed using the four network combinators: *serial*, *closure*, *choice* and *splitter*. Their grammar is defined in Fig. 4.7 while Fig. 4.8 sketches out the corresponding graphical representations.

<i>Combinator</i>	\Rightarrow	<i>Serial</i> <i>Closure</i> <i>Choice</i> <i>Splitter</i>
<i>Serial</i>	\Rightarrow	<i>SNetExpr</i> . . <i>SNetExpr</i>
<i>Closure</i>	\Rightarrow	<i>SNetExpr</i> *
<i>Choice</i>	\Rightarrow	<i>SNetExpr</i> <i>SNetExpr</i> <i>SNetExpr</i> <i>SNetExpr</i>
<i>Splitter</i>	\Rightarrow	<i>SNetExpr</i> !! < <i>TagName</i> > [: < <i>TagName</i> >] <i>SNetExpr</i> ! < <i>TagName</i> > [: < <i>TagName</i> >]

Figure 4.7: Grammar of S-NET network combinators

The binary serial combinator “.” connects the output of the left operand to the input of the right operand. The input of the left operand and the output of the right one become those of the resulting network. The serial combinator establishes computational pipelines. Graphically, it is represented by an arrow connecting the two argument boxes.

The unary closure combinator “*” (conceptually) replicates the operand infinitely many times and connects the replicas by the serial combinator. If the type signature of the operand contains a fixed point rule of the form $\{\langle v \rangle\} \rightarrow \{\langle v \rangle\}$ for some tag v , dynamic replication of the operand stops as soon as all records carry that tag. Otherwise, replication unfolds infinitely and no records can leave the closure, i.e., type-wise the closure becomes a plug. Implementation-wise the closure combinator realises a feedback loop, and the output stream of the operand network is directly connected back to its input stream. This property is emphasised by the graphical representation in Fig. 4.8.

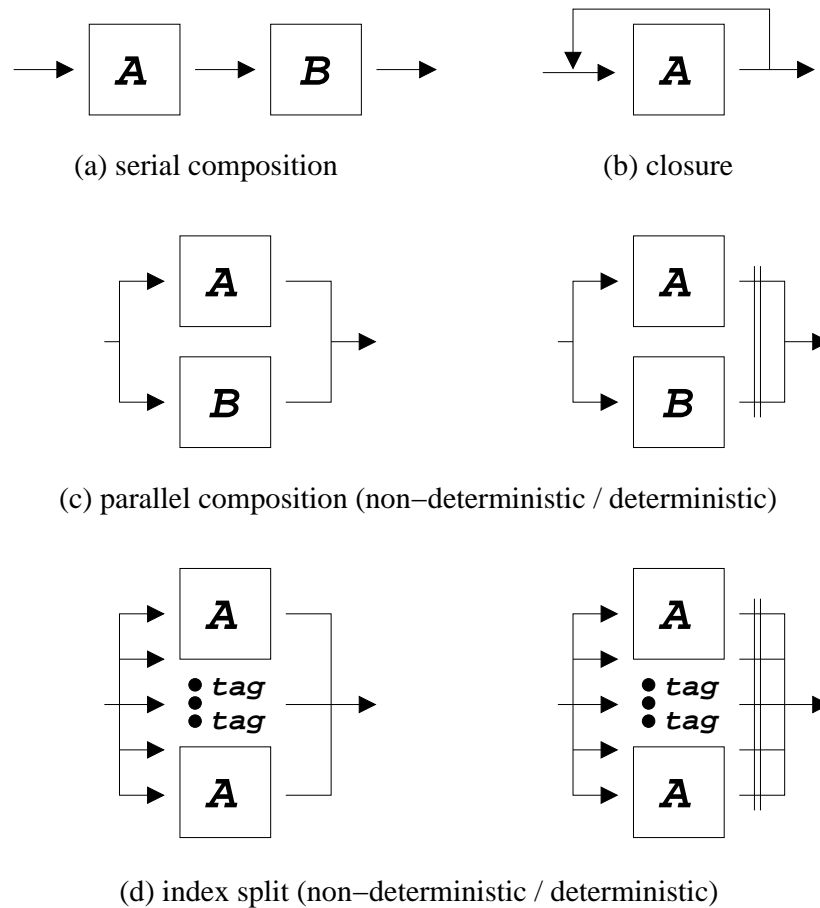


Figure 4.8: Graphical representation of S-NET network combinators

The binary choice combinator “| |” or “|” combines its operands in parallel. If an incoming record matches the type signature of one of the operand networks, it is sent to that network. Should an incoming record match the type signatures of both operand networks, it is non-deterministically sent to one of them. An implementation is free to choose an appropriate scheduling technique in this case. For example, it may send the record to the less loaded subnetwork for proper workload balancing. The graphical representation of the choice combinator employs a split and a merged arrow to visualise the choice operator.

The output streams of the operand networks are merged into a single stream which becomes the output stream of the combined network. Here, the two variants of the choice combinator, “| |” and “|” behave differently. Whereas the “| |” choice combinator merges the two output streams non-deterministically, the “|” variant preserves the sequence of records. More precisely, any output

generated by one of the operand networks in response to an incoming record on the joint input stream is sent to the joint output stream before any records produced by any of the subnetworks in response to a subsequent input record. In the graphical representation an additional vertical double line symbolises the synchronisation necessary to achieve this deterministic behaviour.

Providing these two variants of the choice combinator is motivated by the observation that different application scenarios require different concrete operational behaviours of choice. The non-deterministic variant usually is more efficient since it allows the network to continue processing records as soon as they are available. However, in many situations it is crucial that a compound box behaves as an atomic box in that it preserves the sequence of causality and records may not overtake others. This comes at the price of holding back readily processed records from the output stream and waiting for other records to be sent first.

The binary index split combinator “! $\!$ ” or “!” takes a subnetwork or box as its left operand and one or two tags as its right operand. Like the closure combinator, the index split combinator replicates the box operand, but connects the replicas using the choice operator. The number of replicas is conceptually infinite; each replica is identified by an integer index. Any incoming record goes to the replica identified by the value associated with the named tag, i.e., all records that have the same tag value will be processed by the same replica. If actually two tags are specified, the record is broadcast to all replicas with indices in the range between the value of the first tag and the value of the second tag.

The graphical representation of the index split combinator, as shown in Fig. 4.8, symbolically replicates the argument box with three vertical dots representing the dynamic number of replicas. The name(s) of the tag(s) that control replication are annotated at the vertical dots.

Analogously to the choice operator, the output streams of the replicas are merged into a single output stream of the combined network either non-deterministically (“! $\!$ ”) or under preservation of causality with respect to the sequence of records on the input stream. As in the case of the choice combinator an additional vertical double line visualises the necessary synchronisation in order to achieve this deterministic behaviour.

4.5 Network Operators

Whereas the network combinators allow us to construct complex networks from simpler ones, the network operators can be used to manipulate the interface of a given network to the outside world. Fig. 4.9 shows the syntax of S-NET network operators.

The pass operator restricts the type signature of a subnetwork to certain patterns of incoming records (key word **to**) or outgoing records (key word **from**). Any incoming record must match one of the given patterns. Pattern matching is strictly by subset relationship on record fields. In contrast to the subtyping relationship, as defined in Section 3.2, there is no special treatment of binding

<i>Operator</i>	\Rightarrow	<i>Pass</i> <i>Strip</i> <i>Set</i>
<i>Pass</i>	\Rightarrow	pass (<i>VariantPattern</i>) to <i>SNetExpr</i>
<i>Strip</i>	\Rightarrow	strip (<i>Field</i> [, <i>Field</i>]*) from <i>SNetExpr</i>
<i>Set</i>	\Rightarrow	set (<i>Parameter</i> [; <i>Parameter</i>]*) in <i>SNetExpr</i>
<i>Parameter</i>	\Rightarrow	<i>FieldName</i> := <i>ExternalExpr</i> < <i>TagName</i> > [:= <i>Expr</i>]
<i>Expr</i>	\Rightarrow	<i>ExternalExpr</i> <i>IntegerConst</i>
<i>ExternalExpr</i>	\Rightarrow	<<< <i>LanguageName</i> <i>ForeignExpr</i> >>>

Figure 4.9: Grammar of S-NET network specifications

tags in pattern matching. Matched record fields are sent to the subnetwork while excess fields are flow inherited by the output stream of the subnetwork. Using the key word **as** in the pattern specification, allows us to rename record fields before entering the subnetwork.

If the pass operator is used to filter the output stream of a subnetwork (key word **from**) only those fields matching one of the patterns actually leave the subnetwork whereas all others are discarded. Again, we may rename fields before they leave the subnetwork. The pass operator is a way to create separate name spaces for subnetworks.

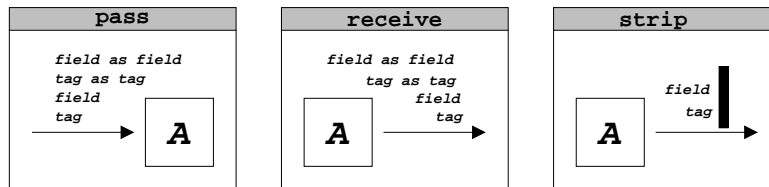


Figure 4.10: Graphical representation of S-NET network operators

The strip operator resembles the second variant of the pass operator. It strips the named record fields from any record on the output stream and discards them. Whereas the pass operator allows us to specify which fields actually remain in outgoing records, the strip operator names those to discard.

Last but not least, the set operator provides a means to introduce initial field values to records. If an incoming record already contains a given field, its value remains unaltered. Otherwise, the field is added and initialised with a value as specified in the set operator. For tags the initialisation value may simply be an integer constant. If it is left out, the tag is set to zero. However, for fields in general the situation is more complicated because S-NET as a pure coordination language deliberately lacks any notion of field value and the means to describe them. Hence, we follow the same path as with atomic box specifications and

exploit the capabilities of a box language. Embraced within the separator symbols <<< and >>> we feature a box language identifier and separated by a single bar an expression in that language.

Chapter 5

Type Inference and Semantics

5.1 Semantics

Define the alphabet of a box $x : (n, m)v^{[i]} \rightarrow w_j^{[i]}$ 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 \mathbf{x} , i.e. its action on any record $v \in V^0 = \aleph(x) \rightarrow D$, where D is the set of all possible field values, can be defined as a semantic function $\hat{x} : V^0 \xrightarrow{?} \alpha(V^0)$, 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 $\emptyset \rightarrow D$, which is the empty set of fields) ϵ , $V = V^0 \cup \epsilon$ and redefining $\hat{x} : V \rightarrow \alpha V$. 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 5.4.1. 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 \mathbf{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 $V^\infty = \Phi^\infty \rightarrow (D \cup \{\epsilon\})$, which results in the following semantic function $\bar{x}_q : V^\infty \rightarrow \alpha(V^\infty)$:

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

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

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

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:

$$\tilde{x}_q = (\odot /) \circ (\text{map } \bar{x}_q).$$

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 form 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 \tilde{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 $\tilde{x}_q a_1$ is a prefix of $\tilde{x}_q a$, i.e. \tilde{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 \tilde{x}_q is taken at the same q for different input prefixes, and is immediately destroyed by nondeterminism.

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

5.2 Serial Combinator

Consider two networks $a : (n, m)v^{[i]} \rightarrow w_j^{[i]}$ and $A : (N, M)V^{[i]} \rightarrow W_j^{[i]}$. The serial combinator $\mathbf{a} . \mathbf{A}$ 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 $\mathbf{a} \cdot A$. Let us define the formal semantics of $\mathbf{a} \cdot A$. Formally it is defined thus:

Definition 5.1 (serial combinator) *The serial combination $\mathbf{S} = \mathbf{a} \cdot A$ of networks \mathbf{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 $S = a \cdot A$, 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 : $a :: (\nu)v^{[k]} \rightarrow w^{[k]}$, 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 $a :: (n)v^{[i]} \rightarrow w^{[i]}$ and $A :: (N)V^{[i]} \rightarrow W^{[i]}$. 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 \dots operator:

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

where

$$R = \{v^{[i]} \cup (V^{[j]} \setminus w^{[i]}) \rightarrow (w^{[i]} \setminus V^{[j]}) \cup W^{[j]} \mid (V^{[j]} \setminus w^{[i]}) \cap v^{[i]} = \emptyset\}, \quad (5.1)$$

and $\Theta(X)$ is an indexed sequence of members $p \rightarrow q$ 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 5.1 depicts the set-theoretical relations between inputs and outputs as defined by Eq 5.1.

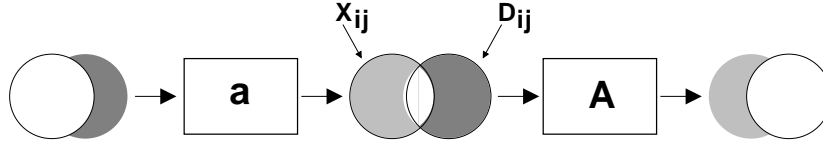


Figure 5.1: Flow inheritance through the \dots combinator

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. 5.1 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 5.1 is modified thus:

$$R = \{v^{[i]} \cup (V^{[j]} \setminus w^{[i]}) \rightarrow (w^{[i]} \setminus V^{[j]}) \cup W^{[j]} \mid (V^{[j]} \setminus w^{[i]}) \cap (v^{[i]} \cup B) = \emptyset\}, \quad (5.2)$$

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

5.3 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:

$$B..B..B.. \dots$$

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

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

Using the above, the closure of \mathbf{B} over a set F is

$$\check{B}_q^\circ = c_q^{[i_\infty]}, \quad (5.3)$$

where $i_\infty = \max_q i_q$, and $i_q = \min\{i \mid c_q^{[i]} \in F\}$. 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 \mathbf{B} unchanged. In other words, F is a set of fixed points of the network \mathbf{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 \mathbf{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 \tilde{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 remedy, let us take a closer look at the recurrent process in Definition 5.2. 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 reached¹. 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 $t \rightarrow t$ 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 $r = v \rightarrow D$ 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 5.3 (hard closure) *A hard closure of a box B is a network B^* whose semantic function \tilde{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.

¹this corresponds to “laziness” in functional semantics

Definition 5.3 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 $B : (n, m)v^{[i]} \rightarrow w_j^{[i]}$ introduce a function $\phi : Q \times \mathbb{N} \rightarrow Q$, 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 k th variant of the output type in response to the given input type variant 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 $\Sigma = (n, m)v^{[i]} \rightarrow w_j^{[i]}$ onto its functional representation $\phi : Q \times \mathbb{N} \rightarrow Q$ as $\Psi(\Sigma)$.

Proposition 5.1 *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 $\Sigma^0 = (n, m)v^{[i]} \rightarrow w_j^{[i]}$, 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 $T_{i, \subseteq} X$ is the i th 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 $v^{[i_1]} \rightarrow w_j^{[i_1]}$ and $v^{[i_2]} \rightarrow w_j^{[i_2]}$ such that $v^{[i_1]} \subset v^{[i_2]}$, $m_{i_1} = m_{i_2}$ and $\forall j_2 \exists j_1 w_{j_2}^{[i_2]} = w_{j_1}^{[i_1]} \cup (v^{[i_2]} \setminus v^{[i_1]})$, 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 $Q \times \mathbb{N} \rightarrow Q$.

Definition 5.4 *Consider two functions $\phi_{1,2} : Q \times \mathbb{N} \rightarrow Q$. 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 n th member of the series. Then the serial combination $\phi_1.. \phi_2$ is a function $\phi_s : Q \times \mathbb{N} \rightarrow Q$ 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 5.1 to the following

Proposition 5.2 Ψ is an isomorphism between the algebras $(\Sigma, ..)$ and $(Q \times \mathbb{N} \rightarrow Q, ..)$ modulo the variant and type orderings that are neutral to the type transformation defined by Σ .

The proof is obtained by comparing Eq 5.1 with Definition 5.4. 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 $(Q \times \mathbb{N} \rightarrow Q, ..)$ is in fact a semigroup:

Proposition 5.3 The operation $..$ as defined by Definition 5.4 is associative.

To prove this, we must prove that for all $\phi_{1-3} : Q \times \mathbb{N} \rightarrow Q$, $(\phi_1.. \phi_2).. \phi_3 = \phi_1..(\phi_2.. \phi_3)$. By applying both sides to some 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, $(n_1^R, k) = \sigma_v^n(\phi_1, \phi_2)$; $(n_2^R, n_3^R) = \sigma_{\phi_1(v, n_1)}^k(\phi_1, \phi_2.. \phi_3)$. 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_{1-3} , 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 interested in hard closure B^* , let us define the projector box for $B : v^{[i]} \rightarrow \tau^{[i]}$ as $B^\rightarrow : v^{[i]} \rightarrow \tau_*^{[i]}$, 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

$$B^* \equiv \underbrace{B..B, \dots, ..B}_{L \text{ times}} ..B^{-} \quad (5.4)$$

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 5.2 and 5.3 establish associativity of the serial combinator viewed as a type constructor. Let us change the evaluation order in Eq 5.4 to achieve back chaining, i.e.

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

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.

5.4 Choice operator

Consider boxes $A: v_a^{[i]} \rightarrow \tau_a^{[i]}$ and $B: v_b^{[i]} \rightarrow \tau_b^{[i]}$. The choice combinator $A|B$ produces a box C that works as follows. A record appearing at the input of C is compared in type with v_a ; if it matches, then the record is directed to A ; if it matches v_b , the record is directed to B ; if the record matches both v_a and v_b , then the maximum match is used; if the maximum match is ambiguous, then a nondeterministic choice is made between A and B in determining the destination of the record. The outputs of A and B are merged into one stream arbitrarily. Here is a formal definition:

Definition 5.5 (choice) *The choice combination $C = A|B$ of networks A and B is a network whose behaviour is represented by the following semantic family \check{C}_q . For every input stream x , the action of the semantic function is*

$$\check{C}_q x = \mathbf{merge}_{q''''}(\check{A}_{q'} x_A, \check{B}_{q''} x_B),$$

where

$$(x_A, x_B) = \mathbf{split}_{q''''}$$

and

$$E(C) = E(A) \times E(B) \times U^\infty \times U^\infty; \quad q = (q', q'', q''', q'''').$$

Here the two auxiliary functions $\mathbf{merge} : (\alpha(V \rightarrow D), \alpha(V \rightarrow D)) \rightarrow \alpha(V \rightarrow D)$ and $\mathbf{split}_q : \alpha(V \rightarrow D) \rightarrow (\alpha(V \rightarrow D), \alpha(V \rightarrow D))$ are defined as follows. The \mathbf{split}_q function splits the stream into two according to the input types of A and B using maximum match; where the choice is ambiguous, the splitting is done according to the event $q \in U^\infty$, the latter being the set of binary strings of unlimited length. Each bit of q represents one instance of choice. The function \mathbf{msort}_q is the merge of two record streams into a single stream under the control of $q \in U^\infty$.

The set of typing rules for a choice combination is straightforward. For convenience we use rule-sets for boxes A and B , Σ_A and Σ_B , rather than lists of rules (i.e. signatures) as before. Given a rule-set the corresponding signature is obtained immediately by topological sorting. The rule set of the choice combination, Σ_C is as follows:

$$\Sigma_C = \left\{ v \rightarrow \tau_* \mid (\exists \tau)(v \rightarrow \tau) \in \Sigma_+ \wedge \tau_* = \bigsqcup_{(v \rightarrow \tau) \in \Sigma_+} \tau \right\},$$

where

$$\Sigma_+ = \{v \rightarrow \tau \mid \exists (v_A \rightarrow \tau_A \in \Sigma_A, v_B \rightarrow \tau_B \in \Sigma_B) v = v_A \sqcup v_B \wedge \tau = \tau_A \sqcup \tau_B\}.$$

Note that the least upper bound $v_A \sqcup v_B$ exists only when $BT(v_A) = BT(v_B)$. Also note that the resulting type signature is complete and monotonic by construction.

5.4.1 Type signature completion

Finally, recall that the input types of boxes are required to be complete (see Section 3.4). Our approach to completeness is slightly different from the way we treat monotonicity. Since in the absence of binding tags any two input variants produce a common subtype, which would require a certain kind of output type in response, it is convenient to rely on a default action rather than painstakingly defining all possible responses. Indeed in most cases the input will not deliver such records, and so no matter what the default solution is it will not be used. In those rare cases when a record does match two variants, it is logical to choose a match nondeterministically, which is how S-NET behaves. Indeed, the record matches both variants and there is no reason to prefer one to the other, but it is useful in implementation to be able to choose an alternative that is ready to proceed (as opposed to an alternative that is busy processing some previous record). In S-NET, it is possible to create a network that would profit from such non-determinism, since flow inheritance makes it possible to preserve the unmatched fields no matter which variant has been selected. One can easily imagine an arrangement under which the output of such a nondeterministic box is fed to a further box, which uses the unmatched fields to do some further processing.

The type signature of a box with the default action taken into account becomes complete in the sense of Definition 3.5. To calculate it, use the following algorithm:

```

repeat
  for all pairs of rules  $(i_1, i_2)$  in  $x$  such that  $BT(v^{[i_1]}) = BT(v^{[i_2]})$  do:
    if  $(\exists k < \min(i_1, i_2)) v^{[k]} = v^{[i_1]} \cup v^{[i_2]}$ 
      add rule  $(v^{[i_1]} \cup v^{[i_2]}) \rightarrow (\tau_*^{[i_1]} \cup \tau_*^{[i_2]})$ ,
      where  $\tau_*^{[i_1, 2]} = \{w \cup (v^{[k]} \setminus v^{[i_2, 1]}) \mid w \in \tau^{[i_1, 2]}\}$ 
    else if  $\tau^{[k]} \sqsubseteq (\tau^{[i_1]} \cup \tau^{[i_2]})$ 
      continue
    else
      fail
    end
  until the signature is monotonic

```

Figure 5.2: Algorithm to make type signature complete

The correctness proof is straightforward, since we only add rules whose absence violates monotonicity and since we can always add those rules if they are

not contradicted by the signature, in which case we fail, and finally since there is only a finite number of rules to add to any finite signature before it becomes monotonic.

If the above process fails, it yields a triplet of rules that violate Definition 3.5. Those rules could be either primary, i.e. coming from the original type signature, or secondary, i.e. added by the process, but the latter can eventually be traced back to primary rules. Thus the programmer gets a complete diagnostic. The maximum number of added rules is exponential in the number of intersecting variants for a given set of binding tags, but the latter number in any real design would be very small. Nevertheless, in an implementation the compiler can refuse to complete the signature if it is too large and when, at the same time, no cut-off information is derivable from the environment. Also, any nondeterminism is easily detected by the compiler when boxes are connected into a network, and the type transformation in each box is made certain. A warning then could be issued in case such behaviour is not intentional.

Hereinafter we assume that all type signatures are completed using the above algorithm and will freely use incomplete input types.

5.5 Index splitter

This operator is written in the form $B!k$, where B is a network and k is a field name, called the *index* hereinafter. Informally, it creates an array of replicas of network B and assigns each replica a unique value from the field type of k . The input stream must contain only records that have field k (among others). Each input record is directed to the replica of B assigned its value of k . The output of all replicas is merged into a single stream. Note that the array is conceptually infinite since SNet has no access to field types, and that the specific replica is selected on the basis of value identity at run time, the more so that in any practical input stream, the variety of k values would be a small set compared to the full type. The assumption of the infinite array is safe since SNet boxes and networks cannot produce output without input, hence the replicas that receive no input are semantically non-existent.

Next we give formal definitions. First the type signature. Let $v_i \rightarrow \tau_i$ be the signature of B . The signature of $B!k$ is then $(v_i \cup \{k\}) \rightarrow \delta(k, v_i, \tau_i)$, where

$$\delta(k, v, \tau) = \begin{cases} (k \cup \tau) & \text{if } k \notin v, \\ \tau & \text{otherwise} \end{cases} .$$

Assume the index takes values from a set V , and, for simplicity, that the set is finite. Now construct set $T = \{t_i\}$ of unique binding tags of the same size. Let $f : V \rightarrow T$ be a bijection between the sets: $T = \{t_i = f i \mid i \in V\}$ Construct a third set, a set of replicas R of network B , as follows:

$$R = \{B_i = \theta(B, t_i) \mid i \in V\},$$

where $\theta(B, t)$ is the same network as B , except each of the input variants v is

augmented with the binding tag t . Now the semantics of $B!k$ is given by the following

Definition 5.6 *The network $B!k$ is semantically equivalent to the following*

$$\text{isplit}..(B_{i_1}|B_{i_2}|\dots|B_{|V|}),$$

where

$$\text{isplit} : \{k\} \rightarrow t_{i_1}\{k\}|t_{i_2}\{k\}|\dots|t_{i_{|V|}}\{k\}$$

is a box that expects single-field records $\{k\}$ and produces single-field records $\{t, k\}$, where $t = f k$. Here $i_1 \dots i_{|V|}$ is some enumeration of set V .

Chapter 6

Interfaces

6.1 Atomic Box Implementation

As pointed out in Section 4.1, is a pure coordination language. As such it provides no means whatsoever to specify computations. For that S-NET relies on an external compute language to implement atomic boxes. S-NET is not fixed to a specific box language and may well combine boxes implemented in different box language in a single network.

However, proper interaction between S-NET and box languages requires that box languages provide a certain infrastructure and code written in box languages obeys to certain rules and restrictions. For example, a box implementation is expected to compute a function mapping an input record to none, one or multiple output records. In particular, the box code must not interact with its execution environment, although a box language may well provide the necessary means to do so. Furthermore, the box code must be reentrant and refrain from leaving any information in persistent storage. Whereas purely functional languages encourage such a programming style in a natural way, the utilisation of imperative languages requires some discipline from the programmer.

The type signature of a box serves as the only specification of the box's behaviour and its interface to S-NET. Even if a specification contains inlined box language code, S-NET is unable to take advantage of the additional information because syntax and semantics of box languages are unknown to S-NET. This is the price for a clear separation between coordination and computation language and the box language independent design of S-NET. Another consequence of this is that concrete types of data and, hence, the representation of data in memory are unknown to S-NET.

To overcome this lack of information all data sent via S-NET streams must be boxed. Hence, S-NET only transfers pointer or references into a shared heap memory while only the box language code is actually able to interpret the data behind a pointer. The only exception to this rule are tags. Both binding and non-binding tags are represented by unboxed integers. Values associated with

tags are visible to both S-NET and the box language(s). This silently assumes a common representation of integer values across diverse box languages and S-NET. However, in practice any reasonable hardware architecture provides some uniform format to store integer numbers, and any reasonable programming language in our targeted application domain provides access to the genuine hardware-supported data types.

```

box example ({a,b,<t>->{{c,<t>},{x,y,z}})
{
  <<< C | snethandle_t *example( snethandle_t *snethandle,
                                sometype *a,
                                sometype *b,
                                int t)
    {
      /* regular C code to compute c and t */

      snethandle = snetout( snethandle, 0, c, t);

      /* more C code to compute x, y and z */

      snethandle = snetout( snethandle, 1, x, y, z);

      return( snethandle);
    }
  >>>
}

```

Figure 6.1: Example of a C implementation of an atomic box

Fig. 6.1 demonstrates the interplay between S-NET box signature and a C binding box implementation. The first line declares an atomic box named `example`. It maps records containing fields `a` and `b` and a tag `t` to none, one or more records that either contain a field `c` and a tag `t` or fields `x`, `y` and `z`. For the purpose of illustration we use inlined box language code in Fig. 6.1; the same code may alternatively reside in a different file.

We assume a function that is named like the S-NET box. The first parameter of the function, regardless of the box signature, is a handle to an S-NET control data structure. This handle is opaque to the box language programmer and must be provided by the language binding. The following parameters are the record fields of the input record type in the sequence of their specification in the box signature. As explained before, record fields are of some pointer type while tags are of type integer. Although it may be useful to name the parameters according to the field names in the box signature, this is not a formal requirement.

Box signatures form a double purpose in S-NET. Firstly, they are a restricted type signature for the type system of S-NET. In this role the input variant and the output types are sets and, hence, box signatures `{a,b,<t>->{{c,<t>},{x,y,z}}` and `{b,a,<t>->{{x,y,z},{<t>,c}}` are equivalent. Secondly, box signatures declare the relevant signature of the box language function that implements a

box. In this role, the sequence of fields in an input variant obviously matters as it describes the concrete sequence of parameters of the box language function.

The box language function may contain any box language code, but must obey to the rules stated above: it neither must draw information from external sources other than its arguments nor must it interact with the outside world in any way.

S-NET boxes may return none, one or multiple records in response to a single input record. As a consequence, we cannot utilise the normal function result for producing an output record because that would enforce a one-to-one correspondence between input and output records. Instead, we use a special function `snetout` that must be provided by the S-NET box language binding. This function receives the S-NET handle provided as an argument to the box language function as its first argument. The second argument is a number referring to the output variant of the box signature. This information is vital for any implementation to interpret the output record. The remaining arguments are the record fields in the sequence of their declaration in the corresponding output variant of the box signature.

Calls to the `snetout` function may occur anywhere in the box language code. Taking C as box language as in our example means that these calls may occur in loops and branches. Depending on the values of the input record the number of calls to `snetout` may vary and so the number of records produced in response. The `snetout` returns the S-NET handle. Eventually, the box language function returns the given handle.

6.2 Input/Output and the Outside World

6.3 Box Language Binding

Chapter 7

Examples

7.1 Defining factorial in S-Net

This chapter outlines the transformation of a tail-end recursive implementation of the factorial function to its S-NET representation. The functional language representation of the function is shown in Fig. 7.1. Following an bottom-up

```
fac (n) =
  let fac2 (x, y) =
    if x <= 1
    then (x, y)
    else
      let u = x - 1
          v = x * y
      in fac2 (u, v)
  in let (x, y) = fac2 (n, 1)
  in y
```

Figure 7.1: Possible implementation of the factorial function using a functional language

approach for the construction of the net, atomic boxes for the computation of values are defined first. Namely, these are DEC: $\{x\} \rightarrow \{u\}$ and MULT: $\{x,y\} \rightarrow \{v\}$. The implementation shown in Fig. 7.1 uses a simultaneous let in line 6 and line 7. Accordingly, the operations may be executed in parallel, which makes the parallel combinator the operator of choice for the resulting net. A synchrocell reassembles the type after the parallel computation.

graphic.here

The computation of the conditional from lines 3 to 5 is split up into two separate boxes. An atomic box computes the boolean value of the condition ($x \leq 1$) and adds a new field to its output. A subsequent box replaces that field with a

binding tag, according to the boolean value of the newly added field.

graphic.here

It relies on the added tag, whether there will be another "recursive" call to the function or if the computation terminates. This will become clear in the following. If the binding tag is $\langle T \rangle$, a new binding tag $\langle \text{Terminate} \rangle$ is added to the type. A computation of values using the net from above will take place otherwise. A closure combinator is applied to the whole net.

graphic.here

A refined link ensures, that no more computation is carried out, once the $\langle \text{Terminate} \rangle$ tag is present. In fact, this realises a fix point and stops the dynamic replication of the net caused by the closure combinator.

graphic.here

The signature of the net so far is $\{x, y\} \rightarrow \{x, y\}$. To conform to the definition of the function as seen in Fig. 7.1, some more work has to be done.

Chapter 8

Conclusions and Future Work

Bibliography

- [1] Broy, M., Stefanescu, G.: The algebra of stream processing functions. *Theoretical Computer Science* (2001) 99–129
- [2] van Rossum, G.: *The Python Language Reference Manual*. Network Theory Ltd (2003)
- [3] 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
- [4] Ashcroft, E.A., Wadge, W.W.: Lucid, a nonprocedural language with iteration. *Communications of the ACM* **20** (1977) 519–526
- [5] Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE* **79** (1991) 1305–1320
- [6] Berry, G., Gonthier, G.: The esternel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming* **19** (1992) 87–152
- [7] Binder, J.: Safety-critical software for aerospace systems. *Aerospace America* (2004) 26–27
- [8] Caspi, P., Pouzet, M.: Synchronous kahn networks. In Wexelblat, R.L., ed.: *ICFP '96: Proceedings of the first ACM SIGPLAN international conference on Functional programming*. (1996) 226–238
- [9] 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
- [10] 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)

-
- [11] Stephens, R.: A survey of stream processing. *Acta Informatica* **34** (1997) 491–541
- [12] Babcock, B., et al.: Models and issues in data stream systems (invited paper). In: Proc. of the 21st ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems (PODS 2002), Wisconsin, May 2002. (2002) 1–16
- [13] 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
- [14] 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
- [15] Stefanescu, G.: *Network Algebra*. Springer-Verlag (2000)
- [16] Shafarenko, A.: Stream processing on the grid: an array stream transforming language. In: *SNPD*. (2003) 268–276
- [17] Cardelli, L., Wegner, P.: On understanding types, data abstraction, and polymorphism. *Computing Surveys* **4** (1985) 471–522
- [18] Mitchell, J.: Type inference with simple subtypes. *Journal of Functional Programming* **1** (1991) 245–285
- [19] Reynolds, J.C.: Using category theory to design implicit conversions and generic operators. In Jones, N.D., ed.: *Semantics-Directed Compiler Generation*, volume 94 of *Lecture Notes in Computer Science*, Springer-Verlag (1980) 211–258
- [20] Fuh, Y.C.C., Mishra, P.: Type inference with subtypes. *Theoretical Computer Science* **73** (1990) 155–175
- [21] Kaes, S.: Type inference in the presence of overloading, subtyping and recursive types. In: *LFP '92: Proceedings of the 1992 ACM conference on LISP and functional programming*, New York, NY, USA, ACM Press (1992) 193–204
- [22] 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
- [23] Lievant, D.: Discrete polymorphism. In: *Proc. 1990 ACM Conference on LISP and Functional Programming*, June 27-29, 1990, Nice, France. (1990.) 288–297

- [24] Rehof, J., Mogensen, T.: Tractable constraints in finite semilattices. *Science of Computer Programming* **35** (1999) 191–221