

Report on S-Net

A Typed Stream Processing Language

—DRAFT—

Alex Shafarenko and Clemens Grelck

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

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.

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.

This concept 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 [1] and the language Lucid [2]. 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 [3] and Esterel[4], 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 [5]. The authors of Lustre broadened their work towards what they termed synchronous Kahn’s networks[6, 7], 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 [8]. The latest comprehensive survey of stream processing and the underlying theory for it can be found in [9]. There is also a growing activity in *database stream processing* [10], 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 OOP, functional programming, etc.

Around the time that Lustre was introduced, David Turner[11] 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 realize the inter-process communication and capture its nondeterministic behaviour.

This arrangement is sketched in fig 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,

¹ but 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

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.

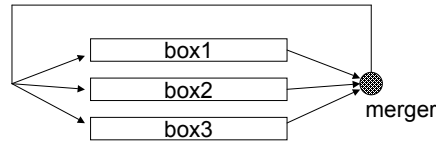


Fig. 1. The Turner scheme

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.

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 [12] and recently including models for nondeterministic stream processing developed collaboratively with Broy [13]. 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 [14]) It is interesting to note that apparently the StreamIt team [8] as well as ourselves [15] were unaware of those and had to re-invent them, albeit for purely pragmatic reasons.

1.2 Proposed Approach

This paper proposes a component technology for stream processing, which we have named S-NET. We treat applications as single-input, single-output (SISO) streaming networks that consist of SISO processing boxes. In this we utilise Turner’s idea, as mentioned previously. S-NET processing boxes are, additionally, stateless, i.e. devoid of the internal state that survives the input-process-output work cycle. This enables them to be deployed cheaply and moved and replicated at will, without raising data integrity concerns. The primary boxes are provided externally to S-NET, by employing a programming language; our methodology has the capability to combine primary boxes into SISO networks which can serve as boxes in further networks. These combinations involve streams of data, which are split according to data types and which are merged non-deterministically. Thus we utilise Turner’s scheme but also set great store by the type system. The combining primitives are called network combinators; they are introduced in the spirit of Stefanescu’s network algebra, with the notable exception that all our entities are SISO.

Since S-NET networks are asynchronous (due to nondeterministic mergers), a synchronisation facility is generally required. It is introduced in the form of a SISO synchro-cell, which is the only kind of ‘stateful’ box in an S-NET network. A synchro-cell expects records of two types to appear at its input; it combines them into a joint record and then outputs the result. Note that synchro-cells have an internal state (which is the record waiting to be synchronised) but perform no computations, while the ‘normal’ 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 subtype. Crucially, S-NET does not require explicit subtype declarations; instead a subtype inference algorithm is applied to determine the most appropriate subtype.

Some of the relevant techniques are defined in our previous publications [16, 15, 17]; these will be summarised here for ease of reference. First we introduce S-NET philosophy and some basic concepts.

1.3 Record Types

In S-NET, streams consist of elements which are treated as non-recursive, tagged variant records with arbitrary non-record fields.

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 order².

² Since 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.

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.

Hence at the interface level, nonrecursive records should be sufficient. On the other hand, the absence of cross references greatly simplifies subtyping analysis as well as alleviating memory management and data distribution concerns.

Even when records are nonrecursive, there is generally a need for a data type to include several nonrecursive record structures under a union. 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*, *height* 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. The reason why the tags alone are not sufficient, which they would be in any conventional algebraic type, is to do with subtyping. We consider it next.

1.4 Subtyping

Subtyping is an important adaptation mechanism of a component technology. S-NET focuses on record streams processed by network boxes. A box can be capable of processing more variants than there are in the incoming data type, which should not make the box typing unacceptable. Equally, if a box receives a valid variant extended with additional fields these fields can be ignored and so the data type should still be considered valid. THose additional fields can be 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 in it 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 [18]. 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 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 time³.

Another important novelty of S-NET is the provision of subtype inference for record fields, which obviates explicit field-subtype declarations. This has been made possible by imposing the homomorphism restriction [17]. This restriction is not severe, as it only forces boxes to be consistent with respect to the output dependency on the input type. We shall discuss homomorphic subtyping in section 3.2.

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 → 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 → 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 → 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.

³ To 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.

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 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. This 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), alternation (several boxes connected in parallel with a single synchro-cell 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.

2 Network Description Language

2.1 Types

The type system of S-NET supports nonrecursive variant records with two forms of subtyping: *record subtyping* and *field subtyping*. A record consists of one or more *variants*. Each variant contains a collection of *fields*. A field is made up of a field name, a field type, and the field types’s *archetype*. Archetypes define chains of types with a linear subtyping relationship. Field subtyping is based on this relationship. Tags are special record fields that are not associated with a value and, hence, neither have a type nor an archetype. In contrast to field subtyping, record subtyping is based on the existence of fields in a record. Any type A that contains the fields of a type B (and potentially some additional fields) is considered a subtype of B if each field type in A is a (field) subtype of the corresponding field type in B. Thus, record subtyping and field subtyping are related to each other. In addition to regular record fields, as defined above, a record may also contain *tags*. Tags have a name but no type and, hence, also no

archetype; they are not associated with values, but their presence (or absence) from a record can be used for control purposes.

In S-NET, an archetype is defined using the key word **archetype**. For example,

```
archetype num := bool < int < double < complex ;
```

defines the archetype **num** as the subtyping chain of standard numerical types. More precisely, it introduces the basic type identifiers **bool**, **int**, **double**, and **complex** and defines their subtype relationship as

$$\text{bool} \sqsubseteq \text{int} \sqsubseteq \text{double} \sqsubseteq \text{complex}.$$

Note that S-NET does not associate any kind of machine representation of data with types or archetypes. This is left to the box languages. The archetype specification merely introduces identifiers and defines their relationship among each other. For example, **int** is a type associated with archetype **num**, or type **bool** is a subtype of type **complex**. Fig. 2 provides a formal definition of the grammar of archetype specifications.

$$\begin{aligned} ATypeDef &\Rightarrow \text{archetype } ATypeName := TypeName [< TypeName]^* \\ TypeDef &\Rightarrow \text{type } TypeName := Type \\ Type &\Rightarrow Variant [| Variant]^* \\ &\quad | TypeName \\ &\quad | \text{bottom} \\ Variant &\Rightarrow \{ [Field [, Field]^*] \} \\ Field &\Rightarrow FieldName [[: ATypeName] : TypeName] \\ &\quad | < TagName > \\ TypeSignature &\Rightarrow Variant \rightarrow Type [, Variant \rightarrow Type]^* \end{aligned}$$

Fig. 2. Grammar for S-NET types and type signatures. The non-terminal symbols *TypeName*, *ATypeName*, *FieldName*, and *TagName* refer to identifiers. They are only distinguished for the purpose of clarity.

Types in S-NET are defined using the key word **type**. Following the definition in Fig. 2, a type consists of a number of variants separated by bars. Each variant essentially is a set of fields enclosed in curly brackets.

A field is either a regular record field or a tag. The former consist of field name, archetype name, and type name each separated by colons. Both archetype and type names must have been declared beforehand using an archetype definition or, in the case of a type name, alternatively by a preceding type definitions. As records in S-NET are always non-recursive, the scope of a type definition excludes the right hand side of the definition itself. For example,

```
size:num:int
```

defines a record field with name `size` of archetype `num` and type `int` as introduced by the example archetype definition above.

Archetype specifications are optional as long as they can be inferred from preceding archetype specifications, i.e., a type name occurs in a single archetype specification only. For example,

```
size:int
```

would make a proper record field if the type `int` does not occur in any other archetype specification as the one above. Likewise, the archetype is missing if the type name does not refer to a basic type introduced on the right hand side of an archetype specification, but as a record type introduced by a preceding type definition.

In order to facilitate the specification of complex variants, several record fields may share the same type and archetypes. For example,

```
x,y,z:num:int
```

defines three record fields `x`, `y`, and `z` to be of archetype `num` and type `int`.

Tags are distinguished from regular record fields by enclosing the tag name within angular brackets. As tags are not associated with values, they have neither archetype nor type. Furthermore, we distinguish between *binding* tags, whose names start with a capital letter, and *non-binding* tags, whose names start with a small letter. Whereas non-binding tags are merely ordinary record fields without a value, binding tags are introduced for control purposes. Their differences will be elaborated on in the remainder of this paper.

As an example for a full record type,

```
type body := {<Triangle>, color:int, x1,y1,dx2,dy2,dx3,dy3:double}
           | {<Rectangle>, color:int, x1,y1,dx2,dy2:double}
           | {<Circle>, color:int, x1,y1,r:double}
```

defines a type `body` for the representation of geometric bodies, which are either triangles, rectangles, or circles. Each body has a color, x, y -coordinates of a reference point, and varying numbers of further relative coordinates.

Let us now formally define subtyping on variants and on records

Definition 1 (Record subtyping).

Record subtyping is defined by the following rules:

1. A variant v_1 is a subtype of some v_2 , $v_1 \sqsubseteq v_2$, if $v_1 \supseteq v_2$ as sets of fields and $BT(v_1) = BT(v_2)$, where $BT(x)$ is the set of binding tags in x , provided that the types of any identically named fields $f_1 \equiv f_2$, such that $f_1 \in v_1$ and $f_2 \in v_2$, are in the subtyping relationship $type(f_1) \sqsubseteq type(f_2)$. All tags are treated as having the same type ‘tag’.
2. A record type t_1 is a subtype of a record type t_2 , $t_1 \sqsubseteq t_2$, if

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

For instance, the variant

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

is a subtype of the variant tagged by `<Triangle>` of the type `body` above. It contains exactly the same binding tags (`<Triangle>`) and all regular record fields of the previous variant plus an additional tag `colored`. The types associated with the regular fields are either identical to those on the earlier definition (`dx2,dy2,dx3,dy3,color`) or field subtypes of them (`x1,y1`). 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:int, x1,y1,r:real, shading:bool}
             | {<Rectangle>, color:bool, x1,y1,dx2,dy2:double}
```

Each variant is in subtype relationship to one variant of the supertype `body`. For the `<Circle>` variant we have added an additional field `shading`. For the `<Rectangle>` variant the field `color` is now of type `bool`, which is a subtype of `int` according to the definition of the common archetype `num`.

Two special types are introduced: \perp , the record type with no variants, and \top , the record type with a single variant having no fields. Clearly, for any type t that does not use binding tags, $\perp \sqsubseteq t \sqsubseteq \top$. Indeed, any variant of t can be used where \top is required since that record is coercible to \top by disposal of all fields. Somewhat less obviously, \perp can be used in place of any record since it does not supply a variant. For this to be consistent each operator must respond with \perp when given \perp as an argument. This corresponds to the fact that if no data is supplied, then none will be contained in the result. In fact, even when binding tags are used in t still $\perp \sqsubseteq t$ holds. With respect to the syntax of types in S-NET, as defined in Fig. 2, the type \top can easily be expressed as `{}`, whereas for the type \perp we introduce the special symbol `bottom`.

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, when 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 following type

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

Indeed, each of the three variants of the type `body` defined above can be shortened to one of the variants of `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 $v_1 \sqsubseteq v_2$, 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:

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

there is a choice of variant to use for a `<Rectangle>`. Not every target type can cause such ambiguities. The following definition introduces a uniqueness condition for type coercions:

Definition 2 (complete record type).

A record type τ is called complete iff

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

where $BT(x)$ again denotes the set of binding tags belonging to variant x . That is, if two variants have the same set of binding tags there must be a third variant having all their fields and tags.

Note that all single-variant types are automatically complete. The input type of an S-NET box is required to be complete, while output types need not be. Hence, the coercion of an input record to the box input type is always unambiguous.

Now, we are ready to define the concept of the box *type signature*, i.e. the type of output records produced in response to a given variant of the input type. The signature of a box which takes records of type $T_{in} = \bigcup_{i=1}^n \{v_i\}$ with variants v_i , and which produces records of type $T_{out} = \bigcup_{i=1}^n \tau_i$, where each τ_i is the type produced in response to receiving variant v_i , would look like $T_{in} \rightarrow T_{out}$ in a conventional programming language. By contrast, 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 complete 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 describe above. As we mentioned before, type T_{in} is required to be complete. A formal definition of the syntax of type signatures can be found in Fig. 2.

It should be noted that the completeness requirement is necessitated by our choice of record type as being one of anonymous variants containing named fields. Record types in conventional languages are based on named variants thus allowing identification by variant name (which is how S-NET tags can be used as well), whereas in our case identification is primarily by a variant field-set. Whilst the conventional approach completely avoids the variant ambiguity, it

also precludes subtyping on the basis of field names only. For instance, in our example of type `body`, conventional subtyping would preclude the construction of a generic box that alters the body position expressed in terms of fields `x1` and `y1` that are common to all variants. As a result a box would require variants to be identified individually and specifically, even when the processing of fields `x1` and `y1` is the same for all variants, for instance, a coordinate shift `x1->x1+a`, `y1->y1+b`. The conventional OOP approach to this would be via a base class with fields `x1` and `y1` and a method `shift` to be inherited by all subclasses. This restricts the design in that there can be more than one set of common fields (which would require multiple inheritance), but more importantly since the significance of a common group of fields may become apparent only when an entirely new processing box is introduced into a streaming network, and in that case a re-design of the class hierarchy may become necessary. Subtyping by subsetting as introduced in the beginning of this section would allow *a posteriori* introduction of a supertype (equivalent to a base class), and so would eliminate the need for the re-design. The price to pay in implementation is the price of a run-time coercion (i.e. a selective copy of fields, or an extra level of indirection to avoid the need to copy), since it can no longer be assumed that the fields to be processed are necessarily a prefix of the field list.

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

Definition 3 (flow inheritance). Consider an n -variant box $X : v^{[i]} \rightarrow \tau^{[i]}$, where $i \in [1, \dots, n]$ and each output type $\tau^{[i]}$ has m_i variants $\tau^{[i]} = \{w_1^{[i]}, \dots, w_{m_i}^{[i]}\}$. Then for any k and any field or non-binding tag $\phi \notin v^{[k]}$ such that

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

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 preceding (in topological order) variants. 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 name⁴. If the output already contains an identically named field, then that field's value supersedes the inherited one. Also note that the flow inheritance is conservative in that it does not allow extension with a field that name-clashes with a preceding variant even though the field *type* of ϕ may be different from the identically named field of the other variant, which, in principle, could make it possible to resolve the clash. Accordingly, we wrote \sqsubset instead of a less restrictive \sqsubseteq . The reason being that we wish record field-sets to be derivable from the network topology without the knowledge of any field types. That, in turn, enables us to connect the boxes first and then apply an efficient algorithm for field type inference, see section 3.3. Recall that the field name was assumed to incorporate the archetype but not the subtype component of the field type.

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 $\tau^{[i]} = \{w_1^{[i]}, \dots, w_m^{[i]}\}$, $i = 1, \dots, n$.

Note that 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* one and 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 non-redundant, since the redundant rules change nothing in the type transformation defined by it.

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 $v_{T_S(j)}$

⁴ obviously the implementation is free to simply switch references

in topological order as j traverses the index range in ascending order. Here is the summary of the subtyping rules:

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

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 well⁵. 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

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

⁵ Note that this rule accounts for a newly introduced variant having extra fields over an existing one, so that these fields would have been flow inherited given the original type.

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

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

and

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

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 typing rules 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:

Definition 5 (monotonicity). Let σv_i be the set of indices j of all supertypes $v_j \sqsupseteq v_i$ of v_i in the box X type signature $X : v^{[i]} \rightarrow \tau^{[i]}$. The signature of X is considered monotonic iff

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

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. There is of course no guarantee of value consistency, for instance a monotonic box 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 desirable.

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 easy to state such constraints explicitly as a restriction on the choice of ϕ , and where appropriate output τ , but we shall not focus on this straightforward modification in this paper.

Finally, recall that the input types of boxes are required to be complete. Our approach to completeness is slightly different to 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 chose 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 nondeterminism, 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 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  $\tau_*^{[i_1, 2]} = \{w \cup (v^{[k]} \setminus v^{[i_2, 1]}) \mid w \in \tau^{[i_1, 2]}\}$ 
    elseif  $\tau^{[k]} \sqsubseteq (\tau^{[i_1]} \cup \tau^{[i_2]})$ 
      continue
    else
      fail
  end
until the signature is monotonic

```

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 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.

2.2 Network Specification

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 behaviour of boxes. This is left to existing computation or box languages like C or SAC. Fig. 3 provides a definition of core S-NET syntax.

$$\begin{aligned}
 SNet &\Rightarrow Net \mid Box \\
 Box &\Rightarrow \mathbf{box} \ BoxName \ (\ BoxSignature \) \ [BoxBody] \\
 BoxSignature &\Rightarrow Variant \ \rightarrow \ Type \\
 BoxBody &\Rightarrow \{ \ \lll \ LanguageName \ | \ LanguageCode \ \ggg \ \} \\
 Net &\Rightarrow \mathbf{net} \ NetName \ \{ \ [SNet]^* \ \} \ NetConnect
 \end{aligned}$$

Fig. 3. Grammar of S-NET specifications. The non-terminal symbols *BoxName* and *NetName* refer to identifiers and are only distinguished for the purpose of clarity.

A box in S-NET is declared by the key word **box** followed by the box name and the box signature, as defined in Fig. ???. The box signature is a simplified form of a type signature that consists of a single variant–type mapping. It serves as the sole specification of a box’s extensional behaviour;

As pointed out above, the specification of the intensional behaviour of a box is outside the scope of S-NET. In general, the code for a 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.

Any box is a (primitive) network by itself. However, non-primitive networks are specified using the key word **net** followed by a network name and a sequence of local (sub-)network definitions and box declarations. Unlike boxes, networks come without a type signature. Type signatures of networks are inferred by the S-NET compiler. A network definition is completed by a specification of the

interconnection topology between the various boxes and subnetworks. Fig. 5 defines the concrete syntax.

```

NetConnect  ⇒ connect SNetExpr

SNetExpr   ⇒ Primitive | SNetInstance | Composite

Primitive  ⇒ Driver | Link | Plug | Sync

Driver     ⇒ [-

Link       ⇒ --

Plug       ⇒ -]

Sync       ⇒ sync Variant with Variant SyncTags

SyncTags   ⇒ [ out TagName ] [ ovfl TagName ] [ fp TagName ]

SNetInstance ⇒ SNetName [ ( Parameter [ ; Parameter ]* ) ]

SNetName   ⇒ BoxName | NetName

Parameter  ⇒ FieldName := ExternalExpr

ExternalExpr ⇒ <<< LanguageName | ForeignExpr >>>

```

Fig. 4. Grammar of S-NET network topology specifications

A network topology specification is initiated with the key word **connect** followed by an S-NET expression, which can be either a primitive box, an instance of a defined box or a subnet, or a composition of S-NET expressions using S-NET topology combinators. S-NET provides four different primitive boxes: *driver*, *link*, *plug*, and *sync*.

1. The driver [- has the type signature $\perp \rightarrow \top$; it produces a stream of empty records. The rationale behind the driver is that all user-defined boxes only act upon receiving data. So, the driver primitive enables us to initiate activity.
2. The link -- realises a simple identity function (defined as $\text{id } x = x$) with the type signature $\top \rightarrow \top$. Due to the flow inheritance rule described in Section 2.1 it can be used in place of the identity function with type $t \rightarrow t$ for any type t free of binding tags.
3. The plug -] has type signature $\top \rightarrow \perp$. It accepts any data and produces absolutely nothing.
4. The synchro-cell is the only “stateful” box in the system. It has storage for exactly one record, which matches one of the given variants. When a

matching record arrives at the synchro-cell, it is kept in this storage. Any record arriving thereafter that matches the same variant as the first one is immediately passed through the synchro-cell. If the `ovf1` option is present, an additional overflow tag is added to each record passed through. If a record arrives that matches the other variant specification, it is combined with the stored record and passed through, i.e., the fields of the two records are merged. In the presence of the `out` option the given tag is added to the output record. A synchro-cell may have a fixed-point, which is defined by the `fp` option (see below).

Instances of S-NET networks and boxes refer to preceding `net` and `box` specifications by their name. Instances may be parameterised. If so, the name is followed by a sequence of parameter definitions within brackets. Any record entering a parameterised subnetwork or box is extended by the given fields. Since S-NET as a pure coordination language has no notion of values, it cannot be used to specify values for these fields. Instead, we employ a box language similar to the definition of boxes themselves. The defining expression is specified in any supported box language. In order to properly separate S-NET code from foreign language code, we again use the separator symbols `<<<` and `>>>` and provide a foreign language identification separated from the code by a bar. These expressions are compiled using an appropriate foreign language compiler and evaluated during network setup.

$$\begin{aligned}
 \textit{Composite} &\Rightarrow \textit{Serial} \mid \textit{Closure} \mid \textit{Choice} \mid \textit{Splitter} \\
 &\quad \mid (\textit{Composite}) \\
 \textit{Serial} &\Rightarrow \textit{SNetExpr} \dots \textit{SNetExpr} \\
 \textit{Closure} &\Rightarrow \textit{SNetExpr} * \\
 \textit{Choice} &\Rightarrow \textit{SNetExpr} \mid \mid \textit{SNetExpr} \\
 \textit{Splitter} &\Rightarrow \textit{SNetExpr} ! \textit{FieldName}
 \end{aligned}$$

Fig. 5. Grammar of S-NET network combinators

Complex network topologies are formed using the four network combinators defined in Fig. 5. As pointed out in the beginning, these network combinators are very much inspired by Stefanescu’s work.

1. 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.
2. The unary closure operator `*` (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 `{<v>->{<v>}`

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 it becomes a plug.

3. The binary choice combinator $||$ combines its operands in parallel. Any incoming record is non-deterministically sent to one operand whose type signature it matches. The output of the operands is merged into a single stream which becomes the output stream of the result.
4. The binary index split combinator takes a subnetwork or box as its left operand and a field name as its right operand. Like the closure combinator, the index split combinator replicates the network operand, but connects the replicas using the choice operator. The number of replicas used corresponds to the number of possible different values the index field can take. Any incoming record goes to the replica identified by its individual value associated with the index field, i.e., all records which have the same value in the index field are also processed by the same replica. Once again, the output of the replicas is merged into a single stream which becomes the output stream of the result.

Next we introduce a simple semantic formalism and discuss each of the above constructs in detail, focusing primarily on their type transformation.

2.3 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 at the end of section ???. To account for the nondeterminism we add an index to the semantic function \hat{x} . A representative of the family \hat{x}_q is a function that corresponds to a particular choice $q \in E(x)$ in nondeterministic variant matching. We will refer to set $E(x)$ as the *event set* of box \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 $V^\infty = \mathcal{P}^\infty \rightarrow (D \cup \{\epsilon\})$, 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:

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

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

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

2.4 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 $\mathbf{a}..A$ 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 $\mathbf{a}..A$. Let us define the formal semantics of $\mathbf{a}..A$. Formally it is defined thus:

Definition 6 (serial combinator). *The serial combination $\mathbf{S} = \mathbf{a}..A$ 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 $S = a..A$, 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

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

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 6 depicts the set-theoretical relations between inputs and outputs as defined by Eq 1.

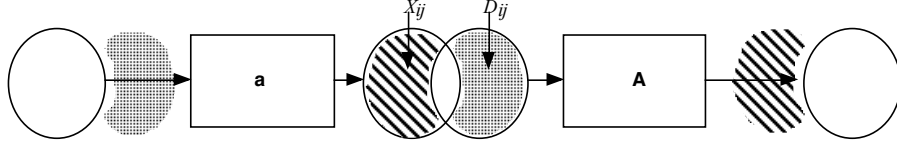


Fig. 6. Flow inheritance through the \dots combinator

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. 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 \mathbf{a} are meant to correspond to perhaps only a single input variant of A . In general it is desirable to be able control the connection between networks when they are combined in series. This is achieved using already familiar binding tags. In their presence, Eq 1 is modified thus:

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

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

2.5 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 7 (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 :*

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

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

$$\check{B}_q^\circ = c_q^{[i_\infty]}, \quad (3)$$

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 non-deterministic 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

B , i.e. solutions of the equation $(\forall q \in E(\check{B}))\check{B}_q x = x$. It is easy to see that the closure of B over $F(\check{B})$ corresponds to the natural intuition of the replica chain introduced at the beginning of the current section. Indeed a sequence of records which at some point reached a fixed point cannot change by going through the network anymore, hence can be “teleported” through the whole infinite chain to the output.

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

In search of a remedy, let us take a closer look at the recurrent process in Definition 7. It is easy to see that if for some $i < i_\infty$, and some q , some $a \in F$ is a prefix of $c_q^{[i]}$, then the eventual fixed point, if it is ever reached, will have a as a prefix, too, thanks to the prefix-monotonicity of semantic functions, which was noted earlier. Indeed, since a fixed point is not sensitive to nondeterminism, $c_q^{[i]}$ will have a as a prefix of the output even though q varies with i . This means that it is possible to output a out of the chain even *before* the fixed point is reached⁶. 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

⁶ this corresponds to “laziness” in functional semantics

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

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

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

First let us introduce an equivalent form of the box type signature. For a box $B : (n, m)v^{[i]} \rightarrow w_j^{[i]}$ 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 $W = \bigcup_{i=1}^m \bigcup_{j=1}^{m_i} w_j^{[i]}$. 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 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 9. 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 1 to the following

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

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

To prove this, we must prove that for all $\phi_{1-3} : Q \times \mathbb{N} \rightarrow Q$, $(\phi_1.. \phi_2).. \phi_3 = \phi_1.. (\phi_2.. \phi_3)$. 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

$$B^* \equiv \underbrace{B..B, \dots, ..B}_{L \text{ times}} ..B^\rightarrow \quad (4)$$

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

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

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

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

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

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

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

and so $B^* = B^{[n]}$. We conclude that the number of iterations does not exceed the size of the resulting signature, which gives the aforementioned linear complexity

bound. Finally observe that the type calculation process can be limited to a reasonable number of output rules, say one hundred: signatures of this size are so unwieldy that they are unlikely to be the result of a deliberate design. Upon reaching the critical size the algorithm could produce appropriate diagnostics (a listing of the rules obtained thus far) and abort.

2.6 Choice Operator

Consider boxes \mathbf{A} : $v_a^{[i]} \rightarrow \tau_a^{[i]}$ and \mathbf{B} : $v_b^{[i]} \rightarrow \tau_b^{[i]}$. The choice combinator $\mathbf{A|B}$ 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 \mathbf{A} and \mathbf{B} are merged into one stream arbitrarily. Here is a formal definition:

Definition 10 (choice). *The choice combination $C = \mathbf{A|B}$ of networks \mathbf{A} and \mathbf{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 \mathbf{A} and \mathbf{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 \mathbf{A} and \mathbf{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:

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

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.

2.7 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 11. *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 .

3 Box Language and Field Subtyping

3.1 Background and Motivation

In the previous section we have described the record processing concept of S-NET and defined record subtyping relations between various components. It is now time to remember that records are not merely sets of field names, which is how we have treated them up to this point, but they are also sets of data associated with fields. Just as record subtyping was important for flexible networking, field subtyping is crucial to flexible processing inside boxes.

S-NET achieves guaranteed field subtype inference by using the concept of homomorphic overloading (h-overloading for short), which is not completely new, although to the best of our knowledge it has not been laid into the foundations of any type system before. The original idea probably goes at least as far back as Reynolds’s paper [19], where he remarked that ”the key to ensuring that implicit conversions ⁷ and generic operators mesh nicely is to require a commutative relationship between implicit conversions and homomorphisms”. To illustrate this, consider the following example. Let a generic operator f be defined on two types: $f_1 : a_1 \rightarrow b_1$ and $f_2 : a_2 \rightarrow b_2$, and let also $a_1 \sqsubset a_2$ and $b_1 \sqsubset b_2$. Under such conditions, the operator application $f x$ is naturally ambiguous. Indeed if $x : a_1$ it has the type a_2 as well so then which of the results $f_1 x$ or $f_2 x$ is expected? The usual principle is to choose the least type, i.e. that of f_1 , so the result is $(f_1 x) : b_1$. However this is coercible to b_2 which gives rise to the question: what is the relationship between the *value* of $f_1 x$ raised to the type b_2 and the value of $f_2 x$?

Reynolds suggests that the results for so overloaded operators must be the same. For instance, if we consider, following [19], $+_1 : (int, int) \rightarrow int$ and $+_2 : (real, real) \rightarrow real$ we find that $x +_1 y$ coerces to type *real* to give precisely the value of $x +_2 y$ (assuming that the available range of integers can be represented as floats without rounding, which is usually the correct assumption). It is easy to see that in this example the coercion from integer to real serves as a homomorphism from $(int, +)$ to $(real, +)$, hence our term “homomorphic overloading”. Paper [19] does not treat this homomorphism as a vehicle of type inference, but rather as a category-theoretical basis for formal semantics of a language that includes generic operators and coercions. By contrast, our concern is exactly the former.

In [20] we showed that a primitive form of h-overloading, where the type signature was constrained to fixed supertypes and subtypes of participating type variables, allowed fast type inference in the presence of unknown external types. The resulting types were inferred as explicit functions of the external types using the longest path algorithm on a constraint graph. We further showed the utility of h-overloading by giving an example of a language for stream processing that benefited from it. However, our solution was not generic, as it limited the variety of overloaded operators to a very restricted set of “offset-homomorphic” operators with a special type signature. Thus arbitrary h-overloading was not

⁷ i.e., coercions

supported, in particular, there was no provision for arbitrary user-defined generic operators.

Paper [17] lifted the restrictions on the h-overloaded signatures and made user-defined families of h-overloaded operators possible, while retaining the original complete inferability of types shown in [20]. We reproduce the main results in this section for completeness.

3.2 H-overloading

We consider field types as being qualified by an ‘archetype’ specification, which is explicitly declared and is not subject to inference (although it is, of course, subject to type checking in a standard way). Here by archetype we mean a set of all subtypes of a well-defined type. For instance, numbers form an archetype with the usual subtyping into integers, reals and complex numbers; pairs of numbers form an archetype which contains a lattice of subtypes, etc.

One archetype may qualify several type attributes at the same time. For instance, numerical arrays can be assumed to have the following attribute structure:

$$narray(etype, rank),$$

where *narray* is an archetype of numerical arrays, which is declared, *etype* is the type of the array element taken from the subtyping hierarchy $int \sqsubset real \sqsubset complex$ and *rank* is the number of array dimensions taken from the hierarchy $0 < 1 < \dots < r_{max}$, where the coercion from lower to higher rank is achieved by infinite replication of the corresponding array in the extra dimensions. This archetype was assumed in [20] in defining a stream processing language, where all operators were overloaded homomorphically in *etype* and *rank*. Another example could be the string archetype: *text(len)*, where *len* is the maximum size of the string, with obvious subtyping. Our subtyping scheme is, at the moment, first-order as we do not allow functional subtyping, the reason being that contravariance of function-argument types destroys the semiring construction described in Section 3, making type inference inefficient. This circumstance prevents our typing scheme from being used in a general functional language. We do nevertheless take full account of contravariance of non-functional types, making our approach applicable to first-order, single assignment languages, such as SAC[21] and ASTL[15]. Here contravariance manifests itself in the *downward* coercion of an assignment target and is the reason that the least type of a variable is required to be sufficiently high.

In this paper it is assumed that the archetype qualifiers of all (sub)expressions in a program to have been deduced from the archetype declarations and the program text, so that they can be omitted from type signatures without creating an ambiguity. We also assume that two types can be in a subtype relation only if they come from the same archetype; in this sense all archetypes are disjoint. An *n*-ary operator is assumed to act on the Cartesian product of types, on which subtyping is defined in the standard way, i.e. component-wise.

Our focus will be on the inference of the least permissible types in a program where all operator overloads are required to satisfy the following

Homomorphism restriction *For any (overloaded) operator F , an instance $F_2 : a_2 \rightarrow b_2$ is said to be homomorphic to an instance $F_1 : a_1 \rightarrow b_1$ iff $a_1 \sqsubseteq a_2$, $b_1 \sqsubseteq b_2$ and $(\forall x : a_2) b_{21} F_1 x = F_2 a_{21} x$, where a_{21} is the type coercion $a_1 \rightarrow a_2$ and b_{21} is the coercion $b_1 \rightarrow b_2$. For any overloaded operator F and any pair of its instances $F_{1,2}$ having identically qualified signatures, one instance must be homomorphic to the other.*

Proposition 2.1 *The set of identically qualified instances of an overloaded operator that satisfies the homomorphism restriction is linearly ordered.*

This follows from the fact that homomorphism is an antisymmetric relation, which is also transitive since the coercions are compositional, i.e. $(\forall t_1 t_2 t_3 : t_1 \sqsubseteq t_2 \sqsubseteq t_3) c_{31} = c_{32} \circ c_{21}$, where c_{ij} is the coercion $t_j \rightarrow t_i$). Note that the linear order of instances induces a linear order on the operand and result subtypes. This does not mean that the subtyping structure of an archetype must be a chain; it only has to *contain* a chain for every overloaded operator family defined on it. Thus, different operator families can potentially use different chains within the archetype without violating the homomorphism restriction. For any h-overloaded n -ary operator family F with k instances, we will write its type signature as follows: $F : \omega_1 \times \omega_2 \times \dots \times \omega_n \rightarrow \omega_0$, where all ω_i are chains of length k in their respective archetypes. The potential confusion with the type signature of a single operator where ω_i are sets of *values* will be avoided by using small Greek letters only for chains of types. A type signature in this form does not by itself define the relationship between the output type of the operator family and its input types, it only defines the ranges of those types within their corresponding archetypes.

The homomorphic restriction has two important consequences. Firstly, it completely disambiguates operator application: $F x$ can always be interpreted as the application of the *lowest* instance of F compatible with the type of x . If the programmer meant a higher instance and applied a further operator to the result assuming that type, this is not a problem, since the result of applying the lower instance is coercible to the output type of the higher one, *yielding exactly the same value*.

Secondly, since Proposition 2.1 places the input and output types on chains in subtyping orders, in any well-typed expression the output type chain ω_1 of an operator F_1 belonging to the expression must *mesh* with the input chain ω_2 of the next operator F_2 up the expression tree. This means that, firstly, the output archetype of F_1 should be the same as the input archetype of F_2 , which is not our concern since the archetype checking is assumed to have been done. Secondly, at least one element of ω_1 must be a subtype of some element of ω_2 so that the result of F_1 can be coerced to an input type of F_2 . Let x_{\max} be the greatest element of ω_1 coercible to ω_2 :

$$x_{\max} = \max_{\omega_1} \{x \mid (\exists y \in \omega_2) x \sqsubseteq y\}.$$

Then the operator F_1 can be restricted (without loss of generality) to just those overloads for which the output type is at most x_{\max} . On the other hand F_2

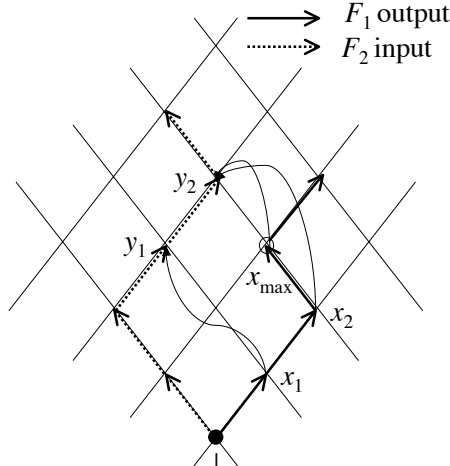


Fig. 7. Meshing type chains

can be restricted, also without loss of generality, to just those overloadings for which the input type is at most x_{\max} (for arity 1). Similar conditions must be satisfied in all operands of F_2 if its arity is greater than 1. Finally, a coercion map $c : \omega'_1 \rightarrow \omega_2$ can be constructed:

$$cx = \min_{\omega_2} \{y \mid x \sqsubseteq y\},$$

where

$$\omega'_1 = \{x \mid x \in \omega_1 \wedge x \sqsubseteq x_{\max}\},$$

and inserted between F_1 and F_2 . It is obvious from the existence of x_{\max} that for any $x \in \omega'_1$ the set on the right-hand side is nonempty, and so the function is well-defined. It is also easy to see that cx is a non-decreasing function. Figure 7 gives an example of two meshed chains, where their common archetype is a lattice. The coercion map is depicted by curvy arrows: $c \perp = \perp$, $cx_1 = y_1$ and $cx_2 = cx_{\max} = y_2$

Another source of coercion is occurrences of program variables. When a variable occurs in a contravariant context, e.g. on the left-hand side of an assignment, the context defines a type chain (corresponding to the top-level operator on the right-hand side) and the type of the variable must be upwards of an output type belonging to that chain. The latter will be subject to type inference and is a priori unknown. Since there can potentially be several contravariant contexts in the program involving the same variable, the variable type must be the least upper bound of the corresponding output types. The variable may also occur in a covariant context, at which point the type derived from the contravariant contexts will be coerced up to the least member of the input type chain assumed by that covariant context.

The difference between meshing a variable with an operator and meshing two operators is subtle. The procedure exemplified in fig 7 effectively maps a

chain onto another chain preserving the order, whereas in the case of variable-to-operator meshing, the least upper bound of the elements of the output chains is represented as a partially ordered subset of the archetype. A coercion map has to map this partially ordered subset onto the input chain of its associated operator. There is a useful factorisation, however, which reduces this kind of meshing to the previous kind. Let us consider the following example program

```
x := F (x,y);
...
x := G y
```

where $F : \alpha_1 \times \alpha_2 \rightarrow \beta$, $G : \alpha_3 \rightarrow \gamma$, the type of x , t_x is given by $t_x \sqsupseteq (b \sqcup g)$, where $b \in \beta$, $g \in \gamma$ are the (unknown) output subtypes of the operators. Note that depending on the shape of the β and γ chains within their common archetype, the least upper bound of b and g can sweep an arbitrary bounded subset, which does not have to be a chain.

For illustration, let us insert coercion functions into the program explicitly:

```
x := CxF F (CFx x, CFy y)
...
x := CxG G (CGy y)
```

Obviously, the output type of CFx is

$$\min_{\alpha_1}\{w \mid w \sqsupseteq (b \sqcup g)\} = \max(\min_{\alpha_1}\{w \mid w \sqsupseteq b\}, \min_{\alpha_1}\{w \mid w \sqsupseteq g\}),$$

which can be simplified to $\max_{\alpha_1}(c_b b, c_g g)$ where c_b and c_g are coercion maps of the kind discussed earlier. Observe that the agreement in type only involves operator output types, b and g with the type of the variable x being directly dependent upon them. Thus the types of program variables can be eliminated from the typing scheme; the output type variables of the corresponding top-level operators hold sufficient information.

In the general case the dependency of an input type of an operator on the output types of other operators via a variable has the form $\max_{i=1}^n (f_i x_i)$ for some n , where f_i is a map from a specific output chain to the common input chain. This construction is very important as it makes it possible to replace f_i by functions mapping a chain *offset* (which is a nonnegative integer representing the distance of a particular type along the chain from its bottom end) onto a chain offset. One can then reason about types solely in terms of those offset numbers. This follows from the factorisation exemplified above, i.e. from the fact that for any chain ω in a partial order P and any bounded set $S \subseteq P$

$$\min_{\omega}\{x \mid x \sqsupseteq (\bigsqcup S)\} = \max_{\omega}\{By \mid y \in S\}$$

where $B : P \rightarrow \omega$ is given by

$$Bx = \min_{\omega}\{y \mid y \sqsupseteq x\}$$

provided that such B exists.

Crucially, under h-overloading, a similar type representation exists for the operators themselves with respect to their multiple operands. It is given by the following

Proposition 2.2. *For any homomorphically-overloaded n -ary operator $F : (a_1, a_2, \dots, a_n) \rightarrow b$, the output type offset \hat{b} can be expressed as a function of the input type offsets \hat{a}_i as follows:*

$$\hat{b}(\hat{a}_1, \hat{a}_2, \dots, \hat{a}_n) = \max(f_1\hat{a}_1, f_2\hat{a}_2, \dots, f_n\hat{a}_n),$$

where $f_i : \mathbb{I}_i \rightarrow \mathbb{I}_0$ are some non-decreasing functions, $1 \leq i \leq n$, $\mathbb{I}_i = [0, k_i]$ is the offset range of the i th operand, k_i is the type offset of the highest overloading in the i th operand relative to the lowest overloading operand type, and $\mathbb{I}_0 = [0, k_0]$ is the output type offset range, with k_0 the difference between the maximum and the minimum output types along the subtype chain.

The proof of Proposition 2.2 follows from the observation that each operand separately demands a certain lowest overloading, and that it is also compatible with all overloadings higher than that one. Consequently, the least output type corresponds to the highest demand, which explains the maximum in the formula. The non-decreasing nature of the functions f_i comes from the fact that raising the type of i th operand along its chain can only make it too high for the current overloading and hence demand a higher one, with a higher output type.

For convenience, we extend the function domains so that $\mathbb{I}_i = \mathbb{I}_0 = \mathbb{Z} \cup \{-\infty, +\infty\} = \mathbb{Z}^\infty$ for all i and assume that $(\forall x < 0, i)f_i x = -\infty$ and $(\forall x > k_i, i)f_i x = +\infty$. The latter assumption models a type error by yielding an infinitely high supertype when the input type range is exceeded, and the former one is motivated by the semiring construction in Section 3. We shall call functions such as f_i and the above-mentioned coercion map *c type maps* when they are expressed in offset form $\mathbb{Z}^\infty \rightarrow \mathbb{Z}^\infty$. The range of $x \geq 0$ in which $f x < \infty$ is the *carrier* of the type map f . Since our type maps are based on finite subtype chains, we shall assume that all carriers are finite. The set of all such functions will be denoted as \mathbb{F} below.

To summarise, the type analysis of a program written in a language with h-homomorphic operators breaks down into the following stages:

1. analysis of the explicit archetype declarations contained in the program.
2. analysis of the operator definitions, including the structure of h-homomorphism within each archetype.
3. archetype checking throughout the program
4. determination of coercion maps induced by meshing, with a subsequent conversion into offset form; elimination of variables by connecting co- and contravariant occurrences by type maps.
5. recording of all type signatures and converting them into offset form; recording of the type maps.
6. subtype inference

Type checking in a language with explicit declarations is well known and does not present a problem. Hence the first three tasks on the list are straightforward.

Item 4 has been explained in previous sections as well as item 5. It is the last stage, item 6, that presents a major challenge, which we focus on in the next section.

3.3 Subtype Inference

The language. To illustrate the subtype inference algorithm, we shall introduce a simple single-assignment language, which models some features of both SAC[21] and ASTL [20] (as well as possibly other single-assignment languages where subtyping can be introduced) that are relevant to the type inference method proposed. The syntax of the language is given in figure 3.3, where a single module is defined. A complete program is a set of modules.

```

⟨module⟩ → function ⟨name⟩ ( ⟨par-tuple⟩ ) ⟨body⟩
⟨par-tuple⟩ → ⟨var⟩ [ , ⟨var⟩ ] *
⟨body⟩ → ⟨assig⟩ [ ; ⟨assig⟩ ] *
⟨assig⟩ → ⟨var⟩ [ ⟨selector⟩ ] := ⟨exp⟩
⟨exp⟩ → ⟨var⟩ | ⟨op⟩ ⟨exp-tuple⟩ | ⟨function⟩ ⟨exp-tuple⟩
⟨exp-tuple⟩ → ( ⟨exp⟩ [ , ⟨exp⟩ ] * )
⟨function⟩ → ⟨id⟩

```

Fig. 8. The model language

A module defines a function whose body is a sets of assignments. An assignment assigns the value of the right-hand side to the object signified by the left-hand side. The optional selector defines which part of the object has been assigned a value, for example which index of an array. All such parts are required to have the same subtype, i.e. the data structure is assumed to be homogeneous. There is a semantic constraint that the the selectors applied to the same variable define the partitioning of that variable associated object, i.e. the selected parts are disjoint and the coverage is complete, hence a single-assignment semantics is assured. SAC achieves this by syntactic means (with a static guarantee), whilst ASTL has a dynamic check for the singleness of assignment, but these details are irrelevant to subtype inference. All that is important for the treatment below is that

1. the same variable can be used repeatedly on the left-hand side of assignment representing different parts of a homogeneous data structure;
2. all these occurrences are type-contravariant, i.e. $\tau_{var} \geq \tau_{RHS}$
3. all occurrences of a variable on the right-hand side are covariant, i.e. $\tau_{var} \leq \tau_{op}$ where τ_{op} is the maximum subtype that the operator applied to the variable in the right-hand side allows it to have.

Each operator $\langle op \rangle$ is predefined with a homomorphic subtype signature defined by Proposition 2.2. Function calls name the function to be called explicitly, and all functions are assumed to be first-order. Function subtype signatures are *not* required to be provided by the programmer but are inferred in the process of subtype inference.

A primary constraint set Subtype inference begins with associating fresh type variables with all subexpressions in the program. In our case, these variables represent type offsets from \mathbb{Z}^∞ rather than type values for the reasons explained earlier. For every operator occurrence, the operator type maps are invoked to produce a type constraint in the form:

$$v_0 = \max_{i=1}^n (f_i v_i),$$

where $v_i, i = 0 \dots n$ are any of the type variables just introduced. The constraints can be broken down into a set of simpler constraints in what we shall call *canonical form*:

$$\tau_0 \geq f_i \tau_i,$$

on the assumption that the minimum type assignment is sought. All canonical constraints in a program constitute the *primary constraint set*. This set can be assumed to contain exactly one constraint for every pair of types a and b . Indeed, if there are two constraints between these types, $a \geq f_1 b$ and $a \geq f_2 b$, then they can be replaced by an equivalent constraint $a \geq f_{1 \oplus 2} b$, where for all $x \in \mathbb{Z}^\infty$, $f_{1 \oplus 2} x = \max(f_1 x, f_2 x) = (f_1 \oplus f_2) x$. (We denote the operator of the pointwise maximum of two functions by \oplus .) On the other hand, if there are no constraints between a and b , then the constraint $a \geq \mathbf{0} b$ can be added, where $\mathbf{0} : \mathbb{Z}^\infty \rightarrow \mathbb{Z}^\infty$ such that for all $x \in \mathbb{Z}^\infty$, $\mathbf{0} x = -\infty$. Thus one can speak of an $n \times n$ constraint matrix C_{ij} defining the primary constraint set for n type variables. Each element of C_{ij} is the function $\mathbb{Z}^\infty \rightarrow \mathbb{Z}^\infty$ that occurs in the constraint between types x_i and x_j in canonical form.

Note that some of the type variable are associated with program variables which are external to the program unit being compiled and which are, consequently, not subject to inference. The purpose of type inference is to express the least type of each program variable as a function of those external types.

Constraint set expansion The simplest type inference procedure would be to initially assign 0 to all type variables associated with internal variables, and then iterate the constraint set until a fixed point is reached or a type variable acquires the value of infinity. In matrix form, we seek a solution to the constraint satisfaction problem $\mathbf{x} = C \mathbf{x}$ as a fixed point of the iterative process:

$$\mathbf{x}^{[0]} = \mathbf{0}; \mathbf{x}^{[k+1]} = C \mathbf{x}^{[k]}.$$

Here $C \mathbf{x}$ denotes $\bigoplus_{i=1}^n C_{ij} x_j$.

The procedure is sound, since at each iteration it delivers a lower bound of all types implied by the primary constraint set. Also, due to the non-decreasing

nature of all matrix elements of C , at each iteration which does not deliver a fixed point, it produces an increased lower bound for at least some type variables. Since the carriers of all matrix elements are finite, a fixed point exists and is reachable. Obviously, the constraint set is satisfiable iff none of the lower bounds delivered at the fixed point is infinite.

This solution has two potential problems. First of all, the number of iterations is only bounded from above by the total length of all type chains, since at each iteration (which does not result in a fixed point) only one type variable has to increase. Secondly, since the numerical values of the external type parameters are unknown, iterations have to be performed with the matrix C by raising it to a power (using function composition as multiplication and \oplus as addition). This by itself is a costly operation, to perform even once.

We propose a more efficient algorithm, based on the algebraic path problem, which we consider next.

Algebraic structure of \mathbb{F} Recall that the elements of the constraint matrix are drawn from the set \mathbb{F} of nondecreasing functions $\mathbb{Z}^\infty \rightarrow \mathbb{Z}^\infty$ that yield $-\infty$ on all $x < 0$ and $+\infty$ on sufficiently large $x \geq 0$. Consider a six-tuple $\Phi = (\mathbb{F}, \oplus, \odot, *, \mathbf{0}, \mathbf{1})$ where \oplus is as defined above, $\odot : \mathbb{F} \times \mathbb{F} \rightarrow \mathbb{F}$ is a function composition, $*$: $\mathbb{F} \rightarrow \mathbb{F}$ is Kleene's star operation:

$$f^* = \mathbf{1} \oplus f \oplus (f \odot f) \oplus (f \odot f \odot f) \oplus \dots,$$

$\mathbf{0} \in \mathbb{F}$ is defined as above and $\mathbf{1} \in \mathbb{F}$ is the identity function⁸: $\mathbf{1}x = x$ for $x \geq 0$, $\mathbf{1}x = -\infty$ otherwise.

Proposition 3.1 \oplus , \odot and $*$ are closed in \mathbb{F} .

Indeed, the \oplus operation is closed in \mathbb{F} since the point-wise maximum of two nondecreasing functions is a nondecreasing function, whose carrier is included in the union of the carriers of the arguments and so is finite. Likewise, the composition of two nondecreasing functions is a nondecreasing function. The behaviour of this function at negative arguments and $\pm\infty$ is proven immediately by substitution; the carrier of the result is the same as that of the first operand, so \odot is closed in \mathbb{F} .

Finally, the star operator is defined in terms of the fixed point of a series, each member of which is computed from elements of \mathbb{F} using the operators \oplus and \odot . Since they are both closed in \mathbb{F} , the star operator itself is closed in \mathbb{F} if the fixed point exists. The fixed point does exist, since the series of partial sums is point-wise nondecreasing and since \mathbb{Z}^∞ includes $+\infty$. In fact, we will show below that the fixed point can be computed in a finite number of steps by an efficient algorithm, which means that the series for the star operator is always finite for any element of \mathbb{F} . This obviates the proof that the star construct is well behaved; such a proof would usually be required for an infinite star series. ‡

⁸ Strictly speaking the identity function is not in \mathbb{F} since it does not have a finite carrier; nor is $\mathbf{0}$. However, we include them in \mathbb{F} as special elements. The use of both $\mathbf{1}$ and $\mathbf{0}$ with \oplus , \odot and $*$ does not lead to further infinite-carrier elements.

Proposition 3.2 $(\mathbb{F}, \oplus, \mathbf{0})$ and $(\mathbb{F}, \odot, \mathbf{1})$ are monoids, the former is commutative. Indeed, function composition is associative and so is point-wise maximum. The elements $\mathbf{0}$ and $\mathbf{1}$ are obviously the identities of the respective operations. ‡

Proposition 3.3 Operation \odot distributes over \oplus both on the left and on the right:

$$a \odot (b \oplus c) = (a \odot b) \oplus (a \odot c) \text{ and } (b \oplus c) \odot a = (b \odot a) \oplus (c \odot a).$$

The proof is by point-wise application, using the nondecreasing nature of functions a , b and c . ‡

Proposition 3.4 $\mathbf{0}$ is a null with respect to \odot : $\mathbf{0} \odot x = x \odot \mathbf{0} = \mathbf{0}$ The proof follows immediately from the construction of the element $\mathbf{0}$. ‡

Propositions 3.1-4 form the proof of the following

Lemma 3.5 Φ is a star semiring.

Inference procedure Now consider the constraint satisfaction problem again. Let us associate every type variable with a vertex of a weighted, directed graph G . Each edge (v_i, v_j, f) of the graph represents the constraint

$$v_i \geq f v_j.$$

Since the (internal) program variables occur in both covariant and contravariant contexts, the graph is not necessarily acyclic, and may contain infinite as well as finite walks. Each walk corresponds to a chain of primary constraints connecting its ends, and hence to a *secondary* constraint corresponding to the (finite or infinite) \odot -product of the weights of the participating edges. The tightest constraint between any types v_i and v_j due to the primary constraint set is the \oplus -sum of the weights of all walks W_{ij} in graph G from vertex i to vertex j :

$$P_{ij} = \bigoplus_{w \in W_{ij}} (\bigodot_{e \in w} f_e),$$

where the selection of edges e from the walk w in the \odot -product is in the walk order. This is a formulation of the classical *algebraic path problem* [22] for the semiring Φ and graph G .

The solution to the algebraic path problem is the matrix P_{ij} of semiring values. We will define an efficient algorithm for its computation below. For now let us assume P_{ij} has been computed, and proceed to the type assignment.

Proposition 3.6. Divide the set of type variables $\{v_k \mid 1 \leq k \leq n\}$, into external ones $k \leq n_e$, which correspond to the program variables from the parameter tuple of the function, see fig 3.3 (and which consequently are not subject to type assignment) and the rest $n_e < k \leq n$. The least type assignment is given by the following formula:

$$v_k = \min_{v_k^*} \{x \mid x \geq \max_{i=1}^{n_e} (P_{ki} v_i)\} = P_{kk} \odot \max_{i=1}^{n_e} (P_{ki} v_i),$$

where v_k^* is the set of solutions of the equation $x = P_{kk} x$. The outline of the proof is as follows. First of all, observe that any type assignment for the variable

v_k has to satisfy the secondary type constraint $v_k \geq P_{kk}v_k$. Since $P_{kk} \geq \mathbf{1}$ point-wise (since at any rate $v_k \geq v_k$), only the fixed points of P_{kk} are suitable as potential type assignments for v_k . Secondly, v_k must be large enough to satisfy all primary and secondary constraints induced by the external types, which explains the above formula. The third part of the equation is due to the fact that P_{kk} is the point-wise maximum of all cyclic chains on vertex k , hence $P_{kk} \odot P_{kk} = P_{kk}$ and so, for all $x \in \mathbb{Z}^\infty$, $P_{kk}(P_{kk}x) = P_{kk}x$. This means that $P_{kk}x$ is a fixed point of P_{kk} . The fact that this fixed point is the least one greater than or equal to x is due to $P_{kk} \geq \mathbf{1}$ point-wise and to its nondecreasing nature.

One might think that v_k must be large enough to satisfy the constraint induced by any other internal variable v_j : $v_k \geq (P_{kj}v_j)$. We claim that this happens automatically. Indeed, assume the contrary, i.e. that for some j , $v_k < (P_{kj}v_j)$. By the above assignment $v_j \geq P_{jj} \odot P_{ji}v_i$ (recall that P_{jj} is a nondecreasing function, so it distributes over the maximum), and so $v_k < P_{kj} \odot P_{jj} \odot P_{ji}v_i$ for any external v_i . The right-hand side reduces to $P_{ki}v_i$ by definition of P and semiring distributivity. Hence $v_k < P_{ki}v_i$, which contradicts our type assignment and proves its validity. ‡

3.4 Implementation

The type inference method proposed in the previous section requires the ability to compute the algebraic path matrix P_{ij} efficiently. This is achieved by Kleene's algorithm in $O(n^3)$ semiring operations using the following iterative process. Set the initial value $P_{ij}^{[0]}$ according to the primary constraint graph. For any edges (i, j) not found in the graph set $P_{ij}^{[0]} = \mathbf{0}$. For $k = 1 \dots n$ do:

$$(\forall i, j) P_{ij}^{[k]} = P_{ij}^{[k-1]} \oplus (P_{ik}^{[k-1]} \odot (P_{kk}^{[k-1]})^* \odot P_{kj}^{[k-1]})$$

The solution is $P_{ij} = P_{ij}^{[n]}$.

At each iteration, the algorithm requires $2N^2$ semiring multiplications and N^2 semiring additions as well as one star operation. We consider the implementation of those next.

We propose the representation of semiring elements as sorted lists of pairs (a, v) where $a \geq 0$ is the value of the function argument and v is its result. The list is sorted in the ascending order of a . The value of the function for the arguments greater than the last one listed are assumed to be $+\infty$. The empty list corresponds to the maximum element of Φ , ϕ_{\max} : $(\forall x \in \Phi) x \oplus \phi_{\max} = \phi_{\max}$. The elements $\mathbf{0}$ and $\mathbf{1}$ are represented as special values recognised by all three operators.

It is easy to see that the \odot operation in this representation is little more than the classical database *join* of the operands equating the v field of the first operand and the a field of the second; it yields a sorted list as a result. Both source lists are only traversed once, thanks to the nondecreasing nature of the semiring elements and the fact that any emerging lists are already sorted. The \oplus operator is implemented as a join in the field a of both lists followed by the

pointwise maximum of the corresponding v fields. Of course the a field does not even need to be stored, as it contains merely the sequential number of the list element.

The star operator is slightly trickier to implement. Observe that since Φ is idempotent (i.e., $(\forall x \in \Phi)x \oplus x = x$), $(f \oplus \mathbf{1})^* = f^*$, which can be proven by substitution. Hence without any loss of generality we can assume that $f x \geq x$ for all nonnegative x . The first step is to identify closed intervals of x , $[b_i, e_i]$ such that:

$$\begin{aligned} f(b_i - 1) &\leq b_i - 1, \\ f k &> k \text{ for } b_i \leq k < e_i \text{ and} \\ f e_i &= e_i. \end{aligned}$$

If no such interval exists, it is easy to see that $f x = x$ for all $x \geq 0$, in which case $f^* = f = \mathbf{1}$. Indeed, since for any $f \in \Phi$, $f(-1) = -\infty$ and $f(+\infty) = +\infty$, there is at least one suitable pair of e_i and b_i . Hence the middle condition is not satisfied, which means that for all k $f k \leq k$, hence $f \oplus \mathbf{1} = \mathbf{1}$.

In the general case, the carrier of f is partitioned into one or more closed intervals of the above sort with possibly intervals where $f x \leq x$ occurring in between those. We then apply the following

Proposition 4.1. *Within each interval $[b_i, e_i]$, $f^* x = e_i$.*

Indeed acting f on any point within the interval will produce a greater result not exceeding e_i (which is the value of a nondecreasing function at the right end of the interval where it is nondecreasing, hence the maximum). Therefore, repeated application of f will eventually reach e_i which is a fixed point.

The star algorithm should consequently proceed in two passes. In the first pass, the closed intervals are identified by scanning the list and comparing the current and previous elements. At the same time any elements for which $v < a$ are adjusted to $v = a$. In the second pass, the answer is computed by filling up the intervals with their final value of a . This is best accomplished by placing the list elements on top of a stack during the first pass, and reading them off the top of the stack in the second, so that the ends of intervals could propagate backwards.

One last observation: in the previous section we stated that f^* maps any x to the nearest fixed point equal or exceeding x . Clearly our algorithm has this property.

From the description of the semiring algorithms, it is clear that their computational cost is $O(L)$ where L is the length of the longest chain in the subtyping system. An obvious optimisation would be to exploit the fact that there are usually much fewer instances to an operator than there are different subtypes in a type. Consequently, the type maps are likely to be step functions with many different a corresponding to the same v . The above algorithms can easily be modified for such functions: only the first record with the same v need be kept, the join algorithm must compare for \geq instead of equality, etc. As a result the computational cost of semiring operations could be reduced to $O(V)$ where V is the maximum number of overloads defined for any operator in the program.

3.5 Linkage

The inference procedure described in Section 3.3 delivers the subtypes of all variables (including the function result) in terms of the types of the function parameter tuple. It can only do that if the subtype transformation of all function applications in the function body is known. If recursive functions are not present, this is easily achievable as the invocation graph for the program is acyclic, and so functions must exist that call no further functions. Inference starts with those. When their subtype signatures (the mapping of the parameter tuple types onto the function result type) becomes known, they can be used to perform subtype inference in functions that depend on those, etc., until inference for the whole program is complete.

Recursive functions can be included into the subtype inference framework by applying the following fixed-point calculation.

1. Assume that each function $F_j : (p_1, \dots, p_{m_j}) \rightarrow t_j$ returns the lowest subtype of the corresponding supertype: $t_j = \max_k f_{jk} p_k$ with all $f_{jk} = f_{jk}^{[0]} = \mathbf{0}$
2. Perform subtype inference in each of the bodies using the above assumption for any function applications occurring in it. The result is a new approximation $f_{jk}^{[1]}$.
3. Iterate step 2 until $(\forall jk) f_{jk}^{[m+1]} = f_{jk}^{[m]}$

The existence of the fixed point follows from the fact that the "sum of product" formula for P_{ij} from Proposition 3.6 is monotonic with respect to all f_i . Indeed, define partial order on Φ thus: $f_1 \sqsubseteq f_2$ iff $f_1 \oplus f_2 = f_2$. Then using semiring distributivity show that for any edge k , its weighting f_k , and any semiring element f , if $f_k \sqsubseteq f$ then $(\forall ij) P_{ij}(f_k) \sqsubseteq P_{ij}(f)$. Informally, this means that if a type function f is replaced by a pointwise same-or-higher function, this can only make any other type function in the constraint set pointwise same-or-higher. This, of course, automatically guarantees a fixed point if the codomain of all type functions here is finite, which is the case.

Note that the fixed-point procedure does not necessarily require a repeated solution of the entire algebraic path problem. Each iteration only changes weightings on a few edges corresponding to function applications. The other edges that correspond to the operator applications retain their weightings, and in any practical program these would be the majority. Consequently, one can introduce a vertex enumeration that leaves all function-application vertices at the end. Only those vertices will require Kleene's algorithm iterations to be re-done, thus giving a cost estimate of $O(mN^2) \ll N^3$, where $m \ll N$ is the number of type variables associated with function applications in the expression tree. In a practical system this estimate can be reduced further by taking into account the locality of type dependencies in the abstract syntax tree.

4 Related Work

The issue of type inference with atomic subtyping has a long history. We cite papers [23, 24, 19, 25, 26] as ones where foundation work was done. Of these,

paper [26] 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 by us in [20]. Our concept of type homomorphism is consonant to Lievant’s idea of “discrete polymorphism” proposed in [27], 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 [28], 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 [28] 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 [28] 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.

5 Conclusions and Future Work

A type inference solution for a general first-order, atomic subtyping with homomorphic overloading has been proposed. We have shown that after archetype checking, an h-overloaded operator produces type constraints characterisable by nondecreasing functions on the expanded integer set. A star semiring Φ was proposed to capture algebraic properties of such functions. Using Φ , we have built a type inference procedure based on Kleene’s algorithm. The procedure infers the least types of all internal variables in the program as explicit functions of the external types.

Future work will proceed towards introducing homomorphic subtyping to SAC and implementing the subtype inference algorithm. More thought is required to improve the efficiency of linking, perhaps by using some heuristics for the first approximation. It would also be interesting to tackle higher-order functional subtyping for which the present technique is not immediately applicable.

References

1. 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

2. Ashcroft, E.A., Wadge, W.W.: Lucid, a nonprocedural language with iteration. *Communications of the ACM* **20** (1977) 519–526
3. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE* **79** (1991) 1305–1320
4. Berry, G., Gonthier, G.: The esternel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming* **19** (1992) 87–152
5. Binder, J.: Safety-critical software for aerospace systems. *Aerospace America* (2004) 26–27
6. 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
7. 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
8. 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)
9. Stephens, R.: A survey of stream processing. *Acta Informatica* **34** (1997) 491–541
10. *et al*, B.B.: 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
11. 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
12. Stefanescu, G.: An algebraic theory of flowchart schemes. In Franchi-Zanettacci, P., ed.: *Proceedings 11th Colloquium on Trees in Algebra and Programming*, Nice, France, 1986. Volume LNCS 214., Springer-Verlag (1986) 60–73
13. Broy, M., Stefanescu, G.: The algebra of stream processing functions. *Theoretical Computer Science* (2001) 99–129
14. Stefanescu, G.: *Network Algebra.* Springer-Verlag (2000)
15. Shafarenko, A.: Stream processing on the grid: an array stream transforming language. In: *SNPD.* (2003) 268–276
16. Shafarenko, A.: Coercion as homomorphism: type inference in a system with subtyping and overloading. In: *Proceedings of the Fourth ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, ACM Press (2002) 14–25
17. Shafarenko, A., Scholz, S.B.: General homomorphic overloading. In: *Implementation and Application of Functional Languages. 16th International Workshop, IFL 2004, Lübeck, Germany, September 2004. Revised Selected Papers.* LNCS'3474, Springer Verlag (2004) 195–210
18. van Rossum, G.: *The Python Language Reference Manual.* Network Theory Ltd (2003)
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. 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

21. Scholz, S.B.: Single assignment c: efficient support for high-level array operations in a functional setting. *J. Funct. Program.* **13** (2003) 1005–1059
22. Rote: Path problems in graphs. In: G. Tinhofer, E. Mayr, H. Noltemeier, and M. M. Syslo (eds.) in cooperation with R. Albrecht, *Computational Graphs Theory*, Springer-Verlag Computing Supplementum 7. Springer-Verlag (1990) 155–198
23. Cardelli, L., Wegner, P.: On understanding types, data abstraction, and polymorphism. *Computing Surveys* **4** (1985) 471–522
24. Mitchell, J.: Type inference with simple subtypes. *Journal of Functional Programming* **1** (1991) 245–285
25. Fuh, Y.C.C., Mishra, P.: Type inference with subtypes. *Theoretical Computer Science* **73** (1990) 155–175
26. 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
27. Lievant, D.: Discrete polymorphism. In: *Proc. 1990 ACM Conference on LISP and Functional Programming*, June 27-29, 1990, Nice, France. (1990.) 288–297
28. Rehof, J., Mogensen, T.: Tractable constraints in finite semilattices. *Science of Computer Programming* **35** (1999) 191–221