Extending the S-Net Type System

Haoxuan Cai and Susan Eisenbach
Department of Computing, Imperial College London

Alex Shaferenko and Clemens Grelck
Department of Computer Science, Hertfordshire University

17th September 2007

Abstract

In this paper we discuss the results of evaluating the S-Net type system. We found some problems that would lead to inefficient S-Net compiler implementation, or even incorrect implementations of S-Net programs. We solved the problems by extending the type system so that it can capture more properties that S-Net was designed with.

1 Introduction

S-Net is not only a language: it comes with a runtime system which implements the various language features of S-Net and controls the setup of streaming networks and the orderly cooperative behaviour of asynchronous components [2].

1.1 S-Net Runtime Properties

The S-Net runtime system features 3 outstanding properties, which are required by the S-Net language: flow inheritance, overwriting, and nondeterministic best match. They will be called 'runtime properties' in this paper because of the tight relationship with the runtime system, although they're design specifications of by the language.

1.1.1 Flow Inheritance

Suppose a network defines its type signature as a single type mapping $v \to w$ where both sides are nonvariant. If a record of type v' enters the network, where v' is a subtype of v (denoted as $v' \sqsubseteq v$), then there will be some extra record labels $\delta = v' \backslash v$ unused by the network. Flow inheritance causes these extras to be flown through the network unchanged and merged into what the network outputs, i.e. the actual output becomes $w' = w \cup \delta$.

1.1.2 Overwriting

The flow-inherited labels may clash with the output specified by the type mapping. In this case, the flow-inherited labels will be overwritten by the output labels, but in mathematic expressions the output is still $w' = w \cup \delta$ where $\delta = v' \backslash v$.

1.1.3 Nondeterministic Best Match

In the cases where the input record to a network matches multiple alternative type mappings, the one that matches the most labels will be chosen. This is the rule of best match. In case the input matches several type mappings equally best, the runtime will choose one according to other, mostly nondeterministic, factors such as rolling a dice or, more practically, checking which participating processor has the most resource available. This is nondeterminism.

1.2 Type Inference and S-Net

S-Net allows programmers to omit type signatures for network declarations. But at runtime, type information is required to decide where and how to send a specific input record. It is also essential to be able to check whether a programmer-provided type signature is valid. These suggest the application of type inference in S-Net, where the network type signatures are inferred from the lowest level boxes and how they're combined.

The type inference stage of the S-Net compiler is to perform this task. It is preceded by the type flattening stage which introduces additional networks to the topology so that each network contains only one box, one filter, one synchrocell, or one combinator [6]. This also simplifies the tasks to be done by the type inference stage, which for each network is either to raise the type signature from the underlying network, or to infer it for one of the four kinds of combinations from the type signatures of the subnetworks.

1.3 Algebraic S-Net Type System

The type inference stage is supported by the S-Net type system, which uses algebraic forms to describe types and type mappings, such as the (nonvariant) record types as sets over the set of labels:

$$RecType = \mathbf{F}Label,$$

and the subtype relationship:

$$v \sqsubseteq w \iff BT(v) = BT(w) \wedge v \supseteq w,$$

where $BT : RecType \mapsto RecType$ is a function that selects all binding tags in a nonvariant record type.

With algebraic representations, the S-Net semantics can be captured in mathematic theories, and thus type inference rules can also be described using algebraic formulae.

1.4 S-Net Type Inference Formulae

In [3] one can also find the formulae for type-inferring the four kinds of network combinations, which are syntactically different, and sometimes inequivalent with the ones shown here. This is because we focus more on implementation than the semantics, and, after careful discussions, some slight improvements were introduced, which could not be included in the S-Net draft reports in time. Before stepping into the formulae, we would like to update the definition of type signature normalisation as a prerequisite.

1.4.1 Type Signature Normalisation

A type signature is **normalised** [4] if there are no variants on the left hand side of any type mappings in the signature. The **normalisation** turns every 'complex' type mapping $V \to W$ into a set of 'simple' type mappings:

$$\{v \to W | v \in V\}.$$

In the expression above and throughout the rest of this paper, we'll use capital letters to denote a (possibly variant) record type with a mathematical meaning of a set of nonvariant record types (a nonvariant record type can be seen as a variant record type with only one variant, so that the capital letters cover the term 'record type' completely), and small letters to denote a nonvariant record type with a mathematical meaning of a set of record labels.

However, the normalisation as in [4] does not normalise the right hand sides. We hereby define another **normalisation** which will be used by this paper as follows.

Definition. A *normalised* type signature is one that uses only nonvariant record types on both sides of every member type mapping. *Normalisation* of a type signature τ is a process that produces a normalised type signature $NTypeSig(\tau)$ which is defined as

$$NTypeSig(\tau) = \bigcup_{(V \to W) \in \tau} \{v \to w | v \in V \wedge w \in W\}.$$

The normalisation process splits all the variants on both LHS and RHS into separate type mappings, making the type signatures easier to be represented and manipulated in the type inference formulae.

1.4.2 Type Inference for Serial Combinator (..)

If we ignore the subtype relationship confinement and the best match rule, serially combining two networks means arbitrarily choosing a type mapping to use for each of the operand networks. For each of these type mapping pairs, we then check if it's possible, and if so, what's the combined type mapping. For simplicity we'll assume the type mapping chosen from the first network to be $v_1 \to w_1$ and the second $v_2 \to w_2$. The checks and calculations are as follows:

1. Check that the binding tags of w_1 and v_2 match. If they don't, outputs of the first network will not be accepted by the second, and this pair will not work.

2. Add the labels in v_2 but not in w_1 into v_1, so they will be flow-inherited to the output of the first network and become the input of the second. But if any of these labels is in v_1, it'll be consumed and not emitted by the first network, so the second one will never receive a valid input, and this pair is a no-go. Moreover, if the augmented input record type matches another type mapping from the first network better, the best match rule will prevent this pair from being chosen.

3. The actual input to the second network is v_2 plus the additional labels from w_1, i.e. $w_1 \cup v_2$. The best match rule also applies here.

4. The extra labels in w_1 not consumed by the second network is then flow-inherited to the output of the combined network.

Formally, if in a serial combination expression, the left operand network has a *normalised* type signature α, and the right operand network β, then the type signature of the combined network is

$$DotDot(\alpha, \beta) = \bigcup_{(v_1 \to w_1) \in \alpha} \bigcup_{(v_2 \to w_2) \in \beta} Join(\alpha, \beta, v_1, w_1, v_2, w_2),$$

where

$$Join(\alpha, \beta, v_1, w_1, v_2, w_2) =$$

$$\begin{cases} \phi, & BT(w_1) \neq BT(v_2) \vee (v_2 \backslash w_1) \cap v_1 \neq \phi \vee \\ & (\exists (v \to w) \in \alpha. [v_1 \cup (v_2 \backslash w_1)] \sqsubseteq v \wedge |v| > |v_1|) \vee \\ & (\exists (v \to w) \in \beta. (w_1 \cup v_2) \sqsubseteq v \wedge |v| > |v_2|); \\ \{[v_1 \cup (v_2 \backslash w_1)] & \\ \quad \to [w_2 \cup (w_1 \backslash v_2)]\}, & \text{otherwise.} \end{cases}$$

1.4.3 Type Inference for Parallel Combinator (|)

Parallel combinators are nothing but providing additional choices. The best match rule and the nondeterminism applies to the combined type signature. There are some constraints that will restrict certain kinds of type signatures to be combined in parallel but they're beyond the scope of this paper. The type signature of a parallel composition of two networks with type signatures α and β, as far as this paper is concerned, is

$$Bar(\alpha, \beta) = \alpha \cup \beta.$$

1.4.4 Type Inference for Serial Replication Combinator (*)

The serial replication of any network `a` is internally an indefinitely long list of binary serial combinations:

`a..a..a..a` ...

The chain terminates when the intermediate record matches any of the record types listed as the right operand of the `*` operator. We'll denote the *normalised* type signature of the network being replicated as α and the collection of termination record types as T (using a variant record type notation since it's perfectly a set of nonvariant record types). The following predicate is needed to check whether a record type matches any of the termination pattern:

$$IsTerm(t, T) \iff \exists v \in T.\, t \sqsubseteq v.$$

We also need two helper functions:

$$Term(\tau, T) = \{(v \to w) | (v \to w) \in \tau \wedge \neg IsTerm(v, T) \wedge IsTerm(w, T)\}$$

and

$$Nonterm(\tau, T) = \{(v \to w) | (v \to w) \in \tau \wedge \neg IsTerm(v, T) \wedge \neg IsTerm(w, T)\}$$

which select terminating type mappings and nonterminating type mappings, respectively. Note that termination patterns found on the LHS are excluded from both of these selections.

We can now define the type signature of a serial replication over a network with the normalised type signature α using the termination patterns T as

$$Star(\alpha, T) = PreTerm(T) \cup star(\alpha, \alpha, T),$$

where

$$PreTerm(T) = \{(t \to t) | t \in T\},$$

and

$$star(\tau, \alpha, T) = Term(\tau, T) \cup star(DotDot(Nonterm(\tau, T), \alpha), \alpha, T).$$

At first glance it might be shocking that it's a nonterminating recursion, but it has been proven that the type signature will stop growing at some stage after a number of serial combinations, which is true if the serial combinators are type-wise associative (i.e. `(x..y)..z` has the same type signature as `x..(y..z)`), which has also been proven true [7]. Once the type signature stops growing, the union operations will not introduce more type mappings into the resulting signature and the recursion can hence terminate.

1.4.5 Type Inference for Parallel Replication Combinator (!)

The parallel replication combinator takes a network as its left operand and one or two tags (simple or binding) as its right operand. If the tag or tags are grouped into a set *tags*, the type signature for the parallel replication of a network with a normalised type signature α is

$$Ex(\alpha, tags) = \bigcup_{(v \to w) \in \alpha} Merge(\alpha, v, w, tags),$$

where

$$Merge(\alpha, v, w, tags) = \begin{cases} \phi, & BT(tags) \not\subseteq BT(v) \vee \\ & \exists (v_0 \to w_0) \in \alpha. \\ & \quad (v \cup tags) \sqsubseteq v_0 \wedge |v_0| > |v|; \\ \{(v \cup tags) \to & \\ \quad (w \cup (tags \backslash v))\}, & \text{otherwise.} \end{cases}$$

2 The Need for Completion

2.1 Discovery of the Need

Suppose we have the following network and we are to type-infer it:

```
net x {
    net a ( {a} -> {b}, {b} -> {c}, {c} -> {d} ) ...;
} connect (a..a);
```

The type signature is by itself normalised. We can directly apply the formula for type-inferring serial combinators found in Section 1.4, and the following will result:

```
( {a} -> {c},
  {a,c} -> {b,d},
  {b,a} -> {c,b},
  {b} -> {d},
  {c,b} -> {d,c} )
```

... *from* 1×2
... *from* 1×3, *also* 3×1
... *from* 2×1
... *from* 2×3
... *from* 3×2

The notion to the right of each line of the type signature signifies how the type mapping was generated. For example, 1×2 means the first type mapping, `{a} -> {b}`, was combined with the second, `{b} -> {c}` to form the resulting type mapping `{a} -> {c}`. Some combinations were missing, namely 1×1, 2×2 and 3×3, because the second networks in these cases cannot receive what they want.

However, it's incomplete. Let's try flowing a record with type `{a,b}` through the network: type mappings 1 and 2 have the equally best match, and suppose type mapping 1 is chosen: `{a} -> {b}`. The label `b` will be flow-inherited to the

output but immediately overwritten by the label `b` in the output, thus leading to the intermediate record type simply `{b}`. Then it's flown through the second network using type mapping 2: `{b} -> {c}` and result in the final output type `{c}`. Therefore there should be a type mapping capturing `{a,b} -> {c}`, but there wasn't one.

2.2 Completion Defined

One can easily associate the adjective 'incomplete' to 'completion', which happens to have been defined in the S-Net type system [5].

Definition. Given a normalised type signature τ, its *completion* is defined as

$$Complete(\tau) = \tau \cup Complete(\tau \cup \bigcup_{(v_1 \to w_1) \in \tau} \bigcup_{(v_2 \to w_2) \in \tau} Check(\tau, v_1, w_1, v_2, w_2)),$$

where

$$Check(\tau, v_1, w_1, v_2, w_2) = \begin{cases} \phi, & BT(v_1) \neq BT(v_2) \vee \\ & \exists (v \to w) \in \tau.\, v_1 \cup v_2 = v \vee \\ & |v_1| \neq |v_2|; \\ Add(v_1, w_1, v_2, w_2) \cup & \\ \quad Add(v_2, w_2, v_1, w_1), & \text{otherwise,} \end{cases}$$

and

$$Add(v_x, w_x, v_y, w_y) = \{(v_x \cup v_y) \to (w_x \cup (v_y \backslash v_x))\}.$$

This is again a 'nonterminating' recursion like the type inference formula for the serial replication combinator in Section 1.4, and likewise it does terminate, and here's an informal proof. The algorithm adds type mappings to the final result through function *Add*, which merges the LHSs of every two existing type mappings to produce the LHSs of the new type mappings. But the additions are guarded by the function *Check* which will first check if the merged LHSs already exist as well as other criteria. The merged LHSs are subsets of the *alphabet* of the source type signature (the set of all labels used in the type signature), which is finite, and therefore there are finite number of LHSs, so there is some point after which the additions will not be able to produce new LHSs and fail due to the guard. This is when the whole process will stop producing extra type mappings, any iteration after which will simply generate the same result, so the process can now be terminated.

After the proof, let's look into more details. For each iteration, the function *Check* checks if a completion between the two selected type mappings is valid and necessary. If so, the function *Add* will be used twice to perform the actual completion, which simulates the result as if the network chooses $v_x \to w_x$ and flow-inherits other input record labels present in v_y.

The completion is extremely useful in theorem proving as it eliminates *a part* of the nondeterminism: if the incoming record equally matches two type mappings with *different* input types, while the runtime applies nondeterminism to break the tie, the theory model applies completion on the two type mappings to generate a best matching type mapping. The keyword 'a part' is because the completion cannot deal with nondeterminism applied to two type mappings with the same input type but different output types, but such type mappings in turn can be combined into one complex type mapping with alternatives on the RHS and be considered as one unity.

Additionally, by completing a type signature we can spell out some extent of flow inheritance: every added type mapping has a part of flow inheritance, as can be observed from the definition of function *Add*.

2.3 Completion for Type Inference

Let's go back to the problem at the beginning of this section on page 6. We should try completing the type signature first and see if the 'mapping missing' problem could be amended. Completing the type signature of `a` yields

```
1  ( {a} -> {b},                  ... 1
2    {b} -> {c},                  ... 2
3    {c} -> {d},                  ... 3
4    {a,b} -> {b},                ... from 1 + 2 using 1
5    {a,b} -> {a,c},              ... from 1 + 2 using 2
6    {a,c} -> {b,c},              ... from 1 + 3 using 1
7    {a,c} -> {a,d},              ... from 1 + 3 using 3
8    {b,c} -> {c},                ... from 2 + 3 using 2
9    {b,c} -> {b,d},              ... from 2 + 3 using 3
10   {a,b,c} -> {b,c},            ... from 4 + 6 using 4, etc.
11   {a,b,c} -> {a,b,d},          ... from 4 + 7 using 7, etc.
12   {a,b,c} -> {a,c} )           ... from 4 + 8 using 8, etc.
```

Again, the notions to the right indicate how each type mapping was generated, for example, '1 + 2 using 2' means that type mappings 1: `{a} -> {b}` was combined with 2: `{b} -> {c}`, and the result of the network choosing to go through the second type mapping and flow-inherit everything else was simulated to obtain the result `{a,b} -> {a,c}`. The top 3 type mappings were simply copied from the original signature, and the following 6 were generated by one iteration of the completion algorithm. Remember that completion is a repetitive process that does not terminate until no additional type mappings are introduced, and therefore the first 9 type mappings had to go through another iteration that generated the last 3 type mappings.

From 3 to 12 type mappings. This is more than a quadratic growth. Nonetheless let's try inferring the serial combination type signature again, using only this completed signature. Note the interesting effect the completion has brought: the LHSs contain all possible combinations from all possible input labels `a`, `b`

and `c` (apart from the empty combination), which in turn means that any serial combination of two type mappings that will introduce new labels to the left (such as type mapping 1 serially combined with 3, which will augment the input `{a}` to `{a,c}`) will cause a better match from other type mappings. The consequence is that for every type mapping, what follows must be the best match of its output, so as to comply with the rules. And the result is:

```
1  ( {a} -> {c},                  ... from 1 × 2
2    {b} -> {d},                  ... from 2 × 3
3    {a,b} -> {c},                ... from 4 × 2
4    {a,b} -> {b,c},              ... from 5 × 6
5    {a,b} -> {a,d},              ... from 5 × 7
6    {a,c} -> {c},                ... from 6 × 8
7    {a,c} -> {b,d},              ... from 6 × 9 and 7 × 1
8    {b,c} -> {d},                ... from 8 × 3
9    {b,c} -> {c,d},              ... from 9 × 2
10   {a,b,c} -> {c},              ... from 10 × 8
11   {a,b,c} -> {b,d},            ... from 10 × 9 and 11 × 4
12   {a,b,c} -> {a,c,d},          ... from 11 × 5
13   {a,b,c} -> {b,c},            ... from 12 × 6
14   {a,b,c} -> {a,d} )           ... from 12 × 7
```

Now, the originally missing type mapping `{a,b} -> {c}` magically appeared at line 3. Completion seemed to be the cure. What was decided after this discovery was that we would perform completion on the type signatures of all operand networks for serial combinators and serial replication combinators, and on the *result* type signatures of the parallel and parallel replication combinators. The reason for this asymmetry is that for parallel combinations, merging two type signatures might introduce more occurrences of 'incompleteness' however the original completion states were for the operand type signatures, and it's better to perform one single completion on top of the combined type signature, which would have the same effect as two pre-completions plus a post-completion.

3 Birth of the Supercompletion

3.1 Incomplete Completion

One question still remained: was the completion algorithm really complete, so that it would help us gain the correct inferred type signatures? Let's investigate three simple type signatures that answered this question negatively and caused some panic: they seemed to have proved wrong the established proof that the serial combinators were type-wise associative (which the type inference for serial replication combinator is based upon, see Section 1.4). The three type signatures were

```
net a ( {x} -> {y,z} ) ...;
```

```
net b ( {y} -> {z} ) ...;
net c ( {z} -> {x} ) ...;
```

By definition, single-mapping signatures like them were automatically complete. Let's try inferring the type signatures for `(a..b)..c` and then `a..(b..c)`:

```
a..b :       {x} -> {z};
(a..b)..c :  {x} -> {x};
b..c:        {y} -> {x};
a..(b..c) :  {x} -> {z,x}.
```

Two different results: serial combinators were 'not' type-wise associative. But was this counterevidence valid? We would not doubt the correctness of the type inference formulae because they were how the network was meant to be designed. The question might lie in the specialty of these type signatures. What if we treated the networks as partial functions from and to the same space $\mathbf{F}\{\text{'x'}, \text{'y'}, \text{'z'}\}\backslash\{\phi\}$? Figure 1 shows side by side the mappings these functions would represent. Flow-inheritance and overwriting were taken into account but nondeterminism not – it wouldn't be necessary for single-mapping signatures.

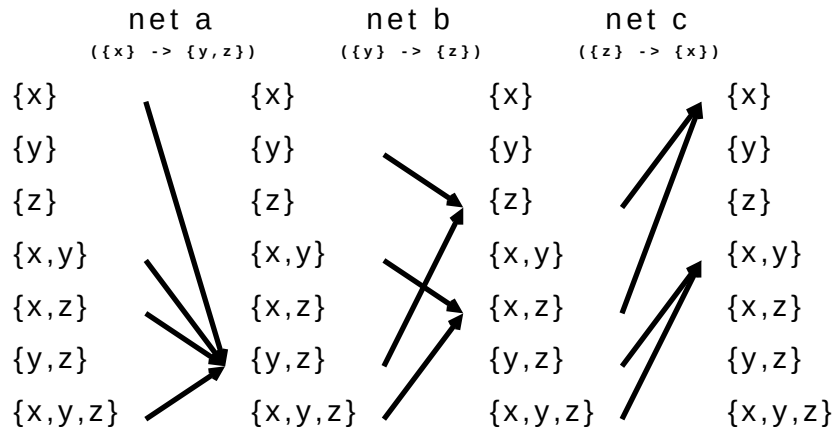


Figure 1: Networks treated as functions from and to $\mathbf{F}\{\text{'x'}, \text{'y'}, \text{'z'}\}\backslash\{\phi\}$

And now we perform the serial combination by observing the arrows. Working with `a..b` first yields 4 type mappings, from `{x}`, `{x,y}`, `{x,z}` and `{x,y,z}`, respectively, but all mapping to `{z}`. Then we combine this with network `c` which simply changes the output `{z}` to `{x}`. The final type signature is

`{x} -> {x}, {x,y} -> {x}, {x,z} -> {x}, {x,y,z} -> {x}.`

Now let's begin with `b..c`, which yields 4 type mappings again, from `{y}`, `{x,y}`, `{y,z}` and `{x,y,z}`, and also all mapping to `{x}`. Then we join in `a` to the left, which can only produce `{y,z}`. The result is again

`{x} -> {x}, {x,y} -> {x}, {x,z} -> {x}, {x,y,z} -> {x}`.

The associativity suddenly came back, and furthermore we obtained 3 more type mappings, which when we came to think about the overwriting property of the S-Net runtime system, were actually correct. So what was the problem?

3.1.1 All Because of Overwriting

If we look back at the specified type signatures for networks `b` and `c`, which are `{y} -> {z}` and `{z} -> {x}`, respectively, a simple merge that produces the type signature of `b..c`, `{y} -> {x}`, somehow shadows the fact that an additional label `z` will be used. The overwriting property then tells us that any incoming `z` in a valid input will be overwritten, and that's exactly the case if we left-join network `a` which produces `{y,z}` for input into the network `(b..c)`. Taking this into account, the combined network `a..(b..c)` should also have a type signature of `{x} -> {x}`, identical to the one produced from `(a..b)..c` using the original methods.

To really complete a type signature, we need some mechanism to capture the overwriting property. Figure 1 suggested that we should write the type signature of network `a` as 4 type mappings, instead of only one. The extra type mappings reflected the overwriting property of the S-Net runtime system by 'porting' the RHS labels to the left. Maybe we could capture this feature and make another level of completion.

3.2 Supercompletion

Recall from Section 2.2 that completion was designed to reduce nondeterminism and emphasize flow inheritance. To further capture the overwriting property, we hereby define 'supercompletion'.

Definition. The *supercompletion* of a normalised type signature τ is defined as

$$Supercomplete(\tau) = \tau_c \cup \bigcup_{(v \to w) \in \tau_c} AddMore(\tau_c, v, w),$$

where $\tau_c = Complete(\tau)$, and

$$AddMore(\tau_c, v, w) =$$

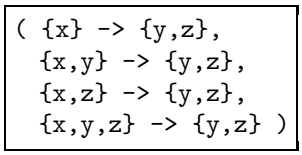
$$\{(v \cup \delta) \to w | \delta \subseteq (w \backslash BT(w) \backslash v) \ \wedge \ \neg \exists (v' \to w') \in \tau_c . (v \cup \delta) = v'\}.$$

The name comes from the fact that for every normalised type signature τ, $Supercomplete(\tau)$ is a superset of $Complete(\tau)$. The function $AddMore$ performs what was suggested by Figure 1, to include all possible combinations of the RHS extra labels into the left. Binding tags are excluded to keep the defined subtyping restriction.

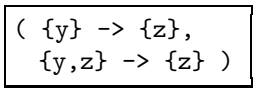
3.3 Evaluation

Let's try our solution using the same 3 networks that made us aware of the problem. Supercompleting their signatures yields:

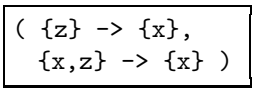
```
net a                net b              net c
({x} -> {y,z})       ({y} -> {z})       ({z} -> {x})
```



```
( {x} -> {y,z},
  {x,y} -> {y,z},
  {x,z} -> {y,z},
  {x,y,z} -> {y,z} )
```



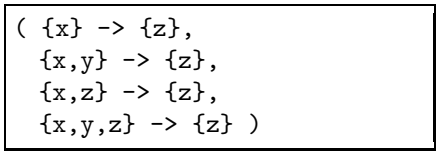
```
( {y} -> {z},
  {y,z} -> {z} )
```



```
( {z} -> {x},
  {x,z} -> {x} )
```

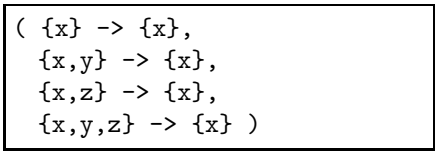
And let's infer the type signatures of `(a..b)..c` and `a..(b..c)` again. Details are omitted; only the results are shown:

```
a..b
```



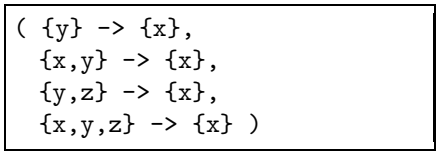
```
( {x} -> {z},
  {x,y} -> {z},
  {x,z} -> {z},
  {x,y,z} -> {z} )
```

```
(a..b)..c
```



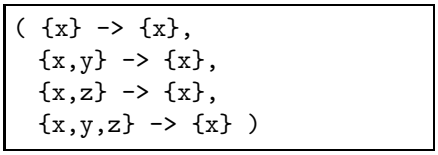
```
( {x} -> {x},
  {x,y} -> {x},
  {x,z} -> {x},
  {x,y,z} -> {x} )
```

```
b..c
```



```
( {y} -> {x},
  {x,y} -> {x},
  {y,z} -> {x},
  {x,y,z} -> {x} )
```

```
a..(b..c)
```



```
( {x} -> {x},
  {x,y} -> {x},
  {x,z} -> {x},
  {x,y,z} -> {x} )
```

Now, with supercompletion, we were able to obtain the correct and most complete result. It was then decided that supercompletion should be applied to type signatures prior to serial combination type inferences, and to the resulting type signatures after parallel combination type inferences.

In fact, supercompletion was the first mechanism to capture all 3 runtime properties into theory. Plus it's describable using the current type system, runtime and theoretic. However, as a side effect it introduces too many 'add-ons' to the original type signature. Completion itself would already grow the signature exponentially, and the supercompletion goes on multiplying the size of the signature by a number roughly equal to the number of all possible combinations of the labels that only exist on the RHSs. In theory the size of the type signatures does not really matter, but in practise, when the type inference stage of the S-Net compiler is to be implemented, the giant type signatures may introduce significant performance degradation. The performance would be especially affected when the giant type signatures go through the serial combination type inference, which has a complexity of $O(M \times N)$, where M and N are the sizes of the type signatures of the two operand networks, which after the supercompletion are too large to handle.

There's one more problem with supercompletion: supercompleted type sig-

nature might become incomplete again. Consider the following type signature which is complete: `{a} -> {b}, {a} -> {c}`. Supercompleting it yields `{a} -> {b}, {a,b} -> {b}, {a} -> {c}, {a,c} -> {c}`, and nondeterminism 'comes back' when the input type is `{a,b,c}`, which means the resulting type signature should go through the completion process again. We may need to refine the definition of supercompletion into a similarly recursive one, adding overwritten labels during the completion process instead of at the final stage of the algorithm. However, before finding a solution, we discovered another fault that tried to persuade us not to use any completion.

4 Disappeared Nondeterminism and Semicompletion

4.1 (Super-)completion Modified Runtime Behaviours

Consider this simple network which is illustrated in Figure 2:

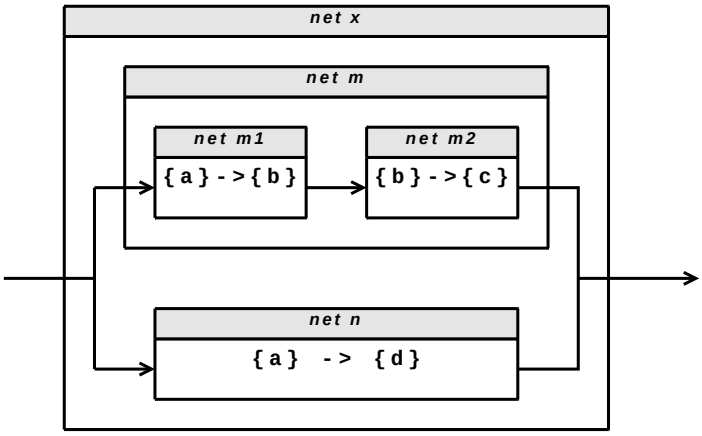


Figure 2: A simple network

```
net x {
    net m {
        net m1 ( {a} -> {b} ) ...;
        net m2 ( {b} -> {c} ) ...;
    } connect (m1..m2);
    net n ( {a} -> {d} ) ...;
} connect (m|n);
```

By observing the figure, we would be convinced that both of the subnetworks accept `{a}`, and nondeterminism should be applied. For example, if a record of type `{a,b}` enters network `x`, it could be sent to either `m` or `n`. The additional

input `b` would be overwritten inside network `m` but this should not affect how the nondeterminism works.

If we perform the type inference for network `m`, which involves supercompleting the type signatures of `m1` and `m2`, we would obtain:

```
net x {
    net m ( {a} -> {c}, {a,b} -> {c}, {a,c} -> {c}, {a,b,c} -> {c} ) {
        net m1 ( {a} -> {b}, {a,b} -> {b} ) ...;
        net m2 ( {b} -> {c}, {b,c} -> {c} ) ...;
    } connect (m1..m2);
    net n ( {a} -> {d} ) ...;
} connect (m|n);
```

As a consequence, network `m` now declares that it accepts, for example, `{a,b}` as input. If the runtime is indeed provided a record of type `{a,b}`, seeing a best match it would never send the input to network `n`, causing the expected nondeterminism to disappear. This would not be amended by further type-inferring `x` and then supercompleting the result afterwards, because supercompletion, like completion, will only reduce nondeterminism further, but not 'recovering' it.

Don't blame supercompletion; the same thing can also happen with the original completion, such as the configuration below:

```
net x {
    net m {
        net m1 ( {a} -> {b}, {b} -> {b} ) ...;
        net m2 ( {b} -> {c} ) ...;
    } connect (m1..m2);
    net n ( {a} -> {d} ) ...;
} connect (m|n);
```

Although now `m1` accept an additional input type `{b}`, it does not stop the property of nondeterminism to give `n` 1/3 of the probability to receive an input record of type `{a,b}`. Such nondeterminism will again disappear had we type-inferred `m` using the original completion, which would generate a better match for `{a,b}` in the type signature of `m1`. A conclusion: completion and supercompletion will modify the runtime behaviour!

4.2 Secret Behind Completion

Should we not use completion or supercompletion for the operand networks of parallel combinations? No, we need them to correctly type-infer serial combinations which might be one operand of a parallel combination. Let's check why it originally worked and now malfunctions.

In Section 2.3 we've seen an example of completion which transformed a 3-mapping signature, `( {a} -> {b}, {b} -> {c}, {c} -> {d} )`, into 12 mappings. We also pointed out that the resulting type signature captured every combination of all input labels. Not all type signatures will produce a 'complete combination', but this suggests how completion deals with nondeterminism: it

uses additional type mappings to explicitly spell out what would happen for every nondeterministic case.

To correctly cover all nondeterministic cases, all alternative type mappings should be at hand when the completion algorithm performs the calculation. Once two type mappings have been merged, a better match will be generated which might cause disruptions. The algorithm avoids the disruptions by not looking at the newly generated type mappings until the next iteration.

Take the same 3-mapping signature for example. First of all, the first two type mappings will merge into two entries beginning with `{a,b}`, but it's not used until the algorithm has finished working with the whole original type signature to further produce type mappings beginning with `{a,c}` and `{b,c}`. Then, during the next iteration the new type mappings are further merged into the ones that begin with `{a,b,c}`.

Had we performed a full completion only on the first two type mappings, and then join in the third, the whole balance will be disrupted. When the algorithm is aware of the input `{c}` from the third type mapping, there exists already a type mapping that takes `{a,b}` as the input, and the merged input type `{a,b,c}` now only matches the double-label input best.

This is exactly what happened to the parallel combination cases, where the nondeterminism disappeared because the completion process was effectively done in two batches for both operand networks instead of as a whole. It's a pity that the completion algorithm is not distributive, and it's worse because we shouldn't stop or pause the completions done to operand networks.

4.2.1 'Hypercompletion'?

The problem may be occasionally relaxed if the two operand networks have the same set of alphabet. As we have seen from the completion example, the completed signatures tend to cover all possible combinations of the alphabet on the LHSs of the member type members. Indeed, if the type signatures are built up using the thought of partial functions, like what we have done in Figure 1, we would have every possible case at hand and the completion algorithm becomes distributive. However, if the alphabets are of different sizes, the balance may be inclined to the network with a larger alphabet. Similarly, the problem will also be magnified if the two alphabets contain different labels.

We could perform something that might be called 'hypercompletion' for parallel combination type inference, to work out the completion (or more properly called, function mapping) on the 'least common alphabet', just like what we did to the network `b` in Figure 1 whose alphabet was {'y', 'z'} but we added the label `x` for calculation. It would be feasible and can solve our burning problem.

But it would worsen the signature explosion problem. We already disliked supercompletion for its explosive effect that applies to even single-mapping signatures. It's time that we thought about why we introduced completion at first.

4.3 Decomposing the Completion

This was the network presented in Section 2.1:

```
net x {
    net a ( {a} -> {b}, {b} -> {c}, {c} -> {d} ) ...;
} connect (a..a);
```

We used completion when type-inferring `x` because we would like to see `{a,b} -> {c}` in the result. How exactly could this type mapping occur? When an input record of type `{a,b}` enters the first network `a`, nondeterminism chooses the first type mapping that causes the input `b` to be overwritten and produces a single-label intermediate result `{b}`, and this result is passed through the second `a` which emits the output `{c}`.

So actually, the input `b` is not exactly an 'input': it's something that's overwritten inside the network. It is written as an input simply because it won't be flow-inherited to the output. It seems that completion can sometimes help us describe the overwriting property, in the absence of supercompletion. But that was the case because the output label `b` occurred in the input of another type mapping: completion never looks at outputs.

Anyway, both kinds of completion could capture all 3 S-Net runtime properties: flow inheritance, overwriting, and nondeterminism. We've seen how capturing nondeterminism could actually modify the runtime behaviour, and we haven't had any trouble with flow inheritance. Maybe what we really need to capture is only the overwriting.

4.4 Discard Label Qualifier

Let's add something to the S-Net language syntax. A label in the LHS record type of a type mapping can be preceded with a backslash '\', to denote the fact that it will be discarded (overwritten) *if the input record contains it.* The choice of symbol is after its mathematical meaning of set subtraction. The labels with such 'discard label qualifier' are not the actual input to the network and thus do not contribute to the match score.

With this handy notation, the previous type mapping `{a,b} -> {c}`, where the label `b` was not an actual input, can then be written as `{a,\b} -> {c}`. This special type mapping also accepts an input of type `{a}` as well as `{a,b}`, capturing two mappings simultaneously. Furthermore, because it outputs label `c`, any input label `c` will also be overwritten, so the type mapping might as well be written as `{a,\b,\c} -> {c}`, capturing 4 mappings at once (the LHSs being `{a}`, `{a,b}`, `{a,c}`, and `{a,b,c}`). We can already see the power of it: a type mapping using the discard label qualifier is actually a 2^n-in-one type mapping, where n is the number of instances of the discard label qualifier.

Now we rewrite the original type signature using the new invention:

```
net x {
    net a ( {a,\b} -> {b}, {b,\c} -> {c}, {c,\d} -> {d} ) ...;
```

```
} connect (a..a);
```

To type-infer network `x`, we'll follow the principle used in the serial combination type inference formula, but now also dealing with the discards. We still begin with arbitrarily choosing a type mapping from each of the participant networks, which are both `a` in this concrete case, and then for every pair of type mapping ($v_1 \to w_1$, $v_2 \to w_2$) we do the followings:

1. Check that the binding tags of w_1 and v_2 match.
2. The second network requires the (non-discard-qualified) labels in v_2 but not in w_1 as additional inputs, so add them to v_1 as the overall input. But if any of these labels is in v_1, regardless of the qualifier, it'll be either consumed or discarded by the first network, and the second network will be stuck waiting for a valid input forever. Furthermore, the augmented input needs to pass the better-match check in the type signature of the first network.
3. The actual input to the second network is those in v_2 without the discard qualifier, plus the additional labels from w_1, and the better-match check should also be done here.
4. The extra labels in w_1 not consumed or discarded by the second network is then flow-inherited to the output of the combined network.
5. Finally, any label NOT in the overall input which is discarded by the second network should also be added to the overall input, with the discard qualifier.

Following these rules we perform the type inference by hand for network `x`:

```
( {a,\b} -> {b} ) * ( {a,\b} -> {b} ): input taken, no-go;
( {a,\b} -> {b} ) * ( {b,\c} -> {c} ) => ( {a,\b,\c} -> {c} );
( {a,\b} -> {b} ) * ( {c,\d} -> {d} ) => ( {a,c,\b,\d} -> {b,d} );
( {b,\c} -> {c} ) * ( {a,\b} -> {b} ) => ( {a,b,\c} -> {b,c} );
( {b,\c} -> {c} ) * ( {b,\c} -> {c} ): input taken, no-go;
( {b,\c} -> {c} ) * ( {c,\d} -> {d} ) => ( {b,\c,\d} -> {d} );
( {c,\d} -> {d} ) * ( {a,\b} -> {b} ) => ( {a,c,\b,\d} -> {b,d} );
( {c,\d} -> {d} ) * ( {b,\c} -> {c} ) => ( {b,c,\d} -> {c,d} );
( {c,\d} -> {d} ) * ( {c,\d} -> {d} ): input taken, no-go.
```

And the resulting type signature is

```
( {a,\b,\c} -> {c},
  {a,c,\b,\d} -> {b,d},
  {a,b,\c} -> {b,c},
  {b,\c,\d} -> {d},
  {b,c,\d} -> {c,d} )
```

... *capturing* $1, 3, 6, 10$
... *capturing* $7, 11$
... *capturing* $4, 13$
... *capturing* $2, 5^*, 8, 14^*$
... *capturing* $9, 12^*$

The type signature is identical with that in Section 6 which was calculated without completion, except the existence of some discarded labels. However, remember that one type mapping using discard label qualifiers can contain multiple 'original' type mappings, and in fact this 5-mapping signature captures the whole type inference result on page 9 which had 14 mappings. The line numbers of the captured mappings are shown to the right of the code above. An asterisk indicates that we need to flow-inherit label `a` (add it onto both sides) for the capture to look identical with the captured. The signature above also includes some entries that could be found by supercompletion, such as the fourth mapping which contains `{b,c,d} -> {d}`, a type mapping that we haven't seen so far.

The discard label qualifier is perfect for serial combinations, but how about parallel? Let's use the network illustrated in Figure 2 for discussion. Using the discard label qualifier, the network can be written as:

```
net x ( {a,\b,\c} -> {c}, {a,\d} -> {d} ) {
    net m ( {a,\b,\c} -> {c} ) {
        net m1 ( {a,\b} -> {b} ) ...;
        net m2 ( {b,\c} -> {c} ) ...;
    } connect (m1..m2);
    net n ( {a,\d} -> {d} ) ...;
} connect (m|n);
```

The type signatures of networks `m1`, `m2` and `n` were rewritten using the discard label qualifier, and those of the networks `x` and `m` were inferred. Since we stated that the labels with discard qualifiers does not contribute to the match score, an input of type `{a,b}` will match the two type mappings in network `x` equally well, and as a result the inner networks `m` and `n` will share such inputs nondeterministically.

So far, so good. We have been too concerned with perfection, while in practise everything could be solved by simply taking care of the overwriting property alone.

4.5 Semicompletion

We would therefore like to formally define 'semicompletion', a kind of completion that captures only the overwriting property. But since we've introduced something new to the record types, almost the whole type system has to be changed. Namely, a nonvariant record type is now a set of optionally discard-qualified labels. However, we can also see it as a pair of sets, one being the original label set, and the other a 'discarded label set'. The output record types are still the original nonvariant record types because the discard label qualifiers will not be used on the RHSs of any type mappings.

As a tentative solution, we define a new format for representing type mappings in mathematic symbols:

$$v[d] \to w$$

where v and w retain their original meanings of input and output label sets, and d is a set of discarded labels. The backslash is not used here to prevent the misunderstanding that it were the set subtraction operator. The set d is constrained to not contain any binding tags, because they're specially used to restrict subtyping: two inputs which differ only by the presence of a binding tag does not have a subtype relationship and cannot be matched to the same type mapping, and therefore specifying a 'discarded' binding tag in a type mapping is completely equivalent to not specifying the binding tag at all.

Then we can define semicompletion as follows.

Definition. The *semicompletion* of a normalised type signature τ is

$$Semicomplete(\tau) = \bigcup_{(v[d]\to w)\in\tau} \{v[(d \cup w)\backslash BT(w)\backslash v] \to w\}.$$

The setup of the function allows it to be applied to type signatures already making use of the discard qualifier. If a type mapping does not contain any discard-qualified labels, we can see it as having $d = \phi$ and apply the same formula.

4.6 Evaluation

The semicompletion simply adds the potentially discarded labels to the LHS of every type mapping. This was exactly what we've done in Section 4.4 when we tried out the discard label qualifier, and the outcome was very satisfactory: capturing only the overwriting property, the use of the discard label qualifier, and thus the semicompletion, was able to help us infer type signatures which were most complete as of our knowledge, yet maintaining all the runtime properties, and more delightfully, did not cause 'signature explosion'.

It was the best completion ever, and it did come with a price: the existing S-Net theory and implementation needed a completely reconstruction due to the extended language grammar and semantics. However, seeing its power, the S-Net development team was willing to adapt to this extension before it becomes too late to correct.

5 Add-on: the Pass-through Label Qualifier

The discard label qualifier divides the labels in a type mapping into two partitions: those to be discarded, and those to be read and written. It would be desirable to insert a buffer between them, which is the group of 'read and not written' labels. The pass-through label qualifier was thus born, soon after the introduction of the discard qualifier.

The pass-through label qualifier is an equal sign '=' placed immediately after the qualifying label, available in the LHS record types. A pass-through qualified label will be input into the network like ordinary, unqualified labels, but at the same time a copy of it is passed through the network to the output, as if it's not used by the network and flow-inherited.

For example, the type mapping `{a,b=} -> {c}` is equivalent with `{a,b} -> {b,c}`, except that the former also emphasizes that the input and output `b` will have the same content. For ease of type checking and inference, it is agreed that the pass-through label qualifier can be symmetrically applied to the labels in the RHS record type in the compiler's internal representation, such as `{a,b=} -> {=b,c}`. Note the symmetry which, when concatenated, forms a meaningful C expression: `b==b`.

Binding tags are again not allowed to carry the pass-through label qualifier, because they're used to restrict subtyping and it may confuse the programmer if a binding tag were passed through to the output generating a different subtype restriction than what can be seen on the RHS of the type mapping – and even if the programmer is aware of it, we would rather have the programmer spell out the full output binding tag set.

5.1 Compatibility with Runtime Properties

5.1.1 Pass-through and Flow Inheritance

The pass-through qualifier actually creates a stronger type of flow inheritance: 'input and flow through' compared to 'unused so flow through'. The original flow inheritance is not affected.

5.1.2 Pass-through and Overwriting

If the RHS record type contains the same label which is pass-through qualified on the LHS, it indicates that the network will output the label, and the one passed through will be overwritten, as specified by the overwriting property. In such case, the label in question can be equivalently changed to be unqualified. For instance, `{a,b=} -> {b,c}` can be rewritten as `{a,b} -> {b,c}`. It is not equivalent to change the LHS to `{a,\b}`, because the label should contribute to the match score, and more importantly, the first subnetwork may indeed demand it as part of the input.

5.1.3 Pass-through and Nondeterministic Best Match

The pass-through qualified labels are actual inputs and contribute to the match score, however, the pass-through qualifier does not raise or lower the value of the label compared to an unqualified one. If two or more best match record types differ only by some pass-through qualifiers, nondeterminism will apply.

5.2 Evaluation

The introduction of the pass-through qualifier served multiple purposes. First, it simplifies the programmers' tasks in some special cases, such as a network that reads a label but does not modify it, which is followed by another network that uses the same label again. Previously the programmer of the first network should remember to emit the unchanged label as output, but now the task can

be handed over to the runtime system. Secondly, it's a way to declare a constant (read-only) field which could be desirable in some circumstances. Finally, the code optimisation stage of the S-Net compiler could use the information to generate faster code, or the runtime system could deploy the processors more wisely, as in the case below:

```
net better_parallel {
    net first ( {a=,b} -> {m,n} ) ...;
    net second ( {a,c} -> {p,q} ) ...;
} connect (first..second);
```

The network `second` takes as input the label `a` which is output by `first`, but because the label `a` is pass-through qualified, the two serially combined subnetworks can in fact be optimised to run in parallel, using identical copies of the label `a`.

6 New Type System

We hereby provide a new mathematic model for the extended S-Net type system, starting with redefining the record types.

6.1 Record Types

Syntax-wise a nonvariant record type is a set of optionally qualified labels, but to capture the semantics of the qualifiers, we would define a nonvariant record type as follows.

- A concrete record has only one nonvariant record type which does not contain any qualified labels.
- A nonvariant record type used as the LHS of a type mapping consists of three sets of labels: the input set v, the pass-through set p having $p \subseteq (v \backslash BT(v))$, and the discard set d. The record type can be represented as a tuple (v, p, d), but for ideographic reasons, in this section it is written as
$$v^p[d],$$
so that the pass-through set stands higher than the input set to mean that it'll 'flow away', and the discard set is wrapped in square brackets which is commonly used in console command syntax to denote optional parts.
- A nonvariant record type used as the RHS of a type mapping consists of only two sets of labels: the output set w and the passed-through set q. Similarly it's written as
$$w^q.$$

To have a unique representation for these nonvariant record types as well as provide some shortcut notations, it is allowed to omit the pass-through or passed-through set if it's empty, and/or omit the discard set together with the square brackets if it's empty. Hence the new nonvariant record type is captured in mathematic terms as

$$RecType' = \{v^p[d] | v \in \mathbf{F}Label \wedge p \subseteq (v \backslash BT(v)) \wedge \\ d \in \mathbf{F}Label \wedge d \cap v = \phi \wedge BT(d) = \phi\}.$$

Variant record types are still a nonempty set of nonvariant record types.

6.2 Subtype Relationship and Match Score

The subtype relationship only checks the main label set:

$$v_1^{p_1}[d_1] \sqsubseteq v_2^{p_2}[d_2] \Longleftrightarrow BT(v_1) = BT(v_2) \wedge v_1 \supseteq v_2.$$

So does the match score: if the actual input type is v_0 and the declared input type in a type mapping is $v^p[d]$, and $v_0 \sqsubseteq v^p[d]$ then the match score for this particular case is $|v|$.

6.3 Flow Inheritance

Flow inheritance occurs if a type mapping accepts an input of a subtype of its LHS. What's declared to be discarded will be discarded, and what's declared as input will be consumed, and the rest is passed through unchanged.

$$\frac{v^p[d] \rightarrow w^q, \qquad v' \sqsubseteq v^p[d]}{(v \cup \delta)^{(p \cup \delta)}[d] \rightarrow (w \cup \delta)^{(q \cup \delta)}, \text{ where } \delta = v' \backslash v \backslash d}$$

This inference rule serves as a tool to understand how the extended format affects flow inheritance; it's not used alone, and wherever flow inheritance should be applied, the inference rule is embedded into the type inference formulae which will be presented later.

6.4 Completion

In Section 5 we allowed variant usages of the pass-through qualifier, such as `{a,b=} -> {c}` which can be complemented to `{a,b=} -> {=b,c}`, and `{a,b=} -> {b,c}` which is better written as `{a,b} -> {b,c}`. Moreover, if we encounter something like `{a} -> {=b}`, the programmer might have meant `{a} -> {b}`. To produce a standardised representation so that the type inference can be expressed concisely, we would like all type mappings to be transformed by the following inference rule before they're used:

$$\frac{v^p[d] \rightarrow w^q}{v^{(p \backslash (w \backslash q))}[d] \rightarrow (w \cup p)^{(p \backslash (w \backslash q))}}$$

so that pass-through qualifiers exist strictly in pairs, i.e. the pass-through qualified labels are identical on both sides.

The inference rule only modifies the main label sets and the pass-through label sets. Recall from Section 4.5 that semicompletion also transforms type signatures on a per-type-mapping basis, and it only modifies the discard label sets. It is therefore logical to group them together in one step. The resulting procedure will be more than performing a semicompletion: it completes the type signature with omitted passed-through output labels as well, thus it might as well be called 'completion', now that the original definition is outdated.

Definition. The new *completion* of a type signature τ is defined as

$$Complete'(\tau) = \bigcup_{(v^p[d] \to w^q) \in \tau} \{v^{(p \backslash (w \backslash q))}[(d \cup w) \backslash BT(w) \backslash v] \to (w \cup p)^{(p \backslash (w \backslash q))}\}.$$

As can be seen from the formula, the new completion simply modifies every type mapping into a more complete form, without introducing any additional type mappings into the signature. The signature explosion problem is officially extinguished.

Likewise, we demand completion to be applied prior to the type inference for serial and serial replication combinators and after type inference for parallel combinators. Unlike before, we now require completion *prior to* the type inference for parallel replication combinators, because the type inference formula will modify the type signature on a per-type-mapping basis, and it's preferable to have them standardised beforehand.

6.5 Type Inference

6.5.1 Type Inference for Serial Combinator '..'

The type signature of a serially combined network `a..b`, where the type signature of `a` after normalisation and completion is α, and for `b` β, is

$$DotDot'(\alpha, \beta) = \bigcup_{(v_1^{p_1}[d_1] \to w_1^{p_1}) \in \alpha} \bigcup_{(v_2^{p_2}[d_2] \to w_2^{p_2}) \in \beta} Join'(\alpha, \beta, v_1^{p_1}[d_1], w_1^{p_1}, v_2^{p_2}[d_2], w_2^{p_2}),$$

where

$$Join'(\alpha, \beta, v_1^{p_1}[d_1], w_1^{p_1}, v_2^{p_2}[d_2], w_2^{p_2}) =$$

$$\begin{cases} \phi, & BT(w_1) \neq BT(v_2) \vee (v_2 \backslash w_1) \cap (v_1 \cup d_1) \neq \phi \vee \\ & (\exists (v^p[d] \to w^p) \in \alpha . [v_1 \cup (v_2 \backslash w_1) \sqsubseteq v \wedge |v| > |v_1|) \vee \\ & (\exists (v^p[d] \to w^p) \in \beta . (w_1 \cup v_2) \sqsubseteq v \wedge |v| > |v_2|); \\ v'^{p'}[d'] \to w'^{p'}, & \text{otherwise,} \end{cases}$$

where

$$\begin{aligned} v' &= v_1 \cup (v_2 \backslash w_1), \\ p' &= (p_1 \backslash (v_2 \backslash p_2)) \cup (p_2 \backslash (w_1 \backslash p_1)), \\ d' &= (d_1 \cup d_2) \backslash v', \\ w' &= w_2 \cup (w_1 \backslash v_2 \backslash d_2). \end{aligned}$$

6.5.2 Type Inference for Parallel Combinator '|'

The type inference formula for parallel combinator remains unchanged:

$$Bar(\alpha, \beta) = \alpha \cup \beta.$$

Again, the constraints that restrict some type signatures to be combined in parallel are omitted.

6.5.3 Type Inference for Serial Replication Combinator '*'

The principle of calculation is unchanged, which is to treat the network as an indefinitely long chain of binary serial combinations. Now that we've defined a new record type format and a new type inference formula for the binary serial combinator, we simply need to update the corresponding places in the original formula for type-inferring serial replication networks.

The right operand of the * operator is a list of nonvariant record types which describe patterns of the records that can cause the replication to terminate, and they're not part of the input or output of any type mappings. Therefore, the extended record type format cannot be used on them.

The type signature of the serial replication over a network with the normalised and completed type signature α using the termination patterns T as

$$Star'(\alpha, T) = PreTerm'(T) \cup star'(\alpha, \alpha, T),$$

where

$$PreTerm'(T) = \{(t^{(t \backslash BT(t))} \rightarrow t^{(t \backslash BT(t))}) | t \in T\},$$

and

$$star'(\tau, \alpha, T) = Term'(\tau, T) \cup star'(DotDot'(Nonterm'(\tau, T), \alpha), \alpha, T),$$

and

$$\begin{aligned} Term'(\tau, T) = \{(v^p[d] \rightarrow w^p) | (v^p[d] \rightarrow w^p) \in \tau \wedge \\ \neg IsTerm(v, T) \wedge IsTerm(w, T)\}, \end{aligned}$$

and

$$\begin{aligned} Nonterm'(\tau, T) = \{(v^p[d] \rightarrow w^p) | (v^p[d] \rightarrow w^p) \in \tau \wedge \\ \neg IsTerm(v, T) \wedge \neg IsTerm(w, T)\}, \end{aligned}$$

and the predicate $IsTerm$ is unchanged as in Section 1.4:

$$IsTerm(t, T) \Longleftrightarrow \exists v \in T.\, t \sqsubseteq v.$$

6.5.4 Type Inference for Parallel Replication Combinator '!'

The one or two tags as the right operand of the ! operator is added to the LHS of every type mapping of the network on which the parallel replication is applied. For each tag, if it's declared as input or discarded, it's remained as it is; otherwise it's passed through unchanged.

The type signature for the parallel replication over a network with the normalised and completed type signature α using one or two tags which are collected in a set $tags$ is

$$Ex'(\alpha, tags) = \bigcup_{(v^p[d] \to w^p) \in \alpha} Merge'(\alpha, v^p[d], w^p, tags),$$

where

$$Merge'(\alpha, v^p[d], w^p, tags) = \begin{cases} \phi, & BT(tags) \not\subseteq BT(v) \vee \\ & \exists (v_0^{p_0}[d_0] \to w_0^{p_0}) \in \alpha. \\ & \quad v' \sqsubseteq v_0 \wedge |v_0| > |v|; \\ \{v'^{p'}[d'] \to w'^{p'}\}, & \text{otherwise,} \end{cases}$$

and

$$\begin{array}{rcl} v' & = & v \cup tags, \\ p' & = & p \cup (tags \backslash v \backslash d), \\ d' & = & d \backslash tags, \\ w' & = & w \cup (tags \backslash v \backslash d). \end{array}$$

7 Conclusion

In this paper, we discussed the story of finding the flaws in the original S-Net type system and solving them.

- The problem that the result of serial combination type inference was incomplete was partly solved by applying completion.
- The problem that the type signatures could not capture the overwriting runtime property was solved by supercompletion.
- The problem that nondeterminism may disappear due to the application of completion or supercompletion, as well as the signature explosion, was solved by the discard label qualifier, i.e. the semicompletion.

The discard label qualifier captured the overwriting runtime property. As an add-on, we introduced the pass-through label qualifier that captured the flow inheritance property. We chose to not capture nondeterminism seeing the potential problem of modifying the runtime behaviour. The label qualifiers led to an extension to the S-Net type system, which made it complete and efficient.

We have also proposed a new mathematic model reflecting this extension to the system.

References

[1] Grelck, C., Shafarenko, A.: Report on S-Net: A Typed Stream Processing Language, Part I: Foundations, Record Types and Networks, draft version 0.5. University of Hertfordshire, UK (2007)

[2] Grelck, C., Penczek, F.: Implementing S-Net: A Typed Stream Processing Language, Part I: Compilation, Code Generation and Deployment, draft version 0.3. University of Hertfordshire, UK (2006)

[3] Grelck, C., Shafarenko, A.: Chapter 4: Type Inference and Semantics, Report on S-Net: A Typed Stream Processing Language, Part I: Foundations, Record Types and Networks, draft version 0.5. University of Hertfordshire, UK (2007) 36-48

[4] Grelck, C., Penczek, F.: 2.2: Preprocessing, Implementing S-Net: A Typed Stream Processing Language, Part I: Compilation, Code Generation and Deployment, draft version 0.3. University of Hertfordshire, UK (2006), 12-13

[5] Grelck, C., Shafarenko, A.: 4.5: Type Signature Completion, Report on S-Net: A Typed Stream Processing Language, Part I: Foundations, Record Types and Networks, draft version 0.5. University of Hertfordshire, UK (2007) 46-47

[6] Grelck, C., Penczek, F.: 1.2 Compiler Architecture, Implementing S-Net: A Typed Stream Processing Language, Part I: Compilation, Code Generation and Deployment, draft version 0.3. University of Hertfordshire, UK (2006), 12-13

[7] Grelck, C., Shafarenko, A.: 4.3: Closure, Report on S-Net: A Typed Stream Processing Language, Part I: Foundations, Record Types and Networks, draft version 0.5. University of Hertfordshire, UK (2007) 40-45