

Graphical and Incremental Type Inference: A Graph Transformation Approach

Silvia Clerici, Cristina Zoltan, and Guillermo Prestigiacomo

Dept. Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
Barcelona, Spain

Abstract. We present a graph grammar based type inference system for a totally graphic development language. NiMo (Nets in Motion) can be seen as a graphic equivalent to Haskell that acts as an on-line tracer and debugger. Programs are process networks that evolve giving total visibility of the execution state, and can be interactively completed, changed or stored at any step. In such a context, type inference must be incremental. During the net construction or modification only type safe connections are allowed. The user visualizes the type information evolution and, in case of conflict, can easily identify the causes. Though based on the same ideas, the type inference system has significant differences with its analogues in functional languages. Process types are a non-trivial generalization of functional types to handle multiple outputs, partial application in any order, and curried-uncurried coercion. Here we present the elements to model graphical inference, the notion of structural and non-structural equivalence of type graphs, and a graph unification and composition calculus for typing nets in an incremental way.

Keywords: type inference, graphical language, process networks, type visualization.

1 Introduction

The data flow view of lazy functional programs as process networks was first introduced in [1]. The graphical representation of functions as processes and infinite lists as non-bounded channels helps to understand the program overall behaviour. The net architecture shows bi-dimensionally the chains of function compositions, exhibits implicit parallelism, and back arrows give an insight into the recurrence relations between the new results and those already calculated. The graphic execution model that the net animation suggests was the outset of the NiMo language design, whose initial version was presented in [2]. It was completely defined with graph grammars and implemented in the graph transformation system AGG [3]. This first prototype NiMoAGG showed the feasibility of a graphical equivalent for Miranda or Haskell, also fully graphically executable. A small set of graphic elements allows dealing with higher order, partial application, non-strict evaluation, and type inference with parametric polymorphism.

As the net represents the code and its computation graph at the same time, users have total visibility of the execution internals in a comprehensible model. Partially defined nets can be executed, dynamically completed or modified and stored at any step, enabling incremental development on the fly. Execution steps can also be undone, acting as an on line tracer and debugger where everything can be dynamically modified, even the evaluation policy. In the current version, five modes of increasing activity can be globally or locally assigned to each process, thus allowing to increase parallelism, reduce channel size (number of elements) and synchronize subnets. Symbolic execution is also admitted. The execution model is defined in [4].

In this context, where incompleteness does not inhibit execution, editing a program is a discontinuous process with execution intervals where code evolves up to the next interaction; hence type inference is by necessity incremental. On the other hand, in NiMo there is no textual code at all. Programs are graphs whose nodes are interfaces of processes or data. Interfaces are graphic tokens with typed in/out ports. Net construction equates to building a bi-dimensional term, where sub-expressions are like puzzle pieces that can be pairwise connected in any order if their shapes fit (both port types unify), thus ensuring type safeness by construction. In the first version, static inference was partial in presence of polymorphism. Now the full type information of each interface port is carried up by means of a second kind of graphs, and updated with each connection. Users can visualize the type information evolution and realize why a connection is rejected. Though based on the same principles, the inference system has significant differences with its functional analogues. Besides being graphical and incremental, the data flow ingredient imposes coping with multiple-output processes and curried-uncurried interpretation of multiple inputs, partial application in any order and partial disconnection for multiple outputs. In the current version this is admitted even in HO parameters. Hence, a process type is a non-trivial generalization of a functional type. The current inference system was also firstly defined with graph grammars [5] and implemented in AGG, since the graph transformation approach is the natural framework to formalize actions in NiMo. They are all subnet transformations, and so is the type inference process as well.



Here we present the type inference system of NiMoToons; the NiMo environment (overviewed in [6]). Graphical typing and incremental inference are described using a textual denotation for type graphs. A type graph unifier operator and a net typing calculus are intended to bridge the gap with the underlying formalism in terms of graph transformation rules. The paper is organized as follows: the next section introduces the syntax and main constructions of NiMo¹. Section 3 presents the graphical representation of types, their interpretation in a textual notation, and the differences between process and function types. Section 4 defines the notion of structural and non-structural equivalence of type descriptors and unification in both cases. Section 5 covers net typing. A set of port connection and composition operators is the basis for the incremental component type

¹ It does not cover evaluation aspects because they are not relevant to the issue of types and can be found in the papers mentioned above.

calculus. All along the paper the topics are illustrated with screen-shot examples. The last section discusses some related work and summarizes our contributions.

2 NiMo Language Elements

NiMo programs are directed graphs with two kinds of nodes: processes and data items. Horizontal arrows represent channels of flowing data streams, and vertical arrows entering a process are non-channel parameters, which can also be processes. Processes can have any number of inputs and outputs, making the use of tuples unnecessary. There are neither patterns nor specific graphical syntax for conditionals. The tokens are: *rounded rectangles* for processes, *circles* (or ovals) for constant values, *black-dots* for duplicators, *hexagons* for data elements, and

green-arrows for non productive outputs () or delayed arguments (). Circles are labelled with their value for atomic types or with their names for symbolic constants of any type, even polymorphic. Hexagon labels are I, R, B, L and F for integers, reals, booleans, lists, and functional processes. Polymorphic data are labelled with ?. In the current version neither user defined types nor Haskell type classes are supported. Ad-hoc polymorphism for functions like > is handled as in Miranda. There are two different processes for real and integer operators. The NiMo syntax makes intensive use of colour. In hexagons and circles it indicates their type, in process names it denotes the evaluation mode, and edges have a state shown as a colored *diamond* to indicate process activation or data evaluation degree. Some program examples can be seen in [7].

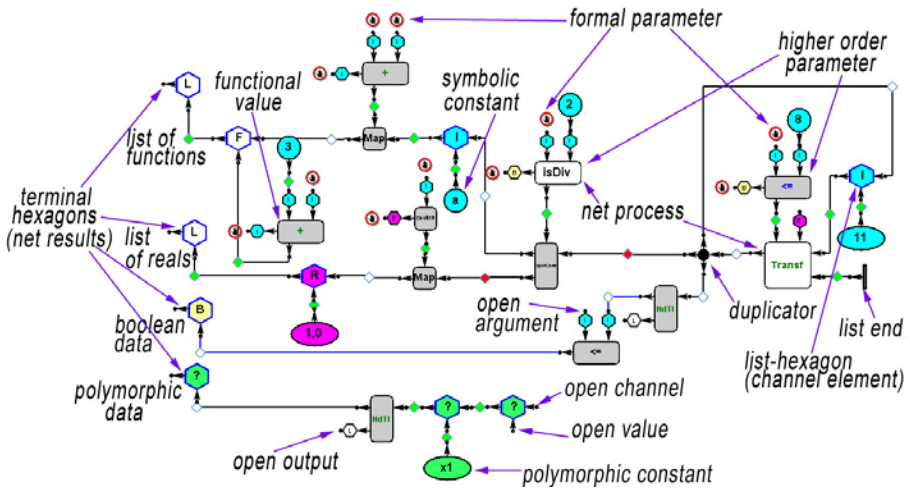


Fig. 1. NiMo program elements

2.1 Interfaces and Connections

All the mentioned nodes are *interfaces* having typed (in/out) connection ports. Interfaces are dragged from a ToolBox and dropped into the workspace where the net is being built (see top of Fig. 2). Clicking on a pair of ports connects them with an edge if both types are compatible; otherwise a failure message is generated. Process interfaces have an F -out port on the bottom. It is not one of their outputs but their value as a functional data.

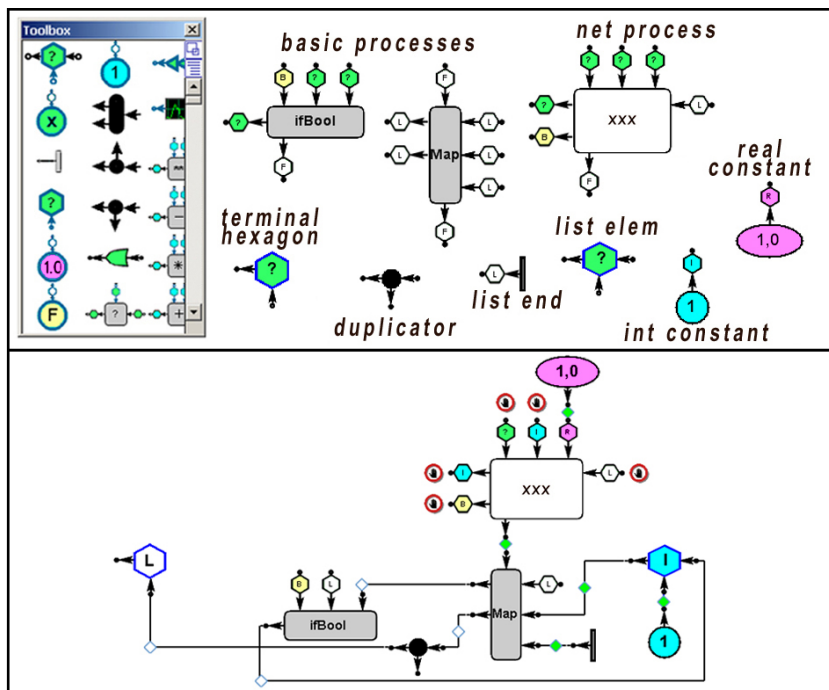


Fig. 2. Interfaces

This special out-port disappears when any output of the process is connected (becoming a potentially active process), or when all its inputs are connected (it is no longer a function). HO parameter processes are connected by their F -out port (as *xxx* on the bottom of Fig. 2). All the other open ports get thus blocked (🔒) to prevent new connections which would change its type. There is a set of built-in processes (grey rounded-rectangles) for basic types and stream processing. It includes multiple output versions of many Haskell prelude functions, as the process *SplitAt*, analogous to the *splitAt* function that can behave also as *take* or as *drop* just by leaving one or the other output open. We call this feature *partial production*, in analogy with the notion of partial application; i.e. there is a symmetry in parameters and results regarding partiality.

Also, some basic processes have configurable arity, as a *Map* with n input and m output channels² (generalizing *map*, *zipWith* and *zipWith3*), *TakeWhile* and *Filter* with n input and output channels, and an *Apply* process.

Terminal hexagon interfaces correspond to the net outputs. Subnets connected to them are considered productive, even being incomplete. In execution all the non-productive subnets are deleted by the garbage collector. For instance, the net in Fig. 2 is productive. Moreover, it is able to produce a result because *Map*₃₋₂ already has enough inputs to act since one of its input channels has a list-end connected, thus it returns a list-end in both outputs. Then the duplicator also returns the list-end.

2.2 Net-Process Definitions

Net processes are user-defined components whose interfaces (the white rounded-rectangles) are defined by means of a parameterisation mechanism. The net in/out open ports that are to be considered as formal parameters or results, are bound to the in/out ports of a configurable interface that is given a name. Afterwards, it can be imported to the Toolbox to be used as a process in a new net and so on, allowing incremental net complexity up to any arbitrary degree.

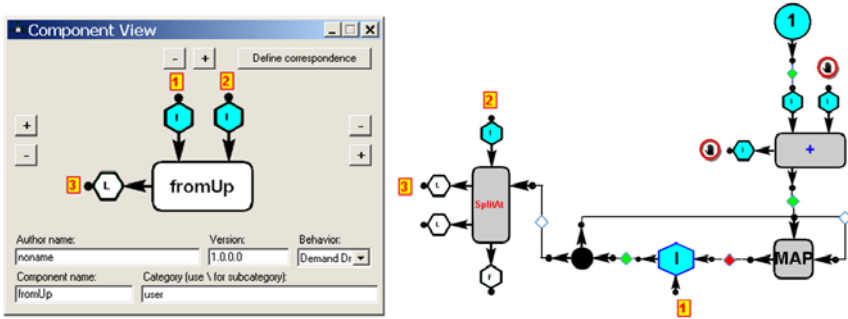


Fig. 3. Net Process definition

Fig. 3 shows an example for the process *fromUp*³ that generates a list with k consecutive integers from the value n , where n and k correspond respectively to the parameters labelled 1 and 2. When the net process acts, the interface is replaced by the net updating the connections according to the bindings. Also, there is a *generic process interface* for building the interface of a not yet defined net process. The user sets the process name and number of channel/non-channel parameters and outputs, and optionally their types (which are all polymorphic by default). In a top down development this allows nets with not yet defined processes to execute. And it is also the means to define recursive processes, i.e. to build a net definition containing the process interface which is being defined.

² We will refer to it as *Map* _{n - m .}

³ In Haskell code: *fromUp n k = x where (x, y) = splitAt k z ; z = n: map (1+) z*.

2.3 Partial Application and Production in HO

In NiMo partial application can be made in any order. In HO parameters, the effective arguments can be delayed by connecting a vertical green-arrow before connecting its F-out port. On the left of Fig. 4, process *ifBool* has the green-arrow at its first input, thus allowing its value to be completed later. It is also the way for binding this port as a second order parameter if the net is defined as a net process⁴. Moreover, in NiMo multiple output processes and even partial

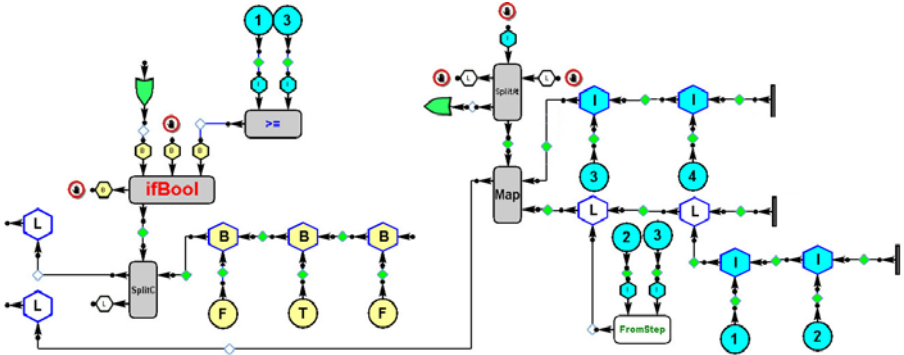



Fig. 4. Delayed argument and partial production

production are allowed in HO parameters. The horizontal green-arrow (in the middle of Fig.4) is connected to the second output of process *SplitAt*. It makes *SplitAt* to behave like *take*, becoming a suitable parameter for a single output *Map*.  is the only interface that can be connected to a process out port without elimination of the F-out port. Once the process is applied the green-arrow disappears.

3 Graphical Typing

As already said, in NiMo type checking and inference is made step by step and locally during net editing. Initially the net is empty. The user adds interfaces and connects pairs of type compatible ports. The full type information of each interface port is carried in a second kind of (optionally visible) graphs, which are updated with each new connection and help to identify what is failing when a connection is rejected. In this section we present the graph representation of types and the textual notation to describe them in a way close to the usual type expressions.

⁴ E.g. if the increment in Fig. 3 were the third *fromUp* parameter instead of being 1.

3.1 Type Graph and Type Descriptors

The net has an associated *type graph*, which is an acyclic and maybe non-connected graph whose nodes are *type hexagons* labelled I, R, B, L, F, O and ?. All ports of every interface are tied to a node in the type graph⁵, and shared subgraphs indicate identical types. In connected ports only the out is tied to the type graph (to avoid arrow duplication). The net type graph is incrementally built during the net construction starting from the type graph associated to each interface that we call its *type descriptor* (TD). TDs fully describe the type of processes and data items. Each interface port is tied to one type hexagon by means of a non-labelled arrow. This hexagon is the root of the port TD and it could be shared by, or included in, another port TD of the interface. In NiMo there are no variable names and this also applies to type variables in polymorphic types. The label ? stands for all the polymorphic types. Sharing a polymorphic hexagon is the graphical equivalent of multiple occurrences of a type variable in a polymorphic type expression. In Fig. 5 we can see the interfaces on the top

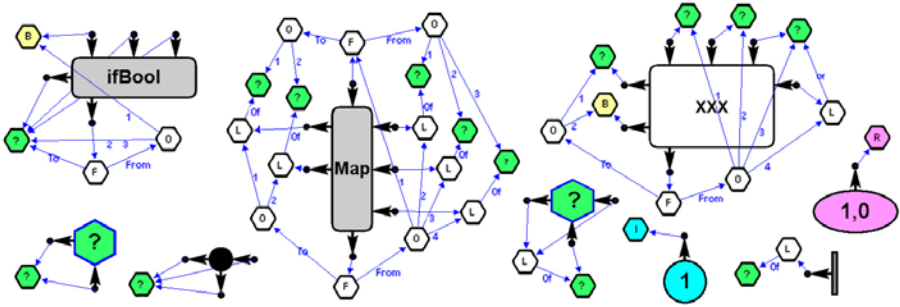


Fig. 5. Type descriptors

of Fig. 2 with their TDs. The F-out port TD of the process interfaces *ifBool*, *Map₃₋₂* and *xxx* describes their type as a functional value. In NiMo a *process type* is a generalization of a functional type, whose graphical representation is a graph rooted with a hexagon F with outgoing edges labelled *From* and *To*. Multiple inputs or outputs in a process type correspond to the subgraphs with an O-hexagon root and edges labelled by numbers. In case of single input or output the corresponding O-hexagon is omitted (as happens with the output of *ifBool*). Note that an O-hexagon never roots a port descriptor; it is not a NiMo type but a subgraph of a F type descriptor. It has as its children the descriptors of the inputs/outputs of the process (thus the F-out port TD contains as sub-graphs the TDs of all the other ports of the interface).

In the textual notation that we will use from now on, \parallel denotes the type constructor O for ordered parallel inputs or results, each ?-hexagon in the TD is denoted by a type variable $?_i$ (or ? if there is only one), and multiple occurrences of

⁵ For an idea of what it looks like, Fig. 10 shows the type graph of the net in Fig. 2.

the same variable in the type expression correspond to a shared $?$ -hexagon. Thus the most general type for processes is denoted by $?_{i1} \parallel \dots \parallel ?_{in} \rightarrow ?_{o1} \parallel \dots \parallel ?_{om}$ where $n, m \geq 0$ $n + m > 0$. The denotation for the *ifBool* type is $B \parallel ? \rightarrow ?$, for *Map₃₋₂* is $(?_1 \parallel ?_2 \parallel ?_3 \rightarrow ?_4 \parallel ?_5) \parallel [?_1] \parallel [?_2] \parallel [?_3] \rightarrow [?_4] \parallel [?_5]$, and for the user process *xxx* is $?_1 \parallel ?_2 \parallel ?_3 \parallel [?_3] \rightarrow ?_4 \parallel B$. Some other examples of process types are $+ : I \parallel I \rightarrow I$; $id : ? \rightarrow ?$; $fibonacci : \rightarrow [I]$ and $sink : ? \rightarrow$. The two last ones are non-functional processes; their interfaces do not have a F-out port. *fibonacci* is a process with no inputs and a single output which is an integer list, and *sink* is a process with no output that consumes its input value. It does not have a Haskell equivalent; its definition would be something like $sink\ x = void$.

4 Type Graph Unification

In order to connect two ports, the editor must first verify that their TDs t and t' can be *unified*; i.e. that there exists a *unifier graph* $t \approx t'$ for them. In this case the connection is made and both ports acquire this common TD; otherwise a failure message is generated. NiMoToons has an option to automatically roll-back partial unifications by recovering the original types each time a connection fails; otherwise they persist to be analyzed and can be explicitly undone afterwards⁶.

The unifier graph exists when the respective TDs are *structurally equivalent*. Roughly, this means that both TDs can be overlapped and all their respective hexagons coincide (same label and number of children), except when one of them is a polymorphic hexagon, in which case the other one hides it. In Haskell-like languages the unification is always structural. A functional type has a single interpretation because all functions have a single result and also a single parameter (the first one), and to be unified both type expressions must be structurally equivalent. Curried and uncurried functions have no equivalent types. But in NiMo processes can be interpreted in one or the other way, and thus non-structural unification is allowed under certain conditions that are described in section 4.2.

4.1 Structural Unification

Fig. 6 shows an example where the F-out port TDs of interfaces f and g , are structurally equivalent. The screen-shot on the right can be obtained by moving the hexagons of both TDs to make them coincide. This allows us to visualize the unifier graph $t \approx t'$ that would result if both TDs were unified⁷. The corresponding port types are $t = I \parallel ?_1 \rightarrow [?_2]$ and $t' = ?_3 \parallel R \rightarrow ?_4$. We can see that the second input of f , the first input of g , and its output, each one having a different polymorphic type on the left, have been replaced by the respective types in the other interface. The resulting type $t \approx t' = I \parallel R \parallel \rightarrow [?_2] = t \langle ?_1 \Leftarrow R; ?_6 \Leftarrow B \rangle = t' \langle ?_3 \Leftarrow I; ?_4 \Leftarrow ?_2 \rangle$

⁶ The same happens when a connection is destroyed (individually or as a result of deleting a connected interface).

⁷ Being both out-ports they cannot be connected but their TDs would be unified e.g. if they were connected as values of two list-items in a same list.

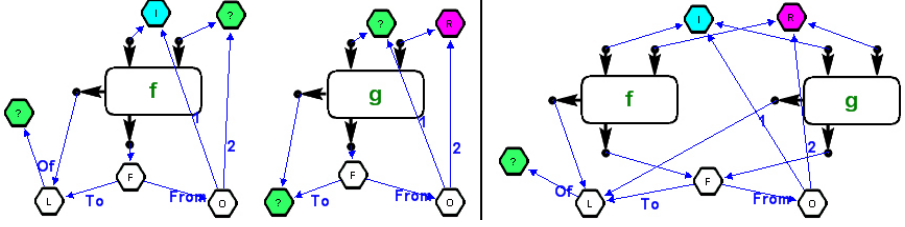


Fig. 6. Structural unification

where the notation $?_i \Leftarrow \delta$ stands for the replacement of the corresponding $?$ -hexagon by a subgraph δ .

If two TDs t and t' unify, the unifier graph $t \approx t'$ is obtained by the fusion of t and t' into a common type graph where, starting from the roots, each pair of corresponding hexagons *collapses* in a single node. This node has as its incoming edges the union of both sets of incoming edges.

The following rules define the (commutative and highest precedence) operator \approx that obtains the unification result in case of structural equivalence:

1. $t \approx t = t$ for t rooted in $\{I, R, B\}$
2. $t \approx ? = t$ (t is not rooted O and $? \notin t$)
3. $[t] \approx [t'] = [t \approx t']$
4. $(t_1 \parallel \dots \parallel t_n) \approx (t'_1 \parallel \dots \parallel t'_n) = t_1 \approx t'_1 \parallel \dots \parallel t_n \approx t'_n$
5. $(ti_1 \parallel \dots \parallel tin \rightarrow to_1 \parallel \dots \parallel to_m) \approx (ti'_1 \parallel \dots \parallel ti'_n \rightarrow to'_1 \parallel \dots \parallel to'_m) =$
 $(ti_1 \parallel \dots \parallel tin) \approx (ti'_1 \parallel \dots \parallel ti'_n) \rightarrow (to_1 \parallel \dots \parallel to_m) \approx (to'_1 \parallel \dots \parallel to'_m)$

Rule 1 is for basic types, i.e two single node TDs with the same label collapsing in a single one. Rule 2 says that a $?$ -hexagon can be substituted by any other TD not rooted O , because O does not represent a tuple type; it is always a subgraph of a process TD. Hence, a single polymorphic input/output cannot be instantiated to multiple inputs/outputs. Besides, the $?$ -hexagon cannot be a proper subgraph of the other TD because a cycle would occur (infinitely recursive type). When a $?$ -hexagon collapses with any other node, the resulting hexagon is the other one (which acquires its incoming edges). This graph replacement of any node $?$ in the TD t by the graph δ ⁸ is denoted as $t \langle ? \Leftarrow \delta \rangle$. Rules 3 and 4 apply when both labels are L or O ; the respective subgraphs are pairwise unified and the collapsed hexagon has (same number of) new outgoing edges, each one of them having as target the respective collapsed hexagons. Rule 5 applies for structurally equivalent TDs rooted F . The collapsed hexagon has as children the unifier graphs of both pairs of children.

4.2 Non Structural Unification

In NiMo two process types with different number of parameters and results can also be unified. Fig. 7 shows that, as happens in Haskell, process $+$ is a

⁸ It can be seen as the equivalent to the Damas-Milner instantiation rule.

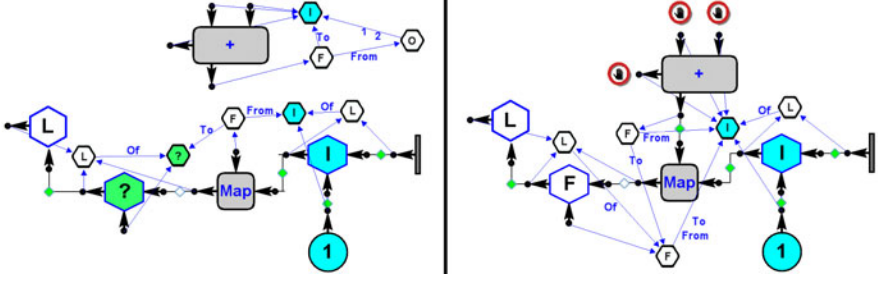


Fig. 7. Carried interpretation of multiple inputs

valid HO parameter for *Map*, in which case the elements in the input channel must be integers, and the result is a channel of functional elements of type $I \rightarrow I$. But the type of $+$ is $I \| I \rightarrow I$, and thus it should unify with $I \rightarrow (I \rightarrow I)$. I.e. in cases like this, there is an implicit conversion among non-structurally equivalent process types. Also the number of outputs could have been different, as happens in Fig. 8. In general, processes with multiple inputs and outputs can be viewed as returning intermediate functional types, i.e. the type of a process with $n > 1$ inputs and m outputs $t_1 \| \dots \| t_n \rightarrow t'_1 \| \dots \| t'_m$ can be implicitly converted to types $t_1 \| \dots \| t_k \rightarrow (t_{k+1} \| \dots \| t_n \rightarrow t'_1 \| \dots \| t'_m)$ for any $k < n$. Thus two non-structurally equivalent process types t and t' could be unified if any of the carried interpretations of t is structurally equivalent to some of those of t' . The idea is that the process with fewer parameters must return a single output, whose type has in turn to unify with the functional type resulting of applying the other one to as many parameters as it has. In this case both *F* nodes collapse, and the new children are the children of the unifier graph root. i.e. the structure of the result changes. The following equation defines the unification result in these cases:

$$6. (t_1 \| \dots \| t_k \| t_{k+1} \| \dots \| t_n \rightarrow to) \approx (t'_1 \| \dots \| t'_k \rightarrow to') = \\ (t_1 \| \dots \| t_k) \approx (t'_1 \| \dots \| t'_k) \rightarrow to' \approx (t_{k+1} \| \dots \| t_n \rightarrow to)$$

Note that all the carried interpretations of a process can be derived from it.

In Fig. 8 the process types of $f : ?_1 \| ?_2 \| [?_3] \rightarrow ?_4 \| ?_5$ and $g : ?_6 \| ?_7 \rightarrow ?_8$ unify because both the first two inputs types unify ($?_1 \| ?_2 \approx ?_6 \| ?_7$) and g has a single output that unifies with a function from the third input of f to its results, i.e. $?_8 \approx ([?_3] \rightarrow ?_4 \| ?_5)$. On the right side⁹ $\tau f \approx \tau g = ?_1 \| ?_2 \rightarrow ([?_3] \rightarrow ?_4 \| ?_5)$, where the collapsed hexagons during the unification correspond to the substitutions $\langle ?_6 \leftarrow ?_1; ?_7 \leftarrow ?_2; ?_8 \leftarrow ([?_3] \rightarrow ?_4 \| ?_5) \rangle$ in the type expression τg , whose result is one of the possible carried interpretations of τf .

⁹ The right side cannot be obtained by overlapping as in Fig. 6. It was obtained by first connecting the *F*-out ports as the values of a pair of connected list-items (then deleted). The unification persists but can be undone by forcing type recalculation.

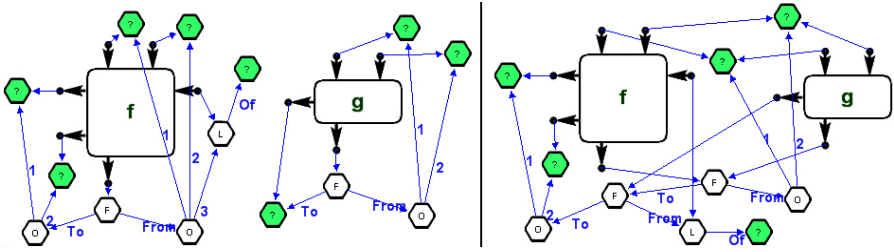


Fig. 8. Non-structural unification

5 Incremental Type Inference for Nets

In functional programming languages variables are used as formal parameters (bound variables), or locally defined elements. Free variables are considered missing definitions and rejected by the compiler. In NiMo there are no variable names. Function parameters are the process interface in-ports, data hexagons with open in-ports can be seen as anonymous free variables, and nets containing open ports are executable. Besides, the multiple outputs of a net can be produced in parallel by non-connected subnets, unlike functional interpreters that always deal with a single (and closed) expression. Hence, the incremental typing of nets has to cover all these cases.

During construction, the net is considered to have as many parameters/results as open in/out ports, which are pairwise closed with each new connection. In terms of graphs the net is a non-connected directed graph. Each new interface adds a component and each connection may reduce the number of connected components (CC). On the other hand, several port TDs in a CC could share subgraphs with ?-hexagons; then, unifying a pair of port TDs can affect any other port TD all along both CCs. But even if the port TDs are identical, the connection changes the types of both interfaces, those of their CCs and thus the net type, because all of them lose (at least) an in or an out open port. In general, connection order is irrelevant except when connecting ports of a process interface having an F-out port. This port makes a difference in the CC type as is discussed in the next section.

5.1 Functional and Non-functional Components

If N is the net under construction, $N = \cup N_i$ where N_i are its CCs. E.g. the net in Fig. 5 has nine single-interface CCs. They are connected in Fig 9 becoming the CCs N_1 and N_2 that result from connecting xxx with $real-const$ in N_2 , and all the other interfaces (in any order) in N_1 . Both CCs are of a different kind.

N_2 is a functional component since it has (a process with) an open F-out port, while all processes in N_1 have lost theirs¹⁰. N_1 has four in and two out open ports. We denote its type as $\{B \parallel [I] \parallel (?_1 \parallel I \parallel ?_2 \rightarrow I \parallel ?_3) \parallel [?_1]\} \rightarrow \{[?_3] \parallel [?_3]\}$,

¹⁰ Because all them have at least one output connected.

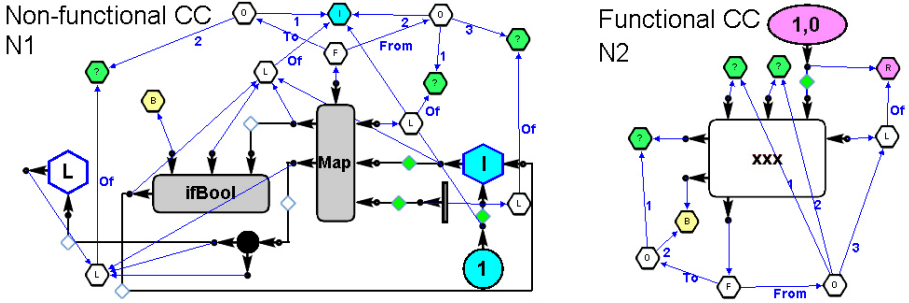


Fig. 9. Both kind of CCs

where curly brackets indicate that the given ordering is arbitrary¹¹. Further connections of these ports can be made in any order; they are *free open ports*. The general form of a non functional CC type τN is $\{t_1 \parallel \dots \parallel t_n\} \rightarrow \{t'_1 \dots \parallel t'_m\}$ with $n, m \geq 0$.

The type of N_2 is different because having a F-out port the connection effect is not uniform (see 2.1 and 2.3). As a functional data xxx can be connected by its F-out port, thus disabling all its open ports. Or xxx could be applied by connecting any of its inputs and the F-out port remains, unless it had only one. But when connecting any output the F-out port disappears, except when connecting a horizontal green-arrow (but not in the only open output). This mutual dependence among the open ports of the interface (*bound open ports*) is denoted in the CC type with a down-arrow preceding the F-out TD (which has as subgraphs all the other open port TDs). In this case $\tau N_2 = \downarrow(?_4 \parallel ?_5 \parallel [R] \rightarrow ?_6 \parallel B)$.

Also, a CC having a F-out port can have non bound in/out open ports as well, as it happens in the net in section 5.5. In this case its type is a compound type of the form $\downarrow(t_1 \parallel \dots \parallel t_n \rightarrow t'_1 \dots \parallel t'_m) \oplus \{t''_1 \parallel \dots \parallel t''_{n_2}\} \rightarrow \{t'''_1 \parallel \dots \parallel t'''_{m_2}\}$ where \oplus is the composition operator described in the next section. Moreover, in the general case a CC could have more than one F-out port and also other free open ports. Therefore the most general type for a CC is:

$\downarrow(T_1 \rightarrow T'_1) \oplus \dots \oplus \downarrow(T_n \rightarrow T'_n) \oplus \{T\} \rightarrow \{T'\}$ where capital T stands for expressions of the form $t_1 \parallel \dots \parallel t_m$.

5.2 Net Type Operators

The operators below perform the transformations on the CC type appropriate for connecting each kind of open port. Operators \neg^{in} , \neg^{out} , \neg^{A-out} and \neg are infix; the 2nd operand is the port index in the given ordering, and \neg^{F-out} is postfix.

¹¹ Ordering is significant for ports of HO parameters, which are clockwise applied, but not for a non-parameterised net. If it finally becomes a net-process (see 2.2) the user selects the open ports to be the parameters and results, and sets both orderings.

1. $\{T\} \rightarrow \{T'\} \neg^{in} k = \{T \neg k\} \rightarrow \{T'\}$ — when connecting the k-th in-port
2. $\{T\} \rightarrow \{T'\} \neg^{out} k = \{T\} \rightarrow \{T' \neg k\}$ — when connecting the k-th out-port
3. $t_1 \parallel \dots \parallel t_n \neg k = \begin{cases} t_1 \parallel \dots \parallel t_{k-1} \parallel t_{k+1} \parallel \dots \parallel t_n & \text{if } n > 1 \\ \emptyset & \end{cases}$ — to remove the k-th parallel input or output
4. $\downarrow (T \rightarrow T') \neg^F \text{-out} = \emptyset$ — when connecting the F-out port
5. $\downarrow (T \rightarrow T') \neg^{in} k = \downarrow (T \neg k \rightarrow T')$ — when connecting the k-th input
6. $\downarrow (T \rightarrow T') \neg^{out} k = \{T\} \rightarrow \{T \neg k\}$ — when connecting the k-th output
7. $\downarrow (T \rightarrow T') \neg^A \text{-out} k = \downarrow (T \rightarrow T' \neg k)$ — when connecting a green-arrow to the k-th output
8. $\downarrow (T \rightarrow \emptyset) = \{T\} \rightarrow \emptyset$ — once all the outputs have green-arrows
9. $\downarrow (\emptyset \rightarrow T) = \emptyset \rightarrow \{T\}$ — once all the inputs have been connected

If the CC has no F-out port it just loses the port (1, 2, 3). When connecting an F-out, all the open ports get closed (4). Any open input can be connected¹² and the F-out persists (5), unless it were the last one (9). When connecting any output the F-out also disappears, thus changing the kind of the CC type (6), except when it is connected with a green-arrow (7 and 8).

On the other hand, if the connected ports belong to different CCs the connection fuses both CCs in a single CC whose free in/out ports are the union of the respective free in/out ports. It is performed by the operator \oplus that groups together the respective sets of types. \oplus is commutative with neutral element \emptyset .

$$\{T_1\} \rightarrow \{T'_1\} \oplus \{T_2\} \rightarrow \{T'_2\} = \{T_1 \parallel T_2\} \rightarrow \{T'_1 \parallel T'_2\}$$

$$\downarrow (T_1 \rightarrow T'_1) \oplus \{T_2\} \rightarrow \{T'_2\} \text{ does not reduce.}$$

5.3 The Type Inference Algorithm

In this section we present the steps to obtain the CC type that results after connecting a pair of unifiable in/out ports. From now on we will denote the connection of an in-port p_1 with an out-port p_2 as $p_1 \prec p_2$. If component N_1 has an in-port p_1 and component N_2 has an out-port p_2 , $N = N_1 p_1 \prec p_2 N_2$ is the CC resulting from the connection $p_1 \prec p_2$.

The type τN is obtained as follows:

1. both TDs are unified: $\tau p_1 \approx \tau p_2 = \tau p_1 \langle \sigma_1 \rangle = \tau p_2 \langle \sigma_2 \rangle$
2. τp_1 and τp_2 are “removed from” τN_1 and τN_2 (applying the fitting \neg operator), thus resulting $\tau N'_1$ and $\tau N'_2$
3. the substitutions σ_1 σ_2 are respectively applied on $\tau N'_1$ and $\tau N'_2$
4. $\tau N = \tau N'_1 \langle \sigma_1 \rangle \oplus \tau N'_2 \langle \sigma_2 \rangle$

Step 1 obtains the unifier graph for both port TDs by performing the substitutions described in section 4. As a result of the unification, other TDs in both CCs might change (if they shared ?-hexagons with the connected ports). In the graph representation such substitutions are made only once on the shared subgraphs. In the equivalent CC type expressions they are performed in step 3,

¹² This rule applies also when connecting a vertical green-arrow; it is not a special case.

once the TDs of the ports closed by the connection have been removed from both CC types, as detailed in the previous section. The last step composes the obtained CC types, thus resulting the single connected component type.

5.4 An Example

The net in Fig. 10 is the result of connecting the components in Fig. 9 by connecting the first in-port¹³ of Map_{3-2} in N_1 and the F-out port of xxx in N_2 .

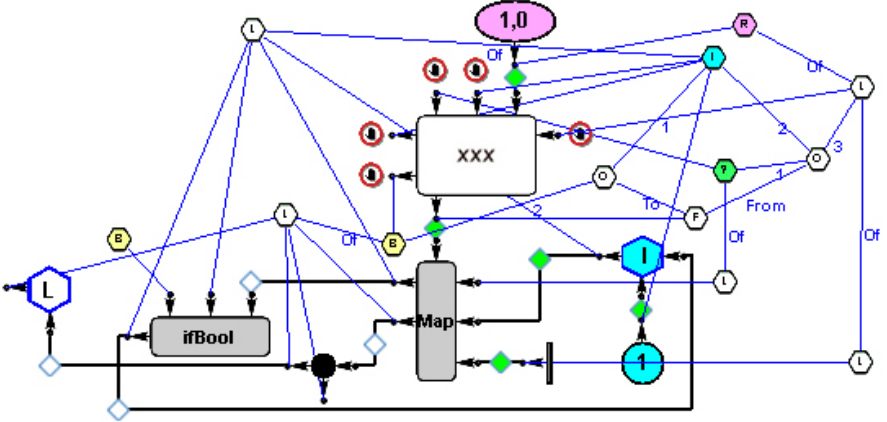


Fig. 10. Single component net

$$\tau p_1 = \tau Map_{3-2}^{in-1} = ?_1 || I || ?_2 \rightarrow I || ?_3, \text{ and } \tau p_2 = \tau xxx^{F-out} = ?_4 || ?_5 || [R] \rightarrow ?_6 || B.$$

The τN calculation proceeds as follows:

1. $\tau p_1 \approx \tau p_2 = ?_1 || I || [R] \rightarrow I || B = \tau p_1 \langle ?_2 \Leftarrow [R]; ?_3 \Leftarrow B \rangle = \tau p_2 \langle ?_4 \Leftarrow ?_1; ?_5, ?_6 \Leftarrow I \rangle$
2. p_1 is the 3rd in-port in the given ordering for τN_1 and p_2 is the N_2 F-out:

$$\begin{aligned} \tau N_1^{-in 3} &= \{B || [I] || \overbrace{(?_1 || I || ?_2 \rightarrow I || ?_3)} \|[?_1]-3\} \rightarrow \{[?_3] || [?_3]\} \\ &= \{B || [I] || [?_1]\} \rightarrow \{[?_3] || [?_3]\} \end{aligned}$$

$$\tau N_2^{-F-out} = \downarrow (?_4 || ?_5 || [R] \rightarrow ?_6 || B)^{-F-out} = \emptyset$$

3. $(\{B || [I] || [?_1]\} \rightarrow \{[?_3] || [?_3]\}) \langle ?_2 \Leftarrow [R]; ?_3 \Leftarrow B \rangle = \{B || [I] || [?_1]\} \rightarrow \{[B] || [B]\}$
 $\emptyset \langle ?_4 \Leftarrow ?_1; ?_5, ?_6 \Leftarrow I \rangle = \emptyset$

4. $\tau N = \{B || [I] || [?_1]\} \rightarrow \{[B] || [B]\} \oplus \emptyset = \{B || [I] || [?_1]\} \rightarrow \{[B] || [B]\}$

Note that the connected ports p_1 and p_2 now have $\tau p_1 \approx \tau p_2$ as their type, and all the port TDs that shared with them a collapsed $?$ -hexagon have also changed. Map_{3-2} has lost the open port, and all the in and out ports of N_2 have been closed with the connection of the F-out port.

¹³ We use the notation X^{in-i} , X^{out-k} and X^{F-out} to refer respectively to the i -th in-port, the k -th output-port and the F-out port of an interface X .

5.5 A Second Example

Fig 11 shows the connection of functional CCs and CCs with green-arrows. On the left side, N_1 contains the horizontal green-arrow $Hgra$, N_2 the process xxx , N_3 the vertical green-arrow $Vgra$, and N_4 the interfaces $Rprod$ (*) and $HdTI$.

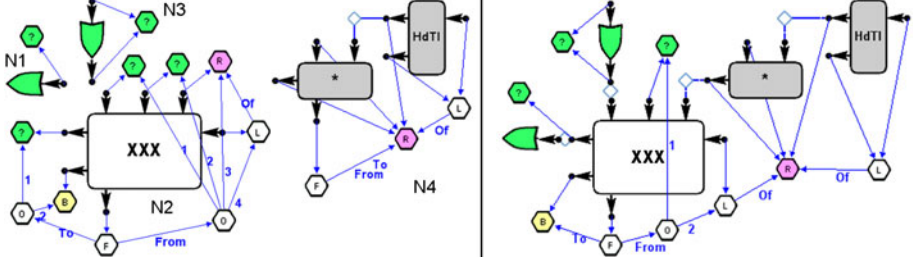


Fig. 11. Connecting green-arrows

$$\begin{aligned} \tau N_1 &= \{?_4\} \rightarrow \emptyset & \tau N_2 &= \downarrow(?_1 \parallel ?_2 \parallel R \parallel [R] \rightarrow ?_3 \parallel B) \\ \tau N_3 &= \{?_5\} \rightarrow \{?_5\} & \tau N_4 &= \downarrow(R \rightarrow R) \oplus \{[R]\} \rightarrow \{[R]\} \end{aligned}$$

The CC N on the right results from having connected in any order¹⁴ the three pairs of ports $p_1 \prec p'_1$, $p_2 \prec p'_2$ and $p_3 \prec p'_3$

$$p_1 = Hgra^{in} \quad p'_1 = xxx^{out1}; \quad p_2 = xxx^{in1} \quad p'_2 = Vgra^{out}; \quad p_3 = xxx^{in3} \quad p'_3 = Rprod^{out}$$

For instance, two of the six possible connection orderings are:

$$((N_1 \ p_1 \prec p'_1 \ N_2) \ p_2 \prec p'_2 \ N_3) \ p_3 \prec p'_3 \ N_4 \quad \text{and} \quad (N_1 \ p_1 \prec p'_1 \ (N_2 \ p_3 \prec p'_3 \ N_4)) \ p_2 \prec p'_2 \ N_3$$

The final result τN is the same; e.g. in the second case it is obtained as follows:

$$(connection\ 1) N_{2..4} = N_2 \ p_3 \prec p'_3 \ N_4$$

$$\tau p_3 \approx \tau p'_3 = R \approx R = \tau p_3 \langle \rangle = \tau p'_3 \langle \rangle$$

$$\tau(N_2 \ p_3 \prec p'_3 \ N_4) = (\tau N_2^{-in3}) \langle \rangle \oplus (\tau N_4^{-out1}) \langle \rangle$$

$$= \downarrow(?_1 \parallel ?_2 \parallel R \parallel [R] \rightarrow ?_3 \parallel B) \oplus (\downarrow(R \rightarrow R^{-1}) \oplus \{[R]\} \rightarrow \{[R]\})$$

$$= \downarrow(?_1 \parallel ?_2 \parallel [R] \rightarrow ?_3 \parallel B) \oplus \{R\} \rightarrow \emptyset \oplus \{[R]\} \rightarrow \{[R]\}$$

$$= \downarrow(?_1 \parallel ?_2 \parallel [R] \rightarrow ?_3 \parallel B) \oplus \{R \parallel [R]\} \rightarrow \{[R]\}$$

$$(connection\ 2) N_{1..4} = N_1 \ p_1 \prec p'_1 \ N_{2..4}$$

$$\tau p_1 \approx \tau p'_1 = ?_4 \approx ?_3 = \tau p_1 \langle ?_4 \Leftarrow ?_3 \rangle = \tau p'_1 \langle \rangle$$

$$\tau(N_1 \ p_1 \prec p'_1 \ N_{2..4}) = (\tau N_1^{-in1}) \langle ?_4 \Leftarrow ?_3 \rangle \oplus (\tau N_{2..4}^{-A-out1}) \langle \rangle$$

$$= (\{?_4^{-1}\} \rightarrow \emptyset) \langle ?_4 \Leftarrow ?_3 \rangle \oplus \downarrow(?_1 \parallel ?_2 \parallel [R] \rightarrow ?_3 \parallel B^{-1}) \oplus \{R \parallel [R]\} \rightarrow \{[R]\}$$

$$= \emptyset \oplus \downarrow(?_1 \parallel ?_2 \parallel [R] \rightarrow B) \oplus \{R \parallel [R]\} \rightarrow \{[R]\}$$

$$= \downarrow(?_1 \parallel ?_2 \parallel [R] \rightarrow) \oplus \{R \parallel [R]\} \rightarrow \{[R]\}$$

$$(connection\ 3) N = N_{1..4} \ p_2 \prec p'_2 \ N_3$$

$$\tau p_2 \approx \tau p'_2 = ?_1 \approx ?_5 = \tau p_2 \langle \rangle = \tau p'_2 \langle ?_5 \Leftarrow ?_1 \rangle$$

$$\tau(N_{1..4} \ p_2 \prec p'_2 \ N_3) = (\tau N_{1..4}^{-in1}) \langle \rangle \oplus (\tau N_3^{-out1}) \langle ?_5 \Leftarrow ?_1 \rangle$$

$$= \downarrow(?_1 \parallel ?_2 \parallel [R] \rightarrow B^{-1}) \oplus \{R \parallel [R]\} \rightarrow \{[R]\} \oplus (\{?_5\} \rightarrow \{?_5^{-1}\}) \langle ?_5 \Leftarrow ?_1 \rangle$$

$$= \downarrow(?_2 \parallel [R] \rightarrow B) \oplus \{R \parallel [R]\} \rightarrow \{[R]\} \oplus \{?_1\} \rightarrow \emptyset$$

$$= \downarrow(?_2 \parallel [R] \rightarrow B) \oplus \{R \parallel [R] \parallel ?_1\} \rightarrow \{[R]\}$$

¹⁴ Since none of the connections closes the other ports.

6 Related Work and Final Remarks

We have presented the graphical type inference system for an incremental and highly interactive development language where editing and execution are interleaved. NiMo programs are graphs that evolve, and so is type information. Hence, the graph transformation approach is the natural framework to model type representation and inference. In this paper we have used a textual notation close to the usual type expressions to describe the type graphs and their evolution. The transformation rules for unification and typing of nets have been presented in terms of a set of operators that perform unification and connection on the equivalent type expressions. However, this textualization shadows some advantages of the graph representation, as having a single shared λ -node instead of multiple occurrences of a quantified variable (hence multiple substitutions).

Regarding the graph transformation approach for modelling types, [8] presents a general framework for typing graph rewriting systems based on the notion of annotated hypergraphs. NiMo nets might be also described in this way, since interfaces can be viewed as directed hypergraphs whose nodes are the ports, internally connected by a hyperedge. Ports are annotated by the corresponding TDs, hence the whole net can be viewed as an annotated hypergraph.

Concerning the graphical and incremental approach, an outstanding asset is that the inference system itself becomes an online visualization tool for type information and failure identification. On this aspect there are several works. GemCut [9] is a graphical viewer for functions in the Haskell-like language CAL; the editor uses CAL compiler's inference system to prevent type errors. TypeTool [10] and System I [11] are web-based tools for visualizing type inference of lambda terms; they are intended to teaching the basis of type inference algorithms for functional languages. Other research focus on tracing the origin of unification failure. [12] proposes a guideline for evaluating the quality of type error diagnosis of type inference systems. It compares several systems and presents the algorithm *Unification Assumption Environments*. The inference process records the local inferences so as to identify all possible sources of inconsistencies. In NiMo, whenever a pair of type hexagons cannot be collapsed, all type ports related to them can be visually identified. Other work on this regard, (not a graphical tool either) is [13] that uses a graph representation with nodes labelled by lambda terms and types from which information is extracted to help in error debugging.

In general, inference systems work on complete terms that can be erroneous, thus producing an error message. In NiMo erroneous nets cannot be edited; messages just indicate incompatibility. Moreover, port compatibility can be tried before connecting simply by moving both TDs hexagons to make them coincide (except in cases of non-structural equivalence).

Another significant point about inference in NiMo is the total absence of type variables; transformations take place directly on the graph structure of the type expressions. The assumption environment is distributed and tied to each term (CC) since every token carries its own type and partially built expressions are always well typed and also carry their type. Besides, NiMo code is bi-dimensional

and can be built in any order; most of the port connections are applications and in NiMo partial application can be made in any order (not only from left to right), hence incremental inference can be made in the user-stated port connection order. On polymorphism handling, interface TDs are originally as polymorphic as they can ever be; hence there is no equivalent for generalization. Instantiation corresponds to the λ -hexagons collapse that occurs when unifying the port TDs.

The other differences come from the data-flow ingredient plus incompleteness. Multiple inputs and outputs required a non-trivial generalization to handle the process type. Non-structural unification is the means to have multiple inputs (then partial application in any order), while keeping the advantages of currying in HO constructs without explicit conversions. On the other hand, typing NiMo nets required treatment of incompleteness and multiple outputs produced by non-connected subnets, in contrast to inference systems that deal with a single and closed term. Application corresponds to connecting a process input in a functional CC. Having multiple inputs and outputs, partial application in any order and partial production also in HO parameters, we needed different operators to define the connection effect vs. the single rule used in functional languages.

Considering the overall development of NiMo, the paradigms fusion was a big challenge that required figuring out many creative solutions to make both models compatible and the graphical realization feasible. But we think it was worth it; the graphic-functional-dataflow nature of NiMo and its incompleteness tolerance result in a very powerful computation model where everything is visible and dynamically modifiable, even the evaluation policy. This allows us to exploit implicit parallelism in a very intuitive way, and to perform symbolic execution in the same framework. We are now exploring its possibilities in simulation and modelling, as well as in generative and multistage programming.

As regards future development, the mixed model opens a range of possible extensions, some of which are hard to imagine in other languages; think for instance that here functions are showable and polymorphic expressions executable. Conversely, some relevant functional language features are not yet included; in particular overloading, type classes, and user defined types (now algebraic types are emulated with functional types), with the consequent implications for inference. But again, the first challenge is making their graphical equivalents stylistic-consistent and manageable, which requires facilities for the compact viewing of complex values. We are now extending the visualization features for net-processes and data channels to cope with any subnet. Besides, in the current version net-process definitions have a single rule with a single interface on the left, whereas Haskell-like languages allow definitions by cases using patterns, making them more modular and readable. The inclusion of this mechanism in NiMo would be a major upgrade far beyond expressiveness, because symbolic execution together with graph patterns open the door to program transformation in the same framework; hence even dynamically.

Acknowledgments. We thank the reviewers for their detailed and helpful comments.

References

1. Turner, D.A.: Miranda: a non-strict functional language with polymorphic types. In: Jouannaud, J.-P. (ed.) FPCA 1985. LNCS, vol. 201, pp. 1–16. Springer, Heidelberg (1985)
2. Clerici, S., Zoltan, C.: A graphic functional-dataflow language. In: Loidl, H.W. (ed.) Trends in Functional Programming. Intellect, vol. 5, pp. 129–144 (2004)
3. AGG: Agg home page (2009), <http://user.cs.tu-berlin.de/~gragra/agg/>
4. Clerici, S., Zoltan, C.: A dynamically customizable process-centered evaluation model. In: PPDP 2009: Proceedings of the 11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, pp. 37–48. ACM, New York (2009)
5. Clerici, S., Zoltan, C.: Graphical type inference. a graph grammar definition. Technical Report LSI-07-24-R, Dept. Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya (July 2007)
6. Clerici, S., Zoltan, C., Prestigiacomo, G.: Nimotoons: a totally graphic workbench for program tuning and experimentation. *Electr. Notes Theor. Comput. Sci.* 258(1), 93–107 (2009)
7. NiMo: Nimo home page (2010), <http://www.lsi.upc.edu/~nimo/Project>
8. König, B.: A general framework for types in graph rewriting. *Acta Inf.* 42(4), 349–388 (2005)
9. Resources (2009), <http://resources.businessobjects.com/labs/cal/gemcutter-techpaper.pdf>
10. Simões, H., Florido, M.: TypeTool - a type inference visualization tool. In: Proceedings of the 13th International Workshop on Functional and (Constraint) Logic Programming (2004), <http://www.dcc.fc.up.pt/typetool/cgi-bin/tt.pl>
11. Church Project: System I (2010), <http://types.bu.edu/modular/compositional/system-i/>
12. Yang, J., Michaelson, G., Trinder, P., Wells, J.B.: Improved Type Error Reporting. In: Proceedings of 12th International Workshop on Implementation of Functional Languages, pp. 71–86 (2000)
13. McAdam, B.J.: Generalising techniques for type debugging. In: Trinder, P.W., Michaelson, G., Loidl, H.W. (eds.) Scottish Functional Programming Workshop. Trends in Functional Programming, Intellect, vol. 1, pp. 50–58 (1999)