Graphical and Incremental Type Inference. A Graph Transformation Approach

Silvia Clerici, Cristina Zoltan, and Guillermo Prestigiacomo

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 visualises 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 analogous 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

1 Introduction

The data flow view of lazy functional programs as process networks was first introduced in [1]. The graphic representation of functions as processes and infinite lists as non-bounded channels, helps to understand the program overall behaviour. The net architecture shows in a bi-dimensional way the chains of function compositions, exhibits the implicit parallelism, and back arrows give an insight of the recurrence relations from the new results and those already calculated. The graphic execution model that the net animation suggests was the starting point for the NiMo language design [2,3]. NiMo is intended to be a workbench for incremental development, testing, debugging and tuning. A small set of graphic primitives allows representing and handling higher order, partial application, non-strict evaluation, and type inference with parametric polymorphism. Since the net is the code but also its computation graph, users have total visibility of the execution internals according to a comprehensible model. Partially defined nets can be executed, dynamically completed or modified and stored at any step, thus allowing incremental development on the fly. Also, execution steps can be undone, acting as an on line tracer and debugger where everything, even the evaluation policy, can be dynamically modified.

In this context, where incompleteness does not inhibit execution, editing a program is a discontinuous process with intervals where code evolves up to the next user interaction, and hence type inference has to be incremental by

force. 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 having typed in and out ports. Net construction is equivalent to building a bi-dimensional term, where sub-expressions are like puzzle pieces that can be pairwise connected in any order, provided their shapes fit together, i.e. both port types unify, thus ensuring type safeness by construction. The full type information associated to each interface port is carried up by means of a second kind of graphs, and updated with each new connection. The user visualises the type information evolution, and can identify the incompatibilities when a connection is rejected. The type inference system, though in essence based on the same principles [4], has some significant differences with its analogous in languages like Haskell. Besides of being graphical and incremental, the data flow ingredient implies to cope with processes with any number of outputs, and curried-uncurried interpretation of multiple inputs. Partial application can be made in any order, and multiple outputs can be left partially disconnected as well. In the current version multiple output processes are also admitted as higher order parameters, maintaining this multiple behaviour. Therefore the process type is a non-trivial generalization of a functional type.

The type inference system implementation was based on a complete graph grammar definition that was initially implemented in AGG. This preliminar version is described in [5]. In fact, the graph transformation approach is the natural framework to formalize actions in NiMo, since they are all visualised as a subnet transformation, and so is type inference as well. On the other hand, in NiMo there are not variable names of any kind (hence not variable substitution either); type identity is represented as a shared type node. In this paper we present the current version of the type inference system of NiMoToons; the NiMo environment [6]. Graphical typing and incremental inference are here described in terms closer to the usual type inference formalism. A textual denotation for type graphs and a typing calculus intend to bridge the gap between the underlaying specification in the graph transformation framework and the classical approach.

The paper is organized as follows: In the next section we introduce the syntax and main constructions of NiMo. Section 3 presents the graphical representation of type information, its interpretation in a textual notation, and discusses the differences between process types and functional types. Section 4 defines the notion of structural and non-structural equivalence of type descriptors, and graphical type unification in both cases. Section 5 covers net typing. A set of port connection typing operators and a composition operator to connect components are the basis for the incremental component type calculus. All along the paper the topics are illustrated with screen-shots examples. The paper ends with a discussion of some type visualization tools and a summary of our contributions.

2 NiMo language elements

NiMo programs are oriented graphs with two kind 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 graphic syntax for conditionals. The kinds of nodes are: rectangles for processes, circles (or ovals)

for constant values, black-dots for duplicators, hexagons for data elements, and green-arrows for open connections or formal parameters. Circles are labelled with

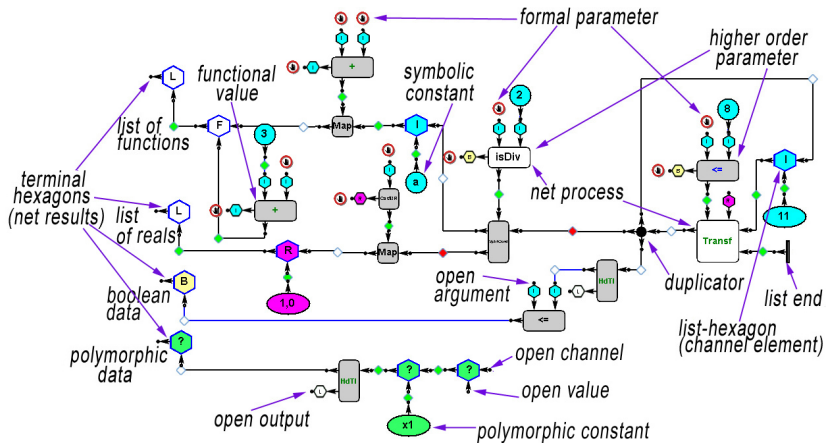

Fig. 1. A NiMo program example

their value for atomic types, and since symbolic evaluation is allowed, labels can also be names for 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 **?**. In the current version neither user defined types nor Haskell type classes are supported. Ad-hoc polymorphism for functions as = or > is handled as in Miranda. For arithmetic operations there are two different processes for reals and integers. The NiMo syntax makes intensive use of colour. Hexagons and circles are coloured according their type, the process name's colour denotes its evaluation mode, and edges have a state indicating process activation or data evaluation degree that is shown as a colored *diamond*.

2.1 Interfaces and Connections

All the mentioned nodes are *interfaces* having typed (in /out) connection ports. Interfaces are dragged from a ToolBox (see left side of Figure 2) and dropped into the workspace where the new net is being built. Clicking on a pair of ports connects them with an edge if both types are compatible; otherwise a failure message is generated. Thus nets are type safe by construction. Let's observe that on the left of Figure 2, the three process interfaces have an *F-out* port on the bottom. It is not one of their outputs but their value as a functional data. This special out-port disappears whenever one output of the process is connected(it is now a potentially active process that cannot be considered data), or all its inputs are connected (once completely applied it is no longer a function). Higher-order parameter processes are connected by its *F-out* port (as it happens with the net process *xxx* on the right of Figure 2). When the connection is made, all the process interface ports not yet connected get blocked (red circle) to prevent new connections, otherwise its value as a functional data (and of course its type)

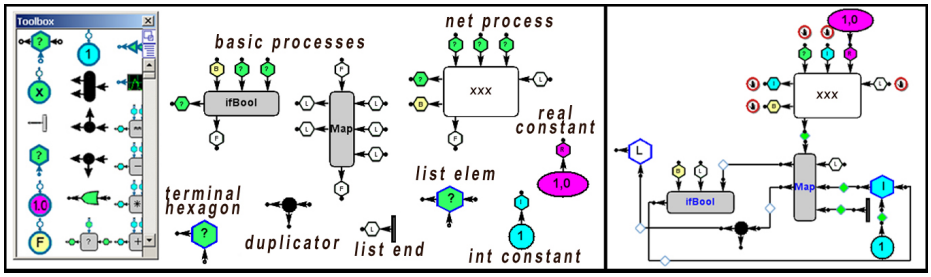

Fig. 2. Interfaces

would change. There is a set of predefined processes (grey rectangles) for basic types and stream processing, including multiple output versions of many Haskell prelude functions. For instance the process *SplitAt* is analogous to the *splitAt* function returning a pair of lists, but it can behave also as *take* or as *drop* just by leaving one or the other output disconnected. We will refer to this multiple behavior as *partial resulting*, in analogy with the notion of partial application, i.e. there is a symmetry in parameters and results regarding partiality. Also, several basic processes have configurable arity, as a *Map* with n input and m output channels (generalizing functions *map*, *zipWith* and *zipWith3*), a *TakeWhile* and a *Filter* with as many input as output channels, and an *Apply* process.

2.2 Net process definitions

Net processes are user-defined components whose interfaces (the white rectangles) are defined by means of a parameterisation mechanism. The open in/out

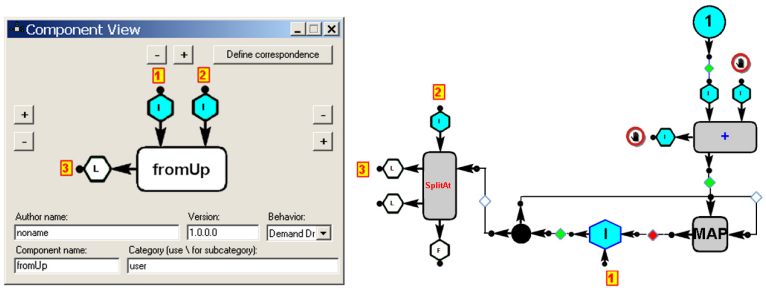

Fig. 3. Net Process definition

ports of the net to be considered the formal parameters and 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.

Figure 3 shows an example for the process *fromUp* 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. The equivalent Haskell code is *fromUp n k = x where(x, y) = splitAt k z ; z= n: map (1+) z*. When the net process has to act, 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 name and number of channels/non-channel parameters and outputs. In a top down development this allows nets with not yet defined processes to execute, and is also the means to define recursive processes.

2.3 Partial application and partial resulting

The multiple inputs of a process can be interpreted in a curried way depending on the context, and partial application can be made in any order. Also, the effective arguments for the application can be delayed. On the left of Figure 4, process *ifBool* acting as a higher order parameter has a 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 process. For instance, if the increment in Figure 3 had been the third parameter

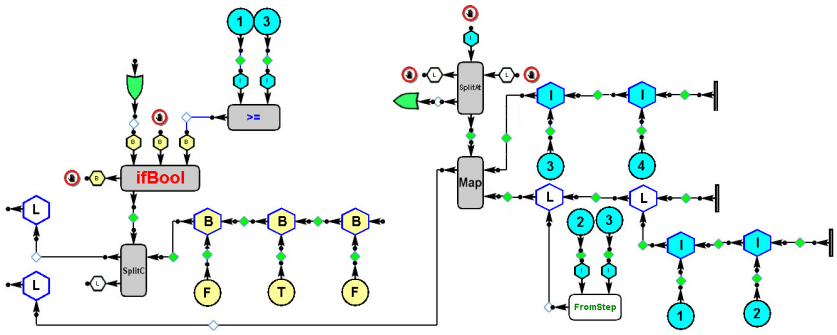

Fig. 4. Delayed argument and partial resulting

of *fromUp,* instead of being 1. Moreover, in NiMo multiple output processes and even partial resulting is allowed in higher order parameters. This is also indicated by a green arrow. At the right in Figure 4 the higher order parameter of *Map* is the process *SplitAt* acting as *take*, i.e. returning a single output, and being therefore a suitable parameter for a single output *Map*. When the process is applied, the horizontal green arrow causes this output to get disconnected.

3 Graphical typing

As already said, in NiMo type checking and inference is made step by step and locally during the net edition. Initially the net is empty. The user adds interfaces (net components) and connects pairs of type compatible ports. To ensure type compatibility, the full type information associated to each interface port is carried up, and updated each time a new connection is made, by means of a second kind of graphs which are optionally visible.

3.1 Type graph and type descriptors

The net has an associated *type graph* that can be made visible. All ports of every interface are tied to a node in the type graph, and shared sub graphs indicate identical types. In connected ports only the out is tied to the type graph (to avoid redundant edges). For an idea of what a type graph looks like, Figure 10 shows the type graph of the net on the right of Figure 2. The net type graph is incrementally built during the net construction starting from the *type descriptor*(TD) of each interface, and its evolution is optionally visible. This makes easy to identify what is failing when a connection is rejected. TDs fully describe the type of processes and data items. They are directed acyclic graphs whose nodes are *type hexagons*, each interface port is tied to one of them 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 into a type expression. In Figure 5 we can see the interfaces on the

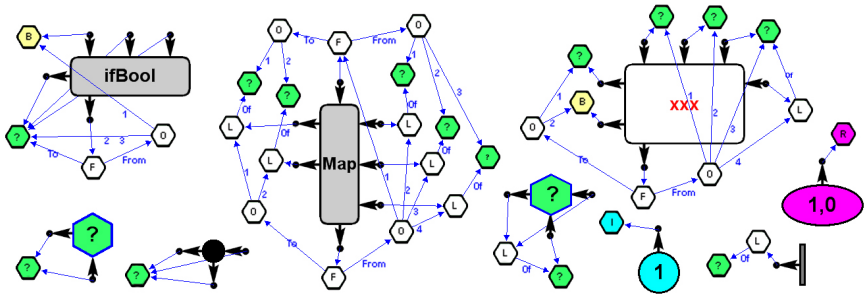

Fig. 5. Type descriptors

left of Figure 2 with their TDs. The type graph tied to the F-out port of the process interfaces *ifBool*, *Map* 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. The O node has as its children the descriptors of the inputs/outputs of the process (thus the F-out port TD contains as sub-graphs all the other ports descriptors of the interface). 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. In the textual notation that we will use from now on, $\|$ denotes the type constructor O for ordered parallel inputs or results, each ?-hexagon in the TD is denoted by a type variable $\mathbf{?}_i$ (or **?** if there is only one), and multiples occurrences of the same variable in the type expression correspond to a shared ?-hexagon. Thus the denotation for the type of process *ifBool* is $B\|\mathbf{?}\|\mathbf{?}\rightarrow\mathbf{?}$,

for Map_{3-2} is $(?_1\|?_2\|?_3\rightarrow ?_4\|?_5)\|[?_1]\|[?_2]\|[?_3]\rightarrow[?_4]\|[?_5]$, and the type of the user process *xxx* is $?_1\|?_2\|?_3\|[?_3]\rightarrow ?_4\|B$. The most general type for processes is $?_{i1}\|\ldots\|?_{in}\rightarrow ?_{o1}\|\ldots\|?_{om}$ where $n, m \geq 0 \quad n+m > 0$, and some other examples of process types are $+ : I\|I\rightarrow I$; $id : ?\rightarrow ?$; $fibonacci : \rightarrow[I]$ 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 for a couple of ports to be connected, the editor must first verify that its TDs t and t' can be *unified*; i.e. that exist an *unifier graph* t≈ t' for them. In this case the connection is made and both ports acquire this common TD, otherwise a failure message is generated. The unifier graph exists when the respective graphs 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. Figure 6 shows an example for the F-out ports of interfaces f and g, which are structurally equivalent. The respective port types are $t = I\|?_1\rightarrow[?_2]$

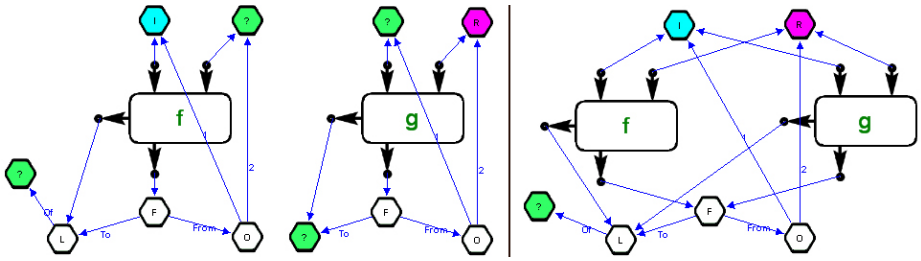

Fig. 6. Structural unification

and t'=$?_3\|R\rightarrow ?_4$ 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 unified. 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 $t \approx t' = I\|R\rightarrow[?_2] = t\langle ?_1 \Leftarrow R\rangle = t'\langle ?_3 \Leftarrow I; ?_4 \Leftarrow [?_2]\rangle$ where the notation $?_i \Leftarrow \tau$ corresponds to the replacement of a ?-hexagon by a subgraph τ.

4.1 Structural unification

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 4.3.

The following rules define the conditions for structural unification of TDs:

1. unify(t , t) for t rooted in {I, R, B}
2. unify (t , **?**) for t not rooted O and **?** $\not\subset t$
3. unify ([t], [t']) $\Leftrightarrow$ unify (t , t')
4. unify$((t_1\|\ldots\|t_n),(t'_1\|\ldots\|t'_n)) \Leftrightarrow$ Unify $(t_k, t'_k)\ \forall\, 1 \leq k \leq n$
5. unify $((ti_1\|\ldots\|ti_n{\rightarrow}to_1\|\ldots\|to_m),(ti'_1\|\ldots\|ti'_n{\rightarrow}to'_1\|\ldots\|to'_m)) \Leftrightarrow$
unify $((ti_1\|\ldots\|ti_n),(ti'_1\|\ldots\|ti'_n)) \wedge$ unify$((to_1\|\ldots\|to_m),(to'_1\|\ldots\|to'_m))$

The restriction in 2 states 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 specialized by multiple inputs/outputs. And the ?-hexagon cannot be a proper subgraph of the other TD because a cycle would occur (infinitely recursive type).

4.2 The unifier graph

If two TDs t and t' unify, the unifier graph t≈t' is obtained by the fusion of t and t' into a common type graph, where 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 (where the new hexagon is now the target node). For identical basic types the unification ends. When a ?-hexagon collapse with any node (not labelled O nor including the ?-hexagon), the resulting hexagon will be the other one (which maintains its outgoing edges). This graph replacement of the node $?_i$ in the TD t by the subgraph τ is denoted as $t\langle ?_i{\Leftarrow}\tau\rangle$. 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 their target the respective collapsed hexagons. And the same happens for structurally equivalent TDs rooted F.

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 **?** $\not\subset t$)
3. [t] $\approx$ [t'] = [t $\approx$ t']
4. $(t_1\|\ldots\|t_n) \approx (t'_1\|\ldots\|t'_n) = t_1 \approx t'_1\|\ldots\|t_n \approx t'_n$
5. $(ti_1\|\ldots\|ti_n{\rightarrow}to_1\|\ldots\|to_m) \approx (ti'_1\|\ldots\|ti'_n{\rightarrow}to'_1\|\ldots\|to'_m) =$
$(ti_1\|\ldots\|ti_n) \approx (ti'_1\|\ldots\|ti'_n)\ \rightarrow\ (to_1\|\ldots\|to_m)\ \approx (to'_1\|\ldots\|to'_m)$

4.3 Non structural unification

In NiMo two process types with different number of parameters and results could also be unified. For instance Figure 7 shows that, as happens in Haskell, process + is a valid actual 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

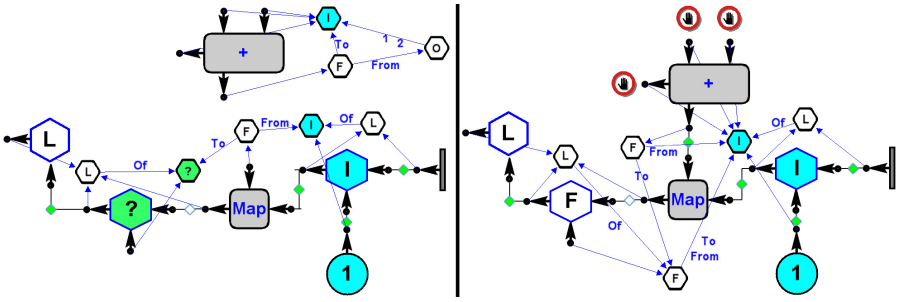

Fig. 7. curried interpretation of multiple inputs

equivalent process types. Also the number of outputs could have been different, as happens in Figure 8. In general, processes with multiple inputs and outputs can also be interpreted as returning intermediate functional types, i.e. the type of a process with $n > 1$ inputs and m outputs $t_1\|\ldots\|t_n \rightarrow t'_1\|\ldots\|t'_m$ can be implicitly converted to types $t_1\|\ldots\|t_k{\rightarrow}(t_{k+1}\|\ldots\|t_n{\rightarrow}t'_1\|\ldots\|t'_m)$ for any $k < n$. Thus two non-structurally equivalent process types can be unified. The idea is that the process with fewer parameters must return a single output, whose type has to unify with the functional type resulting of having applied the second process to as many parameters as the first one 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 rules for non-structural unification complete the predicate *unify* defined in the previous section:

6. unify $(t_1\|\ldots\|t_k\|t_{k+1}\|\ldots\|t_n{\rightarrow}to))$, $((t'_1\|\ldots\|t'_k{\rightarrow}to') \Leftrightarrow$
   $\text{unify}(\text{t}_1\|\ldots\|t_k\ ,\ t'_1\|\ldots\|t'_k)$ & $\text{unify}(\text{to'}\ ,\ t_{k+1}\|\ldots\|t_n{\rightarrow}to)$
7. Anymore unify

And the following equation define the unification result in this case

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

Note that all the possible "curried interpretations" of a process with n inputs and m outputs can be derived from this rule.

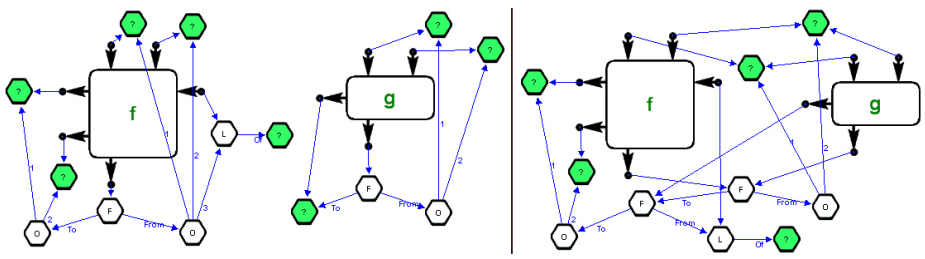

Fig. 8. Non-structural unification

In Figure 8 the process types of f and g unify because the first two inputs types of both functions unify, and g has a single polymorphic output, which can be unified with a function from the third output of f to its results. Their types are $f : ?_1 \| ?_2 \| [?_3] \rightarrow ?_4 \| ?_5$ and $g : ?_6 \| ?_7 \rightarrow ?_8$. The unifier graph on the right side is $\tau f \approx \tau g = (?_1 \| ?_2 \approx ?_6 \| ?_7) \rightarrow ?_8 \approx ([?_3] \rightarrow ?_4 \| ?_5) = ?_1 \| ?_2 \rightarrow ([?_3] \rightarrow ?_4 \| ?_5)$, and the collapsed hexagons during the unification correspond to the following substitutions in the type expression τg , whose result is one of the possible curried interpretations of τf: $(?_6 \| ?_7 \rightarrow ?_8) \langle ?_6 \Leftarrow ?_1; ?_7 \Leftarrow ?_2; ?_8 \Leftarrow ([?_3] \rightarrow ?_4 \| ?_5) \rangle = ?_1 \| ?_2 \rightarrow ([?_3] \rightarrow ?_4 \| ?_5)$

5 Incremental type inference

In functional languages variables are used as formal parameters (bound variables) in function definitions, or locally defined function or constant names. Expressions having free variables cannot be evaluated by the interpreter; they are interpreted as missing definitions and therefore are discharged by the compiler. In NiMo nets containing open ports are executable, and there are no variable names. The function parameters are the process interface in-ports, and data hexagons with open in-ports can be seen as anonymous free variables. During construction, the net can be considered to have as many parameters as open in-ports and as many results as open out-ports. But the ordering of these inputs and outputs is not relevant. Ordering is significant for process interfaces because they can be used as higher order parameters, which are clockwise applied, but not for a non-parameterised net. If it is finally defined as a net-process (see section 2.3) the user decides which subset of open ports are to be the parameters and results and the respective orderings.

Connecting a process input corresponds to function application or function composition and most of the possible parameters and results are progressively cancelled. In terms of graphs the net is a non-connected directed graph. Adding a new interface means adding a new component, and connecting a pair of ports may reduce the number of connected components (CC). On the other hand, since several port TDs in a CC can share subgraphs containing ?-hexagons, when two ports are connected the effect of unifying both types can affect any other port type all along both CCs. But even if both port types are identical, the connection will change the types of both interfaces, those of their CCs and thus the net type, because all of them loose an in and an out port.

5.1 Typing nets

If N is the net under construction, $N = \cup N_i$ where N_i are its CCs. For instance, the net in Figure 2 has nine CCs, each one having a single interface. In Figure 9 there are two CCs N_1 and N_2, which are the result of having connected *xxx* with *real-const* in the CC N_2, and all the other interfaces[1] in CC N_1. The types of both CCs have a different kind because N_2 has an open F-out port. Processes in N_1 are all operative because they all have at least one of their outputs connected, and thus none of the out-ports is an F-out port. Moreover, N_1 though not complete

[1] The respective in and out ports could have been connected in any order and the resulting CC type would have been the same.

is nevertheless executable because it already has a connected terminal hexagon whose value could be produced[2]. N_1 has four in-ports and two out-ports not yet connected and therefore its type is $\{B\|[I]\|(?_1\|I\|?_2{\rightarrow}I\|?_3)\|[?_1]\}{\rightarrow}\{[?_3]\|[?_3]\}$, where curly brackets indicate that the given ordering is arbitrary. On the other

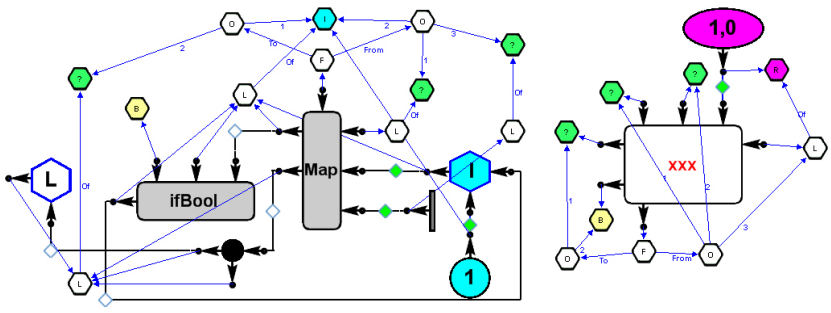

Fig. 9. Two CCs

hand, CC N_2 has a different CC type because *xxx* can still be interpreted as functional data and connected by its F-out port. But also, it could be applied to any of its inputs, or connected by an output port thus becoming an operative process; depending on which kind of port is connected the effect of the connection will be different. As already said, once all the in-ports or at least one of the out ports are connected, the F-out port disappears. Conversely, when it is connected, all the remaining ports become disabled. This mutual dependence among the open ports of the interface is denoted in the CC type with a down-arrow representing the F-out port, whose TD has as subgraphs all the other ones. In this case $N_2 : \downarrow(?_4\|?_5\|[R]{\rightarrow}?_6\|B)$.

5.2 Connecting components

If X_1 and X_2 are interfaces in CCs N_1 and N_2 (which could be the same), and the i-th in-port of X_1 is compatible with the k-th out-port of X_2, their connection $X_1^{in-i} \prec X_2^{out-k}$ modifies both interfaces TDs (each will have at least one open port less). N_1 and N_2 become a single CC N where, due to the unification, all the remaining open port types sharing ?-hexagons with any of them could have changed. For instance, let's suppose that the ports p_1 and p_2 to be connected, are respectively the first in-port of Map_{3-2} in N_1 and the F-out port of *xxx* in N_2. i.e. $\tau p_1 = \tau Map_{3-2}^{in-1} = \mathbf{?}_1\|I\|\mathbf{?}_2{\rightarrow}I\|\mathbf{?}_3$, and $\tau p_2{=}\tau xxx^{F-out} = \mathbf{?}_4\|\mathbf{?}_5\|[R]{\rightarrow}\mathbf{?}_6\|B$. Types τp_1 and τp_2 unify, $\tau p_1{\approx}\tau p_2 = \mathbf{?}_1\|I\|[R]{\rightarrow}I\|B$, and the connection $p_1{\prec}p_2$ produces the fusion of the respective CCs in the single component net N that can be seen in Figure 10. Note that the connected ports p_1 and p_2 now have $\tau p_1{\approx}\tau p_2$

[2] In fact, this is the case here because Map_{3-2} already has enough inputs to act, since one of its input channels is empty (has a list-end connected), and will return a list-end in both outputs whatever its higher order parameter may be. Then the duplicator could return this final result also.

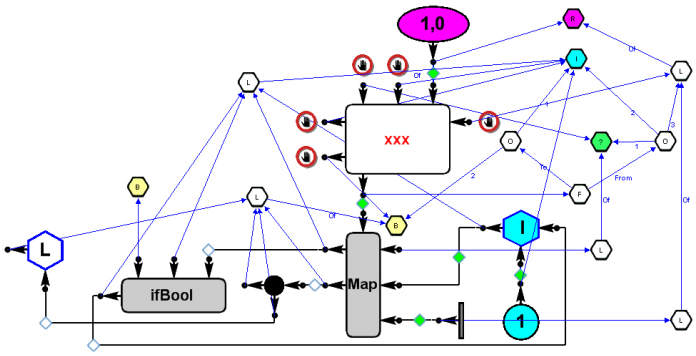

Fig. 10. Single component net

as its type, and all the port TDs that shared with them a collapsed ?-hexagon have also changed. N has one in-port less than N_1 since Map_{3-2} loosed one openne port, and all the in and out ports of N_2 has been cancelled with the connection of the F-port, since they became parameters and results of a higher order parameter. The resulting CC type (up to reordering) is $\{B\|[I]\|[\mathbf{?}_1]\}\rightarrow\{[B]\|[B]\}$.

5.3 Net typing operators

Ports in a functional component are related, and connecting one of them may close some one else. It does not happen in operative components. So the effect on the CC type of a given connection is not uniform.

The following set of operators $\neg$ perform the corresponding transformations for each case. Operators $\neg^{in}$, $\neg^{out}$, $\neg^{A-out}$ and $\neg$ are infix, and $\neg^{F-out}$ is postfix.

1. $\{t\}\rightarrow\{t'\}\neg^{in}k = \{t\neg k\}\rightarrow\{t'\}$ k-th in-port (in a given order) is connected
2. $\{t\}\rightarrow\{t'\}\neg^{out}k = \{t\}\rightarrow\{t'\neg k\}$ k-th out-port (in a given order) is connected
3. $t_1\|\ldots\|t_n\neg k =$ if $n > 1$ then $t_1\|\ldots\|t_{k-1}\|t_{k+1}\|\ldots\|t_n$ else $\emptyset$ k-th parallel input or output is removed
4. $\downarrow(t\rightarrow t')\neg^{F-out} = \emptyset$ the F-out port is connected
5. $\downarrow(t\rightarrow t')\neg^{in}k = \downarrow(t\neg k\rightarrow t')$ partial application in the k-th input
6. $\downarrow(t\rightarrow t')\neg^{out}k = \{t\}\rightarrow\{t\neg k\}$ k-th output is connected
7. $\downarrow(t\rightarrow t')\neg^{A-out}k = \downarrow(t\rightarrow t'\neg k)$ green arrow connected to the k-th output
8. $\downarrow(t\rightarrow\emptyset) = \{t\}\rightarrow\emptyset$ all the outputs have green arrows
9. $\downarrow(\emptyset\rightarrow t) = \emptyset\rightarrow\{t\}$ all the inputs are connected

If the CC has no F-out port it just loose this port (1, 2, 3). Having an F-out, when it is connected all the open ports get closed (4). Any open input can be connected 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). As said in 2.3, the green arrow is the only interface that can be connected to a process output without disappearance of the F-out port. In section 5.6 this case is covered.

On the other hand, connection fuses both CCs in a single CC whose in/out ports are the union of the respective in/out ports. It is performed by the operator

$\oplus$ which groups the respective in as well as out port types of its operands that are not bound to a F-out port. $\oplus$ is commutative with neutral element $\emptyset$:

$\{t_1\}\rightarrow\{t'_1\} \oplus \{t_2\}\rightarrow\{t'_2\} = \{t_1\|t_2\}\rightarrow\{t'_1\|t'_2\}$

$\downarrow(t\rightarrow t') \oplus \{t\}\rightarrow\{t'\}$ does not reduce.

5.4 The type inference algorithm

If $N = N_1 p_1 \prec p_2 N_2$ is the CC resulting from connection $p_1 \prec p_2$, 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. $\tau N = \tau N'_1\langle\sigma_1\rangle \oplus \tau N'_2\langle\sigma_2\rangle$

5.5 Example1

The τN calculus for the net in Figure 10 proceeds as follows:

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

$$\tau N_1 \neg^{in} 3 = \{B\|[I]\|\overbrace{(?_1\|I\|?_2\rightarrow I\|?_3)}\|[?_1]\neg 3\}\rightarrow\{[?_3]\|[?_3]\}$$
$$= \{B\|[I]\|[?_1]\}\rightarrow\{[?_3]\|[?_3]\}$$
$$\tau N_2 \neg^{F-out} = \downarrow(?_4\|?_5\|[R]\rightarrow ?_6\|B)\neg^{F-out} = \emptyset$$

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

5.6 Example2

Figure 11 shows an example where green arrows (see section 2.3) and incomplete subnets are connected to a functional CC. On the left side of the figure there are

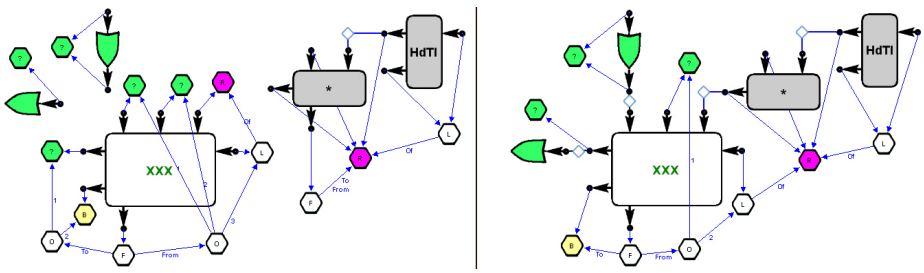

Fig. 11. Connecting green arrows

four CCs, say N_1, N_2 and N_3 respectively containing the horizontal green arrow

$grArw$, the process xxx and the vertical green arrow $grArw2$, and N_4 containing the interfaces $rProduct$ and $HdTl$.

$\tau N_1 = \{?_4\}\rightarrow\emptyset \qquad \tau N_2 = \downarrow(?_1\|?_2\|R\|[R]\rightarrow ?_3\|B)$

$\tau N_3 = \{?_5\}\rightarrow\{?_5\} \qquad \tau N_4 = \downarrow(R\rightarrow R) \oplus \{[R]\}\rightarrow\{[R]\}$

The CC N on the right results from having connected the three pairs of ports $p_1{=}grArw^{in}\, p_1' = xxx^{out1}\, p_2 = xxx^{in1}\, p_2'{=}grArw2^{out}\, p_3 = xxx^{in3}\, p_3'{=}rProduct^{out}$ As can be seen it has two in and one out ports related to the F-out port of xxx, and also other three in and one out ports. Its type (up to renaming and curly brackets reordering) is $\downarrow(?_2\|[R]\rightarrow B) \oplus \{?_1\|[R]\|R\}\rightarrow\{[R]\}$. Connections can be made in any order, for instance c1, c2, c3 thus τN calculus proceeds as follows: $(c1)N_{2.4} = N_2\, p_3{\prec}p_3' N_4 \quad (c2)N_{1.2.4} = N_1\, p_1{\prec}p_1' N_{2.4} \quad (c3)N = N_{1.2.4}\, p_2{\prec}p_2' N_3$

$p_3{\approx}p_3'{=}R{\approx}R{=}p_3\langle\rangle{=}p_3'\langle\rangle \qquad p_1{\approx}p_1'{=}?_4{\approx}?_3{=}p_1\langle ?_4{\Leftarrow}?_3\rangle{=}p_1'\langle\rangle$

$p_2{\approx}p_2'{=}?_1{\approx}?_5{=}p_2\langle\rangle{=}p_2'\langle ?_5{\Leftarrow}?_1\rangle$

$= \downarrow(?_1\|?_2\|R\|[R]\neg 3 \rightarrow ?_3\|B) \oplus (\downarrow(R\rightarrow R\neg 1) \oplus \{[R]\}\rightarrow\{[R]\})$

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

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

$\tau(N_1\, p_1{\prec}p_1'\, N_{2.4}) = (\tau N_1 \neg^{in} 1)\langle ?_4{\Leftarrow}?_3\rangle \oplus (\tau N_{2.4} \neg^{A-out} 1)\langle\rangle$

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

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

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

$\tau(N_{1.2.4}\, p_2{\prec}p_2'\, N_3) = (\tau N_{1.2.4} \neg^{in} 1)\langle\rangle \oplus (\tau N_3 \neg^{out} 1)\langle ?_5{\Leftarrow}?_1\rangle$

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

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

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

6 Related work and final remarks

We have presented a graphic type inference system for a development language where edition an execution are interleaved. Being graphic and incremental, the inference system itself becomes an online visualisation tool for type information and error identification. There are several languages or tools for understanding the type inference process. GemCut [7] 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 [8] and System I [9] are web-based tools for vizualizing type inference of lambda terms, they are oriented to teaching the basis of type inference algorithms for functional languages. Other works focus on tracing the origin of unification failure. [10], 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*. It is in some sense similar as the one in NiMo, since 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 ports related to this hexagon in the type graph can be visually identified. Other work on this regard (not a graphical tool either) is [11], that uses a graph representation with nodes labelled by lambda terms and types from which information is extracted to help in error type debugging. Concerning the NiMo inference process the main differences with other type systems are that every token in the language carries its own type, and partially constructed expressions are always well typed and also carry their type. On the contrary, type inference systems work on

complete terms which, if erroneous, prevent the system from building their types and produce an error message. NiMo has no error reports, just incompatibility messages. Errors are avoided. On the other hand, expressions in NiMo are bi-dimensional, and can thus be constructed in any order, not only left to right, as application in textual languages does. Hence incremental inference is made in the port connection order. The only restriction is that partial application of a process must be made before connecting its F-out port. Inasmuch as the overall aspects of NiMo development, the paradigms fusion was a big challenge that required figuring out many creative solutions to make both models compatible. In particular, dealing with multiple outputs and curried/uncurried compatibility required a non-trivial generalization of the usual notions of polymorphic type inference to handle the process type. Non structural unification is unnecessary in functional languages because functions have a single parameter and curried and uncurried functions have incompatible types. But we think it was worth the try; the graphic-functional-dataflow characteristics of NiMo result in a very powerful computation model where everything can be dynamically changed, even the evaluation policy. The NiMo execution model is described in [12]. We are currently working on several aspects of net visualization, which are critical when nets grow, such as the non-expanded view of net processes. Also, in the next version user defined types are slated for inclusion. A distribution version of NiMoToons (with an interactive tutorial) is now underway.

References

1. Turner, D.A.: Miranda: a non-strict functional language with polymorphic types. In: Proc. of a conference on Functional programming languages and computer architecture, New York, NY, USA, Springer-Verlag New York, Inc. (1985) 1–16
2. Clerici, S., Zoltan, C.: A graphic functional-dataflow language. In Loidl, H.W., ed.: Trends in Functional Programming. Volume 5 of Trends in Functional Programming., Intellect (2004) 129–144
3. NiMo-Home page: (2009), `http://www.lsi.upc.edu/~nimo/Project`
4. Milner, R.: A theory of type polymorphism in programming. Journal of Computer and System Sciences **17** (1978) 348–375
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. Silvia Clerici, C.Z., Prestigiacomo, G.: Nimotoons: a totally graphic workbench for program tunning and experimentation. In: PROLE. (2009) 129–148, Selected to appear in, `http://www.elsevier.nl/locate/entcs`
7. Resources: (2009), (`http://resources.businessobjects.com/labs/cal/gemcutter-techpaper.pdf`)
8. Simões, H., Florido, M.: TypeTool - a type inference visualization tool. In: In Proceedings of the 13th International Workshop on Functional and (Constraint) Logic Programming. (2004)
9. System I: (2009), `http://types.bu.edu/modular/compositional/system-i/`
10. Yang, J., Michaelson, G., Trinder, P., Wells, J.B.: Improved type error reporting. In: In Proceedings of 12th International Workshop on Implementation of Functional Languages. (2000) 71–86
11. McAdam, B.J.: Generalising techniques for type debugging. In: Trends in Functional Programming, Intellect (2000) 49–57
12. Clerici, S., Zoltan, C.: A dynamically customizable process-centered evaluation model. In: PPDP. (2009) 37–48