Graphical Type Inference. A Graph Grammar Definition

Silvia Clerici and Cristina Zoltan

Dept. Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya

Abstract: We present a graph grammar based type inference system for a totally graphic language inspired in the data flow view of lazy functional programs. NiMo (Nets in Motion) can be seen as a graphic equivalent to Haskell that acts as an on-line tracer and debugger. The user not only sees the results but also the way they are calculated according to an understandable model and can interrupt the execution at any point, change data, processes and/or process activation, undo steps, and also execute incomplete programs.
Type inference is incremental; during the net edition (construction or modification) only type safe connections are allowed. The user visualises the type information evolution and, in case of type error, can identify where and why it happened.
The NiMo type system, though similar, has significant differences with systems in functional languages due to the data flow ingredient. It needs to cope with processes with no entries and zero or more that one output and therefore the process type is a generalization of functional types. We present the notion of non-structural type unification, the elements for modelling graphic type inference, and the correspondence with the classical type inference approach.
Construction and execution of NiMo programs are fully defined via an attributed graph grammar. In the previous version type information was incomplete and static type inference was partial in presence of polymorphism. Therefore type inconsistent nets could be executed. Here we present the type descriptor graphs and the graph grammar definition of the complete static type inference system. The grammar has been implemented and successfully tested using AGG as the graph transformation system.

Keywords: Visual Languages, Graph Grammars, Type Inference, Process Networks, Functional Languages.

1 Introduction

It is a long time the data flow process network [1] interpretation of lazy functional programs is well known in the functional world. This graphic representation, where functions are interpreted as processes and infinite lists as non bounded channels, was first introduced by Turner, the author of Miranda in 1984 [2]. In this paper he used the Hamming numbers net as an example. From then on Hamming, and mainly Fibonacci, are the first classical examples for the process network analogy in Functional Programming introductory textbooks [3]. It has proved to be useful to understand the program overall behaviour since 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 laws”, i.e. how new results are obtained from already calculated ones. The graphic execution model that the net animation suggests was the starting point for the NiMo language design.

The main goal in NiMo [4] [5] is to provide totally graphic edition, debugging, execution and experimentation. There is no need for the user to write textual code at all. Solutions of growing complexity can be built using a small set of graphic primitives, which allow representing and handling functional concepts like higher order, partial application, laziness, polymorphism, and type inference. The source code is a graph, which is directly executable. It is not a graphic representation of a given textual

language code as it is the case in several visual languages. The NiMo interpreter acts as an on-line tracer since it can show the program execution step by step, and exhibit all the changes in the program state according to the process network interpretation of a functional program. The user not only sees the results but also the way they are calculated according to an understandable model. Moreover, NiMo allows on-line debugging since the user has direct control on the computation state and can interrupt the execution at any point, change data, processes and/or process activation, undo steps, and also execute incomplete programs.

Regarding typing and type inference the idea is the same. The user can visualize the full type information. Type inference is incremental; during the net edition (construction or modification) only type safe connections are allowed. The user visualises the type information evolution and, in case of type error, can identify where and why it happened. The NiMo type system, though similar to systems used in functional languages, is not exactly the same due to the data flow ingredient. It needs to cope with processes with no entries and zero or more that one output, and therefore the process type is a generalization of the function type in languages like Haskell [6]. On the other hand, currying in NiMo is also different because partial application can be made in any order. In fact, in functional languages functions are considered to have always a single parameter, the first one in case of curried functions and a tuple in the uncurried version. In NiMo, processes equivalent to curried functions can behave as curried but might look like uncurried because they have all their parameters "in parallel". And the same occurs with results. In Nimo processes can produce more than one output in parallel, in which case its type is different to that of a process that returns a single tuple.

NiMo has been completely defined by a graph grammar [7], [8], and a running prototype has been implemented using AGG [9] as the graph transformation system. This meant modelling all the language elements in terms of graphs and graph transformations. Construction rules ensure syntactically correct and consistently typed programs. To do so, these rules need to implement incremental type inference. In the previous version of NiMo type information was incomplete, and static type inference was partial in presence of polymorphism. Therefore type inconsistent nets could be executed. Here we present the type descriptor graphs and the graph grammar definition of the complete static type inference system. The grammar has been implemented and successfully tested in AGG.

The paper is organized as follows: In section 2 we introduce the NiMo language. Section 3 presents the elements to build and type nets and the way they are integrated. Section 4 defines type unification and discusses the differences between process and functional types. Section 5 covers net construction, incremental type inference, and the correspondence with the classical type inference approach. In section 6 we describe the elements of the editing grammar and its structure showing the interaction among transformation units. In section 7 an example illustrates the edition process and, in particular, how incremental type inference is achieved. The paper ends with a summary of the work contributions and its current state.

2 NiMo language overview

MiMo programs are oriented graphs (may be cyclic) whose arrows are (in general) non-bounded channels where data travel in a FIFO way following a demand driven policy. Processes are functional in the sense of referential transparency, i.e. for identical inputs they produce identical outputs, and can have non-channel parameters that can also be

processes, but according to the data flow model they can have no entries and zero or more than one output. Many of the predefined processes are inspired in the Miranda or Haskell higher order functions for lists, but are more suitable for the process network programming style. Multiple outputs avoid the use of tuples, which are rarely used in NiMo and could also be simulated by the multiple outputs of a process. On the other hand, control of conditions and pattern decomposition are encapsulated within the basic processes behaviour and therefore no specific graphic syntax for conditions is provided. An *if-then-else* higher order process can handle all the conditional situations and in the current version a generalized case process is also provided.

The user builds solutions combining processes that can be basic or net themselves. Every new net can be named to be used by another one, (a new process definition) allowing incremental net complexity up to any arbitrary degree, and nets can also be parameterized. Program edition and execution are interleaved, allowing incomplete programs can be executed up to the point where a missing code is needed to proceed. Process activation is explicit. An activated process can demand non-available values to its provider processes, which are activated "in parallel" and, in turn, activate their own providers. Then, process activation can propagate and therefore several processes could work simultaneously in different regions of the net.

At the token level NiMo syntax is very concise, there are only seven kinds of nodes: rectangles for processes, circles for constant values, black-dots for duplicators, hexagons for type information, diamonds for keeping activation or evaluation states, triangles for tupling, and big-arrows for open connections or formal parameters. Except black-dots, all of them are labelled. Circles for atomic type constants have different colours. Horizontal arrows represent channels, and vertical arrows entering a process correspond to parameters that are not channels in the data flow sense of flowing data streams.

Data are type-value-entities (tve), where values (evaluated or not) are tied to its type (hexagon) through a diamond. If the value is a constant the diamond is green. Non-fully evaluated tve's contain at least one rectangle (a non-evaluated process), preceded by a white or red diamond. A red diamond preceding a process or duplicator means that it is required to act.

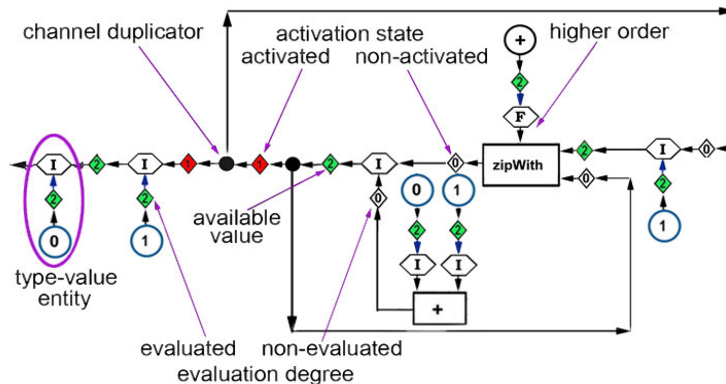

Figure 1. An example of NiMo Graphical Syntax

Labels for hexagons of atomic types are "I" for integers, "R" for reals "B" for Booleans, and "S" for strings. For structured types the identifiers are "L" for lists, "C" for channels, "F" for functional processes and "T" for tuples. Channels and lists are homogeneous and unbounded; they are mutually convertible since they correspond to the data flow and functional view of lists. Horizontal arrows entering a process are

assumed to be channels and therefore the channel hexagon is omitted. It is only used for open (not yet connected) channels. The symbol "**?**" denotes the polymorphic type. In the current version user defined types are not supported and there is no the Haskell type classes notion either. Ad-hoc polymorphism for "cuasi polymorphic" functions as = is handled as in Miranda. For arithmetic operation there are two different processes for real and integer.

The hexagon in a tve fully describe its type in case of atomic or polymorphic values but structured types need a more refined *type descriptor* for their component types whenever type inference is required. It is important to remark that in NiMo there are not variables and this also include type variables in polymorphic types. The label "**?**" is the same for all the polymorphic types, if two of them are the same both hexagons are related by identity arrows or shared, as will be seen in the following section.

Before proceeding with the language description let us see a first example of NiMo program and the way it evolves: Figure 1 shows the state of the net for calculating the Fibonnacci sequence at the time when the two first values have already been calculated and the third one has been produced but not yet evaluated. In the next execution step since the second duplicator is activated (preceded by a red diamond) and has a value at its right, acts passing that value to its left and a copy to the second entry channel of zipWith, afterwards its preceding diamond becomes white. In the next step the first duplicator has now an available value and acts producing the third output value and also a second value in the first entry channel of zipWith. Its preceding diamond becomes white. For the execution to continue this diamond must be set to red again and the activation is automatically propagated up to the diamond in front of zipWith. This activation setting can be made by the user in an interactive way or by a preceding (consumer) process.

As NiMo allows complete higher order and partial application, process parameters can also be parameterised not only by function names as +, but also by partially applied functions (processes with big-arrows in some of its inputs) as the higher order parameter of map in figure 2, which is equivalent to the operator section (*2) in Haskell. The figure shows a net producing the sequence of even numbers. Let's note that *naturals* is a process that produces a channel of integers. In a Haskell-like language its type is *list of integer*, but in NiMo its implicit type is process (F) with 0 entries and a channel (or list) of integer elements as output.

Figure2. Even numbers

3 Type Descriptors and Interfaces

Type descriptors are the analogous to the AST of a type expression. They fully describe the type of processes and values. This complete type information is necessary during the net construction to avoid that type inconsistent nets can be built.

On the other hand, each program unit (type value, process, and duplicator) has a structural descriptor that we will call its *interface*. They are a kind of syntactical templates describing the connections points of each program unit and their types, therefore type descriptors are part of interfaces. In fact, interfaces are used in edition to build syntactically correct pieces of code, and in this sense they could be considered non–terminal graphs. But given that incomplete programs can be executed and interactively completed by need, partially connected interfaces are also terminal graphs, because they are potentially executable.

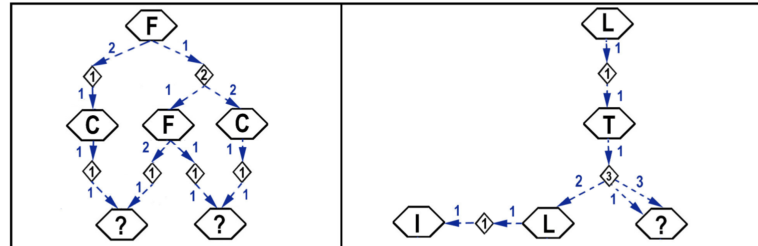

Figure 3. Refined type descriptors

Type descriptors are hierarchical graphs alternating levels with only hexagons and only diamond nodes, and where the root and leaves are hexagons. Arrows inside a type descriptor are dashed (*type refinement arrows*). Values in diamonds indicate the number of its sons (component types) and labels in arrows their order, i.e. for list and channels (L or C in the root hexagon) the type of their elements; for tuples (root T) the number, order and types of their components, and for processes (root F) the number, order and type of both their (parallel) inputs and outputs.

Left side in figure 3 shows the type descriptor for the process map, analogous to the map higher order function in Haskell with type (a →b) →[a] →[b]. Process map has two inputs (a value 2 in the diamond with incoming arrow labelled 1) and a single output (value 1 in the other diamond). The first input is a higher order parameter, the second one is a channel, and the output is also a channel. The parameter process has a single input with the same polymorphic type as the elements in the map input channel (corresponding to *a*), and a single output with the same type as the elements in the map output channel (corresponding to *b*), therefore both pairs of identical polymorphic types share a "**?**" hexagon. Inside a type descriptor shared sub-graphs indicate identical types. A textual version for the process map type expression is F (F ($\mathbf{?}_1$ → $\mathbf{?}_2$) || C($\mathbf{?}_1$) → C($\mathbf{?}_2$)) where same subindex in **?** indicates shared hexagons. Symbol || is the parallel type constructor. We use this symbol instead of arrow because the interpretation of several parameters is not the same as the curried interpretation as it is more deeply discussed in the next section.

The type descriptor on the right side of figure 3 correspond to a list whose elements are triples with the first and third components having the same polymorphic type, and the second one a list of integers. The textual type expression is L(T (**?** ||L(I)||**?**)).

Other examples of process types are

fibonacci: F (→ C(I))

+: F(I || I →I)

id: F(**?** → **?**)

sink: F (**?** →)

sink is a process with no output that does not have an equivalent in Haskell, its definition would be something like sink x = void. It cannot be activated and acts only when has an available value (consuming that value) but it does not activate its provider.

Interfaces describe the connections points of each program unit and their types by means of a couple of hexagon and big-arrow nodes (*connection port*). In input ports the edge between them is oriented to the hexagon while in the output ones the hexagon is the source node. Whenever two polymorphic ports must have the same type, they are linked together signalling their equivalence by means of a pair of identity arrows (in both directions). As is the case of the "=" parameters in figure 4.

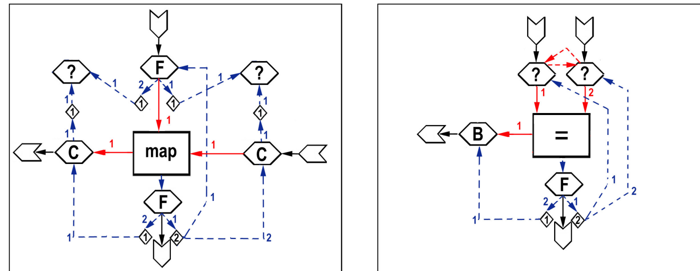

Figure 4. map and = process interface

The left side of figure 4 shows the map process interface. Notice that it contains the type descriptor given in figure 3 as a sub-graph, its root is the lowest F hexagon. This port allows connecting the interface as a (functional) value. Hexagons belonging to ports are *structural* (C an F hexagons in this case), the rest are *inner* (both "**?**" hexagons). Inner hexagons are only connected by refinement arrows, and therefore are not persistent nodes. In a completely connected graph (one without any connection port) there are no inner nodes because refined type descriptors are not longer needed. On the other hand, once a process is connected its process type hexagon only persists if the process is used as a functional value in a tve or as a higher order actual parameter of another process, i.e. in case that its last connected output port was the lowest one. Afterwards its remaining output ports can no longer be connected, they becomes formal parameters of the functional value. This is graphically indicated by changing the big-arrow colour as can be seen in figure 13. Channel hexagons also disappear once connected because horizontal incoming arrows are implicitly typed as channels (see figure 2).

Figure 4 shows other interfaces. This set together with the predefined process interfaces are enough for constructing a NiMo program. The interface for a new user process (having no name yet) is an instance of a generic process interface with n non-channel parameters, m channels and p outputs where n, m, p ≥ 0, n+m+p > 0. In the example n=2, m=1 and p=2. The user sets these values and the process name. The same occurs with generic tuples, the user the user must set the number of elements (3 in the example).

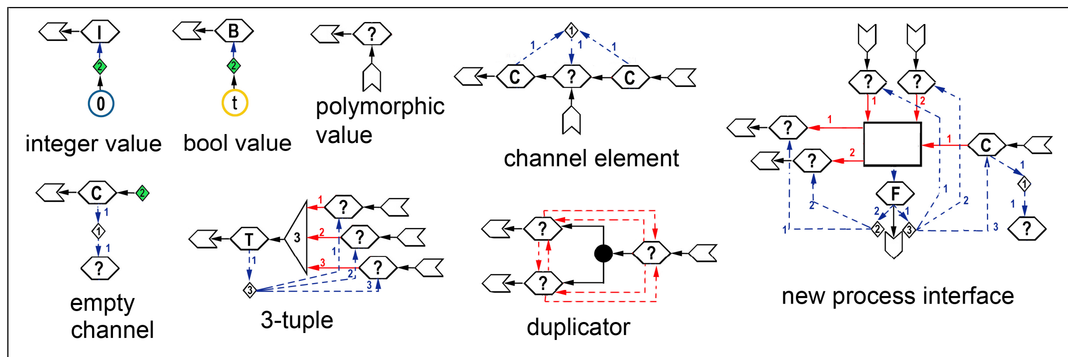

Figure 5. Other interfaces

Channel element: $\mathbf{?} \parallel C(\mathbf{?}) \rightarrow C(\mathbf{?})$
Empty channel: $\rightarrow C(\mathbf{?})$
Polymorphic value: $\mathbf{?} \rightarrow \mathbf{?}$
Duplicator : $\mathbf{?} \rightarrow \mathbf{?} \parallel \mathbf{?}$
any-process :$F(\mathbf{?}_1 \parallel \ldots \parallel \mathbf{?}_n \rightarrow \mathbf{?'}_1 \parallel \ldots \parallel \mathbf{?'}_m)$ where $n, m \geq 0 \quad n+m > 0$
n-tupla :$\mathbf{?}_1 \parallel \ldots \parallel \mathbf{?}_n \rightarrow T(\mathbf{?}_1 \parallel \ldots \parallel \mathbf{?}_n)$ where $n > 1$
int-value: $\rightarrow I$
bool-value: $\rightarrow B$

Other generic interfaces could be added (like one for directly constructing a channel of n elements, a general case, or a general map).
G-case:$F(\mathbf{?}_1 \parallel \ldots \parallel \mathbf{?}_n \parallel C(F(\mathbf{?}_1 \parallel \ldots \parallel \mathbf{?}_n \rightarrow B)) \parallel C(F(\mathbf{?}_1 \parallel \ldots \parallel \mathbf{?}_n \rightarrow \mathbf{?'}_1 \parallel \ldots \parallel \mathbf{?'}_m)) \rightarrow \mathbf{?'}_1 \parallel \ldots \parallel \mathbf{?'}_m)$
G-map:$F(F(\mathbf{?}_1 \parallel \ldots \parallel \mathbf{?}_n \rightarrow \mathbf{?'}_1 \parallel \ldots \parallel \mathbf{?'}_m) \parallel C(\mathbf{?}_1) \parallel \ldots \parallel C(\mathbf{?}_n) \rightarrow C(\mathbf{?'}_1) \parallel \ldots \parallel C(\mathbf{?'}_m))$

4 Type unification

A pair of input and output ports can be connected only if both type are compatible, i.e. if both type descriptors can be unified. The following predicates define this condition

1. Unify(t ,t)
2. Unify (t , **?**)
3. $(X=C \vee X=L) \ \& \ (Y = C \vee Y = L) \Rightarrow$ Unify $(X(t), Y(t')) \Leftrightarrow$ Unify (t , t')
4. Unify $(T(t_1 \parallel \ldots \parallel t_n), T(t'_1 \parallel \ldots \parallel t'_n)) \Leftrightarrow$Unify$(t_1,t'_1)$ & ….Unify(t_n,t'_n)
5. Unify$(F(a_1 \parallel \ldots \parallel a_n \rightarrow b_1 \parallel \ldots \parallel b_m), F(c_1 \parallel \ldots \parallel c_n \rightarrow d_1 \parallel \ldots \parallel d_m)) \Leftrightarrow$
   Unify(a_1,c_1) & …&Unify(a_n,c_n) & Unify(b_1,d_1) & …&Unify(b_n,d_n)
6. Unify$(F(a_1 \parallel \ldots \parallel a_{n1} \rightarrow F(a'_1 \parallel \ldots \parallel a'_{n2} \rightarrow b_1 \parallel \ldots \parallel b_m)), F(c1 \parallel \ldots \parallel c_{n1} \parallel c_{n1+1} \parallel \ldots \parallel c_{nr} \rightarrow$
   $d_1 \parallel \ldots \parallel d_q)) \Leftrightarrow$ Unify(a_1,c_1) & …&Unify(a_{n1},c_{n1}) &
   Unify$(F(a'_1 \parallel \ldots \parallel a'_{n2} \rightarrow b_1 \parallel \ldots \parallel b_m), F(c_{n1+1} \parallel \ldots \parallel c_{nr} \rightarrow d_1 \parallel \ldots \parallel d_q))$
7. Anymore unify

Rules 1 to 5 states that a pair of type descriptors can be unified if they are structurally compatible, i.e. if both roots have the same label (except lists and channels wich are equivalent) or one of them is **?**, and in case of having the same structured type label all the respective subgraph descriptors (component types) are structurally compatible.

In Haskell like languages, thanks to currying, a functional type has a single interpretation because all functions have a single parameter (the first one) and, to be unified, they must be structurally compatible. NiMo processes are not curried in the usual sense. Parameters can be applied in parallel “all at a time”, as in uncurried functions, and can be also partially applied to any subset of their parallel parameters, not only to a prefix of them (as operator sections in Haskell can also do). But processes with more than one parameter can behave as curried as well (i.e. be interpreted as returning successive intermediate functions). For instance, process + is a valid actual parameter for map as in Haskell, in which case the input channel has to be of integers and the result will be a channel of functions $F(I \rightarrow I)$. Therefore the type descriptor of +: $F(I \parallel I \rightarrow I)$ must unify with $F(I \rightarrow F(I \rightarrow I))$. Consequently, in NiMo unification does not imply structural equivalence of the type descriptors and so the types of two processes with different number of parameters could be unified. Rule 6 defines the conditions for such a case. Basically, the process with fewer parameters must return a function whose descriptor has to unify with the resulting type of having applied the second process to as many parameters as the first has.

The following rules define the operator $\approx$ that obtains the unification result.

1. $t \approx t = t$
2. $t \approx \mathbf{?}) = t$
3. $(X=C \vee X=L)$ & $(Y = C \vee Y=L) \Rightarrow X(t) \approx Y(t') = X(t \approx t')$
4. $T(t_1 || \ldots || t_n) \approx T(t'_1 || \ldots || t'_n) = T(t_1 \approx t'_1 || \ldots || t_n \approx t'_n)$
5. $F(a_1 || \ldots || a_n \rightarrow b_1 || \ldots || b_m) \approx F(c_1 || \ldots || c_n \rightarrow d_1 || \ldots || d_m) =$ $F(a_1 \approx c_1 || \ldots || a_n \approx c_n \rightarrow b_1 \approx d_1 || \ldots || b_n \approx d_n)$

6. $F(a_1 || \ldots || a_{n1} \rightarrow F(a'_1 || \ldots || a'_{n2} \rightarrow b_1 || \ldots || b_m)) \approx F(c1 || \ldots || c_{n1} || c_{n1+1} || \ldots || c_{nr} \rightarrow d_1 || \ldots || d_q) =$
$F(a_1 \approx c_1 || \ldots || a_{n1} \approx c_{n1} \rightarrow F(a'_1 || \ldots || a'_{n2} \dashrightarrow b_1 || \ldots || b_m) \approx F(c_{n1+1} || \ldots || c_{nr} \rightarrow d_1 || \ldots || d_q))$

5 Incremental Type inference

As already said, in NiMo type checking and inference is made step by step and locally during net edition. As incomplete programs can be executed, the notion of editing a program is not the usual one. It is a discontinuous process with intervals where code evolves in execution up to the next user interaction. Initially the net is empty. The user adds interfaces (net components) and connects an interface input port with an output port of another one. Net construction is equivalent to build a bi-dimensional expression where sub-expressions are like puzzle pieces that can be pairwise connected in any order (see the example in figure 13), provided their connection point shapes fit together (both port types unify). In textual languages code is unidimentional, expressions whose type can be inferred are "complete" though they can have variables, which in some sense play the role of placeholders. However an expression having variables cannot be evaluated by the interpreter. Variables are used in function definitions as formal parameters (bounded variables) or as function or constant names locally defined. In NiMo there are not variables. The equivalent of function parameters are the process interface input ports. During the net construction the net can be considered as having as many parameters as input ports and as many results as output ports. Connections correspond to function application or function composition and most of the possible "net parameters" are progressively cancelled with the (intermediate) results. 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 be) reducing the number of connected components. Since several port type descriptors in a component can share subgraphs, when two ports are connected the effect of unifying both types can affect any other port all along both connected components. Basically when one of them contain an inner **?** hexagon that changes and it is shared by other descriptors, or when a structural **?** is unified with a different type descriptor and has **?** hexagons related with it by identity arrows. They have to change as well, and propagate the new type to its own related hexagons. But even if both port types are identical and no changes occur after unification, connecting a pair of input and output ports changes the type of both interfaces, the connected components and therefore the net.

If N is the net under contruction $N = \bigcup_i N_i$ where N_i are its connected components. The following *connection rule* stands for connecting two compatible input and output ports.

$p_1 \in N_1 \;\; p_2 \in N_2$
$p_1{:}X(a_1 || \ldots || a_{n1} \rightarrow b_1 || \ldots || b_{m1}) \qquad p_2{:}Y(c_1 || \ldots || c_{n2} \rightarrow d_1 || \ldots || d_{m2}))$ with $X,Y \in \{F, \varepsilon\}$
Unify (b_i, c_k)

$p_1 \;{}_i{>>}_k\; p_2 = \langle p'_1, p'_2 \rangle \qquad p'_1, p'_2 \in N_{<1,2>}$
$p'_1{:}\; X(a'_1 || \ldots || 'a_{n1} \rightarrow b'_1 || \ldots\; b'_{i-1} || b'_{i+1} || \ldots || b'_{m1})$

p'_2: $Y(c'_1 || \ldots || c'_{k-1} || c'_{k+1} || \ldots || c'_{n2} \rightarrow d'_1 || \ldots || d'_{m2})$

where $x'_j = \text{sust}(x_j, b_i, \approx c_k)$ $x_j \in \{a_1, \ldots, a_{n1}, b_1, \ldots, b_{m1}, c_1, \ldots, c_{n2}, d_1, \ldots, d_{m2}\}$

$\forall\ p: t \in N_1 \cup N_2\ p \neq p_1\ p \neq p_2 \ \Rightarrow\ p: t' \in N_{<1,2>}\quad t' = \text{sust}(t, b_i, \approx c_k)$

$N' = N - (N_1 \cup N_2) \cup N_{<1,2>}$

p_1 and p_2 are interfaces belonging respectively to connected components N_1 and N_2 (which could be the same). If the i-th output port of p_1 and the k-th input port of p_2 are compatible, its connection modify both iterfaces (each will have one port less) and its types. N_1 and N_2 becomes a single connected component $N_{<1,2>}$ where, due to the unification of both connected ports, all the remaining port types could have changed.

In type inference systems (like [10] [11]) type definitions are associated to names in a context Γ that initially has all the predefined type information. There is a term e whose T type has to be inferred. T is obtained from the type of its sub-terms e_i whose type T_i has to be inferred using inference rules and their local environment Γ_i. T_i are type expressions that can also include type variables a_k if T_i is a polymorphic type. Each inference rule can add local type information for the term variables x_j (parameters or let definitions) producing a new local environment $\Gamma' = \Gamma.\ x_j: T_k$

In Nimo the net under construction N is both the term e and the context Γ. To simplify let assume that all the interfaces are brought at once. The initial net is a set of typed terms e_i therefore the initial e is a family of terms, and Γ is the union of their type descriptors T_i where the **?** hexagons play the role of a_k. As connections are made the net evolves to become (may be) a single term e. Along this evolution Γ_i are the port types of each connected component. If finally all the ports were connected, Γ would become empty.

Specialization (of polymorphic types) is made when port type are unified, and propagates along the identity arrows way. There is no need for a generalization rule either because port types are born as polymorphic as they can be. The connection rule is a kind of application rule in some cases (when a process non-channel input port is connected) but can also be an equivalent of function composition when two channels are connected. Generate a new process (by means of the generic interface) is the equivalent of adding an annotated variable. And the equivalent of the abstraction rule is only applied over the final net in case that it where defined as a parameterised process to be stored. The net type is F with as many inputs and outputs as input and output ports remain. The user just gives a name for it and bounds its parameters by setting the order for the input and output ports.

6 The Edition Grammar structure

In terms of graph transformations, edition can be thought of as the sequence of editing rule applications enabled by the user actions on the graph. In edition these actions should only be adding a creator node, and setting (or changing) a certain attribute value. Safe deletion can only be ensured via “undo”. The edition grammar has been designed with the goal of minimizing user intervention and ensuring correctness in both syntactical and type consistency aspects. In the following we present the grammar internal organization, the main ideas in the transformation units definition, and representative rules of each one of them.

The editing rules are organized into two main units: Creation and Connection. The last one is also formed by two units: Type-inference and Plugging. Figure 6 shows the grammar structure diagram, where ovals are the initial and final graphs and user interaction is indicated in lower case.

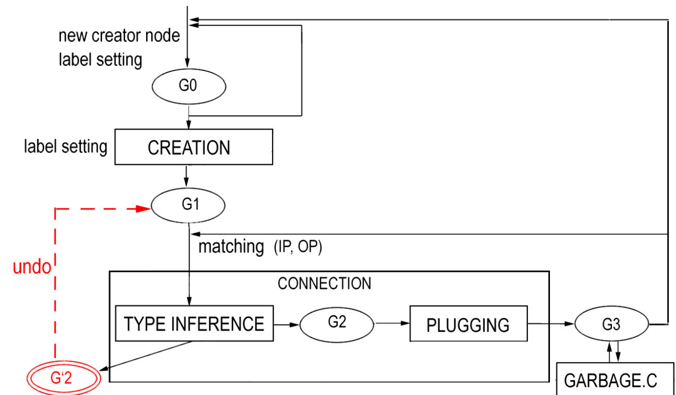

Figure 6 Editing rules organization

Creation unit produces a new interface in the graph (G1). The elementary kinds of interfaces are: constant values, open element, channel element, duplicator, process, end-of-channel, and generic process. For each one of them there is an associated sub-set of creation rules.

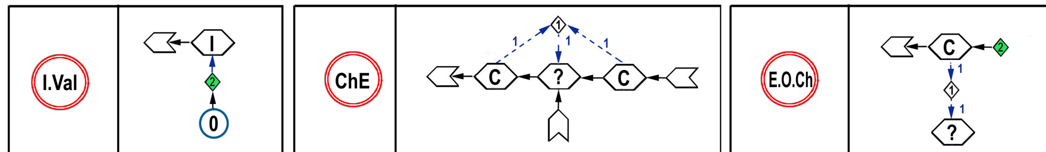

Figure 7. Some creation rules

The user adds a fresh creator node and sets its attribute with the kind of interface to be created (G0). Depending on the kind a single rule may produce the required template (as seen in figure 7), or it may be necessary to set a new attribute to select the specific interface by means of a second rule. Rules in figure 7 create an integer constant with default value 0, a channel element with open value, and the end of channel. Figure 8 shows the creation of a generic interface (G.I) for a new (not yet defined) process with two parameters, one input channel, and two outputs.

For each predefined or already user defined process there is a process interface rule having the name of the process and its interface as left and right sides. They are the only kind of creation rules that are also used in execution, whenever a process name (a circle) is an input parameter for another process (as it is the case of + in figure 1). On the other hand, this set of rules is necessarily extensible; while for the other kinds the set of creating rules is fix.

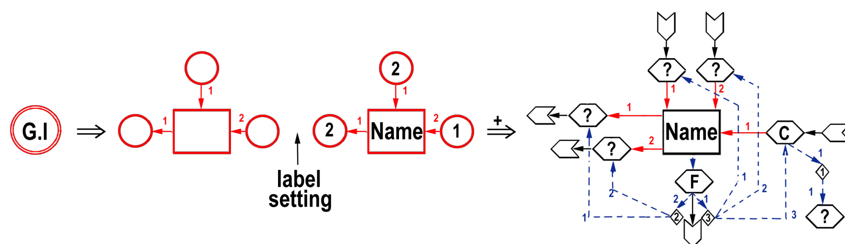

Figure 8. Creating a generic interface

Connection unit plugs a couple of input and output selected ports, once verified that both types are compatible. Ports are selected via matching. Connection uses two transformation units, as seen in figure 6: first, the type inference rules (TI) that perform unification and second, the plugging rules that effectively perform the connection (G3

in figure 6) in case of successful ending of the type inference rules application (G2). If a failure state is reached (G'2) the graph must be rolled back by means of (a single) "undo".Ports of an interface can be connected in any order (see figure13). A connection step starts when the user chooses the pair of input-output ports to be connected and it continues, with no further interaction, until the connection is established or a failure node is produced. The first applicable rule in TI adds a new kind of node with incoming arrows from the type descriptor root of both ports (see left rule on figure 9). Graphs containing this kind of non-terminal nodes are not NiMo programs, and could be hidden from the user. They can be viewed as having unstable states. Consequently, Connection can be regarded as a transaction in the sense of [12].

Type inference rules perform unification between a pair of type descriptors. In the initial graph of this unit, they are the matched pair of corresponding input and output port types.

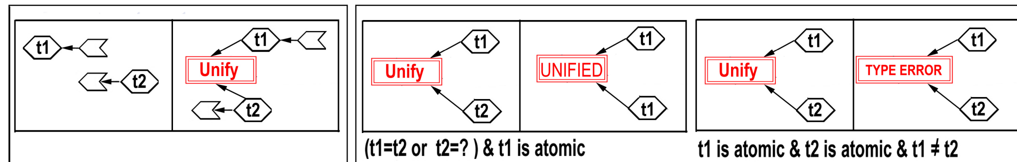

Figure 9. Starting unification and Unifying atomic types rules

During the unification process a pair of hexagons can collapse only if one of them is "**?**". But it is not always the case because persistent hexagons (the root of a port type descriptor) are structural pieces of program units, and two structural pieces cannot collapse in a single one. In both are "**?**" a pair of identity arrows is set between them, else the "**?**" label is set to the other one. If it is a structured type, refinement arrows are created to share the type descriptor sons. Also, if "**?**" hexagon had identity arrows the same effect is propagated, and the arrows are deleted. On the other hand basic hexagons are never collapsed to avoid unnecessary arrows that obscure the graph visualization. The result of unification in this case results in changing the "?" label of the other hexagon, being it persistent or not. Unification rules for atomic types are shown on the right side of Figure 9.

Unification rules for lists (or channels) and for tuples are a simplified version of the process type unification rules. Rules for unifying structurally compatible process types are shown in figure 10. They are the graph transformation equivalent to the rule 5 for "≈" operator (see section 4). Rules R1, R2 and R4 stand for verifying structural equivalence. R3 has as negative condition preventing the use of this rule until both UnifyAll and all the Unify nodes in the next level have disappeared. In this case the Maybe node turns into the successful node UNIFIED.

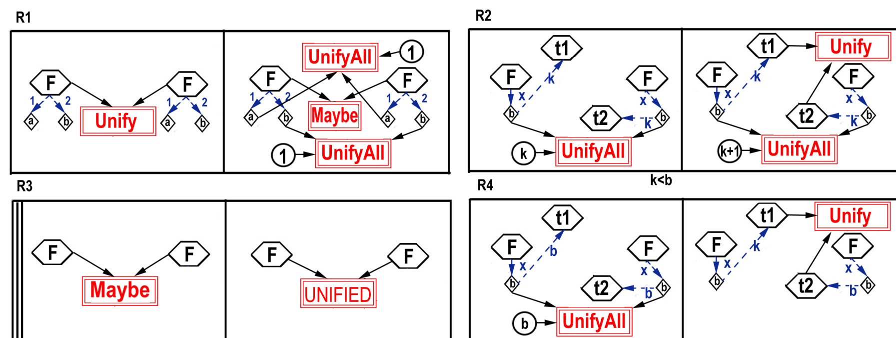

Figure 10. F-Type Unification rules

The unification step for the input/output matched ports ends when the first level success node is produced or whenever a failure node appears. The first one, in turn,

enables the corresponding plugging rule. Conversely, the failure node disallows the connection. Since the type inference rules might have changed the type descriptors while trying to unify them, after a failure state the graph should be restored. It can be rolled back by means of "undo".

Plugging rules connect both ports erasing the big-arrows and, except in the case of channels, collapsing both hexagons. Having a refined type descriptor is no longer needed therefore the first level diamonds are disconnected. The garbage collector rules erase all the no longer needed inner diamonds and hexagons (those that were not shared with another port type descriptor and therefore have no incoming edges).

On the other hand, whenever a connected port belongs to a process interface, the process type descriptor must be updated. From then on the interface has one parameter or one output less, and then the corresponding diamond must be decreased (and perhaps some of its outcoming arrows renumbered).

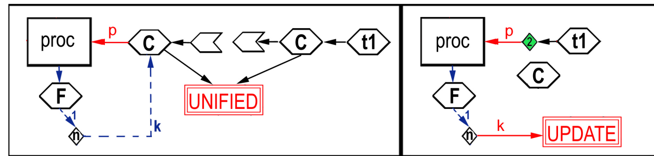

Figure11. Connecting a channel

Rule in figure 11 is applied when the ports to be connected are the input channel of a process and a channel element. The element hexagon is connected to the process input by means of a new green diamond. The output port (C hexagon and big arrow) and the input big arrow are erased. The input "C" hexagon is disconnected from the process box and from the process type descriptor. If the channel hexagon had no incoming arrows garbage collector completes the erasing. UPDATE node enables the rules for updating the process type descriptor.

7 Example

The following example illustrates a piece of NiMo program construction. Greek letters are used to designate ports. Figure 12 corresponds to a G1 graph from figure 6 after creating two process interfaces, a channel element and an integer value. The respective input and output ports could be successfully connected in different ways.

For instance $\langle(\rho, \mu), (\lambda, \theta), (\beta,\tau)\rangle$ or $\langle(\omega, \mu), (\alpha, \theta), (\beta,\tau)\rangle$ are two sequences of port matchings that enable type consistent connections (they produce a sequence of G3 graphs).

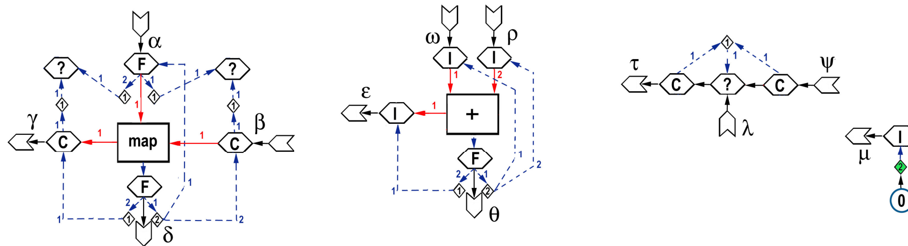

Figure 12. Four templates created

On the contrary (ψ, μ) produces a G'2 graph, because a failure node is obtained when trying to unify C and I hexagons.

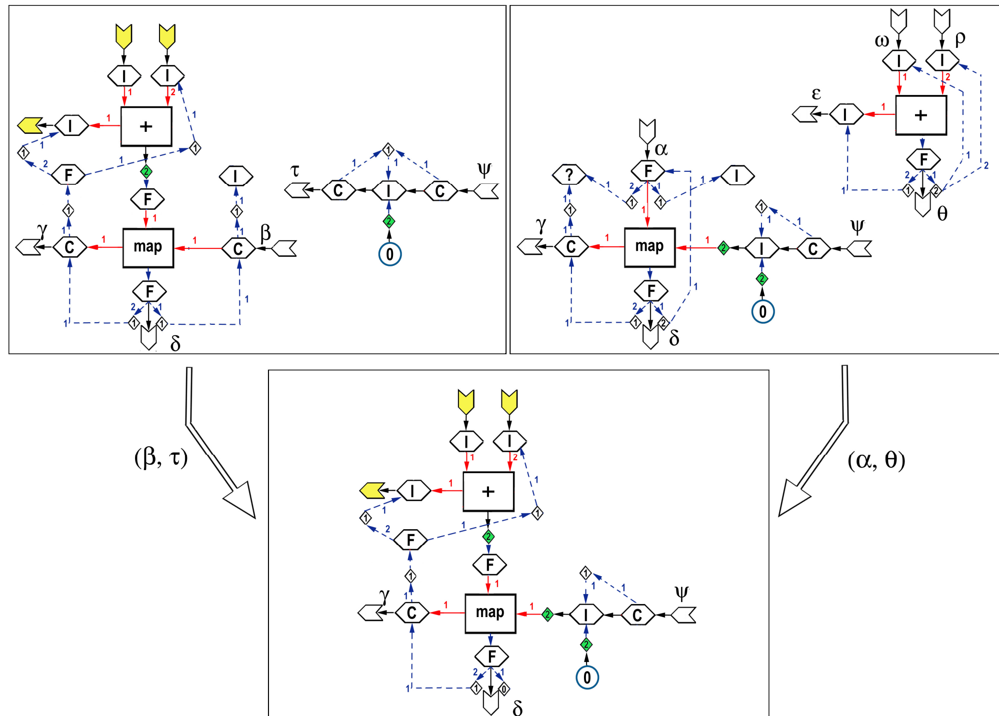

Figure 13 Connections in different order

The left upper graph in figure 13 corresponds to the (G3 graph) obtained after connecting (α, θ) and (λ,μ). Unification of α and θ types was successful even though both type descriptors are no structurally identical (+ has two inputs and the process parameter of map one). As a result of unification, the map output channel type turned into F with input and output type integer. The map type descriptor has also changed because now it has only one input port. Regarding the + interface, it can no longer be connected because once used as a map parameter its type must remain constant. It is graphically indicated by changing the big arrows colour. The right upper graph is the result of connecting (β, τ) and (λ,μ). The inferred type for the input type of the process parameter of map is integer. Its output is still polymorphic.

Now, connecting (β, τ) on the top left graph and (α, θ) on the right one give the graph in the bottom of the figure. In both cases this is the only possible connection.

From now on, new edition units can be added, connected and perhaps executed, or else the net can be considered final and be given a name for defining a new user process. This operation is the equivalent to adding a new library component. It is the only operation that implies extending the grammar and therefore it cannot be handled by means of (first order) rules. Adding a new user defined process gives as a result a pair of new rules; the process interface and the expansion rule (the last one belonging to the execution grammar). Right hand side of both rules can be almost directly obtained from the final net, and a set of rules can perform this task. The idea is the following: the new process interface has a box with the process name, whose input and output ports are the union of the net input and output ports. To build the new process descriptor the only missing information is the enumeration order for the refinement arrows, which should be set by the user. Regarding the expansion rule, it implements the notion of refinement, the left hand side is the interface and the right side is the whole net, with the corresponding port mappings.

8 Conclusions

We have presented the incremental and complete type inference system for the NiMo graphic programming language. The contribution of this work mainly resides on some NiMo language peculiarities. On one hand, NiMo design has required to find a correspondence in terms of graphs for notions as higher order, partial application, polymorphism, laziness and complete static type inference. The NiMo interpreter acts as an on-line tracer since it can show the program execution step by step, and exhibit all the changes in the program state according to the process network interpretation of a functional program. During the net construction the user can visualise the type information evolution and, in case of type error, identify where and why it happened.

From the edition point of view a differential characteristic is the notion of editing itself, in a context where edition and execution are interleaved because incomplete programs can be executed. The double role of interfaces as edition templates and also executable pieces of code is one of the key differences with other graphical editors. Another one is that the NiMo editor kernel is a graphical type inference system, allowing complete static inference be made in an incremental way. This is also based on the interface design where all the typing information is integrated into syntactical templates. This information evolves during edition, propagates all along the graph, and is progressively discarded once it is no longer needed. Other Visual Data-flow languages as Simulink [13], Lab view [14], Prograph [15],VIVA [16] have editors for putting together wires and boxes, but the gluing semantics is quite direct since the language type system is very simple. They do not include functional types, partial application, nor polymorphism and do not deal explicitly and graphically with type checking or inference

On the other hand since NiMo follows a lazy functional-dataflow mix model, in many aspects it does not fit with either parent paradigm and new solutions for known problems have been required. In particular, the type process as a generalization of functional types with a different interpretation of currying, and the notion of non structural equivalence of the type descriptors. Also, the fact of NiMo being totally graphic, has made it necessary to find a correspondence for the type inference system concepts in terms of graphs and without any kind of variables.

The complete grammar definition for both the editor and the interpreter was implemented using AGG. A first version of NiMo and an overall description of its running prototype (not yet including complete static inference) was presented in TFP 2004 and published in [4]. The grammar for the editor here presented is completely implemented. Including a generalized case interface and non-structural identity for processes the grammar has less than a hundred rules (creation unit less than 30, unification around 40, and plugging 20). But, though interesting as a complete graph operational semantic of NiMo, from the point of view of its performance and visualization NiMoAGG can only be considered a prototype. Scaling, zooming, scrolling, and other facilities allowing “pretty printing” of the net, are not present in the graph transformation system graphic interface. Regarding scalability NiMo already has a procedural abstraction mechanism to represent a complex net by a single node, therefore allowing to face more complex problems. But, as in every visual language, as the net grows the limited screen size is a problem that the mechanisms above mentioned (scrolling, etc.) are not powerful enough to solve. There are several possible ways to reduce the number of visible nodes without losing understanding. For instance, optionally compressing basic type-values and process parameters, displaying a graphic shorthand for numeric lists, and having channel windows to scroll over its flowing values.

Currently an integrated environment for NiMo is under development [17]. It is intended to heavily improve the visualization issues and the user interface to become a really useful tool for debugging and testing programs. The environment will provide automatic translation to Haskell and (with some restrictions) vice versa. A new version of the language adding "generic" processes (like a map with any number of input/output channels) is under development. Finally some interesting properties and possible extensions of the current NiMo type system are also being explored.

9 References

[1] Lee, E. and Parks, T. Data-Flow Process Networks. Proc. IEEE 83, 5, 773-799 1995.

[2] D. Turner. Miranda: A Non Strict Functional Language with Polymorphic Types. Functional Programming Languages and Computer Architecture, LNCS 201, Springer-Verlag, *Functional programs as executable specifications* 1985.

[3] Chris Reade Elements of Functional Programming, International Computer Science Series Addison Wesley, 1989

[4] Clerici S., Zoltan C. Trends in Functional Programming, Volume 5. Editor Loidl H-W. Intellect 2006.

[5] Clerici, S., Zoltan C. NiMo home page. http:||www.lsi.upc.edu/~nimo/Project/

[6] P. Hudak, S. Peyton Jones, P. Wadler, et al. Report on the functional programming language Haskell: Version 1.2. ACM SIGPLAN Notices, 27(5), 1992.

[7] Rozenberg, G, Editor, Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations, World Scientific Publishing Co., Inc., 1997.

[8] Andries, Engels, Habel, Hoffmann, Kreowski, Kuske, Plump, Schürr, Taentzer; Graph Transformation for Specification and Programming Science of Computer Programming, Vol. 34, No. 1, April 1999, pp.1- 54

[9] AGG home page, AGG Home page: http:ltfs.cs.tu-berlin.de/agg/ 2006

[10] Basic polymorphic typechecking. Luca Cardelli. ©1987 PDF Science of Computer Programming, 8(2): 147-172, 1987.

[11] R.Milner: A theory of type polymorphism in programming, Journal of Computer and System Sciences, No. 17, 1978.

[12] Paolo Baldan, Andrea Corradini, Fernando Luis Dotti, Luciana Foss, Fabio Gadducci: Towards a Notion of Transaction in Graph Rewriting. Graph Transformation and Visual Modeling Techniques GT-VMT 2006 Electronic Notes in Theoretical Computer Science 2006.

[13] http://www.mathworks.com/

[14] http://www.ni.com/labview/

[15] P. T. Cox, F. R. Giles, and T. Pietrzykowski. Prograph: A step towards liberating programming from textual conditioning. In *IEEE Workshop on Visual Languages*, pages 150–156, 1989.

[16] Tanimoto, S. Liveness in visual languages and VIVA 1990

[17] S. Clerici, C. Zoltan, G. Prestigiacomo, J. García Sanjulián
Diseño de un Entorno integrado de Desarrollo para NiMo Sitges Prole 2006.