A Graphic Functional-Dataflow Language

Silvia Clerici, Cristina Zoltan
Dept. Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
08024 Barcelona
{silvia, zoltan}@lsi.upc.edu

Abstract

NiMo (Nets in Motion) is a visual environment aimed to support totally graphic programming in Data Flow style, with a strong functional inspiration. Solutions of growing complexity can be built using a small set of graphic primitives, which allows dealing with higher order, partial application, laziness, polymorphism and type inference. The net to be executed is exactly the one drawn by the user. It evolves showing all transformations of data and processes. The user has direct control on the computation state and can change data, processes and/or control at any point of the execution. This is feasible because of the combination of several peculiarities: the source code is exactly the computation initial state, activation state and evaluation degree are explicit, execution can be done step-by-step, and between steps the user can modify the net. Therefore even incomplete programs can be run.

1. Introduction

During the last years the interest for visual modeling and programming has resulted in a wide variety of visual languages and environments, in which the term "Visual" has different meanings (system animation, graphic modeling, graphical edition, visual representations of textual programs, data structures animation, graphical display of the internal representation, visualization of how some part of it evolves during execution, etc). This work shares certain characteristics with some of them, but the specific objectives and their implementation are fairly different. NiMo intends to be a workbench for graphically editing, debugging, executing and experimenting. The language is based on the classical process network representations for lazy functional programs. The source code is a graph, which is directly executable; it is not the graphic representation of a textual program as Visual Haskell [4] is. Although close enough to lazy functional languages NiMo follows a functional-data-flow mixed model, and there is no need for the user to write textual code at all. On the other hand, though the graph could be easily translated into a lazy functional language this is not either the case. The interpreter acts directly upon the source code, without any kind of translation. The NiMo environment is based on a graph transformation system; therefore even its implementation is graphical. Execution can be done in free mode or step-by-step, which allows online debugging and experiments on the running net like relaxing laziness for some processes, bounding the execution to a particular subnet, analyzing channel population, testing the need of forcing synchronization, and even executing incomplete programs. Several functional programs tracers also share some functionalities with our work. They transform the code to produce traces; in some of them it is necessary to indicate the functions to observe during the execution, while in others everything can be observed, but after finishing the execution. NiMo naturally acts as an online tracer and as a truly online debugger because, unlike them, code changes do not require recompiling and starting the process all over. A running prototype has been implemented using AGG [1] as the graph transformation system. An extended version of this paper is to appear in [3].

2. Outline of the NiMo Language

The language has been designed to support totally graphic programming in Data Flow style, with a strong functional inspiration. It is based on the process network representations (as proposed by Turner [5]), where data flow metaphor associates functions to single output processes and channels to (usually infinite) lists. The net architecture

shows in a bi-dimensional way the chains of function compositions and exhibits the implicit parallelism. Back arrows give an insight of the "recurrence laws", i.e. how new results are obtained from already calculated ones. The paradigmatic example is the Fibonacci process network. The graphic execution model suggested by its possible animation was the start point for the NiMo language design. Figure 1 shows the Fibonacci NiMo net, which is equivalent to the Haskell code

fibo = 0: rest **where** rest = 1: zipWith (+) fibo rest

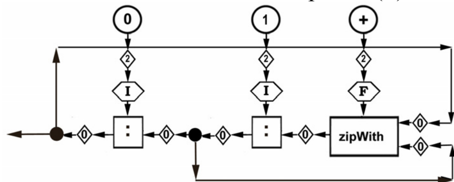

Figure 1. Fibonacci process network

Except for a few new graphic symbols the rest of the elements and their correspondence to the textual code are the usual ones: *rectangles* for functions viewed as processes, *circles* (or ellipses when necessary) for constant values, *arrows* for function parameters and results, avoiding the use of the identifiers needed in textual code (*fibo* and *rest* in this example). Horizontal arrows represent channels and *black-dots* channel duplicators (double occurrence of an identifier in the textual code definition).

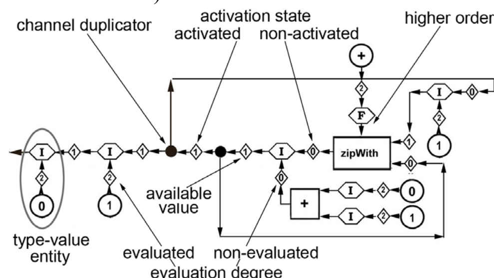

Figure 2. Graphical Syntax

Vertical arrows entering a process correspond to parameters that are not channels in the data flow sense of flowing data streams. The additional symbols allow representing and handling functional concepts like higher order, partial application, laziness, polymorphism, and type inference: *diamonds* carry information about activation or evaluation state. A diamond with 0 precedes a non-activated process or a non-evaluated value. Values in a channel are always preceded by a diamond with code 1. *Hexagons* carry type information. Data are type-value-entities (*tve*), where values (evaluated or not) are tied to its type (hexagon) through a diamond. Code 2 in the diamond indicates that the value is an atomic constant. Fully evaluated values (values in normal form) are tve's having only hexagons, circles and diamonds. Non-fully evaluated values are tve's containing at least one rectangle (a non-evaluated function), as seen in figure 2. Formal parameters are represented by incomplete tve's; i.e. having big arrows (as shown in figure 3). In fact, big arrows represent incompleteness in a more general sense. They are open connections that can be bound later, and so, incomplete programs can run and be completed by need.

NiMo allows complete higher order and partial application. Process parameters can also be parameterized; i.e. not only function names as +, but also partially applied functions (a box with big-arrows in some of its inputs) can be a valid parameter for another process, as >= in figure 3. Processes in NiMo can have multiple outputs, but only single output processes can be a valid parameter.

Atomic types are *integers* (I), *reals* (R), *booleans* (B), *strings* (S). Circles of different colors represent constant values of each type.

Structured types are *lists* (L), *channels* (C), *tuples*(T), and *functions*(F) all being polymorphic. Channels and lists are homogeneous and unbounded. They correspond to the data flow and functional view of lists and are mutually convertible. In the current version user defined types are not supported. Static type inference is partial, which means that polymorphic processes could be non-correctly instantiated and non-type-safe nets could be executed. Type inference and checking at runtime is therefore needed. In the NiMo next version (under preparation) full static inference will be possible. The system provides a repertory of powerful *basic processes* including equivalent versions for most functions in Haskell prelude, but processes in NiMo can have more than one output, like the process splitCond in figure 3. The language also offers additional basic processes like a fixpoint, a generalized zipWith-merge, and a flip-flop-merge, helpful for the process network programming style. The intensive use of the higher-order basic processes promotes a

static solution style, without explicit recursivity in processes; recurrence is done on channels (feedback arrows). The idea is that casuistry and pattern decomposition are encapsulated within the basic processes behavior, they are powerful enough to handle stream processing in the data flow sense; only the simplest user functions (i.e. not involving control flow) should need conditionals, and an if-then-else higher-order process suffices for handling all the conditional situations. However, explicit recursivity, and therefore dynamic nets, can also be represented in NiMo, and a specialized graphic syntax for channel pattern conditional is provided (see figure 3). 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, allowing incremental net complexity up to any arbitrary degree, and nets can also be parameterized. In addition to basic processes some useful parameterized nets are provided in a library that can be extended with user-defined nets.

3. The Execution Model

NiMo follows a functional-data-flow mixed model. In the basic data flow model, control is data driven, i.e. processes act by data availability; the precedence graph allows to fully exploit the implicit parallelism. In lazy functional languages functions produce results only when they are required. To produce the result *(w.h.n.f)* only the needed parameters are evaluated with the same criterion. In the mixed model, laziness means that processes only produce results under demand, i.e. the control flow follows a demand-driven policy.
On the other hand, lazy languages have primitives that annotate a sub expression to be eagerly evaluated; it is the only way the user can explicitly modify the control. In NiMo each subnet is explicitly preceded by its evaluation state, and the user can set an initial activation marking or change the current activation state of any process during execution. Nevertheless basic processes have a lazy behavior; activation propagates by need, and persists until a "sufficiently evaluated" result is obtained. An activated process can demand non-available values to its provider processes, which are activated "in parallel" (all those needed, not only the "leftmost one", as in sequential lazy languages) 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. Besides, during execution the user can activate/deactivate any process by simply changing the code of its preceding diamond. This facility opens a wide range of experimental possibilities not present in the offline style debuggers. Some regions of the net can be tested in an isolated manner, and inactive subnets could be activated for parallel execution.

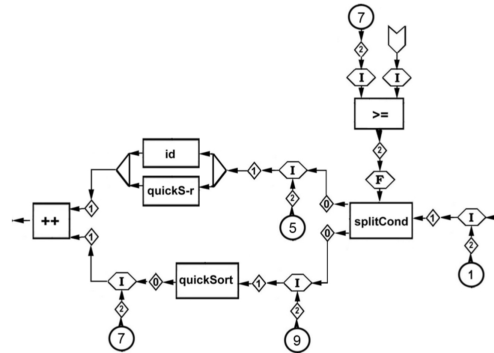

Figure 3. Running quickSort algorithm

4. The NiMo Environment

The environment is based on a graph transformation system. It has two kernel attributed graph grammars: *Construction Grammar* (CGG), and *Execution Grammar* (EGG). EGG is the interpreter, and also gives the complete semantic definition of the language, while CGG acts as an interactive syntax directed editor that checks a first level of type-consistency. Once a new net is built, it can be directly executed or else be given a name to be further used. This action generates two new rules, an *interface rule* giving its type declaration (input and output channels and parameters types), and an *expansion rule*. The interface rule is needed to construct other nets that have it as a process. It is also used in execution to allow higher order parameterization. The expansion rule will be added to EGG to be used while executing a net having a box with the same name. When this process is activated, its whole net replaces the box that already has the necessary connections. According to the demand-driven policy, the initial activation of the net is done by

activating its output channels (although a different initial marking could be set). If the first process in a channel is not basic, it is expanded, and so on until a basic process can act. Once activated it is responsible for activating its provider processes. The set of rules defining the basic processes behavior is analogous to the equations defining a function in Haskell. Different configurations in its adjacent nodes give the patterns.

5. Current state and future work

A running prototype has been implemented using AGG [1]. The system has been tested on classical examples as Hamming numbers, prime numbers in both static and dynamic versions, prefixes, quicksort in different dynamic and static versions, obtained by derivation from the classical one, other sorting algorithms, and of course, Fibonacci. Full higher order and partial application were tested with a set of examples as the equivalent to the following Haskell codes:

```
zipWith id (zipWith const(zipWith id
(repeat (+)) [1..])  [1..]) [1..]
zipWith id (map (zipWith (const id))
[[1],[2],[3,4]]) [[5],[6,7,8],[9],[ ]]
```

The work done shows the feasibility of the approach and the experimental possibilities it opens. However, although the main design and implementation challenges have been overcome, for the time being, the current state of the system is far from being a final product. One of the main drawbacks of the current version is, precisely, the graphical interface. To be a really useful tool this aspect should be heavily improved. As the net grows, it is necessary to have control on the scaling, zooming, scrolling and other facilities allowing "pretty printing" of the net, not present in the graphic interface of the current transformation system. Nevertheless, the graph transformation engine can be used isolated and be combined with another interface. In `http://www.lsi.upc.edu/˜NiMo/Project/` executions of some of the examples tested in the prototype are shown as they would look like with an improved graphic interface. Regarding the extension of the language itself, a more powerful type inference system is under development. And, in general, other Haskell features not having equivalence in NiMo, should also be considered. In particular, user defined types and therefore data abstractions mechanisms, which are a key issue for scalability [2]. On this regard, 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 usual mechanisms (scrolling, zooming, 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. Besides, once expanded, a net process currently remains so whether active or not. Another possible approach for a better use of the screen real estate would be to provide operations to reverse the expansion, and in general to control the visibility of the internal transformations of subnets. Finally, translation from NiMo to Haskell is almost direct. NiMo is specially suited for debugging and experimentation, not for being the final execution system. The last step of the workbench should be to produce an equivalent but efficiently executable textual version, ideally in a parallel language.

References

[1] AGG HOMEPAGE,
AGG Home page: http//tfs.cs.tu-berlin.de/agg/

[2] Burnett M., Baker M.J., Bohus C., Carlson P., Yang S., and. Van Zee P. Scaling up visual Programming languages. *Computer*, 28:45–54, 1995.

[3] Clerici S., Zoltan C. A Graphic Functional-Dataflow Language.Trends in Functional Programming, Volume 5. Editor Loidl H-W. Intellect 2005. (to appear).

[4] Reekie H., Visual Haskell: A First Attempt. Technical Report 94.5, Key Center for Advanced Computing Sciences, Univ. Tech. Sydney, 1994.

[5] Turner D., Functional programs as executable specifications. Proc. of a discussion meeting of the Royal Society of London on Mathematical Logic and Programming Languages, Prentice-Hall,1985.