

NiMo: un lenguaje gráfico para usuarios finales

Silvia Clerici

Universitat Politècnica de Catalunya, Dept. Llenguatges i Sistemes Informatics
Barcelona, España, 08028
silvia@lsi.upc.edu

y

Cristina Zoltan

Universitat Politècnica de Catalunya, Dept. Llenguatges i Sistemes Informatics
Barcelona, España, 08028
zoltan@lsi.upc.edu

Resumen

En este trabajo presentamos un lenguaje de programación visual inspirado en redes de procesos, con construcciones sencillas, bien adaptado a usuarios finales y completamente gráfico. El lenguaje tiene una semántica muy clara y el hecho de que la prueba de programas se pueda hacer paso a paso, corrigiendo el código sin tener que recomenzar la ejecución, y visualizando las transformaciones que sufre el programa a medida que va ejecutándose hace que sea muy fácil la programación y la puesta a punto de programas. Su estructura jerárquica permite la construcción de aplicaciones de envergadura. En la literatura de programación funcional, vemos con frecuencia que los programas funcionales textuales se describen como redes de procesos, a modo de asistencia visual a la comprensión del mismo. NiMo (Nets in Motion) es la extensión de las redes de procesos a un lenguaje de programación con un ambiente que actúa como un banco de desarrollo y prueba de programas. La potencia de sus primitivas algorítmicas, a la vez muy intuitivas, junto con la claridad y expresividad gráfica del modelo Data-flow lo convierten en un lenguaje sencillo y seguro para que programen usuarios finales.

Palabras claves: Lenguaje de programación, gráfico, funcional, Data-flow, paralelismo implícito, usuario final

Abstract

In this paper we present a visual programming language inspired in process networks, with simple constructs well suited to end users and totally graphic. The language has a very clear semantics, and the fact that programs can be executed step by step, modifying code without starting execution again, seeing all the way the program transformation during execution, makes programming and program tuning an easy task. The language hierarchical structure allows construction of rather complex applications. In functional programming literature we see very often that textual programs are described as process networks, as a visual aid to its understanding. NiMo (Nets in Motion) is an extension from process networks to a complete programming language, with an environment acting as a workbench for program development and testing. The power of its algorithmic primitives, very intuitive also, combined with the Data-flow model graphical expressiveness turn NiMo into a simple and safe language for end users.

Keywords: Programming language, graphical, functional Data-flow, implicit parallelism, end user.

1. INTRODUCCIÓN

NiMo [4,5,6] está inspirado en las representaciones en redes de proceso de programas funcionales perezosos, donde los programas son grafos orientados que admiten ciclos. Los nodos del grafo son funciones vistas como procesos conectados por canales (listas usualmente infinitas), implementando la metáfora Data-flow de proveedor-consumidor. Tal como sucede en los lenguajes funcionales y los modelos Data-flow en NiMo no existe la noción de almacenamiento. La red de programa incluye los datos con los que opera. Pero a diferencia de casi todos los lenguajes funcionales en NiMo no se usan variables para indicar parámetros. Los únicos identificadores requeridos son los identificadores de procesos.

Los programas NiMo son grafos orientados, donde los arcos son canales de capacidad no acotada, por donde viajan en forma FIFO los datos (que pueden ser valores simples, estructurados, o funcionales). Los procesos pueden tener parámetros que pueden ser otros procesos, valores o canales.

El grafo o red, en todas sus transformaciones durante la ejecución sigue siendo un programa NiMo, por lo que el ambiente de desarrollo permite la ejecución paso a paso de un programa y por ende se constituye en un mecanismo de “Lo que ves es lo que pruebas” (WYSIWYT). Este modo de operación puede ayudar a comprender el algoritmo, hacer pruebas e incluso permite interactuar cambiando partes y completando redes incompletas.

El lenguaje dispone de un conjunto de procesos básicos que están bien adaptados al procesamiento Data-flow e inspirados en los transformadores de listas de lenguajes como Haskell [8]. Puesto que el efecto de estos procesos es muy intuitivo y fácil de describir, aprender a combinarlos gráficamente es sencillo y al alcance de los no programadores. Como los procesos pueden estar parametrizados por procesos, con un número reducido de procesos básicos, bien escogidos, basta para cubrir todos los casos posibles de tratamiento de flujos de datos.

En el modelo Data-flow clásico los procesos actúan por disponibilidad de datos. NiMo sigue un modelo de ejecución Data-flow -perezoso, los procesos actúan sólo por demanda pese a que puedan tener ya sus datos disponibles. La activación de un proceso se visualiza en la red y permanece así hasta satisfacer el requerimiento. La ejecución de una red muestra paso a paso todas las transformaciones que ésta sufre, es decir, es una sucesión de redes en las que las redes intermedias tienen uno o más procesos que pueden ejecutarse. El usuario puede interrumpir la ejecución, modificar la red actual y retomar la ejecución directamente. Esta es una de las características más peculiares de NiMo posible gracias a que desde el punto de vista del usuario el código es a la vez el código y el estado de la computación.

Un ejemplo de programa NiMo puede verse en la figura 2 y ejecuciones paso a paso y animaciones se encuentran en la página del proyecto NiMo [5].

En la sección siguiente se describen los elementos que constituyen el lenguaje. A continuación se desarrolla un ejemplo de aplicación. En la sección 4 se comentan algunos lenguajes visuales en alguna medida relacionados con NiMo, y en la sección siguiente se describe el estado actual de desarrollo del lenguaje y su entorno, y se mencionan las futuras líneas de trabajo.

2. ELEMENTOS DEL LENGUAJE

2.1 Elementos gráficos

El lenguaje utiliza un repertorio muy reducido de símbolos gráficos para construir las redes. Los siguientes 9 símbolos corresponden a la variedad de tipos de nodo NiMo, como puede verse en la figura 1:

- rectángulos para representar procesos
- círculos (u óvalos) para valores constantes
- puntos llenos para duplicadores de canal
- diamantes para información de activación o estado de evaluación
- hexágonos para información de tipo
- nodos-flecha para conexiones abiertas o parámetros formales
- triángulos para condicionales sobre patrones de canal

Las conexiones entre nodos siguen las siguientes convenciones:

arcos horizontales para canales de datos y arcos verticales entrando a un proceso para parámetros que no son canales en el sentido Data-flow. Para conectar nodos de tipo con su valor y el estado de evaluación del mismo se utilizan también arcos verticales, a estos subgrafos los denominaremos *valor-tipado*.

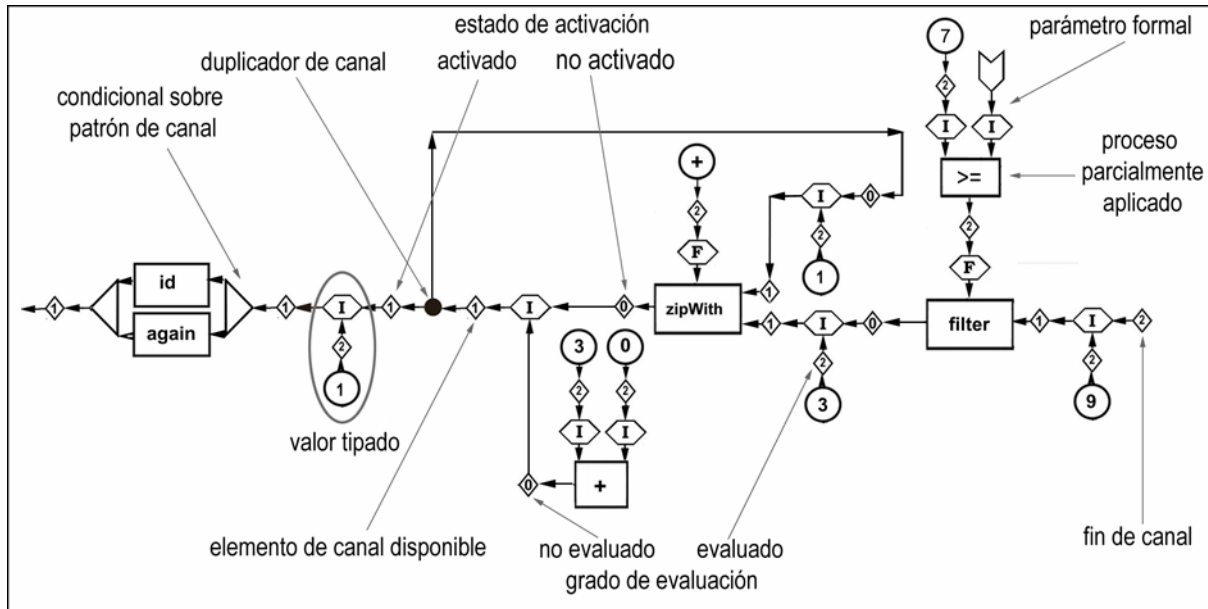


Figura 1. Sintaxis gráfica

Los elementos de canal son valores tipados que están precedidos en el canal por un diamante que indica que el elemento está presente (ya sea que tenga su valor evaluado o no). Los diamantes tienen en los grafos NiMo diferentes colores según el grado de activación o disponibilidad. En este artículo usaremos una convención numérica: 0 en un diamante que precede a un proceso indica no activado, 1 indica activado. Si el diamante precede a un elemento de canal (valor tipado) tiene código 1. Si el diamante precede a un valor constante en un valor tipado tiene código 2. El fin de canal también se indica con un diamante con código 2.

En el grafo de los programas NiMo se representa el flujo de resultados de derecha a izquierda para visualizarlos tal como se leen las secuencias, es decir con el primer elemento más a la izquierda, y por lo tanto las salidas de la red son los arcos de más a la izquierda.

2.2 Procesos Básicos y Redes predefinidas

El lenguaje NiMo posee una colección de procesos básicos que incluye los necesarios para el procesamiento de los tipos escalares (operadores aritméticos, relacionales, lógicos, etc.), y otros adecuados al procesamiento Data-flow. Una buena parte de estos últimos están inspirados en las funciones (usualmente polimórficas y de orden superior) para el tratamiento de listas de los lenguajes funcionales, y en general se les ha mantenido el nombre que tienen en el Prelude de Haskell.

Por ejemplo **map** es un proceso que realiza la misma transformación a todos los elementos del canal. En ejecución **map** aplica su primer argumento (otro proceso) a cada uno de los elementos de su canal de entrada produciendo un canal de salida con la transformación de estos según el proceso parámetro. El proceso ha de ser aplicable a elementos del tipo de los elementos del canal. Por ejemplo, si es el operador de valor absoluto y la entrada es un canal de enteros lo transformará en uno de naturales, y si es el proceso **hd** (cabeza de una lista) y los elementos del canal son listas de tipo T se obtendrá un canal de elementos de tipo T con los primeros elementos de cada lista del canal de entrada. El proceso **zipWith** es el análogo a **map** para dos canales de entrada, en ejecución el proceso parámetro se aplicará en cada paso al primer elemento de ambos canales. En las figuras 3 y 4 puede verse un paso de ejecución del proceso **zipWith** en la red Fibonacci.

Otros procesos básicos permiten seleccionar los elementos de un canal de acuerdo a una propiedad controlada por su parámetro que es un proceso de resultado booleano: **filter** selecciona todos los elementos que la cumplen, **takeWhile** devuelve sólo los primeros (descartando todos los demás desde el primero que deja de cumplirla), y su inverso **dropWhile** elimina los primeros que cumplen la propiedad. Los procesos **take** y **drop** con parámetro n permiten respectivamente tomar o eliminar los primeros n elementos, y **merge** permite mezclar dos canales ordenados y del mismo tipo.

Sin embargo en NiMo no existe la restricción funcional de que los procesos tengan una sola salida por lo que muchos de los procesos básicos no tienen un equivalente directo en Haskell, por ejemplo el proceso **split-cond**, que dada una

propiedad descompone un canal de salida en dos canales, el primero con los elementos que la cumplen y el otro con los que no, o el proceso **takeDropWh** descrito en el ejemplo de la sección 3. Otros procesos NiMo tampoco tienen equivalente porque están especialmente pensados para la programación Data-flow, posible pero no especialmente fomentada en Haskell, como por ejemplo el proceso **shuffle** que intercala dos canales del mismo tipo, o su inverso **split** que separa los elementos del canal de entrada alternándolos en dos canales de salida. Otro proceso muy útil para la construcción de redes de procesos es **gate2** para el tratamiento genérico del caso “dos canales de entrada y uno de salida”, del cual **zipWith**, **shuffle**, y **merge** son casos particulares.

Asimismo, el hecho de que el modelo de ejecución sea perezoso promueve el uso de redes capaces de producir secuencias infinitas. Entre los procesos de esta naturaleza está por ejemplo **nat** que generara los naturales ordenadamente

Además de los procesos básicos, NiMo provee un conjunto de redes predefinidas como por ejemplo **partAcum** que calcula resultados parciales de cálculos por acumulación como sumas parciales, mínimos hasta el momento etc., (ver su uso en el ejemplo de la sección 3) y **acum** que produce el resultado total. La red **sort** permite ordenar un canal según el orden indicado en el proceso parámetro. La red **prefix**, que produce los prefijos de su canal de entrada, es especialmente útil para “mantener memoria” de los elementos ya obtenidos o procesados de un canal.

La enumeración anterior de procesos básicos y redes predefinidas no es exhaustiva, en particular en la sección 3 se desarrolla un ejemplo completo en el cual se describen y utilizan algunos no mencionados aquí. Como se habrá podido observar la mayoría de los procesos y redes predefinidos tienen una funcionalidad tan sencilla de describir que los hace comprensibles por personas sin ningún conocimiento previo de programación.

2.3 MODELO DE EJECUCIÓN

NiMo sigue un modelo de ejecución Data-flow-perezoso, los procesos actúan sólo por demanda pese a que puedan tener ya sus datos disponibles. La activación de un proceso se produce por la demanda ejercida por alguno de los procesos del cual es proveedor, o del exterior si el proceso genera alguna de las salidas de la red. El proceso activado se ejecutará si ya dispone de los parámetros necesarios para hacerlo, de lo contrario el proceso propagará la demanda a los proveedores de los parámetros que necesite para actuar. La activación de procesos se visualiza en la red mediante diamantes coloreados que marcan los caminos donde se espera flujo de respuesta.

Todos los procesos en condiciones de actuar se pueden ejecutar en paralelo por lo que no es necesario poner anotaciones para indicar ejecución paralela, o sincronización, ni el establecimiento de un orden de ejecución externo a la red. El control de ejecución viene dado por la política dirigida por demanda y regulado por la disponibilidad de datos. Sin embargo el usuario puede establecer una activación inicial de procesos que no obedezca a la demanda por propagación e incluso cambiar el estado de activación de cualquier proceso deteniendo para ello la ejecución.

Cuando un proceso es activado, si es un proceso básico y está en condiciones de actuar produce la transformación correspondiente. Si el proceso es una red se produce su expansión reemplazándose en el grafo la caja del proceso por la red que lo implementa.

En el grafo de los programas NiMo se representa el flujo de resultados de derecha a izquierda, mientras que el orden por defecto de ejecución por propagación es de izquierda a derecha.

Un ejemplo clásico es la red Fibonacci que en NiMo se expresa del siguiente modo

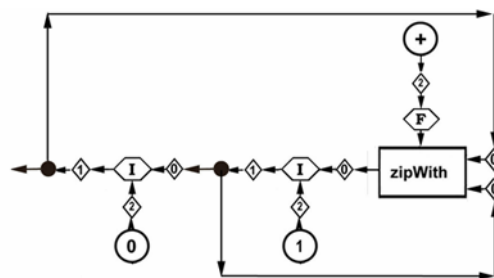


Figura 2. Red Fibonacci

Cuando se requiere un valor a la red, se activará el primer proceso duplicador, que al tener un valor 0 disponible en su canal de entrada lo colocará en el canal de salida y también en el primer canal de entrada de **zipWith**. Al requerirse un segundo valor el primer duplicador (ya sin dato disponible en su canal) activará al segundo duplicador que actuará sobre su valor 1 situándolo en el canal de entrada del primer duplicador y en el segundo canal de entrada de **zipWith**. Dado que el primer duplicador mantiene su demanda el valor 1 es también duplicado por éste situándose en el canal de

salida y encolándose en el primer canal de **zipWith**. Ante una nueva solicitud a la red la demanda se propaga hasta el proceso **zipWith** resultando la red de la figura 3. En la figura 4 se ve el siguiente paso de ejecución.

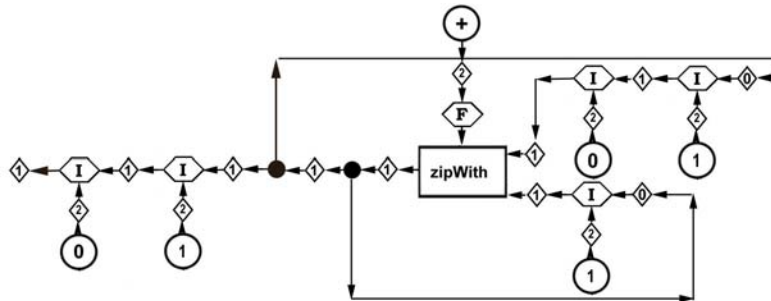


Figure 3. Fibonacci cuando el tercer valor es requerido

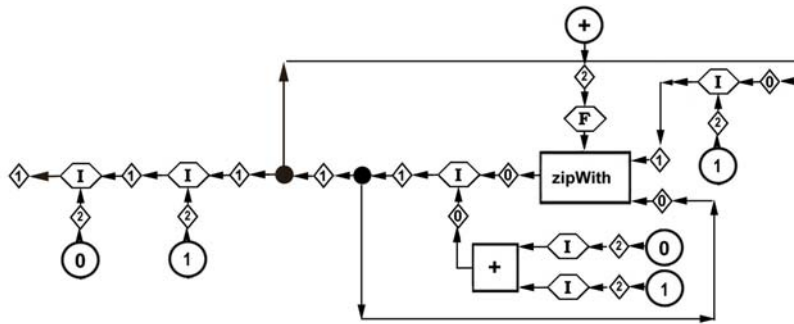


Figure 4. Fibonacci con el tercer valor producido

Nótese que el parámetro actual (proceso +) de **zipWith** es la función suma, es decir un valor constante de tipo función. Cuando **zipWith** (que está activado y tiene datos disponibles) la aplica a sus dos primeros elementos de canal se produce en el canal de salida un resultado de tipo entero cuyo valor se obtendrá por aplicación del proceso + (nodo rectángulo). Como ese valor aún no ha sido calculado el rombo que lo precede tiene un diamante con valor 0.

3. EJEMPLO DE DESARROLLO DE UNA APLICACIÓN NiMo

A efectos de dar una visión de la metodología de trabajo en NiMo tomaremos un caso del tipo de problemas de interés para usuarios de gestión, simplificado por el espacio disponible en este artículo, pero suficiente para mostrar la potencialidad de NiMo para atacar problemas de complejidad creciente.

Una empresa está interesada en controlar el costo diario acumulado de sobregiro bancario para evitar que supere un costo máximo admisible. Cada día el departamento de cobros emite un detalle de los cobros realizados, y el de pagos el correspondiente a los pagos de ese día. Se dispone diariamente de la tasa de interés que determina el banco Central por sobregiro (costo del dinero en ese día), y del saldo anterior al primer día de utilización de la aplicación.

El diseño a primer nivel de la red puede verse en la figura 5.

Los procesos **dtoCobros** y **dtoPagos** producen en sus respectivos canales de salida elementos que son listas de reales, una lista por cada día a partir de la fecha inicial. El proceso **tasaDiaria** produce una lista de reales con las tasas diarias de interés a partir de la fecha inicial. Los tres procesos proveen los flujos de datos de entrada a la red.

El proceso **balance diario** deberá totalizar los cobros y pagos diarios y realizar su diferencia para obtener el balance diario, que será un canal de reales. Las redes utilizadas para este proceso pueden verse en la figura 6.

La red **saldo bancario** recibe como parámetro el saldo inicial, y utilizando los balances diarios calcula el saldo bancario diario produciendo un canal de reales. Dicho canal, junto con las tasas diarias de interés son los canales de entrada a **control intereses**, que tiene además como parámetro el umbral (máximo admisible por sobregiro).

control intereses primero debe seleccionar los saldos negativos y determinar el costo de sobregiro en cada uno de esos días, para luego calcular los costos parciales acumulados y producir un canal de salida de la red. En cuanto se supere el umbral admisible la aplicación se detiene produciendo en su segunda salida (que no es un canal sino un dato escalar de tipo real) el valor acumulado que superó el umbral.

Los procesos de salida **costo intereses** y **umbral superado** son los consumidores del sistema que actuará respondiendo a su demanda. Esta demanda se propagará en la red de izquierda a derecha y por lo tanto el primer proceso que actuará (en cuanto tenga sus entradas disponibles) será **balance diario**.

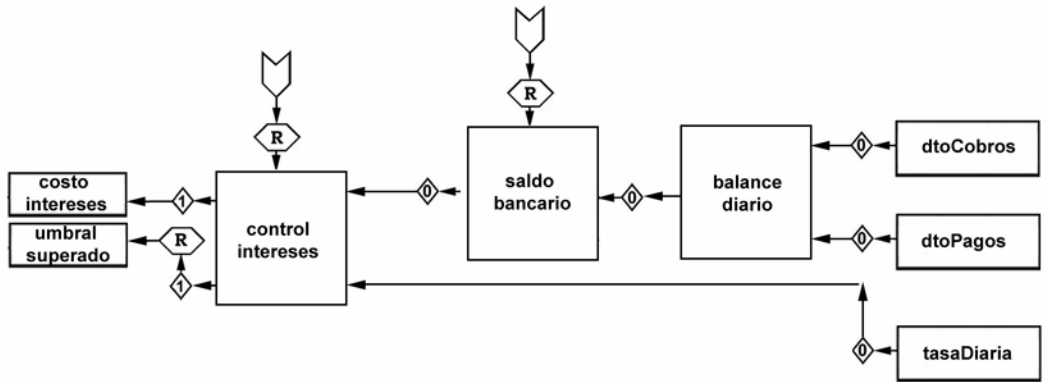


Figura 5. Red de primer nivel

El proceso **balance diario** (figura 6) utiliza para totalizar los cobros y los pagos del día el mismo proceso **totalizar**. Este proceso aplica a cada lista de detalle de cobros / pagos (mediante el proceso **map**) el cálculo de su sumatoria. Para ello tiene como parámetro el proceso **acum**, que dada una operación y un valor inicial da como resultado el valor total acumulado de su canal de entrada.

En la red **totalizar** el proceso **acum** está parcialmente aplicado a sus dos primeros parámetros (operador de suma y valor inicial 0) quedando libre el tercero que (según el tipo de los dos primeros) ha de ser un canal de números reales. Puesto que **acum** es el proceso que **map** aplicará a los elementos de su canal de entrada, estos deben ser listas de números reales.

El proceso **zipWith** a cada par de elementos de sus canales de entrada (total de cobros y pagos diarios respectivamente) le aplica el operador de resta, obteniendo un canal con la sucesión de balances diarios

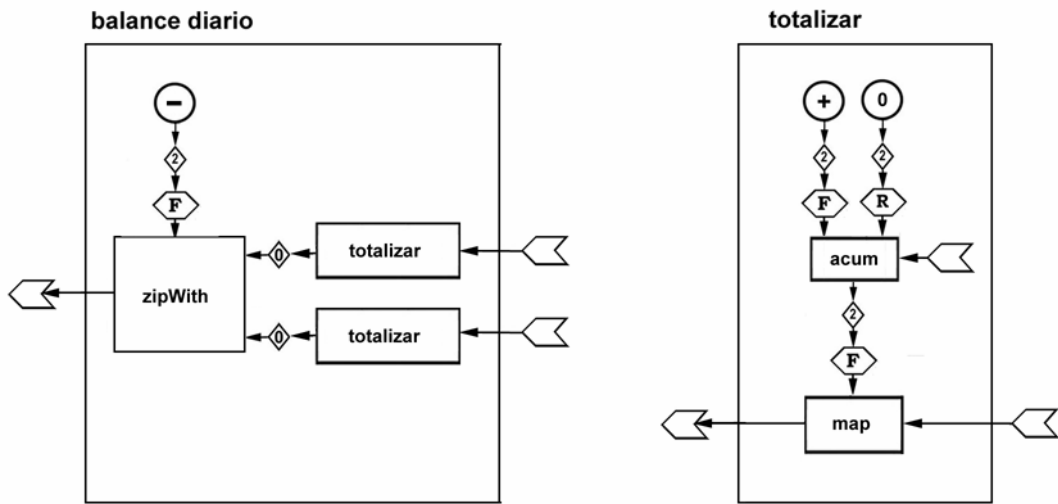


Figura 6. Red balance diario y subred totalizar

La red **saldo bancario** (figura 7) recibe como parámetro el saldo inicial, y utilizando los balances diarios obtiene el saldo bancario diario, que se calcula sumando al saldo del día anterior el balance diario, es decir cada uno de los saldos es la acumulación parcial de los balances diarios mas el saldo inicial.

El proceso **partAcum** es una red predefinida del sistema que dada una operación, un valor inicial y un canal de valores de entrada produce la secuencia de resultados parciales aplicando reiteradamente dicha operación. El refinamiento de esta red no forma parte del diseño del usuario pero hemos considerado interesante mostrarla para ilustrar el uso de tipos

polimórficos, nociones de deducción de tipos, los mecanismos de construcción de canal a partir de parámetros valor, y para dar otro ejemplo de red que reutiliza los valores ya calculados (red con ciclos). Sin embargo un usuario final no necesita dominar estos conceptos y no es necesario que sepa cómo funcionan internamente las redes predefinidas, sino simplemente el efecto que producen.

partAcum construye un canal con el valor inicial como primer elemento seguido de los que vaya produciendo en su canal de salida (que son duplicados para retroalimentar dicho canal). Este canal se construye con el proceso “:”, que dado un elemento y un canal de entrada lo pone a la cabeza del canal. El tipo polimórfico de su parámetro valor inicial se indica con el símbolo ?. Cualquiera sea el tipo concreto con que se instancie ha de tener el mismo tipo de los elementos del canal de salida, y el proceso parámetro debe ser aplicable a pares de elementos del tipo del valor inicial y de los elementos del canal de entrada, tipos que podrían ser diferentes.

El proceso **zipWith** aplica la operación a cada par de elementos del canal de entrada y del canal de resultados parciales produciendo en cada momento el siguiente resultado acumulado. La primera vez se opera sobre el primer valor del canal de entrada y el valor inicial.

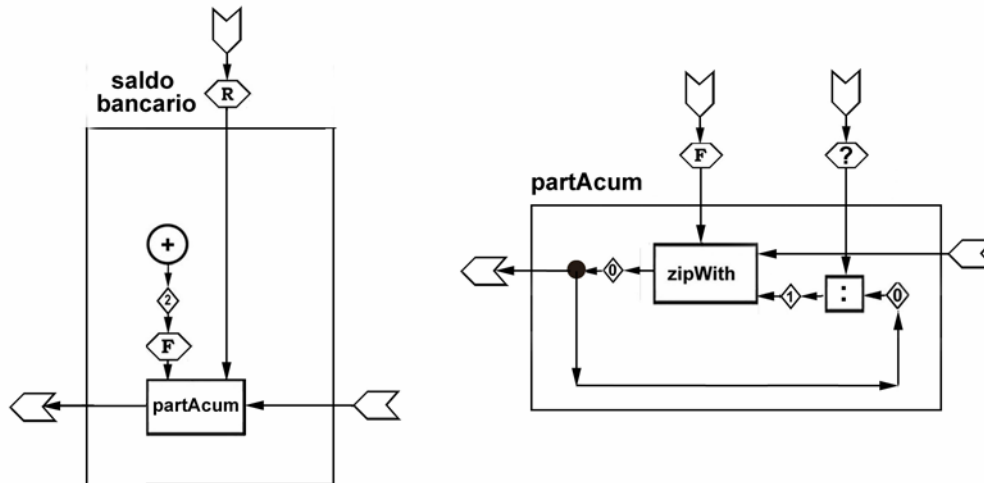


Figura 7. Red saldo bancario y subred predefinida **partAcum**

El saldo bancario diario junto con las tasas diarias de interés y el umbral son las entradas al proceso **control intereses** (figura 8), que primero debe seleccionar los saldos negativos y calcular el costo de sobregiro en cada uno de esos días. Para ello utiliza el proceso **filterZip** que usa su primer parámetro proceso para decidir si debe aplicar a ambos primeros valores de sus canales de entrada el proceso que es su segundo parámetro. De ser así calcula el correspondiente elemento de salida, y en cualquier caso consume ambos primeros elementos de sus canales entrada.

En esta red la condición de **filterZip** es que el saldo sea negativo, y el cálculo a aplicar es simplemente cambiar el saldo de signo y multiplicarlo por la tasa de ese día. Los respectivos procesos **sdoneg** e **interes**, parámetros de **filterZip**, son suficientemente simples como para que la versión textual resulte mucho más sencilla que la gráfica. El lenguaje NiMo admite que se los defina de forma textual (siguiendo la notación Haskell) y genera la red correspondiente.

En la medida en que **filterZip** produce la sucesión de intereses pagados, estos son parcialmente acumulados por el proceso **partAcum**. Esta sucesión, mientras no supere el umbral seguirá siendo calculada y colocada en el primer canal de salida. Si el umbral se supera se detiene el cálculo y el valor que supera el umbral será el resultado de la segunda salida de la aplicación. Para ello se utiliza el proceso **takeDropWh**, que pasa a su primer canal de salida los primeros elementos de su canal de entrada que cumplen la condición indicada en su parámetro proceso. A partir del momento en que un elemento de la entrada no cumpla la condición se cierra el primer canal de salida, y el canal de entrada (incluyendo el elemento de corte) se conecta al segundo canal de salida.

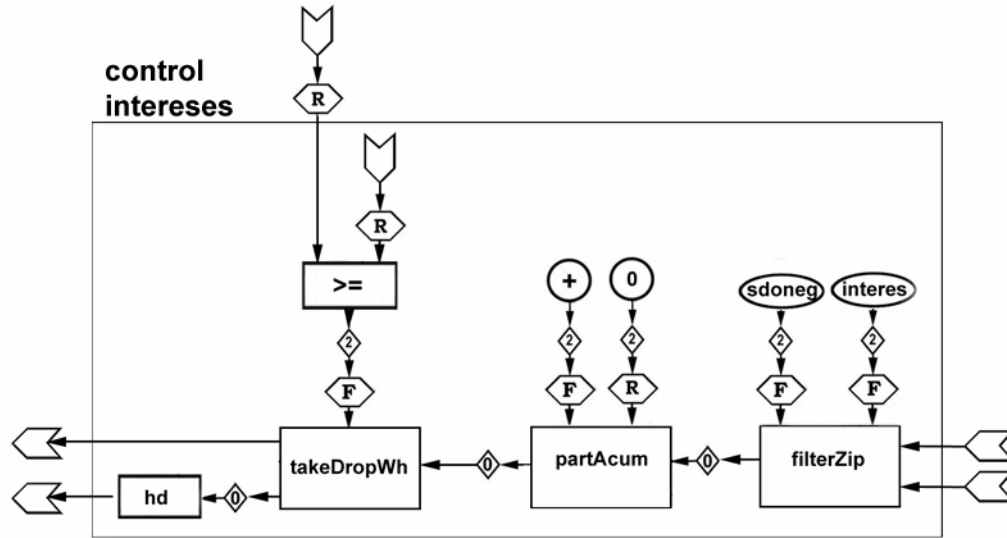


Figura 8. Red control de intereses

En nuestro caso la condición es que el umbral, que es el primer parámetro de **control intereses**, sea mayor o igual que los elementos del canal de entrada, pasando todos los que la cumplan al canal de salida de la red **control intereses** y por ende al de la aplicación. De superarse el umbral el proceso **hd**, que tiene como entrada el canal con saldos que superan el umbral, tomará sólo el primero de ellos para producir el valor que será el segundo resultado de la aplicación. Como ese segundo canal de **takeDropWh** queda ya sin demanda, deja a su vez de demandar a sus proveedores y por lo tanto se detiene la red en su totalidad.

De no superarse nunca el umbral, el proceso continuaría produciendo nuevos intereses acumulados en la medida en que se sostenga la demanda de resultados y se produzca un nuevo sobregiro bancario. Es decir, podría darse el caso, en una empresa muy exitosa, que esta red no llegara a producir nunca resultados pues nunca se produjo un sobregiro.

Como puede observarse en el ejemplo anterior se ha seguido un diseño descendente muy modular, las redes no superan los 3 o 4 procesos y cada uno de ellos puede ser naturalmente asociado a un proceso básico o red predefinida, o a una combinación simple de ellos, a partir de los cuales ya no se requiere seguir refinando. El entrenamiento de un usuario final para poder resolver (o entender soluciones para) problemas del grado de complejidad ejemplificado, se deberá centrar en comprender a profundidad la funcionalidad de estos “bloques algorítmicos prefabricados”, y saber identificar en el problema concreto qué combinación de ellos le permite alcanzar la solución. Razonamientos del estilo de “esto es un resultado acumulado, hay que usar un **acum**”, “aquí hay que seleccionar según esta propiedad, es un **filter** o un **splitCond**”, o “hay que aplicarles a todos los deudores este proceso, esto es un **map**”.

4. OTROS LENGUAJES GRÁFICOS

Con el desarrollo de las interfases gráficas y la enorme capacidad de procesamiento gráfico de los ordenadores, aun los más sencillos, en los últimos tiempos se ha desarrollado una gran actividad en el área de lenguajes visuales. Sin embargo muchos de los llamados lenguajes visuales no tienen una sintaxis gráfica sino textual, el término visual en esos casos se refiere a que tienen un ambiente de desarrollo con elementos visuales.

Los lenguajes funcionales parecen muy apropiados para tener una expresión visual en lugar de textual, y varios intentos de este tipo encontramos en [12,8]. Visual Haskell de [13] fue un intento de disponer de una notación visual para los programas Haskell (no confundir con Visual Haskell de .NET, que es un ambiente integrado para desarrollo de programas Haskell sobre Visual Studio). Los trabajos de Kelso [9] desarrollan un ambiente que incluye interpretación de los árboles de expresiones en presentación grafica, útil para la comprensión de algoritmos.

Entre los lenguajes gráficos basados en el paradigma Data-flow, que se desarrollaron inicialmente orientados a aplicaciones de procesamiento de señales, podemos mencionar a LabVIEW [10], Agilent Vee[2] y Ptolomy[12]. En particular LabVIEW tiene un lenguaje completo gráfico llamado G con primitivas gráficas for, while loops y estructuras case. Estos lenguajes tienen un componente importante de conexiones con dispositivos o instrumentos

debido a que inicialmente fueron diseñados como lenguajes de propósito específico. Todos ellos tienen una estructuración jerárquica del grafo de programa (con flujo de izquierda a derecha), pero las componentes pueden estar escritas en otros lenguajes: MATLAB®, C etc. La instanciación de parámetros de los procesos se realiza mediante formas textuales que no son visibles durante la ejecución. Una característica destacada de Ptolomy es la de tener una gran variedad de modelos de computación. A cada componente se le asigna un modelo de computación, por lo que una aplicación puede realizarse usando diferentes modelos. NiMo posee un modelo uniforme Data-flow-perezoso por defecto y localmente modificable en ejecución.

Como la principal orientación de estos lenguajes es al tratamiento de señales, los datos se visualizan con los dispositivos usuales (Ej. una pantalla simulando la de un osciloscopio). En particular en LabVIEW para visualizar los valores que viajan por los hilos de conexión, se le asigna un identificador al hilo y en una zona de la pantalla distinta al grafo, llamada panel, se tienen los visores de todos los hilos identificados. En NiMo, los valores mismos viajan por los canales en el grafo de programa. De utilizarse un visor como forma abreviada de visualización, este está en el propio canal (ver figura 10). El hecho de que todos los componentes de un programa NiMo se puedan visualizar en una misma pantalla, hace que sea más sencillo seguir el comportamiento de un programa en ejecución.

5. ESTADO ACTUAL Y TRABAJO FUTURO

Se implementó un prototipo usando AGG[1]. AGG es un sistema de transformación de grafos, que dada una gramática de grafos permite visualizar paso a paso la secuencia de transformaciones de un grafo inicial dado. La gramática desarrollada tiene dos componentes: la gramática de construcción de programas NiMo sintácticamente correctos (editor dirigido por sintaxis) y la gramática de ejecución que actúa como un intérprete. Dado que un programa NiMo puede ejecutarse antes de ser completado, ambas componentes se utilizan en la transformación del grafo. El sistema desarrollado incluye tratamiento de orden superior completo, instanciación parcial e inferencia de tipos. El prototipo demostró la factibilidad del enfoque y su potencialidad como banco de transformación y prueba de programas. Asimismo, dio como subproducto una especificación (como gramática de grafos) del lenguaje en su totalidad, es decir una semántica operacional completa ejecutable.

Sin embargo, la visualización de la secuencia de ejecución que produce el prototipo es muy deficiente. Esto se debe a que en la transformación no se respeta el diseño gráfico inicial de la red, ya que no se pueden establecer restricciones sobre la ubicación relativa de los nodos, la verticalidad, horizontalidad y ortogonalidad de los quiebres en los arcos, etc. Incluso en la ejecución paso a paso las facilidades para reordenarlo son muy elementales, lo que lo hace poco útil como herramienta de experimentación. En la actualidad se está desarrollando un ambiente para NiMo llamado NiMo Toons [11], que tiene como uno de sus objetivos más destacados el conservar lo mejor posible la estructura de un grafo que va siendo transformado en ejecución.

Además de las facilidades clásicas de visualización de ventanas (*scaling*, *zooming*, *scrolling*) imprescindibles para tratar con gráficos que crecen mucho [3], el diseño incluye herramientas de control sobre la visualización de la red, como subredes que puedan compactarse o no expandirse en ejecución, ocultamiento opcional de los nodos de tipo, plegado de parámetros, o visores de canal. Un ejemplo de cómo se reduciría el tamaño de la red aplicando este tipo de transformaciones sin perder la estructura del flujo de datos se puede ver en la figura 10. Los recuadros de la red de la figura 9 señalan procesos que pueden compactar sus parámetros y los canales que pueden visualizarse con visores. La primera red de la figura 10 muestra dichas transformaciones y el recuadro señala un subgrafo que corresponde a la expansión de una red que puede revertirse. En la segunda se ve el resultado y se puede apreciar la reducción de espacio de visualización respecto de la red inicial.

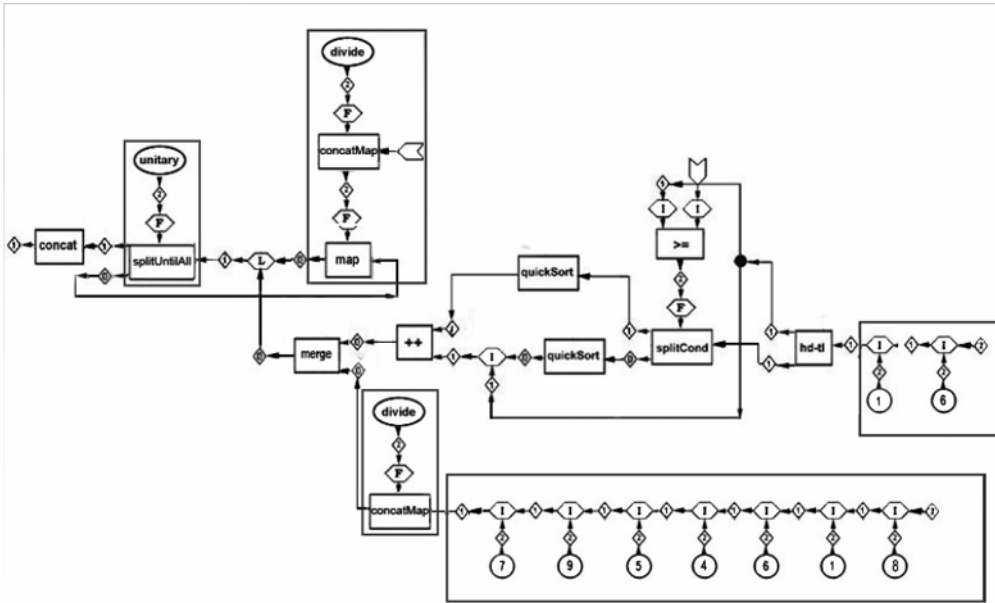


Figura 9 Red sin compactar

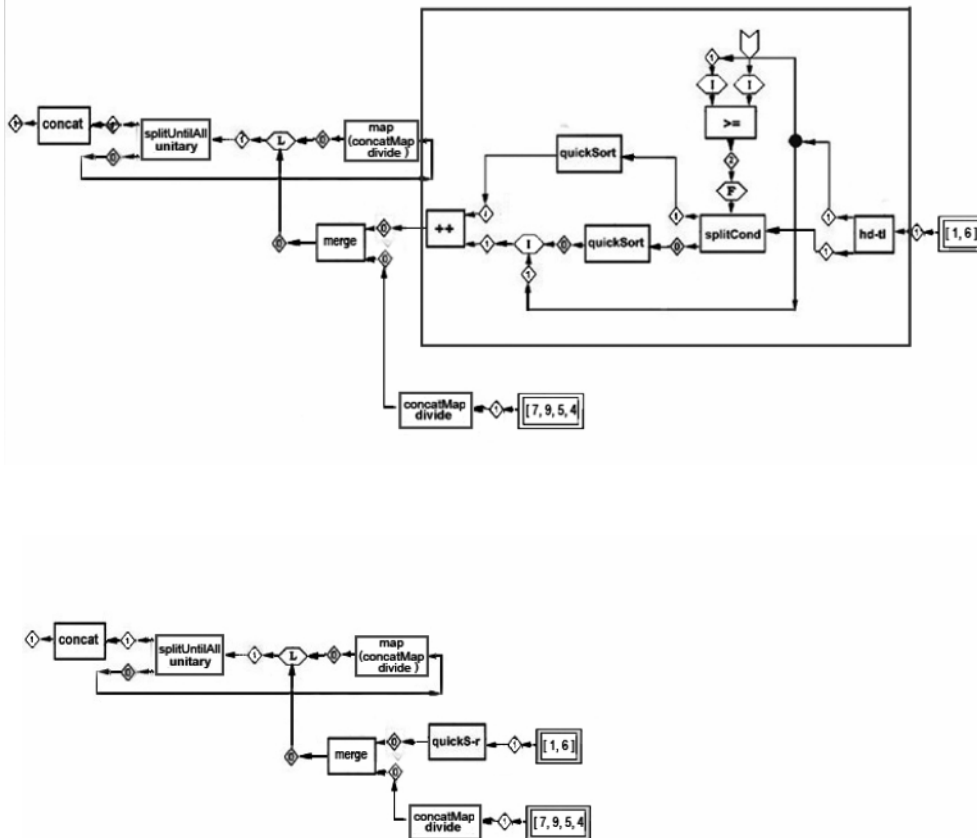


Figura 10. Compactación de una red mediante opciones de visualización

Una facilidad de gran utilidad para la puesta a punto es la capacidad de deshacer los pasos de ejecución, en el caso de NiMo ésta es particularmente útil ya que se puede corregir la red y retomar la ejecución desde ese punto directamente. Otro aspecto crítico en el ambiente es disponer de un editor que facilite la construcción de grafos, y además de las operaciones clásicas de este tipo de editores asista en el ajuste con las convenciones gráficas de redes NiMo y a la vez con las restricciones semánticas dadas por la tipificación fuerte del lenguaje, por ejemplo que no permita situar en un canal un elemento de tipo incompatible con su predecesor, o poner como argumento de **map** un proceso incompatible con su canal de entrada. Obviamente, el menú de procesos básicos y redes predefinidas agiliza enormemente la construcción de redes de usuario. El ambiente dispondrá además de facilidades para el manejo de librerías y proyectos, como así también de herramientas de conversión entre programas en NiMo Toons y NiMo-AGG, y traducción de ambos a Haskell y viceversa [7].

6. CONCLUSIONES

Se ha presentado un lenguaje gráfico inspirado en las redes de procesos como representación de programas funcionales con evaluación perezosa, y el ambiente para asistir en la construcción de programas en este estilo. Las características más importantes de este lenguaje fueron ilustradas mediante un ejemplo completo de desarrollo de una aplicación. Se han delineado las ideas fundamentales del diseño del ambiente de desarrollo y el estado actual de implementación.

Entre las características que hacen a NiMo particularmente atractivo para usuarios finales está el hecho de ser totalmente gráfico, y que los programas son grafos directamente ejecutables que se construyen a partir de nodos proceso predefinidos usando una variedad muy reducida de símbolos gráficos. Los procesos básicos son primitivas algorítmicas de un alto nivel de abstracción, adaptados a redes Data-flow y muy sencillos de comprender y combinar. El usuario puede describir su solución sin llegar a involucrarse en detalles de bajo nivel.

Pero quizás la característica más original de NiMo es que el usuario puede visualizar y controlar totalmente la ejecución en línea de su programa y ver el flujo de los datos en la medida en que se van generando o transformando. La ejecución puede hacerse en modo paso a paso, interrumpirse, cambiar o completar cualquier elemento y retomarse a partir de ese punto. Asimismo en la red se señalan los procesos activados en cada instante, y el usuario puede cambiar localmente el orden de activación por defecto, por ejemplo para hacer pruebas de subredes. Los modos de edición y ejecución se alternan sin tener que pasar por procesos de compilación ni que ello obligue a recomenzar la ejecución desde el principio, como sucede incluso en los lenguajes interactivos actuales. Todo esto es posible ya que el código (la red) es a la vez el programa y su estado.

Referencias

- [1] AGG home page, AGG Home page: <http://tfs.cs.tu-berlin.de/agg/>
- [2] Agilent Vee <http://www.home.agilent.com/>
- [3] Burnett M., Baker M.J., Bohus C., Carlson P., Yang S., y Van Zee P. *Scaling up visual Programming languages. Computer*, 28:45–54, 1995.
- [4] Clerici, S, Zoltan C. *A Graphic Functional-Dataflow Language (Extended Abstract)* [PROLE, Granada Spain September 2005](#)
- [5] Clerici, S., Zoltan C. *Página del Proyecto NiMo*. <http://www.lsi.upc.edu/~nimo/Project/>
- [6] Clerici S., Zoltan C. *A Graphic Functional-Dataflow Language*. Trends in Functional Programming, Volume 5. Editor Loidl H-W. Intellect 2006.
- [7] García San Julián, J. *Conversión de NiMo a Haskell y viceversa*. Proyecto Final de carrera FIB-UPC 2006.
- [8] Haskell home page <http://www.haskell.org/>
- [9] Kelso, J. *A Visual Programming Environment for Functional Languages*, Phd. Murdoch University 2002.
- [10] LabVIEW <http://www.ni.com/labview/>
- [11] Pestigiaco, G., *NiMo Toons, el ambiente de NiMo*, Tesis de grado UBA, convenio FIB-UPC-UBA. En curso 2006.
- [12] Ptolemy II *Página del proyecto* <http://ptolemy.berkeley.edu/ptolemyII/>
- [13] Reekie H., *Visual Haskell: A First Attempt*. Technical Report 94.5, Key Center for Advanced Computing Sciences,

Univ. Tech. Sydney, 1994.