MVC en O'Haskell

Germán E. Ruiz Federico Feller Pablo E. Martínez López

LIFIA, Facultad de Informática
Universidad Nacional de La Plata,
CC.11, 1900, La Plata, Argentina,
E-mail:{gruiz,ffeller,fidel}@lifia.info.unlp.edu.ar

Resumen

Tanto la programación orientada a objetos como la programación funcional son paradigmas muy estudiados que contribuyen a la construcción de mejor software. Aquí se presenta un estudio sobre el lenguaje O'Haskell, el cual combina elementos de ambos paradigmas para brindar una herramienta robusta. El propósito de este artículo es analizar al lenguaje desde un punto de vista práctico mediante ejemplos. Se presentan conclusiones sobre la verdadera utilidad del mismo para ser usado en el desarrollo cotidiano de aplicaciones.

Palabras Clave: programación orientada a objetos, programación funcional, O'Haskell, combinación de paradigmas.

1. Introducción

Hoy en día la programación orientada a objetos [Boo94, AC96] ha cobrado gran importancia debido a las facilidades que brinda para el diseño de sistemas complejos y con necesidades de cambios permanentes. Dualmente la programación funcional [BW88, Bir98] también resulta significativa ya que permite razonar sobre los programas de forma matemática, ayudando a la modularización de los mismos. Sin embargo, sería interesante contar con un lenguaje que combine las ventajas de ambos paradigmas.

Este trabajo describe la investigación realizada en el LIFIA de la facultad de Informática de la UNLP acerca de la factibilidad de usar el lenguaje O'Haskell con diseños orientados a objetos. El objetivo principal fue la de adquirir experiencia en el uso del lenguaje O'Haskell, con la idea de establecer las condiciones para poder desarrollar algunos puntos críticos del mismo. Se intentó usar O'Haskell como un lenguaje alternativo para la implementación de programas diseñados según las bases del paradigma de orientación a objetos. Esto resultaba en un desafío ya que, si bien el lenguaje estudiado cuenta con herramientas de orientación a objetos básicas como las nociones de identidad y estado, carece de otras características como los conceptos de clase y de herencia.

Como metodología de trabajo se decidió la elección de dos casos de uso que poseyeran características representativas del problema a abordar, su estudio, diseño orientado a objetos, e implementación utilizando el lenguaje, para luego comparar dichas implementaciones en O'Haskell con otras similares en otros lenguajes orientados a objetos (como Smalltalk o Java). Se pensó que la comparación con otros lenguajes serviría para tomar dimensión de la verdadera utilidad del nuevo lenguaje al momento de desarrollar aplicaciones de cierta complejidad.

Aquí se presentan los resultados de la primera parte de la investigación que consiste en la implementación en O'Haskell, quedando como futuro trabajo la implementación y comparación de las implementaciones en otros lenguajes.

En primer lugar se introduce O'Haskell con sus características más importantes. Luego se detalla la técnica MVC y se describe su implementación en el lenguaje estudiado. Las siguientes secciones explican los ejemplos elegidos y muestran algunas consideraciones sobre su implementación. Finalmente se presentan conclusiones sobre O'Haskell.

2. O'Haskell

El lenguaje O'Haskell [Nor99] surgió como una necesidad de contar con un lenguaje de programación que se adapte a los requisitos actuales para la construcción de software. Para su creación se extendió del lenguaje funcional Haskell [PJH99, Jon03] mediante la incorporación de tres características básicas:

- la noción de objeto reactivo, la cual es una unificación de las nociones de objetos y concurrencia.
- una capa implementada en base a mónadas [Wad95] para manejar los efectos computacionales de los objetos, la cual permite separar de manera bien definida los objetos con estado de los valores.
- un sistema polimórfico de tipos que soporta registros y subtipos junto a un algoritmo de inferencia parcial.

Los objetos reactivos encapsulan un estado y funcionan como servidores que se ejecutan concurrentemente con otros objetos de una misma aplicación, admitiendo el pedido de requerimientos a través del envío de mensajes asincrónicos. Los métodos que ejecutan los objetos reactivos están siempre disponibles y el orden en que se ejecutan respeta el orden en que fueron enviados los mensajes. Además la ejecución de un método no puede esperar más que por la invocación a algún otro método, por lo que la ausencia de deadlock y de loops infinitos en la ejecución concurrente de los objetos está garantizada. Puede que sea necesario obtener una respuesta inmediatamente después a la invocación de un método, por lo que O'Haskell también provee métodos sincrónicos para ser usados en estos casos. Una característica notable del lenguaje es que la propiedad de sincronía/asincronía de un método es capturada por el sistema de tipos.

En O'Haskell, además de las construcciones estándar de Haskell, es posible declarar nuevos tipos mediante la construcción *struct*, que permite definir registros en general con cualquier combinación de tipos y en particular permiten definir interfases de objetos especificando la signatura de los métodos. Los nombres usados para definir un *struct* son únicos, por lo que el tipo de la interfase requerida para un objeto puede ser inferida analizando las invocaciones a métodos del mismo.

```
struct Example =
  method1 :: Int -> Action
  method2 :: (Int, Int) -> Request Int
```

El tipo `Action` corresponde a un acción asincrónica, mientras que para expresar un pedido sincrónico que retorna un valor de tipo `a` se utiliza el tipo `Request a`.

Es posible especificar herencia de interfases mediante el uso de subtyping.

```
struct Example2 < Example =
  method3 :: Action
```

En el ejemplo, la interfase `Example2` extiende la interfase `Example`, por lo que un objeto que la implemente deberá contener métodos para responder a los mensajes `method1`, `method2` y `method3`. Además siempre que se requiera un objeto `Example` se podrá usar un objeto `Example2`.

La forma de crear objetos es a través de la instanciación de *templates*. En su declaración se define la estructura del estado de los objetos (compuesta de valores y otros objetos) y se especifican los métodos correspondientes a la interfase que implementa. Por ejemplo,

```
myExample = template
               val1 := 0
               obj1 <- otherExample   -- Se instancia un objeto colaborador
                                      -- al instanciar este template
            in struct
                 method1 x     = action
                                    val1 := x
                 method2 (x,y) = request
                                    val1 := x + y
                                    return val1
```

declara el template `myExample` cuyas instancias serán objetos que tengan en su estado un entero y un objeto instancia del template `otherExample`. Además estas instancias podrán responder a los mensajes `method1` y `method2`.

La siguiente expresión instancia un objeto del template `myExample`, luego ejecuta el método `method2` obteniendo su resultado en la variable `res` y por último ejecuta el metodo `method1`:

```
do m <- myExample
   res <- m.method2 (1,2)
   m.method1 res
```

El constructor `do` permite crear una expresión que represente una secuencia de comandos los cuales son ejecutados según el orden de aparición. Esto es una adición sintáctica permitida por las construcciones monádicas ([PJH99], sección 3.14).

Una característica esencial que deben tener los lenguajes que soportan objetos es el encapsulamiento. O'Haskell logra este requerimiento a través de reglas de scoping y de tipado que se verifican estáticamente. Se garantiza que las variables de estado de un objeto solo sean accedidas por el propio objeto a través de sus métodos.

El lenguaje también cuenta con una semántica formal bien definida lo que constituye una característica muy importante para poder razonar sobre los programas.

Por sus características, O'Haskell no resulta ser un lenguaje orientado a objetos más, ya que difiere drásticamente en la manera en que se consideran los objetos. La principal diferencia se encuentra en la posibilidad de envío de mensajes asincrónicos que permite que un objeto siempre esté disponible para recibir mensajes, sin la necesidad de que el emisor deba esperar. Otra diferencia se encuentra en que O'Haskell no sigue la filosofía de *todo es un objeto*, sino que hace uso de estructuras matemáticas como pares, listas, registros, funciones y tipos algebraicos cuyos valores no son considerados objetos ya que se piensa que no tienen las propiedades de los mismos (como estado e identidad). Por último, O'Haskell no soporta ningún mecanismo de herencia o de delegación automática de mensajes.

3. Model-View-Controller

Dado que los problemas seleccionados para ser implementados requerirían la construcción de aplicaciones con interfases gráficas, se optó por usar como estrategia de implementación al MVC (Model-View-Controller) [GHJV95], del paradigma orientado a objetos, la cual se basa en separar la aplicación en tres componentes bien diferenciables:

1. el modelo, el cual incluye a los objetos que se encargan de cumplir el objetivo del programa,
2. la vista, la cual está formada por las clases que proveen una forma de mostrar la información generada en pantalla, y por último,
3. el controlador, el cual se compone de los objetos que obtienen el input de los usuarios y definen la forma en que la interfase reacciona ante tales entradas.

Con esta separación de los objetos, la vista y el modelo de la aplicación quedan bien diferenciados. Dado que la vista debe reflejar en todo momento el estado actual del modelo, se utiliza un protocolo de subscripción/notificación, en el cual la vista se subscribe para recibir notificaciones por parte del modelo de los cambios de estado ocurridos en el mismo; al recibir tales notificaciones, la vista se actualiza quedando sincronizada con el modelo.

Este estilo de diseño permite obtener programas con una mayor facilidad al cambio y a las mejoras, ya que no fuerza la solución del problema a depender de una única interfase con el usuario.

3.1. Implementación del MVC en O'Haskell

El lenguaje O'Haskell y, en particular, la implementación usada, O'Hugs, carece de facilidades para realizar aplicaciones usando la estrategia del MVC. Dado que la intención era la de usar MVC en los dos casos de estudio, se pensó en implementar un diseño genérico para soportar MVC que pueda ser reusado en el futuro. A la descripción básica del MVC anterior se le agregó la posibilidad de que la vista pueda diferenciar cambios en distintos aspectos del modelo. O'Hugs provee librerías para manejar vistas de ventanas, por lo que el esfuerzo se concentró en implementar un mecanismo de subscripción/notificación que permita mantener sincronizados al modelo con sus vistas. Se pensó en el modelo siguiente: todo objeto modelo que esté pensado para tener dependientes debería cumplir con una interfase (`Model`) que permita tanto la subscripción a las notificaciones de los cambios.

```
struct DependencyActions a =
  addDependent :: (Aspect, Maybe a -> Action) -> Action

struct Model a < DependencyActions a =
  changed :: Aspect -> Maybe a -> Action
```

Cuando un objeto se subscribe mediante el método `addDependent` envía como parámetro un aspecto y una función que retorna la acción a realizar (las acciones se pueden tratar como valores) cuando se produzca la actualización del aspecto especificado. El parámetro de la función representa información relacionada con la actualización (que puede no estar presente en caso de que no sea necesario). Cuando el objeto modelo se modifica, notifica los cambios ejecutando las acciones de los dependientes que correspondan de acuerdo al aspecto modificado. Las acciones necesarias para mantener las subscripciones, como así también aquellas realizadas para manejar

las notificaciones, son comunes para todos los objetos modelo, por lo que se pensó en implementarlas en una sola clase o template `DependencyImplementor`. Por esta razón se optó por separar la interfase `DependencyActions`:

```
struct DependencyImplementor a < DependencyActions a =
  update :: Aspect -> Maybe a -> Action

dependencyImplementor =
    template
       dependents := [] :: [(Aspect, Maybe a -> Action)]
       i:= 0
     in struct
          addDependent n = action
                           let nuevo = n:dependents
                           dependents := nuevo
          update aspect p = action
                           doList (map snd $
                                   filter ((==aspect) . fst) $
                                   dependents
                                  ) p
                           done

doList :: Monad m => [a -> m b] -> a -> m ()
doList []     p = done
doList (f:fs) p = do f p; doList fs p
```

Es importante observar cómo se pueden componer acciones de manera puramente funcional antes de invocarlas a través del método `update`; esto es una ventaja provista por el poder resultante de la combinación entre paradigmas que O'Haskell provee.

La falta de herencia o de algún otro mecanismo de delegación implícito forzó a que en cada objeto modelo se deban agregar los métodos que delegan el comportamiento a un objeto `DependencyImplementor`:

```
struct MyModel a < Model a =
  method1 :: Action
  ...

mymodel = template
            di <- dependencyImplementor
            ...
          in struct
              method1 = action ...
                ...
              changed asp p  = action di.update asp p
              addDependent d = action di.addDependent d
```

El lenguaje ofrece una facilidad sintáctica para completar la definición de un *struct* automáticamente. Por ejemplo, si tenemos declarado el registro

```
struct A = x, y :: Int
```

entonces la expresón `struct x = 4 ; ..A` equivale a `struct x = 4 ; y = y`, donde la segunda aparición de la variable `y` debe estar ligada a algún binder dentro del alcance permitido (la ocurrencia izquierda expresa la variable de estado, mientras que la derecha expresa una variable funcional.) Se pensó que esta ventaja podría ser utilizada para automatizar la escritura de los métodos que simplemente delegan el comportamiento. Sin embargo la necesidad de acceder al `DependencyImplementor` que forma parte del template que sirve de modelo hace imposible la escritura de tales métodos en otro lugar que no sea dentro del scope del template en cuestión. Esta restricción impide escribir una implementación común para los métodos que delegan que puedan usar todos los objetos modelos.

4. Casos de Estudio

Como se mencionó anteriormente, se han elegido dos problemas a implementar; su elección se fundamenta en el hecho de que fueron considerados problemas que poseían las características básicas que permitiesen hacer un diseño orientado a objetos completo no trivial.

4.1. Mines

La elección del juego de Buscaminas se basa en que el mismo combina el tratamiento lógico propio de un juego con un uso fundamental de la interfase gráfica.

Se pensó en un diseño simple para este juego, en donde el modelo es un objeto `MinesGame` con un colaborador que representa la matriz de celdas. Este último tiene el comportamiento básico que permite acceder a las diferentes posiciones de la matriz. Cada celda de la matriz es a su vez un objeto cuyo estado permite saber si tiene una mina, cuántas minas tiene alrededor esa celda (en caso que tenga). En el estado del `MinesGame` se guardan datos como estado de juego (jugando o perdido) y cantidad de minas encontradas. El comportamiento básico de un objeto `MinesGame` es la de responder a un método marcar, que recibe una posición de la matriz, y responde cambiando, quizás, el estado de la celda en esa posición. La interfase se representa con otro objeto, que crea un `MinesGame` de quien es dependiente. Cada vez que una celda del juego cambia de estado, la interfase es notificada para actualizar su vista según lo visto en MVC.

En principio, el diseño descripto se adapta a un lenguaje orientado a objetos tradicional. Una optimización posible gracias a la combinación de la programación funcional y la orientación a objetos que provee O'Haskell es la de usar donde fuese posible valores sin estado en lugar de objetos. De esta forma los objetos celdas pasarían a ser directamente datatypes, y el objeto matriz se reemplazaría usando Arrays polimórficos (que se instancian con el tipo de las celdas) que ya vienen incorporados en el lenguaje. Por lo tanto, sólo permanecerían como objetos el modelo y la interfase. La implementación del método marcar consiste en hacer pattern-matching sobre el valor del array en la posición recibida para saber qué acción realizar.

4.2. Red Neuronal

Como segundo ejemplo se implementó una red neuronal para el reconocimiento de figuras, siguiendo el modelo de Hopfield [Hop82]. El problema consiste en realizar un programa que maneje una red neuronal que pueda "memorizar"patrones con el fin de reconocer nuevas entradas, asociándolas con los patrones ya aprendidos. Cada patrón consiste en un arreglo binario (que

puede ser visto como una matriz para representar caracteres alfabéticos) y para el aprendizaje se vincula a cada neurona con todas las demás.

Dado que la intención es reconocer figuras (e.g. letras del alfabeto) que son patrones poco relacionados, el aprendizaje se hace a través de la regla de Cooper-Heb:

Definición.

Sean $p_1 \dots p_m$ arreglos de patrones de aprendizaje, y sea w_i la matriz de vínculos entre la neurona i-ésima y el resto, se calculan los vínculos de la siguiente forma:

$$\begin{cases} w_{ij} = w_{ji} = \sum_{k=1}^{m} (2 * p_i^m - 1) * (2 * p_j^m - 1) & si\ i \neq j \\ 0 & si\ i = j \end{cases}$$

Una vez hecho el aprendizaje, se pueden cargar patrones para que sean reconocidos por la red. Para ello se utiliza el algoritmo de Hopfield que consiste en primero asignar a cada unidad el valor correspondiente al patrón de entrada. Luego se ejecuta reiteradamente un ciclo que consiste en elegir una posición al azar i y calcular su salida de la siguiente manera:

$$\begin{cases} out_i = 1 & \text{si } net_i > 0 \\ out_i = 0 & \text{si } net_i \leq 0 \end{cases} \quad \text{donde } net_j = \sum_i w_{ij} * out_i$$

El algoritmo utilizado itera indefinidamente, obteniendo el resultado por convergencia. Por esto, se fija una condición de finalización consistente en ejecutar $N * I$ iteraciones, donde N es representa la cantidad de entradas del arreglo del patrón e I es un parámetro indicado por el usuario.

Para la implementación de este problema se puede pensar en un diseño básico, con un objeto principal que sirva de modelo y que conserve en su estado los vínculos entre las neuronas que simbolizan la memoria de aprendizaje. Este objeto deberá poder aprender un nuevo patrón mediante la actualización de los vínculos según la fórmula vista y deberá ser capaz de reconocer una nueva muestra ejecutando el ciclo antes descripto. El resto del modelo se completa con objetos que modelen las neuronas, los patrones y las muestras y los vínculos entre las neuronas.

Aquí nuevamente puede hacerse uso del potencial de O'Haskell para lograr una optimización mediante el uso de valores sin estado. De hecho, en el programa implementado hizo falta sólo un objeto que represente al modelo y que implemente el comportamiento necesario para el aprendizaje y el reconocimiento. Para el resto de los objetos del modelo se usaron las facilidades del lenguaje como Arrays y tipos algebraicos. Respecto a la visualización del programa, se implementaron dos interfases distintas (una modo texto y otra modo ventana) aprovechando los mecanismos de MVC previamente implementados.

5. Conclusiones

Luego de haber adquirido experiencia trabajando sobre el lenguaje O'Haskell, se pueden concluir algunas cuestiones sobre el mismo.

En primer lugar se observó que O'Haskell es un lenguaje robusto, el cual puede ser perfectamente usado para implementar cualquier tipo de aplicación. Puede argumentarse que la falta de pureza en el lenguaje sea un inconveniente; sin embargo la combinación entre elementos del paradigma funcional y objetos resulta en una herramienta poderosa y útil. El máximo aprovechamiento de esta característica surgió al momento de combinar los objetos con los valores (como listas, funciones y tipos algebraicos), lo que permitió hacer optimizaciones sobre

los diseños, usando valores sin estados en lugar de objetos, cuando estos últimos no tuviesen comportamiento definido.

También vale la pena mencionar y hacer hincapié en el sistema de tipos que provee O'Haskell, el cual brinda seguridad al escribir los programas gracias a su algoritmo de inferencia y permite flexibilidad gracias al subtyping y los tipos polimórficos.

Uno de los puntos negativos encontrados que ya se ha mencionado es la falta de herencia o de algún otro mecanismo de autodelegación que permita hacer reuso de código. Al principio resulta un impacto muy fuerte para el usuario acostumbrado a otros lenguajes orientados a objetos; sin embargo, diferentes facilidades del lenguaje como la posibilidad de manejar funciones como valores permiten encontrar soluciones alternativas a la herencia, aunque quizás con la desventaja de perder posibilidad de reuso. La implementación del MVC resultó un claro ejemplo de cómo la carencia de ciertas facilidades primitivas necesarias para trasladar un diseño en objetos, pueden ser reemplazadas mediante la utilización de las características poderosas resultantes de la combinación de un lenguaje funcional y uno que soporte objetos reactivos.

Referencias

[AC96] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1996.

[Bir98] Richard Bird. *Introduction to Functional Programming Using Haskell*. Prentice Hall, 1998.

[Boo94] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, Redwood City, CA, 2nd edition, 1994.

[BW88] R. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice Hall, 1988.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, 1995.

[Hop82] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79:2554–2558, 1982.

[Jon03] Simon Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.

[Nor99] Johan Nordlander. *Reactive Objects and Functional Programming*. PhD thesis, Chalmers University of Technology, 1999.
URL: `http://www.cs.chalmers.se/~nordland/ohaskell/`.

[PJH99] Simon Peyton Jones and John Hughes (editors). Haskell 98: A non-strict, purely functional language, February 1999.
URL: `http://www.haskell.org/onlinereport/`.

[Wad95] P. Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming: 1st International Spring School on Advanced Functional Programming Techniques*, pages 24–52. Springer, Berlin, Heidelberg, 1995.

A. Código de la red neuronal

Anexamos parte del código que implementa la red neuronal con el fin de que puedan observarse algunos detalles del trabajo.

```
--=======================================================
-- Módulo del modelo
--=======================================================
module Hopfield where
import Random
import Array
import DependencyImplementor

type State = Array Int Int
type Links = Array Int State

struct NeuronalNet a < Model a =
    learnNet   :: State -> Action    -- La red es "educada" con una nueva
                                     -- imagen o estado de memoria
    stepNet    :: Action             -- Ejecuta un paso del loop de evolución
    runNet     :: Action             -- Arranca la ejecucion
    stopNet    :: Action             -- Detiene la ejecucion
    loadState  :: State -> Action    -- Carga un nuevo estado
    getNeurons :: Request State      -- Retorna el conjunto de neuronas
    getNeuron  :: Int -> Request Int -- Retorna el valor de una neurona
    getLinks   :: Request Links      -- Retorna la coleccion de vinculos

neuronalNet :: Int -> Template (NeuronalNet (Int, Int))
neuronalNet x =
   template
      neurons := array (0,99) 0                -- Neuronas de la red
      links   := array (0,99) (array (0,99) 0) -- vínculos entre las neuronas
      rand <- uniformGenerator x               -- Generador aleatorio.
      di   <- dependencyImplementor
   in let loadState newState = action neurons:= newState
          getLinks   = request return links
          learnNet e = action forall i <- [0..99] do
                                links!i := actLinks (links!i,i) e
              where actLinks (ys,x) e = listArray (0,99) list
                    list = [ if x == y then 0
                                       else (ys!y)+(2*(e!x)-1)*(2*(e!y)-1)
                           | y <- [0..99] ]
```

```
          runNet  = action changed "run" Nothing; done
          stopNet = action changed "stop" Nothing; done
          stepNet = action -- Un paso en el loop de "reconocimiento"
                           rnd <- rand.next
                           let
                              acc = acumulate 0 99 neurons  (links ! pos)
                              val = if acc > 0 then 1 else 0
                              pos = toInt $ truncate (rnd * 100)
                           neurons!pos:=val
                           changed  "step" (Just (pos,val))
          getNeurons     = request return neurons
          getNeuron pos  = request return $ neurons ! pos
          changed asp  p = action di.update asp p
          addDependent d = action di.addDependent d

       in struct ..NeuronalNet

---- Calcula la sumatoria de los vinculos para una neurona en particular
acumulate :: Int -> Int -> Array Int Int -> Array Int Int -> Int
acumulate acc pos neu vin =
   if pos == -1
    then acc
    else acumulate (neu ! pos * (vin!pos) + acc) (pos - 1) neu vin

--=======================================================
-- Módulo de la vista (incompleto)
--=======================================================
module Hopwin where
import Tk
import Hopfield

struct Grid = initialize    :: Request ()
              changeEdition :: Action
              setMem        :: Action
              getMem        :: Action
              clean         :: Action
              learn         :: Action
              paintPoint    :: (Int, Int) -> Action
              load          :: State -> Action

grid :: Canvas -> NeuronalNet (Int, Int) -> Template Grid
grid canv nnet = template
                   canvas := canv           -- Canvas donde se dibujará
                   edit   := False          -- Estado de edición
                   state  := array (0,99) 0 -- Estado del widget.
                 in let <...omitido...>
                     in struct ..Grid
```

```
-- Ventana principal
main tk = do win <- tk.window [Title "Neuronal Net in O'Hugs!"]
             cv <- win.canvas [Background white, Width 300, Height 300]
             (_,seed) <- tk.timeOfDay
             nnet <- neuronalNet seed    -- Red Neuronal
             gd <- grid cv nnet
             gd.initialize
             sch <- tk.periodic 15 nnet.stepNet
             nnet.addDependent ("run", \_ -> sch.start)
             nnet.addDependent ("stop",\_ -> sch.stop)
             nnet.addDependent ("step",\x -> gd.paintPoint (fromJust x))
             <...omitido...>
```