Modelización conceptual declarativa en un lenguaje lógico-funcional avanzado*

S. Escobar M. Alpuente S. Lucas

{sescobar,alpuente,slucas}@dsic.upv.es

30 de diciembre de 1998

Resumen

Los lenguajes de programación declarativos proporcionan un estilo de programación potente que reduce el tiempo de desarrollo de programas y facilita su mantenimiento en comparación con los lenguajes imperativos clásicos. Estas propiedades, junto con el uso de técnicas de optimización han facilitado la aplicación industrial de los lenguajes declarativos en los últimos años. En este trabajo resolvemos un problema de modelización real en el lenguaje lógico-funcional integrado Curry, una propuesta moderna de lenguaje integrado auspiciada por una iniciativa internacional que aspira a convertir el lenguaje en un estándar para el área. La aplicación abordada consiste en la *Modelización Conceptual del Ecosistema de la Cuenca Hidrográfica del río Segura*, una aplicación de interés tanto ecológico como educativo.

1 Introducción

En este informe resolvemos un problema de modelización real en el lenguaje lógico-funcional integrado Curry [HAK+98]. El problema considerado es la *Modelización del Ecosistema de la Cuenca Hidrográfica del río Segura*, una aplicación de interés tanto ecológico como educativo que se plantea con la doble finalidad de ayudar a controlar la calidad del medio y generar actitudes y valores que favorezcan una mayor armonía entre el sistema y su entorno. La especificación del modelo, siendo ejecutable, puede entenderse como el prototipo operacional de una aplicación, que se pretende explotar con fines educativos. El sistema facilita el desarrollo de experiencias de simulación que puedan colaborar al surgimiento de actitudes

*Este trabajo ha sido desarrollado parcialmente bajo el proyecto CICYT TIC 98-0445-C03-01, la Acción Integrada hispano-alemana HA1997-0073 y el convenio de colaboración en Formación de Empleo suscrito por la Universidad Politécnica de Valencia y la empresa Consoft S.A.

de concienciación y promover que sociedad y administraciones públicas asuman las medidas necesarias para la solución definitiva de los problemas del río.

Este trabajo apunta a un segundo objetivo importante, de índole más técnica, consistente en comprobar la viabilidad del lenguaje lógico–funcional integrado Curry para abordar una aplicación industrial. Cabe destacar que ésta es la primera experiencia documentada de desarrollo de una aplicación industrial en un lenguaje lógico-funcional integrado. El trabajo proporciona también un marco de trabajo útil para el desarrollo de futuras aplicaciones de interés para otras empresas y para futuras extensiones de la aplicación desarrollada, que va a ser utilizada para tareas de estudio del medio ambiente considerado (influencia de las actividades humanas sobre la calidad del medio ambiente, relaciones, interdependencias, etc.). Se espera que el prototipo generado ayude a promover la sensibilización social hacia dicho entorno y sus problemas.

La estructura de esta memoria es como sigue. En la Sección 2 presentamos los preliminares del informe introduciendo el lenguaje Curry y sus características más relevantes. En la Sección 3 revisamos las ideas fundamentales de la modelización conceptual utilizando técnicas de programación declarativa. La Sección 4 plantea el modelo conceptual de la cuenca hidrográfica del río Segura desarrollado en Curry. Por último, la Sección 5 muestra las conclusiones obtenidas tras la realización de este trabajo.

2 El lenguaje Curry

Curry es un lenguaje lógico funcional avanzado resultado de los esfuerzos coordinados de un grupo de trabajo multinacional que aspira a convertir el lenguaje Curry en un estándar para la programación lógico funcional.

El lenguaje de programación Curry [HKMN95, HAK+98] incorpora un mecanismo operacional híbrido que permite integrar tanto el estrechamiento (necesario) como la residuación y considerar ambos como casos particulares de dicho mecanismo [Han97b, HAK+98]. De esta forma se consigue reunir todas las características de la programación lógica, funcional, de resolución de restricciones y concurrente de una forma clara, sencilla y completa.

Curry combina en un mismo sistema las mejores características de la programación funcional (expresiones anidadas, evaluación perezosa, funciones de orden superior), de la programación lógica (variables lógicas, estructuras de datos parciales, búsqueda incorporada) y de la programación concurrente (evaluación concurrente de restricciones con sincronización sobre las variables lógicas).

Informalmente, dado que la sintaxis de Curry es muy semejante a la de Haskell, la utilización de Curry por parte de un programador funcional es bastante sencilla, mientras que un programador lógico debe preocuparse de expresar los predicados como funciones que devuelven una restricción.

Ejemplo 2.1 *Dada la siguiente cláusula de Horn:*

```
mujer(ana) ← .
```

su codificación en Curry es:

```
mujer ana = success
```

siendo `ana` *un constructor perteneciente a algún tipo abstracto de datos definido por el usuario.*

Ejemplo 2.2 *La siguiente cláusula de Horn:*

```
hermanas(X,Y) ← mujer(X), mujer(Y),
                padres(X,P,M), padres(Y,P,M).
```

se codificada de la siguiente forma:

```
hermanas x y = (mujer x) & (mujer y) &
               (padres x _p _m) & (padres y _p _m)
```

que corresponde a una función expresada usando restricciones.

Los programas en Curry están constituidos por declaraciones de tipos de datos y ecuaciones de definición de funciones. Pasamos a describir con más detalle la sintaxis de dichos componentes.

2.1 Tipos

El sistema de tipos del lenguaje Curry sigue las directrices de los sistemas de tipos (algebraicos) empleados comúnmente en los lenguajes funcionales [Dav92, PvE93]. En los tipos de datos algebraicos, los *valores* se representan mediante términos construidos únicamente a partir de símbolos constructores de datos, respetando la aridad de cada símbolo. Por eso, los valores corresponden con *términos constructores*.

En una misma declaración se especifican los *constructores de tipo* y los *constructores de datos*, que describen los *valores* que constituyen el tipo de datos *algebraico*. Las declaraciones de tipos de datos son de la forma:

$$\texttt{data}\ \ T\ \tau_1\ \cdots\ \tau_n = C_1\ \tau_{11} \cdots \tau_{1n_1}\ \texttt{|}\ \cdots\ \texttt{|}\ C_k\ \tau_{k1} \cdots \tau_{kn_k}$$

donde T es un nuevo constructor de tipo de datos de aridad n (polimórfico si $n > 0$) cuyos constructores de datos son $C_1, \ldots, C_k$. La aridad de un constructor de datos $C_i, 1 \leq i \leq k$ es n_i.

Por ejemplo, la declaración:

```
data TreeInt = Leaf Int | Branch TreeInt TreeInt
```

establece un nuevo tipo de datos `TreeInt` que consta de un número infinito de valores definido recursivamente con la ayuda de los símbolos constructores de dato `Leaf` (unario) y `Branch` (binario) que toman como argumentos un número entero y un par de árboles `TreeInt` respectivamente.

La declaración:

```
data List t = Nil | Cons t (List t)
```

donde `List` es el constructor (unario) de un tipo polimórfico cuyos valores se construyen empleando el constructor constante `Nil` y el constructor binario `Cons` (aquí `t` es una variable de tipo).

Otro ejemplo de declaración de tipo de datos algebraicos es la siguiente:

```
data Nat = Cero | S Nat
```

Observar que `Bool`, un tipo estándar de Curry, puede ser definido como un tipo de usuario sin ningún problema:

```
data Bool = True | False
```

Notación 2.3 *Habitualmente, las listas se representan, de una forma abreviada, empleando los corchetes '*`[`*' y '*`]`*' y las equivalencias* `[ ]` $\equiv$ `Nil` *y* `[x|xs]` $\equiv$ `Cons x xs`*. También suele utilizarse el operador infijo '*`:`*' que es completamente equivalente a* `Cons`*, es decir:* `Cons x xs` $\equiv$ `x:xs`*. De esta forma, la lista* `Cons 0 (Cons 1 (Cons 2 Nil))` *se representa como* `[0,1,2]` *y también como* `0:1:2:[]`*.*

Notación 2.4 *Para mejorar la legibilidad de los programas y expresiones que involucren el tipo* `Nat` *emplearemos* `0` *en lugar del constructor* `Cero` *que es el constructor que realmente corresponde según las normas de asociación de identificadores a constructores de datos.*

2.2 Definición de funciones

En Curry, las funciones se describen mediante *ecuaciones*, apoyándose en la existencia de ciertas funciones primitivas, de comportamiento conocido. De forma análoga a la existencia de tipos primitivos (como `Int`, `Bool`, etc.) existen *funciones primitivas* (o *básicas*) cuya evaluación se supone soportada por el entorno donde se ejecutan los programas. Ejemplos típicos son las funciones `+`, `*`, `div`, etc.

Las funciones de usuario se definen por medio de ecuaciones de diversos modos.

Ecuaciones incondicionales

Las declaraciones de funciones más simples son ecuaciones de la forma:

$$f\ p_1 \cdots p_n = e$$

donde $p_1, \ldots, p_n$ son *patrones* (términos constructores con variables) que verifican la restricción de que ninguna variable ocurre más de una vez en la n-tupla $p_1, \ldots, p_n$ y donde la parte derecha, e, de la ecuación es una expresión cualquiera. La regla de programa $l \to r$, de acuerdo a la sintaxis de los sistemas de reescritura (SRT), se representa en Curry de la forma l = r según el esquema anterior.

Ejemplo 2.5 *La función* `append` *puede definirse mediante las siguientes ecuaciones:*

```
append []     ys = ys
append (x:xs) ys = x : append xs ys
```

observemos los patrones `[]` *y* `(x:xs)` *de tipo lista, utilizados para distinguir el comportamiento de la función según los distintos valores que puede adoptar el primer argumento.*

Ecuaciones condicionales

Podemos definir funciones sujetas a ciertas restricciones mediante ecuaciones condicionales de la forma:

$f\ p_1\ \cdots\ p_n$ | c = e

donde c es la restricción a cumplir. Las restricciones se expresan como conjunciones de ecuaciones t_1=:=u_1 & t_2=:=u_2 & $\cdots$ & t_n=:=u_n. Una regla de programa condicional $l \ \to \ r \ \Leftarrow \ c$, de acuerdo a la sintaxis de los sistemas de reescritura condicionales (SRTC) [DJ90, Kap87], se representa en Curry de la forma l | c = r según el esquema anterior.

Ejemplo 2.6 *La función* `ismother` *se define mediante una ecuación condicional como sigue:*

```
mother John = Christine
mother Mary = Christine
mother Christine = Alice

ismother x y | mother x =:= y = True
```

Otra forma de definir funciones sujetas a restricciones es:

$f\ p_1\ \cdots\ p_n$ | b = e

donde b es una expresión booleana. Podemos expresar este tipo de restricción en el formato anterior si se reescribe como:

$f\ p_1\ \cdots\ p_n$ | b =:= True = e

Ejemplo 2.7 *Un ejemplo de definición de función que emplea una expresión booleana para restringir la definición es:*

```
isgrandmother x y | ismother (mother x) y = True
```

Ecuaciones condicionales con múltiples condiciones

La inclusión de múltiples partes condicionales en una ecuación condicional es una característica típicamente funcional [PJ87], que facilita la elección de una expresión según la condición cumplida. En Curry dicha facilidad merece una distinción dependiendo de si las condiciones son expresiones booleanas o restricciones [HAK$^+$98].

De esta forma, una ecuación con parte condicional múltiple cuyas condiciones sean expresiones booleanas, será interpretada (visto desde una aproximación funcional pura) como una expresión `if then else` anidada.

$$\begin{array}{lllll} l \mid b_1 = r_1 & & l = & \texttt{if } b_1 \texttt{ then } r_1 \texttt{ else} \\ \vdots & \Rightarrow & & \vdots \\ \mid b_n = r_n & & & \texttt{if } b_n \texttt{ then } r_n \texttt{ else undefined} \end{array}$$

La función `undefined` se define como *evaluable a fallo* (distinto del término *no evaluable* en el contexto del modelo computacional de Curry).

En cambio, una ecuación con parte condicional múltiple cuyas condiciones sean restricciones, será interpretada como un conjunto de ecuaciones condicionales con parte condicional simple que poseen la misma parte izquierda.

$$\begin{array}{lll} l \mid c_1 = r_1 & & l \mid c_1 = r_1 \\ \vdots & \Rightarrow & \vdots \\ \mid c_n = r_n & & l \mid c_n = r_n \end{array}$$

Especificación del tipo de una función

Se permite especificar el tipo de una función n-aria mediante expresiones de tipo de la forma:

$$f :: \tau_1 \texttt{ -> } \tau_2 \texttt{ -> } \cdots \texttt{ -> } \tau_n \texttt{ -> } \tau$$

donde τ es el tipo resultante de la función y $\tau_1, \ldots, \tau_n$ son los tipos de cada uno de los n argumentos de f.

Anotaciones de evaluación para una función

Es posible especificar cómo se evaluarán las llamadas a una función f mediante anotaciones de la forma:

f `eval` α

donde α es `flex` o `rigid` indicando si se permite o no instanciar las variables de las subexpresiones situadas bajo el símbolo de función f.

Si no se indica ninguna anotación para un símbolo de función definido, se adopta un criterio por defecto. Éste consiste en suponer el modo `flex` para todas aquellas funciones que devuelven una restricción y el modo `rigid` para el resto de funciones. De esta manera, se otorga un carácter eminentemente lógico a los símbolos de funciones que devuelvan una restricción (ya que la resolución de restricciones implica tratar las ocurrencias de variables como variables lógicas), y se otorga un carácter puramente funcional a los restantes símbolos de función (empleando reescritura para evaluar las expresiones).

2.3 Expresiones condicionales

La función condicional estándar en el formato habitual:

`if` *cond* `then` e `else` e'

Ejemplo 2.8 *El siguiente ejemplo muestra el uso de expresiones condicionales:*

```
fact x = if x==0 || x==1
         then 1
         else x*(fact (x-1))
```

2.4 Restricciones

Las restricciones se especifican de la forma t_1`=:=`s_1 `&` t_2`=:=`s_2 `&` $\cdots$ `&` t_n`=:=`s_n que expresa la conjunción de restricciones ecuacionales elementales. Una restricción elemental $t =:= s$ puede resolverse si las expresiones t y s pueden ser evaluadas a términos constructores unificables [HAK+98]. La resolución de conjunciones se realiza concurrentemente, de forma que si en el proceso de resolución de c_1 `&` $\cdots$ `&` c_n (donde cada $c_i, 1 \leq i \leq n$ es una restricción elemental $t_i =:= s_i$) la resolución de c_i queda suspendida, se espera a la resolución de alguna c_j con $i \neq j$ por si produjese una instanciación que reactivase el proceso de resolución de c_i.

Curry incorpora el operador booleano '`==`' que representa la *igualdad estricta* definida de la siguiente forma:

C	`==` C	= `True`	% para todo constructor C de aridad 0
$C\ x_1 \cdots x_n$	`==` $C\ y_1 \cdots y_n$	= x_1`==`y_1 `&&` $\cdots$ `&&` x_n`==`y_n	% para todo constructor n-ario
$C\ x_1 \cdots x_n$	`==` $D\ y_1 \cdots y_n$	= `False`	% para todo par de constructores C y D distintos

El operador '`=:=`' sólo puede aparecer en las restricciones (como un operador de resolución de restricciones elementales). Mientras el operador '`=:=`' instancia variables, el operador

'==' no lo hace [HAK+98]. En este sentido (la búsqueda de soluciones para las restricciones) Curry difiere significativamente de los lenguajes funcionales (que se limitan a comprobar si la instancia actual de la restricción se satisface).

2.5 Orden Superior

En programación funcional, la expresión *orden superior* alude al hecho de que las funciones son *ciudadanos de primera clase* (*first-class citizens*). Con esto se quiere decir que las funciones recibirán el mismo tratamiento que el resto de objetos que se puedan definir: podrán ser pasadas como argumentos de otras funciones y devueltas como resultado de evaluar una expresión. A veces también se dice que una función es de *orden superior* si toma como uno de sus argumentos alguna función.

Ejemplo 2.9 *Un ejemplo típico de función de orden superior es:*

```
map f []     = []
map f (x:xs) = (f x):(map f xs)
```

La función `map` *toma como argumento una función* `f` *y una lista y forma la lista que resulta de aplicar la función* `f` *a cada elemento de la lista que se le pasa como segundo argumento.*

En la implementación de las características de orden superior de los lenguajes es muy importante la noción de *currificación*. La expresión 'currificación' hace mención al lógico británico Haskell B. Curry que fue el primero en aplicar a los lenguajes de programación la propiedad matemática que permite escribir el tipo de una función:

$$f :: \tau_1 \times \cdots \times \tau_k \rightarrow \tau$$

en su 'forma currificada':

$$f :: \tau_1 \rightarrow \cdots \rightarrow \tau_k \rightarrow \tau$$

Ejemplo 2.10 *Escribimos a continuación el tipo de algunas de las funciones previamente definidas empleando su versión currificada [Rea93].*

```
+ :: Int -> Int -> Int

if :: Bool -> t -> t -> t

map :: (t -> s) -> [t] -> [s]
```

El operador $\rightarrow$ es asociativo por la derecha, es decir, $\tau_1 \rightarrow \tau_2 \rightarrow \cdots \rightarrow \tau_n$ se interpreta como $\tau_1 \rightarrow (\tau_2 \rightarrow (\cdots \rightarrow \tau_n))$

La currificación justifica la sintaxis libre de paréntesis que se suele adoptar en los lenguajes de programación funcional (y también en Curry) para denotar la aplicación de una función a (algunos) argumentos. La aplicación de una función $f :: \tau_1 \rightarrow \tau_2 \rightarrow \cdots \rightarrow \tau_k \rightarrow \tau$ a sus argumentos se denota simplemente yuxtaponiendo éstos uno detrás de otro en el orden deseado, sin paréntesis. La aplicación de una función a sus argumentos es asociativa por la izquierda, esto es, $f\ e_1\ e_2\ \cdots\ e_k$ se interpreta como $(\cdots((f\ e_1)\ e_2)\cdots\ e_k)$. Observar que ello es consistente con la asociatividad de '$\rightarrow$': esto hace que una expresión como $f\ e_1$ tenga sentido: $f\ e_1$ es una función cuyo tipo es $\tau_2 \rightarrow \cdots \rightarrow \tau_k \rightarrow \tau$.

En Curry es posible manejar expresiones de *orden superior* empleando aplicaciones parciales de función y mediante el uso de *lambda* abstracciones.

Ejemplo 2.11 *Empleando la definición anterior para la función* `map` *podemos definir la función* `suma`*:*

```
suma = map (1+) [1,2,3]
```

que utiliza la aplicación parcial `(1+)` *del operador* `+` *que añade la unidad a cualquier número pasado como argumento.* `suma` *es una función constante que devuelve como resultado* `[2,3,4]`.

Las lambda abstracciones se representan de la forma:

\ $x_1\ \cdots\ x_n$ -> e

donde $x_1, \ldots, x_n$ son variables distintas que representan los argumentos de una función sin nombre.

Ejemplo 2.12 *Podemos definir* `resta`

```
resta = map (\x -> x-1) [1,2,3]
```

empleando una lambda abstracción que define una función que resta 1 al argumento que se le pasa. Es interesante ver que no hay forma de definir esta operación usando una aplicación parcial del operador '`-`*', pues no es lo mismo* `1-x`*, que corresponde a la aplicación parcial* `(1-)`*, que* `x-1`*.*

2.6 Entrada y salida de datos

Curry proporciona un modelo declarativo para la E/S considerando las operaciones de E/S como transformaciones del mundo exterior. Para asegurar que no se trabaja con diferentes versiones del mundo exterior, se exige que exista una única versión accesible de éste. Utilizando la aproximación monádica de E/S [PJW93] de Haskell y prohibiendo la existencia de funciones de E/S en los puntos donde pueda haber una evaluación no determinista podemos garantizar las condiciones de trabajo deseadas.

En la aproximación monádica de E/S el mundo exterior no es directamente accesible, sino que se modifica por la acción de determinadas funciones del lenguaje. Por lo tanto, el mundo exterior queda representado como un tipo abstracto de datos que proporciona acciones para modificarlo [GTW78, EM85]. El tipo abstracto es `IO t`.

Toda acción sobre `IO t` genera un nuevo valor del tipo `t` y un nuevo mundo exterior. De hecho `IO t` puede verse como una abreviatura de `World -> (t,World)`. Por ejemplo, la función `getChar` lee un carácter de la entrada estándar cuando se ejecuta, es decir, cuando es aplicada al mundo exterior, por eso tiene el tipo `IO Char`.

Lo más importante es que el programador no puede acceder al mundo exterior, sólo puede componer acciones sobre él para realizar la entrada o salida de datos.

Ejemplo 2.13 *Las siguientes funciones son de entrada de datos:*

```
getChar :: IO Char
getLine :: IO String
```

ya que tienen asociadas tipos de datos obtenidos del mundo exterior. Las siguientes funciones realizan operaciones de salida de datos:

```
putChar  :: Char -> IO ()
putStr   :: String -> IO ()
putStrLn :: String -> IO ()
```

ya que tomando como entrada determinados tipos de datos (`Char`, `String`) *modifican el mundo exterior sin devolver ninguna información* (*esto último viene representado por el tipo vacío* `()`).

Como un programa interactivo va a consistir en una secuencia de acciones de entrada y salida aplicadas sobre el mundo exterior, se proporcionan dos operadores de composición de acciones de E/S. El operador:

```
(>>) :: IO a -> IO b -> IO b
```

toma dos acciones y las compone, obteniendo una nueva versión del mundo exterior. La composición de acciones es secuencial, ignorando el valor obtenido por la primera acción. Pero si se desea utilizar el valor devuelto por una función de entrada, se dispone de la función:

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

que pasa el valor obtenido por la primera acción a la segunda acción.

Ejemplo 2.14 *La siguiente expresión:*

```
expr1 = getLine >>= putStrLn
```

lee una linea de texto de la entrada estándar y la escribe en la salida estándar. La siguiente función, algo más compleja,

```
expr2 = putStr "Opción[a,b,c]:" >> getChar >>=
  \x -> if x=='a' then putStrLn "Opción a"
        else if x=='b' then putStrLn "Opción b"
             else if x=='c' then putStrLn "Opción c"
                  else putStrLn "Erróneo" >> expr2
```

serviría como pequeño menú de opciones gracias al uso de una lambda abstracción. Se puede apreciar cómo se componen tres acciones sobre el mundo exterior, una de salida, que muestra el texto inicial del menú, compuesta con una de entrada, cuyo valor leído se pasa a una lambda abstracción para generar una determinada salida.

3 Programación declarativa y modelización conceptual

La modelización de sistemas no es una tarea fácil, pues requiere obtener información precisa y definida acerca de algo que puede ser muy complejo o cuya comprensión por parte del cliente es sólo aproximada. Para acometer esta tarea, se han propuesto multitud de técnicas de modelización, métodos de análisis de requisitos y entornos de descripción de sistemas, pero muchos de ellos generan modelos que no facilitan el estudio formal de la corrección de la especificación ni la derivación automática de propiedades de interés del sistema.

La *modelización conceptual* es la primera fase en el desarrollo de cualquier aplicación de sistemas reales y su salida, el modelo conceptual, describe información, restricciones o requerimientos que representan nuestro conocimiento acerca del sistema de interés.

El contenido de esta Sección repasa algunas técnicas básicas de especificación formal, especialmente las que entroncan con la programación declarativa. Posteriormente se resumen las ventajas derivadas del uso de las técnicas de programación declarativa en el campo de la modelización conceptual.

3.1 Modelización conceptual

Un modelo es un objeto formal que interpreta una parte del mundo y que sólo refleja (aunque parcialmente) el conocimiento que un observador construye sobre un cierto dominio atendiendo a un cierto criterio de utilidad.

La Figura 1 traza un esquema visual de la interrelación de estos conceptos.

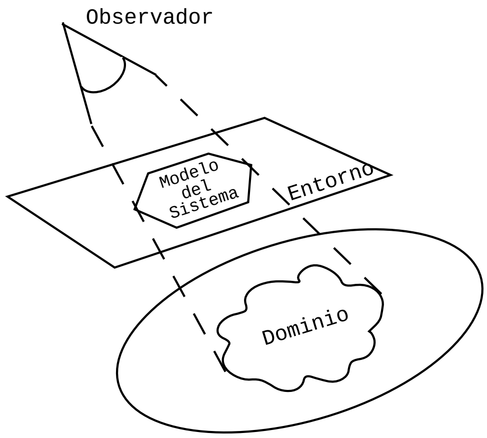

Figura 1: Vista del modelado de sistemas

3.1.1 Lenguajes de descripción

La especificación de un modelo conceptual se hace en un lenguaje descriptivo que utiliza generalmente una notación y reglas sintácticas precisas. Sin embargo, muy frecuentemente el lenguaje no dispone de una semántica igualmente precisa, de forma que la calidad de la descripción depende dramáticamente del buen hacer de la persona que la redacta.

Los lenguajes utilizados en la descripción de sistemas, más allá del uso de un lenguaje natural (cuyos defectos en términos de contradicción, ambigüedad, sobreespecificación, etc. son ampliamente reconocidos) o de los así llamados *lenguajes con formato* (lenguajes textuales o con gráficos que incorporan cierta estructura sintáctica para asociar características semánticas a determinadas construcciones pero no son ejecutables) son los lenguajes formales. Entendemos por lenguajes formales aquellos que incorporan tres partes bien diferenciadas:

- *sintaxis*, una notación para escribir la especificación,
- *semántica*, que da significado a las construcciones sintácticas tanto en términos declarativos como operacionales,
- una *relación* bien definida entre ambas.

La ventaja de utilizar lenguajes de especificación formales es que el modelo obtenido es ejecutable y que es posible:

1. estudiar la corrección de la especificación,
2. estudiar matemáticamente sus propiedades formales (consistencia, completitud, equivalencia entre especificaciones, ...) y

3. realizar transformaciones mecánicas (capaces de producir algún tipo de optimización) que no alteran el significado de la especificación.

Por otra parte, el lenguaje de especificación debe facilitar la descripción del problema sin tener que entrar en detalles de diseño e implementación y debe permitir analizar las propiedades de la especificación de forma sencilla y eficiente. Además, la especificación debe tener una semántica operacional bien definida que pueda mecanizarse de forma efectiva. Todos estos requisitos son cumplidos satisfactoriamente por los lenguajes declarativos.

3.1.2 Construcción de la especificación

Esta fase es crucial en la modelización conceptual, pues debe garantizarse la corrección de lo que se escribe en el lenguaje de especificación utilizado respecto a la concepción del modelo.

Una de las líneas de investigación más atractivas en este área es el estudio de técnicas, metodologías y herramientas para derivar un sistema ejecutable a partir de sus especificaciones. Es una tesis ampliamente aceptada que este objetivo puede alcanzarse de forma elegante siguiendo una aproximación formal. Puesto que una especificación formal sí es ejecutable, ésta puede considerarse como un prototipo (de muy alto nivel) del sistema.

Desde el momento en que la especificación escrita en un lenguaje declarativo es ejecutable, no hay ninguna diferencia notable entre especificación y programa, de tal forma que parece razonable pensar en realizar el proceso de modelización y codificación usando un mismo lenguaje, lo que simplifica el proceso y permite ahorrar costes [Bal85].

En la siguiente Sección reunimos las características más importantes de los lenguajes declarativos que, como lenguajes formales con una semántica operacional y declarativa simples, son aptos para nuestros propósitos.

3.2 Programación Declarativa

La investigación, desarrollo y uso de los lenguajes de programación declarativos han aumentado considerablemente durante la última década debido a una creciente convicción de que el estilo declarativo de programación puede contribuir esencialmente a la resolución de muchos problemas críticos en la producción de software, particularmente en lo que se referiere a la productividad y fiabilidad. Esto se debe, fundamentalmente, a que el estilo declarativo soporta un nivel de abstracción que facilita conceptualmente la tarea de programar. Si la complejidad de un lenguaje de programación se refleja en la complejidad de su descripción semántica, lo que caracteriza a los lenguajes declarativos es la claridad, sencillez y concisión de sus semánticas, que permiten entender rápidamente las construcciones del lenguaje y el significado de los programas [Bac78]. La razón de esta simplicidad está en la lógica matemática subyacente, que aporta al lenguaje su sintaxis, su semántica y su mecanismo de cómputo. Una ventaja práctica adicional de los lenguajes declarativos es que facilitan la explotación del paralelismo, en la medida en que la semántica de los programas no impone un orden secuencial determinado.

Los dos estilos tradicionales de programación declarativa son la programación funcional (basada en algún tipo de lógica ecuacional[1]) y la programación lógica relacional (basada en fragmentos de la lógica de primer orden, fundamentalmente cláusulas de Horn). En un contexto más laxo, es habitual denominar a la programación relacional con cláusulas de Horn simplemente "programación lógica"; en lo que sigue, vamos a mantener esta terminología en todos aquellos casos en los que no se preste a confusión. En programación lógica, la presencia de variables lógicas, soportada por el algoritmo de unificación, es una característica potente y altamente expresiva e, incluso, puede verse como un mecanismo primitivo de comunicación en un eventual modelo concurrente del lenguaje. Por otro lado, es el origen de la mayor parte de las diferencias entre los lenguajes relacionales y los funcionales, donde el enlace de las variables se produce en un único sentido. Como referencias estándar para la programación lógica podemos mencionar [AP90, Kow74, Llo87].

En la programación funcional, existe una diversificación mucho más grande de aproximaciones y estilos, desde la aproximación ecuacional pura [O'D77, O'D85, O'D94] hasta los trabajos clásicos de programación funcional y λ-cálculo [Bar90, FH88, PJ87], pasando por diferentes enfoques algebraicos, una panorámica informal de los cuales puede encontrarse, por ejemplo, en [BKL$^+$91, Wir90].

Las tres aproximaciones, ecuacional, funcional y algebraica, de alguna manera, compiten pero también se complementan. Mientras en los enfoques ecuacional y algebraico, las familias de funciones se especifican, tradicionalmente, por medio de ecuaciones condicionales, en los programas funcionales se definen las funciones por combinación de otras funciones predefinidas, usando algunas construcciones clásicas como el *if–then–else* y la abstracción de funciones. Sin embargo, es posible un tratamiento operacional de estos programas por técnicas de reescritura [BKL$^+$91] y, de hecho, más allá de las cuestiones de presentación o de estilo, y del diferente énfasis puesto en las técnicas de evaluación, los estilos ecuacional, algebraico y funcional conducen, esencialmente, a la misma clase de programas [O'D94], que llamaremos simplemente programas funcionales.

En un lenguaje lógico–funcional integrado se pueden escribir y ejecutar tanto programas funcionales puros como programas similares a los que se escriben en programación lógica. De la programación lógica, los lenguajes integrados obtienen la potencia de las variables lógicas, un mecanismo de búsqueda automática y las estructuras de datos parciales. De la programación funcional, se obtiene la expresividad de las funciones y tipos y un mecanismo de evaluación más eficiente, gracias a la reducción determinista de expresiones funcionales. Algunas ventajas adicionales de la integración son el contribuir a la mayor aceptación y difusión de la programación declarativa y el evitar la duplicación de esfuerzos en el desarrollo

[1] Incluimos aquí el λ–cálculo, que puede verse como una lógica ecuacional de orden superior.

de técnicas de implementación, análisis, transformación de programas, interfaces de usuario y otras muchas herramientas.

En los últimos años, se han formulado muchas propuestas para combinar los estilos lógico y funcional con el fin de ganar expresividad y permitir soluciones más naturales para ciertos problemas de programación. La gran variedad de escuelas que se aprecia en este campo refleja una diversidad importante también de motivaciones. Pero los métodos más populares se basan en combinar los mecanismos de reducción y búsqueda. Una gran parte de las propuestas para la integración hacen uso de los sistemas de reescritura de términos condicionales (SRTC) como programas y (alguna variante del) estrechamiento (*narrowing*) como semántica operacional, soportando así unificación y variables lógicas en un contexto funcional. El uso de *estrechamiento* como mecanismo de cómputo supone una extensión muy potente de los programas lógicos tradicionales y el modelo resultante tiene muchas oportunidades para el paralelismo. Algunos lenguajes que siguen este método son Alf [Han91], Babel [MR88] y Toy [CRLFSH97a]. Pese a que el algoritmo de *estrechamiento* es un método correcto y completo para obtener las soluciones a un sistema de ecuaciones, su terminación no está asegurada. El procedimiento puede ser controlado por varias estrategias de cómputo, cuyos comportamientos difieren desde el punto de vista de la completitud y la eficiencia. Un principio operacional alternativo es la *residuación*. La residuación retrasa aquellas llamadas a funciones que no estén lo suficientemente instanciadas para efectuar un paso (determinista) de reducción. El principio de residuación se utiliza en lenguajes como Escher [Llo95], Life [AP90] y Oz [Smo95]. La residuación combina la reducción determinista de funciones con el manejo de estructuras de datos parciales y soporta de forma muy natural computaciones concurrentes con sincronización sobre las variables lógicas. La principal desventaja del método de residuación es que se trata de un procedimiento incompleto. Recientemente, la resolución y la residuación generalizada se han combinado en un marco uniforme [Han97b]. El *estrechamiento* perezoso y la residuación pura pueden verse como casos particulares del nuevo modelo de computación integrado. El lenguaje de programación Curry [HAK+98] incorpora este nuevo modelo de computación. En [AN89, BL86, DP88, Han94, Red86] se puede encontrar una revisión, análisis y clasificación de las propuestas básicas para la integración. La colección en [DL86] constituye una referencia estándar en este campo. La panorámica más actualizada de propuestas de integración se encuentra en [Han94].

Los últimos avances en las técnicas de compilación para cláusulas de Horn y lenguajes funcionales permiten desarrollar compiladores que hacen la eficiencia de estos lenguajes comparable a la de los lenguajes más convencionales ejecutados sobre las mismas máquinas. También, la aparición de nuevos modelos y arquitecturas para la computación paralela contribuyen a incrementar sustancialmente la velocidad de los cálculos, ya que el paralelismo está presente implícitamente en la semántica operacional de los lenguajes declarativos. Una primera consecuencia es que la misma caracterización semántica sirve para las versiones secuencial y paralela de dichos lenguajes: los conceptos de consecuencia lógica, sustitución de respuesta computada, paso de resolución, forma normal, etc, ... son independientes de la forma (secuencial o paralela) en que son evaluados o computados los programas declarativos.

A finales de los años 80 surgió un nuevo paradigma de programación lógica que intentaba incorporar métodos eficientes de resolución de restricciones sobre dominios específicos, conservando el carácter declarativo y las buenas propiedades semánticas de los programas lógicos. La idea de base es que la programación lógica tiene estas propiedades debido al uso de cláusulas definidas; la unificación y el universo de Herbrand tienen poco que ver con dichas propiedades. Trabajar sobre estructuras específicas puede resultar muy intuitivo y las diferentes implementaciones del modelo pueden beneficiarse ampliamente de la extensa literatura referente al problema de la resolución de restricciones sobre diferentes dominios de interés (desarrollada, e.g., en los campos de la inteligencia artificial, la investigación operativa o el álgebra computacional), tales como los árboles regulares, enteros, reales, booleanos, dominios finitos y conjuntos regulares, entre otros. Esta flexibilidad contrasta con la rigidez del modelo tradicional, donde la computación se realiza siempre sobre un dominio simbólico, el universo de Herbrand. Se han desarrollado técnicas sofisticadas para la resolución de restricciones sobre los nuevos dominios [JM94, VH89, VH92], gracias a las cuales la eficiencia de los lenguajes mejora radicalmente con respecto a Prolog y se aproxima a la de los lenguajes imperativos. Michael Jampel mantiene una extensa bibliografía sobre Constraint (Logic) Programming, accesible en la dirección: `http://www.cirl.uoregon.edu/constraints/`.

La programación lógica tradicional carece de mecanismos bien establecidos de estructuración, necesarios para satisfacer algunos requerimientos estándar de la ingeniería del software como modularidad, ocultación de información, reusabilidad, etc. Los trabajos sobre módulos y herencia para teorías lógicas constituyeron los primeros intentos para llevar mecanismos de estructuración a la programación lógica. La integración de la programación lógica y la programación orientada a objetos (POO) es un medio natural para explotar la sinergia entre ambas. Los aspectos dinámicos de los objetos se introducen generalmente en los lenguajes orientados a objetos a través de la noción de clase. Es por ello que en una visión unificadora de los paradigmas de programación declarativos sea necesario plantearse un mecanismo potente que recoja el concepto de modularización de programas (cfr. [BLM94], [Mes93]). Los estudios habidos a este respecto se han ocupado, por una parte, de características metalingüísticas como la independencia, combinación o equivalencia de módulos (ver, e.g. [BGLM94, OK85]) y, por otra, del enriquecimiento de los lenguajes y de su base lógica [Mil89].

Curry es un nuevo lenguaje lógico–funcional que pretende incorporar las características más importantes de los lenguajes lógicos y funcionales modernos, como son Gödel [HL94] y λ-Prolog [NM88] (lenguajes lógicos), Haskell [HPJW92] y SML [MTH90] (lenguajes funcionales) y Alf [Han91], Babel [MR88] y Toy [CRLFSH97b] (lenguajes lógico–funcionales).

Más específicamente, el lenguaje incluye facilidades como los sistemas de tipos polimórficos, módulos, restricciones, orden superior, entrada/salida declarativa monádica y búsqueda encapsulada. Estos componentes permiten reemplazar muchas de las características impuras de lenguajes lógicos bien conocidos como Prolog (por ejemplo, el *corte*), mejorando así la naturaleza declarativa de los lenguajes lógicos. El desarrollo del lenguaje es una iniciativa internacional que aspira a construir una plataforma común para la investigación, enseñanza y aplicación de los lenguajes lógicos y funcionales integrados [Han97a, HKMN95]. El diseño del lenguaje acaba de empezar, pero existe una intensa cooperación a nivel internacional para que pronto llegue a convertirse en un estándar en el área.

3.3 Modelización conceptual declarativa

En las Secciones precedentes hemos recordado las ventajas de integrar la programación lógica y funcional de manera flexible, sencilla y con buen soporte formal, utilizando algún procedimiento especializado en resolver ecuaciones bajo una teoría ecuacional, como el *narrowing*. Como una aplicación interesante de las técnicas propuestas, en esta sección se estudia la utilidad de las mismas en el modelado conceptual de sistemas. En particular, mostramos que dichas técnicas son capaces de operar con especificaciones ejecutables en dos modos distintos, uno de los cuales es bien conocido que ofrece una capacidad inferencial que puede utilizarse para resolver problemas de generación autómatica de planes.

Introducimos la idea de describir el esquema conceptual como una teoría formal en una lógica de primer orden con igualdad. El formalismo empleado debe proporcionar, al menos, las siguientes facilidades, que deben ser ofrecidas dentro de un marco formal uniforme [AR92, VF85]:

- igualdad y otras relaciones y funciones (para dar soporte a la especificación de las operaciones de actualización y de las respuestas a los requerimientos de consulta).
- distinción entre constructores y funciones definidas (para dar soporte a la metodología constructiva de especificación algebraica).
- axiomas entre constructores (para el rechazo de requerimientos de actualización inaceptables y, por lo tanto, improductivos).
- mecanismos de resolución y simplificación de restricciones (para verificar y hacer cumplir las restricciones de integridad).
- algún tipo de tratamiento de la negación (que permita definir y deducir la falsedad de una relación, en particular de la igualdad).

El lenguaje Curry es capaz de soportar todas estas características haciendo posible la especificación ejecutable de modelos conceptuales (declarativos). En nuestra aproximación, basada en el paradigma de programación lógico funcional con restricciones, la teoría ecuacional representa el esquema conceptual que será codificada como un programa del lenguaje Curry. De esta manera, las semánticas declarativa y operacional del lenguaje dan soporte formal a los aspectos de interpretación y ejecutabilidad de la teoría. Los estados del sistema se denotarán mediante composición de operaciones de actualización que, presentadas por medio de símbolos de función, transforman estados en estados. Esta noción de estado recoge la máxima cantidad de información sobre la evolución del sistema que el modelo conceptual permite representar. La igualdad y otros predicados se utilizarán para afirmar propiedades y relaciones sobre un estado. La satisfacción de las restricciones de integridad será verificada mediante técnicas de resolución de restricciones. Mostraremos también como la representación intensional (por medio de restricciones) del conjunto de respuestas a un requerimiento de consulta (a un objetivo Curry) proporciona facilidades adicionales que enriquecen este campo.

4 Modelo conceptual de la cuenca hidrográfica del río Segura

Uno de los objetivos de este informe es contribuir al desarrollo de una propuesta educativa para concienciar a la sociedad sobre los problemas del río Segura, ofreciendo un modelo operativo del río para su uso en centros de enseñanza.

A lo largo de esta sección, desarrollamos el modelo conceptual de la cuenca hidrográfica del río Segura siguiendo una aproximación declarativa tal como se presentó en la Sección 3. Partimos de una perspectiva inicial del ecosistema del río Segura que presta atención a la importancia de éste en el entorno económico y social en que se ubica (cuestión de interés para enfocar adecuadamente un producto que pretende ser un instrumento educativo útil), para posteriormente explicar el proceso de desarrollo del modelo conceptual para el citado ecosistema, abordando la identificación de los factores estáticos y dinámicos que mejor caracterizan el sistema y su evolución.

4.1 El río Segura

Los problemas del agua tienen un especial interés para los habitantes de la Región de Murcia, pues son la base de su economía. Es un bien esencial en la región debido a las características climáticas de esta zona y a los problemas que genera su escasez. Es un tema de gran actualidad que se trata habitualmente en los medios de comunicación. Trasvases, riegos de socorro, lluvias y riadas están en las conversaciones de la calle. El río Segura es un ecosistema emblemático en la Región de Murcia pero en torno a él se plantean problemas ambientales graves. En la cuenca del Segura se consume más agua de la que hay porque se extrae agua de los acuíferos y aguas subterráneas. La situación de éstos es de gravísima sobreexplotación. Esta circunstancia se agrava en las épocas de sequía, pero en el caso de esta cuenca la situación de sequía no es esporádica sino endémica. Grupos y asociaciones de todo tipo han denunciado en repetidas ocasiones el lamentable estado en que se encuentra, pero los ciudadanos, aunque han avanzado en la toma de conciencia, no han conseguido todavía que los responsables pongan en práctica las soluciones idóneas.

Hay que realizar un esfuerzo educativo para que, a partir de actitudes de análisis y comprensión, la sociedad y las administraciones públicas puedan asumir las medidas necesarias para solucionar definitivamente los problemas del río. El río resulta de un elevado interés ecológico y educativo para generar actitudes y valores en favor de una mayor armonía entre las personas y su entorno. Es necesario integrar los conceptos ecológicos y de desarrollo y analizar las causas de los principales problemas ambientales vinculados al desarrollo local.

4.2 Modelización de ecosistemas

Desde un punto de vista biológico, un *ecosistema* puede definirse como aquella unidad del medio ambiente que contiene una comunidad biológica en la que el flujo de energía se manifiesta en la cadena trófica y en los ciclos de la materia [CC87]. Utilizando un enfoque conceptual, un *ecosistema* se describe como un conjunto de objetos interrelacionados, cuya evolución se ajusta a ciertas reglas de transformación de dichos objetos y sus relacioness [CC87]. Un ecosistema es un sistema complejo con acciones externas de salida dependientes tanto de acciones externas de entrada (acciones del entorno hacia el ecosistema) como de acciones internas (acciones entre elementos del ecosistema), que identifican un conjunto de interacciones entre el ecosistema y su entorno. Dicha información determina los distintos estados por los que traviesa el ecosistema.

De acuerdo con esto, el modelo conceptual queda definido en función de la noción de *estado del sistema*, que se representa como la secuencia de interacciones ocurridas entre el ecosistema y su entorno. Cada interacción supondrá, por lo tanto, una modificación del estado, que resulta de la composición de la nueva información generada y de la información representada por el estado anterior a ésta. También debemos considerar que la 'aceptación' de una interacción (en el modelo) debe ir precedida de la comprobación de ciertas restricciones que deben satisfacerse para asegurar una equivalencia entre el modelo y el comportamiento real del ecosistema.

4.3 Planteamiento del modelo conceptual

En este informe un ecosistema se estudia como un caso particular de sistema que interacciona con un entorno. En el modelo, el usuario que lo maneja desempeña el papel de entorno del mismo. El usuario, es pues, el encargado de desencadenar las acciones de entrada que actúan sobre el ecosistema.

La consideración de todo el conjunto de acciones internas y externas (de entrada y salida) que el río posee produciría un modelo casi intratable, sobre todo desde el punto de vista computacional. Por tanto, nos limitaremos a un modelo simplificado del ecosistema del río Segura con la intención de hacerlo más adecuado a los intereses mostrados previamente en la introducción de esta sección.

Modelamos el ecosistema del río Segura como un sistema reactivo (sin acciones internas) donde cada acción de entrada implica una acción de salida. El modelo asociado reflejará únicamente la respuesta del ecosistema ante una determinada acción del usuario. De esta manera, se refuerza la orientación educativa del modelo, insistiendo en la repercusión de las acciones externas (del usuario) sobre el ecosistema del río. Aún así, la respuesta del ecosistema se restringe a la modificación del valor asociado a determinadas propiedades de interés, en vez de generar inmediatamente un conjunto de salidas hacia el entorno como respuesta a la acción de entrada recibida por el ecosistema. Ésto permite representar el estado del ecosistema como un conjunto de valores asociados a ciertas propiedades (escogidas previamente) que sintetizan el ecosistema.

El modelo conceptual del río Segura constará, por tanto, de un conjunto de acciones aplicables y un conjunto de propiedades que utilizaremos para observar el estado del sistema y su evolución. Las acciones han sido escogidas considerando su relación con las propiedades de interés y asegurando una correspondencia con la realidad. La caracterización del estado del ecosistema se realiza empleando dos parámetros: el índice de calidad y el caudal existente. Éstos engloban de una forma clara diferentes propiedades cuya naturaleza compleja hubiese dificultado la comprensión rápida del estado del ecosistema. Dichas propiedades corresponden a los dos problemas que padece el río Segura: contaminación y sequía.

A continuación, mostramos al lector los diferentes tramos en que dividimos el cauce del río Segura, los parámetros empleados para caracterizar el estado del ecosistema, las acciones escogidas para el modelo conceptual y la información necesaria para calcular los valores asociados a dichos parámetros.

4.3.1 Tramos del río Segura

El río Segura está dividido en varios tramos que abarcan desde su nacimiento hasta su desembocadura en el mar Mediterráneo. Éstos son [Ibo96]:

1. Cabecera
2. Cenajo a Almadenes
3. Cieza a Abarán
4. Blanca

5. Blanca a Alguazas
6. Alguazas
7. Molina a Alcantarilla
8. Murcia
9. Murcia a Orihuela
10. Orihuela
11. Orihuela a Rojales
12. Guardamar

Cada tramo tiene unas características propias de caudal, aportaciones de agua, extracciones de agua, tipo de vertidos contaminantes que recibe, etc., según el entorno local en el que está situado. Sin embargo, hemos simplificado el modelo conceptual considerando que todos los tramos tienen las mismas características a priori, permitiendo que cualquier acción de las existentes pueda ser aplicada indistintamente a cualquiera de los tramos.

4.3.2 Índice de Calidad Mixto

El uso de un indicador de calidad del agua ayuda a clasificar rápidamente (aunque con menor precisión) el estado del río. De esta forma, podremos realizar comparaciones averiguando qué estado es mejor o peor.

Para definir un indicador de la calidad del agua es preciso considerar multitud de parámetros biológicos y químicos y tener en cuenta su variabilidad geográfica respecto a cada tramo así como su dependencia de la época del año. Además, desde la perspectiva de los usos a que se destina, la calidad del agua en los cauces de la cuenca se puede clasificar según su aptitud para el consumo humano, los baños, la vida de los peces y el regadío. Sin embargo, es común el uso de indicadores globales como el Índice de Calidad General elaborado por el M.O.P.T. que permiten disponer de un valor numérico que simboliza la calidad del agua. La legislación española vigente sobre calidad de aguas continentales es la contenida en la Ley de Aguas (29/1985 de 2 de Agosto) y en el Reglamento de la Administración Pública del Agua y de la Planificación Hidrológica que se desarrolla de los Títulos II y III de la Ley de Aguas (R.D. 927/1988 de 29 de Julio) [Ibo96].

La determinación del Índice de Calidad General requiere medir un total de 23 parámetros divididos en básicos y complementarios, unidos a un conjunto de consideraciones asociadas según el valor de algunos parámetros críticos. Si consideramos el uso que vamos a hacer del índice de calidad general, éste no ayuda de forma alguna a la simplificación del cálculo del índice de calidad para la obtención de un valor único y representativo. Por ello, con los datos obtenidos para las aguas de la cuenca del río Segura (tanto biológicos como físicos y químicos) se ha propuesto el uso del Índice de Calidad Mixto (ICM) para la cuenca del río Segura que, tomando sólo tres parámetros físico-químicos y uno biológico, obtiene un valor representativo de la calidad del agua. Dicho Índice de Calidad Mixto es el que utilizamos en el modelo para el cálculo del índice de calidad de las aguas del ecosistema.

Los parámetros que utilizaremos para representar el estado del río son los siguientes:

- concentración de fosfatos,
- demanda biólogica de oxígeno (DBO_5),
- oxígeno disuelto y
- organismos indicadores

La Tabla 1 muestra la relación existente entre los tres parámetros físico-químicos y el parámetro biológico. En realidad, sólo utilizaremos los tres parámetros físico-químicos pues el biológico actúa sólo como indicativo, de tal forma que dado un índice de calidad se supone que los organismos asociados estarán presentes sobre la superficie, orilla, etc. La Tabla 1 se utiliza para calcular el valor del ICM, comprobando de arriba a abajo las distintas condiciones y asignando el valor asociado a la primera condición cumplida. Si no se cumple ninguna condición, el valor asignado es 0.

Además, se establece una relación entre el índice de calidad mixto y la clasificación que deseamos obtener de la siguiente forma:

Máxima calidad:	9 y 10
Calidad buena:	7 y 8
Calidad media:	5 y 6
Calidad baja:	3 y 4
Mínima calidad:	0, 1 y 2

4.3.3 Caudal

El conocimiento del caudal del agua aporta información no recogida por un indicador de la calidad del agua. Los incesantes cambios de caudal que normalmente soporta el río Segura, debidos a la sequía existente y a los incrementos habituales de los afluentes, precipitaciones, etc., hacen necesario considerar esta propiedad del río. La Tabla 2 muestra el caudal medio para cada uno de los diferentes tramos del río Segura.

Fosfatos (mg/l)	DBO_5 (mg/l)	O_2 disuelto (mg/l)	Organismos indicadores	ICM
$< 0,4$	$< 3,5$	$> 10,0$	Plecóptera, Epeorus o Rhithrogena	10
$< 0,4$	$< 4,0$	$> 8,5$	Hidropsyche o Potamanthus luteus	9
$< 0,4$	$< 4,0$	$> 8,0$	Ephemeroptera (excepto Baetidae y Caenidae)	8
$< 1,5$	$< 11,0$	$> 8,0$	Echinogammarus	7
$< 3,5$	$< 30,0$	$> 3,0$	Odonata o Diptera (excepto Chironomus y Psychodidae)	6
$< 3,5$	$< 30,0$	$> 3,0$	Ephemeroptera (Baetidae y Caenidae)	5
$< 3,5$	$< 30,0$	$> 2,5$	Asellidae, Physella acuta o Potamopyrgus jenkinsi	4
$< 5,0$	$< 50,0$	$> 2,0$	Psychodidae y Chironomus	3
$< 5,0$	$< 50,0$	$> 2,0$	Tubificidae	2
$< 15,0$	$< 110,0$	$\geq 0,0$	Sin vida macróspica	1

Tabla 1: Rango de valores para el Índice de Calidad Mixto de la Cuenca del Río Segura

4.3.4 Acciones sobre el río Segura

De entre las posibles acciones a considerar en el modelo del río, hemos escogido las más representativas para el ecosistema en estudio. Para expresar su efecto sobre el valor de los parámetros empleados en la descripción del estado del río, emplearemos ecuaciones de la forma $P_{i+1} = f_P(P_i)$ donde P_i es el valor del parámetro P antes de que se realice la acción y P_{i+1} es el valor del parámetro una vez completada ésta. La función f_P relaciona ambos valores (pudiendo tener en cuenta el valor actual de otros parámetros distintos a P). F representa la concentración de fosfatos, D la demanda biológica de oxígeno (DBO_5), O la concentración de O_2 disuelto y C el caudal existente. De acuerdo con esto, algunas de las acciones consideradas junto con el efecto que las acompaña son las siguientes:

1. aumento del caudal al doble con agua de temperatura muy baja procedente de un pantano (tramos 2 y 3) que aporta fosfatos (+3, 5) en abundancia y produce un aumento del DBO_5 (+30, 0),

$$\begin{aligned} F_{i+1} &= \tfrac{F_i+3,5}{2} \\ D_{i+1} &= \tfrac{D_i+30,0}{2} \\ C_{i+1} &= C_i * 2 \\ O_{i+1} &= O_i \end{aligned}$$

Cabecera	282
Cenajo a Almadenes	433
Cieza a Abarán	725
Blanca	734
Blanca a Alguazas	741
Alguazas	759
Molina a Alcantarilla	790
Murcia	790
Murcia a Orihuela	854
Orihuela	854
Orihuela a Rojales	854
Guardamar	871

Tabla 2: Caudales medios del río Segura (en hm^3/s)

2. disminución drástica del caudal (-20%) por un riego incontrolado (tramos 7, 8, 9, 10, 11) que puede hacer bajar el caudal al 3% o incluso al 1% (considerando en ese caso que el valor de ICM no puede ser calculado),

$$\begin{aligned} C_{i+1} &= C_i * 0,8 \\ F_{i+1} &= F_i \\ D_{i+1} &= D_i \\ O_{i+1} &= O_i \end{aligned}$$

3. vertido de aguas sin depurar procedentes de una empresa de frutas o verduras (tramos 4, 5, 6, 7) con aguas ricas en materia orgánica que produce un aumento considerable de fosfatos ($+5,0$).

$$\begin{aligned} C_{i+1} &= C_i \\ F_{i+1} &= F_i + 5,0 \\ D_{i+1} &= D_i \\ O_{i+1} &= O_i \end{aligned}$$

4. vertido de aguas urbanas sin depurar (tramos 7, 8, 9, 10, 11) con detergentes disueltos que dificultan la disolución de gases, disminuyendo la concentración de oxígeno disuelto ($-2,0$).

$$\begin{aligned} C_{i+1} &= C_i \\ F_{i+1} &= F_i \\ D_{i+1} &= D_i \\ O_{i+1} &= O_i - 2,0 \end{aligned}$$

5. vertidos de aguas calientes, de refrigeración de industrias (tramos 6, 7, 8, 9) que disminuyen la solubilidad de los gases en el agua, es decir, disminuyen la concentración de oxígeno disuelto $(-1, 0)$.

$$\begin{aligned}
C_{i+1} &= C_i \\
F_{i+1} &= F_i \\
D_{i+1} &= D_i \\
O_{i+1} &= O_i - 1,0
\end{aligned}$$

4.4 Modelo Declarativo del ecosistema del río Segura

El modelo declarativo debe incorporar una representación del estado del ecosistema que incluya la información acerca de qué acciones han sido realizadas, en qué tramos del recorrido del río y en qué fechas. Esto facilita la obtención del valor del ICM y del valor del caudal para un determinado tramo y una determinada fecha.

Para representar el estado utilizaremos estructuras de datos algebraicas propias que nos permitan reunir la información necesaria para codificar el modelo. A continuación, identificamos las estructuras de datos asociadas a los diferentes conceptos pertenecientes al modelo. La representación del estado incluye, como veremos a continuación, la representación de las acciones.

4.4.1 Tramos

El conjunto de tramos en que está dividido el cauce del río se representa mediante el tipo `Tramo`:

```
data Tramo = trA | trB | trC | trD | trE | trF | trG | trH |
             trI | trJ | trK | trL
```

La correspondencia entre los constructores del tipo `Tramo` y la división geográfica presentada en la Sección 4.3.1 es la mostrada en la Tabla 3.

4.4.2 Fechas

La falta de un tipo estándar del lenguaje Curry para representar las fechas se resuelve fácilmente, definiendo un nuevo tipo de datos `DateTime` que almacene la información necesaria:

```
data Time = tm Int Int Int
data Date = dt Int Int Int
data DateTime = dtime Time Date
```

Los tipos auxiliares `Time` y `Date` se utilizan para representar, respectivamente, las horas y las fechas. El tipo `Time` representa un valor de hora usando el constructor `tm`. Dicho

Cabecera	`trA`
Cenajo a Almadenes	`trB`
Cieza a Abarán	`trC`
Blanca	`trD`
Blanca a Alguazas	`trE`
Alguazas	`trF`
Molina a Alcantarilla	`trG`
Murcia	`trH`
Murcia a Orihuela	`trI`
Orihuela	`trJ`
Orihuela a Rojales	`trK`
Guardamar	`trL`

Tabla 3: Correspondencia entre constructores y tramos.

constructor tiene asociados tres números enteros que representan la hora, los minutos y los segundos. El constructor `dt`, perteneciente al tipo `Date`, es el encargado de representar las fechas. Para ello asocia los valores de día, mes y año, respectivamente, a los tres números enteros que posee como argumentos. El tipo de datos `DateTime` representa un valor de hora y fecha, y utiliza el constructor binario `dtime` donde almacena la hora y la fecha.

4.4.3 Estado del ecosistema

La representación del estado se realiza mediante "composición de actualizaciones", siguiendo la forma estándar usada en el modelado declarativo algebraico.

El estado del ecosistema se representa como un término que acumula las acciones realizadas sobre el río. Cuando deseemos obtener información sobre las propiedades de interés, procesaremos el término que representa la secuencia de acciones. Para representar dicho término, se utiliza un tipo de datos algebraico cuyos constructores representan cada una de las posibles acciones, incorporando en sus argumentos el tramo al que se aplica, la fecha en que se realizó y el estado anterior del modelo. De esta forma, la nueva acción se acumula sobre las realizadas hasta el momento. La representación en secuencia necesita de un constructor que indique el inicio de ésta. En nuestro caso, hemos obtado por el constructor `Cini` que no posee ningún argumento.

```
data Rio = Cini |
           Cpantano Tramo Date Rio |
           Criego   Tramo Date Rio |
           Cfrutas  Tramo Date Rio |
```

```
Curbanas Tramo Date Rio |
Crefri   Tramo Date Rio
```

donde, de acuerdo con las acciones enumeradas en la Sección 4.3.1:

- `Cpantano` corresponde al travase de agua procedente de un pantano,
- `Criego` corresponde a un riego incontrolado,
- `Cfrutas` corresponde a un vertido de una empresa de frutas,
- `Curbanas` corresponde a un vértido de aguas urbanas,
- `Crefri` corresponde a un vertido de aguas usadas para refrigeración.

La consideración efectiva de una acción externa a efectos de modificación del estado del modelo necesita que se cumplan un conjunto de restricciones previas asociadas a ella. La presencia de un constructor en el término que representa el estado del ecosistema reflejará la aplicación de una actualización válida, mientras que usaremos un símbolo de función indicando la aplicación tentativa de la acción externa, incluyendo en la definición de ésta la comprobación del conjunto de restricciones asociado a ella. De esta manera, utilizaremos los símbolos de función: `pantano`, `riego`, `frutas`, `urbanas` y `refri` para indicar que vamos a comprobar la validez de la composición de la acción asociada con el estado actual.

En nuestro modelo declarativo del ecosistema, las restricciones asociados a cada acción son todas del mismo tipo. Éste asegura una correcta inserción de la acción en el término que representa el estado actual del modelo del ecosistema.

4.4.4 Caudal

El caudal no ha sido representado como un tipo de datos puesto que únicamente necesitamos conocer el valor medio en cada tramo para después obtener el valor actual deduciéndolo a partir del estado. Representamos esta asociación mediante una función `caudal` definida en la Figura 2.

El procesamiento de la secuencia de acciones de entrada que representa el estado del ecosistema permite obtener un valor de concentración de fosfatos, DBO_5 y O_2 junto con el valor de caudal para un tramo concreto y tomando como referencia una fecha concreta. Dicho procesamiento está codificado en la función `obt_datos` de acuerdo con las transformaciones del valor de los parámetros especificados en la Sección 4.3.4.

El cálculo del valor del ICM se realiza mediante la función `icm` de acuerdo con la Tabla 1.

```
caudal trA = 282.0
caudal trB = 433.0
caudal trC = 725.0
caudal trD = 734.0
caudal trE = 741.0
caudal trF = 759.0
caudal trG = 760.0
caudal trH = 790.0
caudal trI = 854.0
caudal trJ = 854.0
caudal trK = 854.0
caudal trL = 871.0
```

Figura 2: Código asociado a la función `caudal`

4.5 Manual de usuario de la especificación

Tras la carga de la especificación en Curry procederemos a introducir la llamada "`menu`" para iniciar la ejecución, obteniendo en pantalla el menú inicial de la especificación:

```
Menu - Modelo cuenca hidrografica
p - trasvase de agua del pantano
r - riego incontrolado con agua del rio
f - vertido de aguas sin depurar de una empresa de frutas
u - vertido de aguas urbanas sin depurar
e - vertido de aguas calientes de refrigeracion de industrias
i - ver indice calidad mixto
v - ver valores: Fosfatos DBO5 Oxigeno Caudal
x - salir
?
```

Las distintas opciones del menú nos permiten introducir todas las acciones mostradas en la Sección 4.3.4 (opciones `p`, `r`, `f`, `u`, `e`) y comprobar el valor de las distintas propiedades de interés (opciones `i` y `v`).

Cada una de las opciones implica la especificación del tramo a considerar y del momento temporal al que se hace referencia.

Un ejemplo de ejecución sería el mostrado a continuación, en el que procedemos, inicialmente, a comprobar el valor del ICM y de los factores físicos para continuar añadiendo una acción contaminante, tras la cual comprobamos nuevamente el valor del ICM y de los factores físicos:

```
Menu - Modelo cuenca hidrografica
p - trasvase de agua del pantano
r - riego incontrolado con agua del rio
f - vertido de aguas sin depurar de una empresa de frutas
u - vertido de aguas urbanas sin depurar
e - vertido de aguas calientes de refrigeracion de insdustrias
i - ver indice calidad mixto
v - ver valores: Fosfatos DBO5 Oxigeno Caudal
x - salir
?v
Tramo A - Cabecera
Tramo B - Cenajo a Almadenes
Tramo C - Cieza a Abaran
Tramo D - Blanca
Tramo E - Blanca a Alguazas
Tramo F - Alguazas
Tramo G - Molina a Alcantarilla
Tramo H - Murcia
Tramo I - Murcia a Orihuela
Tramo J - Orihuela
Tramo K - Orihuela a Rojales
Tramo L - Guardamar
Elige tramo?a
(hh:mm:ss)? 0:00:00
(dd/mm/yy)? 1/1/98
Fosfatos:0.0 DBO5:0.0 Oxigeno:11.0 Caudal:282.0

Menu - Modelo cuenca hidrografica
p - trasvase de agua del pantano
r - riego incontrolado con agua del rio
f - vertido de aguas sin depurar de una empresa de frutas
u - vertido de aguas urbanas sin depurar
e - vertido de aguas calientes de refrigeracion de insdustrias
i - ver indice calidad mixto
v - ver valores: Fosfatos DBO5 Oxigeno Caudal
x - salir
?i
Tramo A - Cabecera
Tramo B - Cenajo a Almadenes
Tramo C - Cieza a Abaran
Tramo D - Blanca
```

```
Tramo E - Blanca a Alguazas
Tramo F - Alguazas
Tramo G - Molina a Alcantarilla
Tramo H - Murcia
Tramo I - Murcia a Orihuela
Tramo J - Orihuela
Tramo K - Orihuela a Rojales
Tramo L - Guardamar
Elige tramo?a
(hh:mm:ss)? 0:00:00
(dd/mm/yy)? 1/1/98
Calidad: Maxima
Indice Calidad Mixto: 10
Organismos Indicadores: Plecoptera, Epeorus o Rhithrogena

Menu - Modelo cuenca hidrografica
p - trasvase de agua del pantano
r - riego incontrolado con agua del rio
f - vertido de aguas sin depurar de una empresa de frutas
u - vertido de aguas urbanas sin depurar
e - vertido de aguas calientes de refrigeracion de insdustrias
i - ver indice calidad mixto
v - ver valores: Fosfatos DBO5 Oxigeno Caudal
x - salir
?p
Tramo A - Cabecera
Tramo B - Cenajo a Almadenes
Tramo C - Cieza a Abaran
Tramo D - Blanca
Tramo E - Blanca a Alguazas
Tramo F - Alguazas
Tramo G - Molina a Alcantarilla
Tramo H - Murcia
Tramo I - Murcia a Orihuela
Tramo J - Orihuela
Tramo K - Orihuela a Rojales
Tramo L - Guardamar
Elige tramo?a
(hh:mm:ss)? 0:00:01
(dd/mm/yy)? 1/1/98
```

```
Menu - Modelo cuenca hidrografica
p - trasvase de agua del pantano
r - riego incontrolado con agua del rio
f - vertido de aguas sin depurar de una empresa de frutas
u - vertido de aguas urbanas sin depurar
e - vertido de aguas calientes de refrigeracion de insdustrias
i - ver indice calidad mixto
v - ver valores: Fosfatos DBO5 Oxigeno Caudal
x - salir
?i
Tramo A - Cabecera
Tramo B - Cenajo a Almadenes
Tramo C - Cieza a Abaran
Tramo D - Blanca
Tramo E - Blanca a Alguazas
Tramo F - Alguazas
Tramo G - Molina a Alcantarilla
Tramo H - Murcia
Tramo I - Murcia a Orihuela
Tramo J - Orihuela
Tramo K - Orihuela a Rojales
Tramo L - Guardamar
Elige tramo?a
(hh:mm:ss)? 0:00:02
(dd/mm/yy)? 1/1/98
Calidad: Media
Indice Calidad Mixto: 6
Organismos Indicadores: Odonata o Diptera (excepto Chironomus y Psychodidae)

Menu - Modelo cuenca hidrografica
p - trasvase de agua del pantano
r - riego incontrolado con agua del rio
f - vertido de aguas sin depurar de una empresa de frutas
u - vertido de aguas urbanas sin depurar
e - vertido de aguas calientes de refrigeracion de insdustrias
i - ver indice calidad mixto
v - ver valores: Fosfatos DBO5 Oxigeno Caudal
x - salir
?v
Tramo A - Cabecera
Tramo B - Cenajo a Almadenes
```

```
Tramo C - Cieza a Abaran
Tramo D - Blanca
Tramo E - Blanca a Alguazas
Tramo F - Alguazas
Tramo G - Molina a Alcantarilla
Tramo H - Murcia
Tramo I - Murcia a Orihuela
Tramo J - Orihuela
Tramo K - Orihuela a Rojales
Tramo L - Guardamar
Elige tramo?a
(hh:mm:ss)? 0:00:02
(dd/mm/yy)? 1/1/98
Fosfatos:1.75 DBO5:15.0 Oxigeno:11.0 Caudal:564.0

Menu - Modelo cuenca hidrografica
p - trasvase de agua del pantano
r - riego incontrolado con agua del rio
f - vertido de aguas sin depurar de una empresa de frutas
u - vertido de aguas urbanas sin depurar
e - vertido de aguas calientes de refrigeracion de insdustrias
i - ver indice calidad mixto
v - ver valores: Fosfatos DBO5 Oxigeno Caudal
x - salir
?x

Solution: [ IO () ] 52930 milliseconds
```

4.6 Especificación del modelo conceptual del río Segura en Curry

Lo que sigue a cotinuación es el código Curry para el modelo del ecosistema de la Cuenca Hidrográfica del río Segura. Este código está especificado en UPV-Curry. En concreto, en una versión especial del intérprete que incluye los reales (`Float`).

```
-- Modelo Declarativo del Segura

infixr 9 `less`

---------------------------------------
--- Definiciones generales
---------------------------------------
data Time = tm Int Int Int -- hora/minutos/segundos
```

```
data Date = dt Int Int Int -- dia/mes/anyo
data DateTime = dtime Time Date
---------------------------------------

-- funcion `less` para comparar fechas (dates)
(dtime (tm h1 mi1 s1) (dt d1 m1 y1)) `less` (dtime (tm h2 mi2 s2) (dt d2 m2 y2)) =
   if y1<y2 then True
   else if m1<m2 then True
        else if d1<d2 then True
             else if h1<h2 then True
                  else if mi1<mi2 then True else s1<s2

---------------------------------------
-- Indice de calidad medio.
---------------------------------------
icm fosfatos dbo5 o2 =
   if fosfatos < 0.4  && dbo5 < 3.5   && o2 > 10.0 then ("Maxima",10) else
   if fosfatos < 0.4  && dbo5 < 4.0   && o2 > 8.5  then ("Maxima",9)  else
   if fosfatos < 0.4  && dbo5 < 4.0   && o2 > 8.0  then ("Buena",8)   else
   if fosfatos < 1.5  && dbo5 < 11.0  && o2 > 8.0  then ("Buena",7)   else
   if fosfatos < 3.5  && dbo5 < 30.0  && o2 > 3.0  then ("Media",6)   else
   if fosfatos < 3.5  && dbo5 < 30.0  && o2 > 3.0  then ("Media",5)   else
   if fosfatos < 3.5  && dbo5 < 30.0  && o2 > 2.5  then ("Baja",4)    else
   if fosfatos < 5.0  && dbo5 < 50.0  && o2 > 2.0  then ("Baja",3)    else
   if fosfatos < 5.0  && dbo5 < 50.0  && o2 > 2.0  then ("Minima",2)  else
   if fosfatos < 15.0 && dbo5 < 110.0 && o2 >= 0.0 then ("Minima",1)
                                                   else ("Minima",0)

-- informacion adicional al indice de calidad
organismos 10 = "Plecoptera, Epeorus o Rhithrogena"
organismos  9 = "Hidropsyche o Potamanthus luteus"
organismos  8 = "Ephemeroptera (excepto Baetidae y Caenidae)"
organismos  7 = "Echinogammarus"
organismos  6 = "Odonata o Diptera (excepto Chironomus y Psychodidae)"
organismos  5 = "Ephemeroptera (Baetidae y Caenidae)"
organismos  4 = "Asellidae, Physella acuta o Potamopyrgus jenkinsi"
organismos  3 = "Psychodidae y Chironomus"
organismos  2 = "Tubificidae"
organismos  1 = "Sin vida macroscopica"

{-----------------------------------------------------------------
```

```
                         Modelo del Rio
 -------------------------------------------------------------------}

-- Tramos
data Tramo = trA | trB | trC | trD | trE | trF | trG | trH |
             trI | trJ | trK | trL

strTramo trA = "Cabecera"
strTramo trB = "Cenajo a Almadenes"
strTramo trC = "Cieza a Abarn"
strTramo trD = "Blanca"
strTramo trE = "Blanca a Alguazas"
strTramo trF = "Alguazas"
strTramo trG = "Molina a Alcantarilla"
strTramo trH = "Murcia"
strTramo trI = "Murcia a Orihuela"
strTramo trJ = "Orihuela"
strTramo trK = "Orihuela a Rojales"
strTramo trL = "Guardamar"

---------------------------------------
-- Tipo de datos del modelo del rio
---------------------------------------
data Rio =
  Cini |                        -- estado sin vertidos
  Cpantano Tramo DateTime Rio | -- trasvase de agua de un pantano
  Criego   Tramo DateTime Rio | -- disminucion del caudal por riego
  Cfrutas  Tramo DateTime Rio | -- vertido empresa de frutas o verduras
  Curbanas Tramo DateTime Rio | -- vertido de aguas urbanas
  Crefri   Tramo DateTime Rio   -- vertido de industrias de refrigeracion

---------------------------------------------
-- Calculo del icm a partir de un estado dado
---------------------------------------------
obt_datos tramo date Cini = (0.0,0.0,11.0,caudal tramo)
obt_datos tramo date (Cpantano tramo2 date2 s) |
  (_f,_d,_o,_c) =:= obt_datos tramo date s =
        if tramo==tramo2 && (date2 `less` date)
        then ((_f+3.5)/2.0,(_d+30.0)/2.0,_o,_c*2.0)
        else (_f,_d,_o,_c)
obt_datos tramo date (Criego tramo2 date2 s) |
```

```
  (_f,_d,_o,_c) =:= obt_datos tramo date s =
        if tramo==tramo2 && (date2 `less` date)
        then (_f,_d,_o,_c*0.8)
        else (_f,_d,_o,_c)
obt_datos tramo date (Cfrutas tramo2 date2 s) |
  (_f,_d,_o,_c) =:= obt_datos tramo date s =
        if tramo==tramo2 && (date2 `less` date)
        then (_f+5.0,_d,_o,_c)
        else (_f,_d,_o,_c)
obt_datos tramo date (Curbanas tramo2 date2 s) |
  (_f,_d,_o,_c) =:= obt_datos tramo date s =
        if tramo==tramo2 && (date2 `less` date)
        then (_f,_d,_o-2.0,_c)
        else (_f,_d,_o,_c)
obt_datos tramo date (Crefri tramo2 date2 s) |
  (_f,_d,_o,_c) =:= obt_datos tramo date s =
        if tramo==tramo2 && (date2 `less` date)
        then (_f,_d,_o-1.0,_c)
        else (_f,_d,_o,_c)

-- auxiliar para caudal
caudal trA = 282.0
caudal trB = 433.0
caudal trC = 725.0
caudal trD = 734.0
caudal trE = 741.0
caudal trF = 759.0
caudal trG = 760.0
caudal trH = 790.0
caudal trI = 854.0
caudal trJ = 854.0
caudal trK = 854.0
caudal trL = 871.0

-- funciones de insercion (mantienen la ordenacion por fechas)
pantano tramo date Cini = Cpantano tramo date Cini
pantano tramo date (Cpantano tramo2 date2 s) =
        if (date2 `less` date)
        then Cpantano tramo date (Cpantano tramo2 date2 s)
        else Cpantano tramo2 date2 (pantano tramo date s)
pantano tramo date (Criego tramo2 date2 s) =
```

```
        if (date2 `less` date)
        then Cpantano tramo date (Criego tramo2 date2 s)
        else Criego tramo2 date2 (pantano tramo date s)
pantano tramo date (Cfrutas tramo2 date2 s) =
        if (date2 `less` date)
        then Cpantano tramo date (Cfrutas tramo2 date2 s)
        else Cfrutas tramo2 date2 (pantano tramo date s)
pantano tramo date (Curbanas tramo2 date2 s) =
        if (date2 `less` date)
        then Cpantano tramo date (Curbanas tramo2 date2 s)
        else Curbanas tramo2 date2 (pantano tramo date s)
pantano tramo date (Crefri tramo2 date2 s) =
        if (date2 `less` date)
        then Cpantano tramo date (Crefri tramo2 date2 s)
        else Crefri tramo2 date2 (pantano tramo date s)

riego tramo date Cini = Criego tramo date Cini
riego tramo date (Cpantano tramo2 date2 s) =
        if (date2 `less` date)
        then Criego tramo date (Cpantano tramo2 date2 s)
        else Cpantano tramo2 date2 (riego tramo date s)
riego tramo date (Criego tramo2 date2 s) =
        if (date2 `less` date)
        then Criego tramo date (Criego tramo2 date2 s)
        else Criego tramo2 date2 (riego tramo date s)
riego tramo date (Cfrutas tramo2 date2 s) =
        if (date2 `less` date)
        then Criego tramo date (Cfrutas tramo2 date2 s)
        else Cfrutas tramo2 date2 (riego tramo date s)
riego tramo date (Curbanas tramo2 date2 s) =
        if (date2 `less` date)
        then Criego tramo date (Curbanas tramo2 date2 s)
        else Curbanas tramo2 date2 (riego tramo date s)
riego tramo date (Crefri tramo2 date2 s) =
        if (date2 `less` date)
        then Criego tramo date (Crefri tramo2 date2 s)
        else Crefri tramo2 date2 (riego tramo date s)

frutas tramo date Cini = Cfrutas tramo date Cini
frutas tramo date (Cpantano tramo2 date2 s) =
        if (date2 `less` date)
```

```
        then Cfrutas tramo date (Cpantano tramo2 date2 s)
        else Cpantano tramo2 date2 (frutas tramo date s)
frutas tramo date (Criego tramo2 date2 s) =
        if (date2 ‘less‘ date)
        then Cfrutas tramo date (Criego tramo2 date2 s)
        else Criego tramo2 date2 (frutas tramo date s)
frutas tramo date (Cfrutas tramo2 date2 s) =
        if (date2 ‘less‘ date)
        then Cfrutas tramo date (Cfrutas tramo2 date2 s)
        else Cfrutas tramo2 date2 (frutas tramo date s)
frutas tramo date (Curbanas tramo2 date2 s) =
        if (date2 ‘less‘ date)
        then Cfrutas tramo date (Curbanas tramo2 date2 s)
        else Curbanas tramo2 date2 (frutas tramo date s)
frutas tramo date (Crefri tramo2 date2 s) =
        if (date2 ‘less‘ date)
        then Cfrutas tramo date (Crefri tramo2 date2 s)
        else Crefri tramo2 date2 (frutas tramo date s)

urbanas tramo date Cini = Curbanas tramo date Cini
urbanas tramo date (Cpantano tramo2 date2 s) =
        if (date2 ‘less‘ date)
        then Curbanas tramo date (Cpantano tramo2 date2 s)
        else Cpantano tramo2 date2 (urbanas tramo date s)
urbanas tramo date (Criego tramo2 date2 s) =
        if (date2 ‘less‘ date)
        then Curbanas tramo date (Criego tramo2 date2 s)
        else Criego tramo2 date2 (urbanas tramo date s)
urbanas tramo date (Cfrutas tramo2 date2 s) =
        if (date2 ‘less‘ date)
        then Curbanas tramo date (Cfrutas tramo2 date2 s)
        else Cfrutas tramo2 date2 (urbanas tramo date s)
urbanas tramo date (Curbanas tramo2 date2 s) =
        if (date2 ‘less‘ date)
        then Curbanas tramo date (Curbanas tramo2 date2 s)
        else Curbanas tramo2 date2 (urbanas tramo date s)
urbanas tramo date (Crefri tramo2 date2 s) =
        if (date2 ‘less‘ date)
        then Curbanas tramo date (Crefri tramo2 date2 s)
        else Crefri tramo2 date2 (urbanas tramo date s)
```

```
refri tramo date Cini = Crefri tramo date Cini
refri tramo date (Cpantano tramo2 date2 s) =
        if (date2 `less` date)
        then Crefri tramo date (Cpantano tramo2 date2 s)
        else Cpantano tramo2 date2 (refri tramo date s)
refri tramo date (Criego tramo2 date2 s) =
        if (date2 `less` date)
        then Crefri tramo date (Criego tramo2 date2 s)
        else Criego tramo2 date2 (refri tramo date s)
refri tramo date (Cfrutas tramo2 date2 s) =
        if (date2 `less` date)
        then Crefri tramo date (Cfrutas tramo2 date2 s)
        else Cfrutas tramo2 date2 (refri tramo date s)
refri tramo date (Curbanas tramo2 date2 s) =
        if (date2 `less` date)
        then Crefri tramo date (Curbanas tramo2 date2 s)
        else Curbanas tramo2 date2 (refri tramo date s)
refri tramo date (Crefri tramo2 date2 s) =
        if (date2 `less` date)
        then Crefri tramo date (Crefri tramo2 date2 s)
        else Crefri tramo2 date2 (refri tramo date s)

---------------------------------------------
-- Menu
---------------------------------------------
menu = menu1 Cini

menu1 state =
  putStrLn "" >>
  putStrLn "Menu - Modelo cuenca hidrografica" >>
  putStrLn "p - trasvase de agua del pantano" >>
  putStrLn "r - riego incontrolado con agua del rio" >>
  putStrLn "f - vertido de aguas sin depurar de una empresa de frutas" >>
  putStrLn "u - vertido de aguas urbanas sin depurar" >>
  putStrLn "e - vertido de aguas calientes de refrigeracion de industrias" >>
  putStrLn "i - ver indice calidad mixto" >>
  putStrLn "v - ver valores: Fosfatos DBO5 Oxigeno Caudal" >>
  putStrLn "x - salir" >>
  putChar '?' >> getLine >>=
 (\str ->
   if str=="P" || str=="p"
```

```
   then askTramo _tramo >>
        askDateTime _date >>
        menu1 (pantano _tramo _date state)
   else
   if str=="R" || str=="r"
   then askTramo _tramo >>
        askDateTime _date >>
        menu1 (riego _tramo _date state)
   else
   if str=="F" || str=="f"
   then askTramo _tramo >>
        askDateTime _date >>
        menu1 (frutas _tramo _date state)
   else
   if str=="U" || str=="u"
   then askTramo _tramo >>
        askDateTime _date >>
        menu1 (urbanas _tramo _date state)
   else
   if str=="E" || str=="e"
   then askTramo _tramo >>
        askDateTime _date >>
        menu1 (refri _tramo _date state)
   else
   if str=="I" || str=="i"
   then askTramo _tramo >>
        askDateTime _date >>
        check_icm _tramo _date state >> menu1 state
   else
   if str=="V" || str=="v"
   then askTramo _tramo >>
        askDateTime _date >>
        show_state _tramo _date state >> menu1 state
   else
   if str=="X" || str=="x"
   then done
   else putStrLn "Opcion no valida" >> menu1 state )

show_state tramo date state
 | (_fosfatos,_dbo5,_o2,_caudal) =:= obt_datos tramo date state =
  putStr "Fosfatos:" >> putStr (show _fosfatos) >>
```

```
  putStr " DBO5:"    >> putStr (show _dbo5)     >>
  putStr " Oxigeno:" >> putStr (show _o2)       >>
  putStr " Caudal:"  >> putStr (show _caudal)   >>
  putStrLn ""

check_icm tramo date state |
 (_f,_d,_o,_c) =:= obt_datos tramo date state &
 (_Texto,_ICM) =:= icm _f _d _o =
   if (_c/(caudal tramo)) <= 0.01
   then putStrLn "Caudal a 1%, no se puede calcular ICM"
   else putStr "Calidad: " >> putStrLn _Texto >>
        putStr "Indice Calidad Mixto: " >> putStr (show _ICM) >>  putStrLn "" >>
        putStr "Organismos Indicadores: " >> putStrLn (organismos _ICM)

---------------------------------------------
-- auxiliares para manejo de estados
---------------------------------------------
askTramo t = putStr "Tramo A - " >> putStrLn (strTramo trA) >>
             putStr "Tramo B - " >> putStrLn (strTramo trB) >>
             putStr "Tramo C - " >> putStrLn (strTramo trC) >>
             putStr "Tramo D - " >> putStrLn (strTramo trD) >>
             putStr "Tramo E - " >> putStrLn (strTramo trE) >>
             putStr "Tramo F - " >> putStrLn (strTramo trF) >>
             putStr "Tramo G - " >> putStrLn (strTramo trG) >>
             putStr "Tramo H - " >> putStrLn (strTramo trH) >>
             putStr "Tramo I - " >> putStrLn (strTramo trI) >>
             putStr "Tramo J - " >> putStrLn (strTramo trJ) >>
             putStr "Tramo K - " >> putStrLn (strTramo trK) >>
             putStr "Tramo L - " >> putStrLn (strTramo trL) >>
             putStr "Elige tramo?" >> getLine >>= askTramo1 t

-- es necesario que no sea una func lambda para poder instanciar al tramo
askTramo1 x c |
   (if (c=="a") || (c=="A") then trA else
     if (c=="b") || (c=="B") then trB else
      if (c=="c") || (c=="C") then trC else
       if (c=="d") || (c=="D") then trD else
        if (c=="e") || (c=="E") then trE else
         if (c=="f") || (c=="F") then trF else
          if (c=="g") || (c=="G") then trG else
           if (c=="h") || (c=="H") then trH else
```

```
            if (c=="i") || (c=="I") then trI else
             if (c=="j") || (c=="J") then trJ else
              if (c=="k") || (c=="K") then trK else
               if (c=="l") || (c=="L") then trL else
                 askTramo2 ) =:= x = done

-- es necesario que sea una func para poder usar la forma condicional
askTramo2 | _io=:=askTramo _t = _t

------------------------------------------------------
-- auxiliares para manejo de enteros, reales y fechas
------------------------------------------------------
askInt i = putStr "(entero)? " >> getLine >>= askInt2 i
askInt2 i s | i =:= str2int s 0 = done

askFloat f = putStr "(real)? " >> getLine >>= askFloat2 f
askFloat2 f s | f =:= str2float s = done

askDate d s | d =:= str2date s = done
askTime t s | t =:= str2time s = done

askDateTime eval flex
askDateTime (dtime t d) =
       putStr "(hh:mm:ss)? " >> (getLine >>= askTime t) >>
       putStr "(dd/mm/yy)? " >> getLine >>= askDate d

-- string to integer
str2int [c] n = (n*10) + ((ord c)-(ord '0'))
str2int (c1:c2:cs) n = str2int (c2:cs) ((n*10) + ((ord c1)-(ord '0')))

-- string to float
str2float s
  | (_i,_r) =:= findSym '.' s "" "" =
        if _r==[]
        then (float (str2int _i 0))
        else (float (str2int _i 0))+(str2float2 _r 0.1)

str2float2 [] _ = 0.0
str2float2 (x:xs) Mult =
        (float ((ord x)-(ord '0')))*Mult
          +
```

```
        (str2float2 xs (Mult*0.1))


-- split string by a symbol character
findSym sym [] xs ys = (xs,ys)
findSym sym (c:cs) xs ys =
        if c==sym
        then (xs,ys++cs)
        else findSym sym cs (xs++[c]) ys

-- string to date
str2date str
 | findSym '/' str [] [] =:= (_day,_rest) &
   findSym '/' _rest [] [] =:= (_month,_year)
 = dt (str2int _day 0) (str2int _month 0) (str2int _year 0)

-- string to time
str2time str
 | findSym ':' str [] [] =:= (_hour,_rest) &
   findSym ':' _rest [] [] =:= (_min,_sec)
 = tm (str2int _hour 0) (str2int _min 0) (str2int _sec 0)
```

5 Conclusiones

En este informe hemos mostrado al lector el resultado de la modelización del comportamiento del ecosistema del río Segura bajo el enfoque de la programación declarativa. Esperamos que la especificación ejecutable del modelo de la cuenca hidrográfica del río Segura desarrollada en el lenguaje Curry constituya una herramienta útil para ayudar a la concienciación social de los problemas que afectan al río Segura, así como para la inclusión de posibles mejoras y adaptaciones de dicho modelo a sus necesidades específicas. La especificación ejecutable resultante sirve como precedente para futuros desarrollos orientados a potenciar las ventajas del uso de lenguajes lógico funcionales integrados.

6 Bibliografía

[AN89] H. Aït-Kaci and R.Ñasr. Integrating Logic and Functional Programming. *Lisp and Symbolic Computation*, 2(1):51–89, 1989.

[AP90] J.M. Andreoli and R. Pareschi. Linear Objects: logical processes with built-in inheritance. In D.H.D. Warren and P. Szeredi, editors, *Proc. of Seventh Int'l Conf. on Logic Programming*, pages 495–510. The MIT Press, Cambridge, MA, 1990.

[AR92] M. Alpuente and M.J. Ramírez. An Equational Constraint Logic Approach to Conceptual Modelling. In 3^{rd} *Int'l Conf. on Database and Expert Systems Aplications DEXA'92*, pages 26–32. Springer Verlag, Wien New York, 1992.

[Bac78] J. Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Comm. of the ACM*, 21(8):613–641, 1978.

[Bal85] K. Balzer. A 15 Year Perspective on Automatic Programming. In *IEEE Transactions on Software Engineering*, volume 2, pages 1257–1268, 1985.

[Bar90] H. Barendregt. λ-calculus and Functional Programming. In *J. van Leeuwen, ed., Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 321–363. Elsevier, Amsterdam and The MIT Press, Cambridge, MA, 1990.

[BGLM94] A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The s-semantics approach: Theory and applications. *Journal of Logic Programming*, 19-20:149–197, 1994.

[BKL+91] M. Bidoit, H.J. Kreowski, P. Lescanne, F. Orejas, and D. Sannella. *Algebraic System Specification and Development*. Springer LNCS 501, 1991.

[BL86] M. Bellia and G. Levi. The relation between logic and functional languages. *Journal of Logic Programming*, 3:217–236, 1986.

[BLM94] M. Bugliesi, E. Lamma, and P. Mello. Modularity in logic programming. *Journal of Logic Programming*, 19/20:443–502, 1994.

[CC87] J. Catalán Lafuente and J. M. Catalán Alonso. *Ríos: Caracterización y calidad de sus aguas*. Dihidrox, 1987.

[CRLFSH97a] R. Caballero Roldán, F.J. López Fraguas, and J. Sánchez-Hernández. User's Manual for Toy. Technical Report SIP-9757, UCM, 1997.

[CRLFSH97b] R. Caballero-Roldán, F.J. López-Fraguas, and J. Sánchez-Hernández. User's manual for Toy. Technical Report SIP-5797, UCM, Madrid (Spain), April 1997.

[Dav92] A.J.T. Davie. An Introduction to Functional Programming Using Haskell. *Cambridge University Press*, 1992.

[DJ90] N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 243–320. Elsevier, Amsterdam, 1990.

[DL86] D. DeGroot and G. Lindstrom, editors. *Logic Programming, Functions, Relations and Equations.* Prentice Hall, Englewood Cliffs, NJ, 1986.

[DP88] J. Darlington and H. Pull. A Program Development Methodology Based on a Unified Approach to Execution and Transformation. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Proc. of the Int'l Workshop on Partial Evaluation and Mixed Computation*, pages 117–131. North-Holland, Amsterdam, 1988.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science.* Springer, 1985.

[FH88] A.J. Field and P.G. Harrison. Functional Programming. *Addison-Wesley*, 1988.

[GTW78] J.A. Goguen, J.W. Thatcher, and E.G. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In R. Yeh, editor, *Current Trends in Programming Methodology*, volume 4, pages 80–149. Prentice Hall, Englewood Cliffs NJ, 1978.

[HAK+98] M. Hanus, S. Antoy, H. Kuchen, F.J. López-Fraguas, and F. Steiner. Curry An Integrated Functional Logic Language (November 1998). Available at `http://www-i2.informatik.rwth-aachen.de/~hanus/curry`.

[Han91] M. Hanus. Efficient Implementation of Narrowing and Rewriting. In *Proc. Int'l Workshop on Processing Declarative Knowledge*, pages 344–365. Springer LNAI 567, 1991.

[Han94] M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[Han97a] M. Hanus. Teaching functional and logic programming with a single computation model. In *Proc. Ninth International Symposium on Programming Languages, Implementations, Logics, and Programs (PLILP'97)*, pages 335–350. Springer LNCS 1292, 1997.

[Han97b] M. Hanus. A unified computation model for functional and logic programming. In *Proc. 24st ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 80–93, 1997.

[HKMN95] M. Hanus, H. Kuchen, and J.J. Moreno-Navarro. Curry: A truly functional logic language. In *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, pages 95–107, 1995.

[HL94] P. Hill and J.W. Lloyd. *The Gödel Programming Language.* The MIT Press, Cambridge, MA, 1994.

[HPJW92] P. Hudak, S. Peyton-Jones, and P. Wadler. Report on the Functional Programming Language Haskell: a non–strict, purely functional language. *Sigplan Notices*, 27(5), 1992.

[Ibo96] M. B. Iborra Botía. Viaje por el Segura. Master's thesis, Proyecto Fin de Curso para optar al título de Master en Educación Ambiental de la Universidad Nacional de Educación a Distancia y la Fundación Universidad-Empresa, 1996.

[JM94] J. Jaffar and M.J. Maher. Constraint Logic Programming: a Survey. *Journal of Logic Programming*, 19-20:503–581, 1994.

[Kap87] S. Kaplan. Simplifying Conditional Term Rewriting Systems: Unification, Termination and Confluence. *Journal of Symbolic Computation*, 4:295–334, 1987.

[Kow74] R. Kowalski. Predicate logic as a programming language. In *Proc. IFIP '74*, pages 569–574. North-Holland, 1974.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming.* Springer, second, extended edition, 1987.

[Llo95] J.W. Lloyd. Declarative Programming in Escher. Technical report cstr-95-013, University of Bristol, 1995.

[Mes93] J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In *G. Agha, P. Wegner, and A. Yonezawa, eds., Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. The MIT Press, Cambridge, MA, 1993.

[Mil89] D. Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, pages 79–108, 1989.

[MR88] J.J. Moreno-Navarro and M. Rodríguez-Artalejo. BABEL: A Functional and Logic Programming Language based on a constructor discipline and narrowing. In I. Grabowski, P. Lescanne, and W. Wechler, editors, *Proc. of the Int'l Conf. on Algebraic and Logic Programming*, volume 343 of *Lecture Notes in Computer Science*, pages 223–232. Springer-Verlag, Berlin, 1988.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of ML.* The MIT Press, 1990.

[NM88] G.Ñadathur and D. Miller. An Overview of λProlog. In R.A. Kowalski and K.A. Bowen, editors, *Proc. of Fifth Int'l Conf. on Logic Programming ICLP'88*, pages 810–827. The MIT Press, Cambridge, MA, 1988.

[O'D77] M. J. O'Donnell. *Computing in Systems Described by Equations*, volume 58 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1977.

[O'D85] M. J. O'Donnell. *Equational Logic as a Programming Language.* The MIT Press, Cambridge, MA, 1985.

[O'D94] M. J. O'Donnell. Introduction: Logic and Logic Programming Languages. In *D. Gabbay, ed., Handbook of Logic in Artificial Intelligence and Logic Programming, Vol. 5 on Logic Programming, Chapter 1.* Oxford Science Publications, 1994.

[OK85] R. O' Keefe. Towards an algebra for constructing logic programs. In *Proc. IEEE Symp. on Logic Programming*, pages 152–160. IEEE Press, NY, 1985.

[PJ87] S.L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[PJW93] S.L. Peyton Jones and P. Wadler. Imperative functional programming. In *Proc. 20th Symposium on Principles of Programming Languages (POPL'93)*, pages 71–84, 1993.

[PvE93] R. Plasmeijer and M. van Eekelen. *Functional Programming and Parallel Graph Rewriting.* Addison-Wesley, 1993.

[Rea93] C. Reade. Elements of Functional Programming. *Addison-Wesley*, 1993.

[Red86] U.S. Reddy. On the relationship between logic and functional languages. In *[DL86]*, pages 3–36, 1986.

[Smo95] G. Smolka. The Oz programming model. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, pages 324–343. Springer LNCS 1000, 1995.

[VF85] P. Veloso and A. Furtado. Towards simpler and yet Complete Formal Specifications. In *Information systems: Theoretical and Formal aspects.* North-Holland, Amsterdam, 1985.

[VH89] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[VH92] P. Van Hentenryck. Constraint Satisfaction using Constraint Logic Programming. In *Artificial Intelligence*, volume 58, pages 113–159, 1992.

[Wir90] M. Wirsing. Algebraic Specification. In *J. van Leeuwen, ed., Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 674–788. Elsevier, Amsterdam and The MIT Press, Cambridge, MA, 1990.