mente, proceder a su verificación mediante técnicas formales (e.g., *model checking*). Es en este contexto, donde el programador debe abordar el desarrollo de aplicaciones de varios miles de líneas—o incluso de varias decenas de miles de líneas—, cuando las técnicas de fragmentación pueden ser realmente útiles, prácticamente imprescindibles (como ya lo son en el ámbito de los lenguajes imperativos). Por ello, pretendemos abordar la adaptación de dichas técnicas al caso del lenguaje multi-paradigma Curry y, posteriormente, proceder al desarrollo de las herramientas software asociadas, las cuales se integrarán en alguno de los entornos de programación existentes (e.g., en el entorno PAKCS [HAK+00] del lenguaje Curry). Cabe destacar que los componentes del grupo de investigación tienen una amplia experiencia en el desarrollo de herramientas software; en particular, el entorno PAKCS incluye una herramienta de especialización de componentes software [AHV02a] desarrollada por el equipo de la Universidad Politécnica de Valencia en colaboración con el grupo de investigación liderado por Michael Hanus, el principal impulsor y coordinador del lenguaje Curry.

No cabe duda que el desarrollo de nuevas técnicas formales en informática puede revertir beneficiosamente en las empresas de la Comunidad Valenciana, mejorando así el nivel competitivo de las mismas. Son muchas las empresas que hoy en día hacen uso de métodos formales en el desarrollo de software con el objeto de mejorar los tiempos de producción y, especialmente, la fiabilidad del software desarrollado. En este marco se engloban los objetivos prioritarios del presente proyecto.

2 El lenguaje multi-paradigma Curry

El lenguaje de programación Curry [Han00c] es un lenguaje declarativo moderno que integra características de los paradigmas de programación declarativa más populares: la programación lógica, funcional y concurrente. El origen del lenguaje es relativamente reciente (1995) y se puede considerar, hoy en día, la referencia estándar en el área de la programación declarativa multi-paradigma. Existen otros lenguajes de características similares (e.g., Toy [LS99] o Escher [Llo95], entre otros), pero no gozan de la misma popularidad y/o no disponen de entornos de desarrollo tan completos como en el caso del lenguaje Curry.

Curry dispone de facilidades para el desarrollo de aplicaciones de un tamaño considerable gracias al uso de tipos, módulos y búsqueda encapsulada. Desde un punto de vista sintáctico, un programa Curry es muy similar a un programa funcional "puro"; la principal diferencia se encuentra en la posibilidad de incluir variables *libres* (i.e., variables lógicas) en las condiciones y en las partes derechas de las ecuaciones que definen las funciones del programa. El lenguaje sigue la sintaxis del lenguaje funcional Haskell [Pet97], i.e., las variables y los nombres de función comienzan en minúscula y los nombres de los tipos y los constructores de datos en mayúscula; la aplicación de una función f sobre un argumento e se denota simplemente "$f\ e$" (i.e., sigue una notación *currificada*). Por ejemplo, el siguiente fragmento de un programa Curry define la función "`qsort`" que implementa el conocido algoritmo de ordenación *quicksort*:

```
split _ [] = ([],[])
```

```
split e (x:xs) | e>=x  = (x:l,r)
               | e<x   = (l,x:r)
               where (l,r) = split e xs

qsort []     = []
qsort (x:xs) = let (l,r) = split x xs
               in qsort l ++ (x:qsort r)
```

La función auxiliar "`split`" se emplea para dividir una lista en dos sublistas con los elementos menores y mayores, respectivamente, de un valor dado. Las variables cuyo nombre no resulta relevante—las variables "anónimas" según la terminología del lenguaje Prolog—se representan con un subrayado. La segunda ecuación de la función "`split`" es una ecuación *condicional*. En general, las ecuaciones condicionales tienen la siguiente forma:

$$\begin{array}{lcl} func\ t_1 \ldots t_n & | & cond_1 = exp_1 \\ & | & cond_2 = exp_2 \\ & & \ldots \\ & | & cond_m = exp_m \end{array}$$

Cuando se realiza una llamada a la función "$func$", se intentan satisfacer las condiciones; si alguna condición tiene éxito, se devuelve la expresión correspondiente. Asimismo, Curry permite la definición local de funciones (y enlaces) mediante las construcciones

`let` $decls$ `in` exp $\qquad$ exp `where` $decls$

donde "exp" es una expresión arbitraria y "$decls$" es un conjunto de declaraciones (enlazando variables a valores o bien definiendo una función local).

En general, un programa Curry consta de un conjunto de ecuaciones definiendo las funciones del programa y los tipos de datos sobre los que operan dichas funciones. Las funciones se evaluan de forma "perezosa" (*lazy*), i.e., el paso de parámetros en las llamadas a función es siempre por nombre; los argumentos de una llamada a función sólo se evalúan si éstos son realmente necesarios para computar el resultado final. Con el objeto de proporcionar a Curry todo el poder de la programación lógica, las funciones se pueden invocar con un conjunto parcialmente instanciado de argumentos. El comportamiento de una función cuando se invoca con argumentos variables depende de las *anotaciones de evaluación* asignadas a cada función del programa:

- Si una función está anotada como "rígida" (*rigid*), entonces la computación procede de una forma puramente funcional, i.e., la función sólo se podrá evaluar si los argumentos están lo suficientemente instanciados como para poder ser emparejados con alguna parte izquierda de las ecuaciones que definen la función. En otro caso, la computación se "suspende" (hasta que las variables tomen un valor, típicamente en el marco de una ejecución concurrente).

- Si la función está anotada como "flexible" (*flex*), entonces la computación procederá instanciado las variables libres a los valores adecuados de forma indeterminista. El principio de ejecución subyacente recibe el nombre de "estrechamiento" (*narrowing*) y es la base de los lenguajes declarativos que integran características de la programación lógica y funcional [Han94a]. La posibilidad de definir funciones flexibles le proporciona al lenguaje Curry la potencia expresiva de los lenguajes lógicos, incluyendo el uso de variables lógicas y la búsqueda de soluciones a un objetivo dado.

Por defecto, todas las restricciones (i.e., funciones cuyo resultado es de tipo `Success`) son flexibles y el resto rígidas. Esta asignación por defecto se puede modificar fácilmente incluyendo anotaciones en el programa.

El siguiente programa Curry define, en las dos primeras líneas, los tipos de datos para los valores booleanos y para las listas (polimórficas). A continuación, aparecen las ecuaciones que definen las funciones "`conc`"—para concatenar dos listas—y "`last`"—para calcular el último elemento de una lista:

```
data Bool   = True | False
data List a = []   | a : List a

conc :: [a] -> [a] -> [a]
conc eval flex

conc []     ys = ys
conc (x:xs) ys = x : conc xs ys

last xs | conc ys [x] =:= xs  = x   where x,ys free
```

La definición del tipo de datos "`Bool`" nos indica que sólo existen dos posible valores: `True` y `False`. El tipo de datos (recursivo) "`List a`" define las listas de elementos del tipo (variable) `a`; se trata, por tanto, de un tipo de datos polimórfico. Por ejemplo, "`List Bool`" denotaría el tipo de datos que contiene todas las posibles listas cuyos elementos son valores booleanos. En Curry, las listas se pueden denotar siguiendo una notación funcional, `x : xs`, o bien una notación lógica, `[x|xs]`; en ambos casos, `x` representa el elemento en cabeza de la lista y `xs` la lista que contiene el resto de elementos. A continuación, encontramos la *declaración* (opcional) de la función "`conc`". En este caso, se indica el perfil de la función (currificado) y se anota como una función flexible. Las siguientes dos líneas definen la función `conc` de manera recursiva. Por último, la función `last` se define haciendo uso de una ecuación condicional, en la que la condición es una restricción que debe entenderse de la siguiente forma: "devolver el valor `x` si es posible encontrar una lista `ys` tal que al concatenarla con la lista `[x]` obtenemos el argumento de la llamada a la función, `xs`, donde `x` e `ys` son dos variables libres". En este caso, la restricción "`conc ys [x] =:= xs`" tiene éxito (únicamente) si instanciamos la variable `ys` al mismo valor de la lista `xs` excepto el último elemento y la variable `x` al último elemento de la lista `xs`.

Los siguientes apartados exponen brevemente las principales facilidades y extensiones existentes en el lenguaje Curry.

2.1 Programación concurrente

El lenguaje Curry dispone del operador concurrente "&" sobre restricciones, e.g., una expresión de la forma "c_1 & c_2 & ... c_n" denota que las restricciones $c_1, c_2, \ldots, c_n$ deben evaluarse de manera concurrente. En principio, se intenta la evaluación de las restricciones de izquierda a derecha. Si la evaluación de alguna de ellas no puede proceder (porque aparece una llamada insuficientemente instanciada a una función rígida), entonces se continúa con la evaluación de las restantes restricciones; una vez la restricción suspendida está lo suficientemente instanciada, su evaluación puede reanudarse. Por tanto, Curry sigue un modelo de concurrencia basado en la sincronización de restricciones usando variables lógicas, de forma similar a los lenguajes lógicos con restricciones concurrentes [Sar93].

Por ejemplo, el siguiente programa (concurrente) [Han00c] implementa el conocido problema de coloreado de un mapa, de forma que los países con una frontera común, tengan un color diferente (los comentarios comienzan con "--"):

```
-- tipos de datos para los colores
data Color = Rojo | Verde | Amarillo | Azul

-- la funcion (flexible) isColor permite generar los colores
isColor :: Color -> Success
isColor Rojo     = success
isColor Verde    = success
isColor Amarillo = success
isColor Azul     = success

-- coloring genera todos los posibles coloreados para l1, l2, l3 y l4
coloring :: Color -> Color -> Color -> Color -> Success
coloring l1 l2 l3 l4 = isColor l1 & isColor l2 & isColor l3 & isColor l4

-- correct comprueba que el coloreado sea correcto, i.e., que no
-- hayan dos paises adyacentes con el mismo color
correct :: Color -> Color -> Color -> Color -> Success
correct l1 l2 l3 l4
   = diff l1 l2 & diff l1 l3 & diff l2 l3 & diff l2 l4 & diff l3 l4

-- diff implementa la desigualdad (i.e., diff x y = not (x == y))
diff :: a -> a -> Success
diff x y = (x == y) =:= False

-- el mapa tiene la siguiente forma:
--
```

```
--  --------------------------
--  |       |        |       |
--  |       |   L2   |       |
--  |       |        |       |
--  |  L1   |--------|  L4   |
--  |       |        |       |
--  |       |   L3   |       |
--  |       |        |       |
--  --------------------------

-- solucion estilo Prolog:
goal1 l1 l2 l3 l4 = coloring l1 l2 l3 l4 & correct l1 l2 l3 l4

-- solucion mas eficiente:
goal2 l1 l2 l3 l4 = correct l1 l2 l3 l4 & coloring l1 l2 l3 l4
```

El primer objetivo, "`goal1 l1 l2 l3 l4`", genera correctamente todas las posibilidades, pero no resulta muy eficiente. Se trata de la típica solución estilo "generación y prueba", donde primero se van generando todas las posibilidades y, luego, se procede a comprobar la corrección de cada una de ellas, descartando las que no sean correctas. Por tanto, el espacio de búsqueda es muy grande. Por el contrario, el segundo objetivo, "`goal2 l1 l2 l3 l4`", implementa una estrategia de "prueba y generación", la cual es mucho más eficiente. Concretamente, la evaluación de la restricción "`correct l1 l2 l3 l4`" se suspende inicialmente, con lo cual comienza a ejecutarse la restricción "`coloring l1 l2 l3 l4`". Sin embargo, a diferencia del objetivo anterior, tan pronto como disponemos de un valor para `l1` y `l2`, se reactiva la primera restricción y, si el coloreado no es correcto, la computación en curso se aborta y comienza la búsqueda de nuevas posibilidades. De este modo, los posibles coloreados se generan a medida que son demandados por la función `correct` y son eliminados tan pronto como es posible comprobar que es imposible formar un coloreado correcto.

El operador concurrente "&" es un elemento básico para la creación de una red de actividades concurrentes. Sin embargo, esta primitiva es demasiado limitada para permitir la implementación de aplicaciones distribuidas *realistas*.

2.2 Programación distribuida

Las aplicaciones distribuidas requieren, a menudo, modelar un número variable (de manera dinámica) de procesos concurrentes con estructuras de comunicación "muchos–a–uno". En este caso, se necesita la combinación de los flujos de mensajes procedentes de diferentes procesos en un único flujo de mensajes. Si realizamos esta combinación mediante una función, aparecen toda una serie de problemas (ver [Han99, JMH93]). Por ello, [JMH93] propuso la utilización de *puertos* para el lenguaje lógico concurrente AKL. El modelo ha sido recientemente generalizado en [Han99] para cubrir las características del lenguaje Curry. En principio, un puerto es una restricción entre un multiconjunto y

un flujo de mensajes que se satisface si el multiconjunto y el flujo contienen lo mismos elementos (mensajes).

Los puertos se crean mediante la función "`openPort p s`", donde `p` y `s` son variables lógicas libres. Esta función crea un multiconjunto y un flujo y los combina sobre un puerto. Entonces, es posible añadir nuevos elementos en el multiconjunto y el flujo enviándolos a `p` mediante la función "`send m p`". Cuando un mensaje se envía a `p`, se añade automáticamente al flujo `s` de modo que se satisfaga la condición anterior (i.e., que `p` y `s` contienen los mismos elementos). Para permitir la implementación de aplicaciones distribuidas, donde los procesos se ejecutan en máquinas diferentes, es posible hacer que los puertos sean accesibles externamente asignándoles un nombre simbólico. Por ejemplo, la acción (`openNamedPort` "*name*") abre un puerto externo cuyo nombre simbólico es "*name*" y devuelve un flujo de mensajes de entrada a dicho puerto. Si el puerto se ha abierto en una máquina m, entonces los clientes pueden acceder al puerto ejecutando la acción (`connectPort` "*name*@*m*"). Esta acción nos devolverá un puerto `p` al cuál enviar los mensajes. En particular, los mensajes pueden contener variables libres, las cuales proporcionan un mecanismo de alto nivel para devolver valores por instanciación (en lugar de crear "canales de respuesta"). El concepto de puerto resulta imprescindible para la extensión del lenguaje Curry con características de orientación a objetos, como veremos en el siguiente apartado.

El ejemplo que presentamos a continuación muestra una aplicación distribuida sencilla en la cual el servidor pone un contandor a disposición de los clientes. El contador reacciona a tres tipos de eventos: `Set v`, para inicializar el contador al valor `v`; `Inc`, para incrementar su valor en una unidad; y `Get x`, para consultar el valor actual del contador:

```
-- tipos de datos para los mensajes
data CounterMessage = Set Int | Inc | Get Int

-- implementacion del contador (servidor)
counter eval rigid     -- el contador es rigido porque actua como un
                       -- consumidor (de mensajes)
counter _ (Set v : ms) = counter v ms
counter n (Inc   : ms) = counter (n+1) ms
counter n (Get v : ms) = v=:=n & counter n ms
counter _ []           = success

-- arranque del contador (servidor)
counter_server =
  do stream <- openNamedPort "counter"
     doSolve (counter 0 stream)

-- envio de mensajes (cliente)
cc msg =
  do port <- connectPort "counter@xxx.dsic.upv.es"
     doSend msg port
```

```
-- ejemplos de mensajes (cliente)
cinc   = cc Inc
cset n = cc (Set n)
cget x = cc (Get x)
```

2.3 Programación orientada a objetos

Es bien conocido de la programación lógica concurrente [ST83] que los objetos pueden implementarse fácilmente como predicados que procesan un flujo de mensajes de entrada. El estado interno del objeto se puede implementar como un parámetro, el cual puede cambiar en las sucesivas llamadas recursivas que se producen mientras se procesa el flujo de mensajes. Puesto que las restricciones juegan el papel de predicados en el lenguaje Curry, los objetos se consideran como funciones cuyo resultado es de tipo `Success`. Estas funciones toman como entrada el estado actual del objeto y un flujo de mensajes. Si el flujo no está vacío, la función "objeto" se llama recursivamente con un nuevo estado, el cual dependerá del primer elemento del flujo de mensajes. Así, el tipo general de un objeto tiene la siguiente forma:

$$\texttt{obj} :: \texttt{State} \rightarrow [\texttt{MessageType}] \rightarrow \texttt{Success}$$

donde `State` es el tipo del estado interno del objeto y `MessageType` es el tipo de los mensajes que acepta el objeto. En general, se deberá definir un nuevo tipo de datos algebraico para los mensajes de cada objeto. Por ejemplo, la función "`counter`" de la aplicación mostrada en el apartado anterior se puede ver como un objeto.

Aunque efectiva, la técnica esbozada tiene una serie de problemas, especialmente cuando el estado del objeto viene determinado por un número elevado de variables, ya que el programador debe repetir el estado completo en las llamadas recursivas. Esto motivó la introducción de una sintaxis especial para la definición de los patrones de objetos en Curry, dando lugar a la extensión del lenguaje conocida como ObjectCurry [HHN01]. Se emplea la palabra "patrón" en lugar de la más común "clase" para evitar la confusión entre las clases de un lenguaje orientado a objetos y las clases (de tipos) en el lenguaje Haskell. Por ejemplo, un patrón para el objeto `counter` se puede definir en ObjectCurry como sigue:

```
template Counter =

constructor
   counter init = x := init

methods
   Inc   =  x := x + 1
   Set s =  x := s
   Get v =  v =:= x
```

La definición de un patrón comienza con la palabra reservada "`template`" seguida del nombre del patrón. De forma similar a la declaración de un tipo de datos, el nombre

del patrón es su propio tipo. El constructor del patrón es una función que se emplea para definir nuevas instancias de un objeto. La parte izquierda de la definición de un constructor tiene la forma de una función estándar; la parte derecha, sin embargo, consta de un conjunto de asignaciones describiendo los atributos del objeto y sus valores iniciales. Los mensajes aceptados por el objeto se definen en el apartado "`methods`"; cada método se define de forma similar a la función constructora. La única diferencia consiste en que los métodos pueden contener restricciones en la parte derecha, con el objeto de permitir la respuesta a un mensaje instanciando una variable lógica del mismo.

La creación de nuevas instancias de un objeto se realiza mediante la función "`new`", cuyo perfil es el siguiente:

```
new :: Constructor a → Object a → Success
```

La función toma como entrada una función constructora y una variable libre, y tiene como efecto enlazar la variable libre con una nueva instancia del patrón. Los mensajes se pueden enviar al nuevo objeto mediante la primitiva "`send`", cuyo perfil es

```
send :: Message a → Object a → Success
```

Por ejemplo, la evaluación de la siguiente expresión enlazaría la variable `v` al valor 42:

```
new (counter 41) o
& (send Inc o &> send (Get v) o &> send Stop o)
```

ObjectCurry también dispone de un mecanismo de herencia. Un patrón puede heredar los atributos y métodos de otro patrón permitiendo, además, que se modifique la definición de los métodos o se añadan nuevos atributos. Por ejemplo, podemos definir un nuevo patrón "`maxCounter`" que herede el atributo `x` y los métodos `Inc`, `Set` y `Get` del patrón `Counter` de la siguiente forma:

```
template MaxCounter extends Counter =

constructor
   maxCounter init maxInit = counter init
                             max := maxInit

methods
   Inc           = x := (if x < max then x+1 else x)
   SetMax newMax = max := newMax
                   x := (if x<max then x else max)
```

En este caso, hemos introducido un nuevo atributo, `max`, que representa el límite máximo del contador. El método `Inc` se redefine para evitar que se incremente `x` por encima del límite y, adicionalmente, se añade un nuevo método, `SetMax`, para permitir la modificación del límite máximo del contador.