Universidad Nacional de Rosario

Tesina de grado
Licenciatura en Ciencias de la Computación

Generación de monitores C embebidos para especificaciones Lola

Autora:
Aldana Ramirez

Director:
Dr. César Sánchez
Co-Director:
Lic. Martín A. Ceresa

Departamento de Ciencias de la Computación
Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Av. Pellegrini 250, Rosario, Santa Fe, Argentina

12 de junio de 2022

Resumen

Los sistemas embebidos se utilizan en la actualidad en diversas áreas de aplicación, desde dispositivos hogareños como televisores y lavarropas, a drones y equipos médicos de alta complejidad. Si bien en todos los casos se busca que el funcionamiento del sistema sea correcto, en el caso de los sistemas críticos resulta imprescindible proveer garantías de correctitud.

Una manera de lograrlo es mediante técnicas de verificación formal, que permiten demostrar matemáticamente propiedades de correctitud de un modelo. Sin embargo, a pesar de los enormes avances en la materia, continúa siendo impráctico, y en algunos casos incluso imposible, aplicar este tipo de métodos formales sobre sistemas completos de escala industrial.

La verificación en tiempo de ejecución es un área de investigación activa que estudia cómo generar formalmente monitores a partir de especificaciones declarativas. En lugar de demostrar formalmente la correctitud del sistema para cualquier ejecución posible, se analiza la traza de una (única) ejecución para verificar su correctitud. Este enfoque resigna completitud para proveer un mecanismo formal práctico que permite obtener información precisa del comportamiento del sistema en tiempo de ejecución.

En los últimos 30 años se han estudiado diversos enfoques y técnicas en el área de verificación en tiempo de ejecución, con diferentes niveles de expresividad y dirigidas a distintos dominios de aplicación. Tal es el caso de Lola, un lenguaje de especificación junto con algoritmos para el monitoreo de sistemas síncronos. Desde su publicación en 2005, han surgido numerosos trabajos basados en este lenguaje. Uno de los más recientes es hLola, un lenguaje de dominio específico embebido en Haskell para escribir especificaciones basado en Lola junto con un motor para ejecutar los monitores. Esta implementación presenta varias características atractivas, entre las que se destaca la extensibilidad del lenguaje a nuevos tipos de datos. hLola ha sido usado con éxito en diversas áreas de aplicación, incluyendo vehículos aéreos no tripulados. Sin embargo, en el contexto de sistemas críticos como en el caso de la aviación, existen regulaciones que prohíben el uso de garbage collectors y otras características intrínsecas al sistema de ejecución de Haskell.

A partir de estos antecedentes y de la experiencia de uso del lenguaje C en sistemas críticos, esta tesina presenta un prototipo, llamado mcLola, para la síntesis de monitores C a partir de especificaciones de alto nivel basadas

en el lenguaje Lola. Concretamente, mcLola se ha implementado como un lenguaje de dominio específico embebido en Haskell.

Agradecimientos

Quiero agradecer a César Sánchez, Martín Ceresa y Felipe Gorostiaga por toda su ayuda, sus ideas y su dedicación para este proyecto. Ha sido un gran placer trabajar con ellos.

A los profesores de la LCC y a mis compañeros por todo el tiempo compartido. Me llevo muchos lindos recuerdos de los años de cursado.

Por último, a mi familia por el apoyo durante todos estos años. En especial a mis padres, mi hermana y mi novio. Y por supuesto, a nuestra querida Samy por su compañía en tantas jornadas de estudio.

Índice general

Capítulo 1

Introducción

En los últimos años se ha incrementado notablemente el interés en el uso y desarrollo de *vehículos aéreos no tripulados* (*UAVs*, *Unmanned Aerial Vehicles*) [55, 24]. Se trata de vehículos sin tripulación que se desplazan por vías aéreas de manera autónoma o mediante control remoto para llevar a cabo una tarea específica. Su creciente popularidad puede atribuirse a múltiples factores, entre los que se destaca la posibilidad de realizar tareas típicamente peligrosas o fuera del alcance físico de los humanos. Algunas de sus aplicaciones recientes incluyen la extinción de incendios [66], actividades de control urbano [81] y el monitoreo de obras de ingeniería civil [52].

Uno de los componentes fundamentales en el software para vehículos no tripulados es un sistema de monitoreo confiable, que permita recolectar información sobre el sistema en ejecución. Esta información es usada para llevar a cabo tanto tareas de verificación y diagnóstico (por ejemplo, verificar que determinada propiedad se cumple en el sistema o detectar violaciones de la misma), como de planeamiento (por ejemplo, hacer análisis estadísticos o generar predicciones sobre el comportamiento del sistema).

La *verificación en tiempo de ejecución* (*RV, Runtime Verification*) [3, 50] es un área de investigación activa que estudia cómo generar formalmente monitores a partir de especificaciones declarativas. A diferencia de las técnicas de verificación formal clásicas que demuestran matemáticamente propiedades de correctitud de un modelo, RV se ocupa de analizar la traza correspondiente a una (única) ejecución del sistema. De esta manera, resigna completitud para proveer un mecanismo formal práctico para obtener información precisa del comportamiento del sistema en tiempo de ejecución, aplicable a sistemas de escala industrial.

hLola [9, 36] es un lenguaje de dominio específico embebido en Haskell que ofrece una solución integral para este paradigma. Provee un lenguaje para escribir especificaciones basado en el lenguaje Lola [19], junto con un engine temporal para ejecutar los monitores. hLola presenta varias características atractivas; entre ellas, mantiene por separado las dependencias temporales de

las computaciones de datos, facilitando la extensión de nuevos tipos de datos y el cálculo estático de recursos espaciales y temporales necesarios para la evaluación de los monitores. hLola ha sido usado con éxito en diversas áreas, incluyendo el desarrollo de misiones complejas con drones y UAVs [82].

Sin embargo, muchas regulaciones de aviación prohíben el uso de garbage collectors y otras características intrínsecas al sistema de ejecución de Haskell. Considerando, además, ciertas características del lenguaje C atractivas para el desarrollo de sistemas críticos, como la eficiencia, la portabilidad y la experiencia de uso [57], resulta de interés trabajar en el desarrollo de un prototipo para generar código C embebido a partir de una especificación Lola.

Objetivo

En este trabajo se desarrolla un lenguaje de dominio específico embebido en Haskell para la generación automática de monitores escritos en C99 a partir de especificaciones de alto nivel basadas en el lenguaje Lola. En particular, con el objeto de aportar seguridad al código generado, se siguieron los lineamientos del estándar MISRA-C:2012 [58].

Además, se ha estudiado cómo extender el lenguaje con nuevas teorías de datos y cómo incorporar características adicionales, como la parametrización de streams, el uso de librerías y la anticipación en el cómputo de resultados.

La herramienta mcLola, que implementa el lenguaje de dominio específico desarrollado en el presente trabajo, puede accederse libremente desde el repositorio *Git* `https://github.com/imdea-software/McLola`.

Organización del trabajo

Este documento realiza un reporte general del proyecto. Puntualmente, se optó por explicar los principales desafíos encontrados haciendo algunas abstracciones para focalizar la atención en las técnicas utilizadas para su resolución. Por este motivo, los fragmentos de código presentados en este informe son parciales y pueden diferir levemente con el código fuente del proyecto.

El resto del documento se estructura según se describe a continuación. En el capítulo 2 se discuten los conceptos relevantes al problema en cuestión y se repasa brevemente el estado del arte. En el capítulo 3 se describe el lenguaje Lola, sobre el que se basa esta tesina. El capítulo 4 ofrece una introducción a mcLola, comentando brevemente los componentes principales del proyecto y explicando cómo usar la herramienta. Los capítulos 5 y 6 abordan la implementación del proyecto, comenzando por el diseño del lenguaje de dominio específico y siguiendo con la generación de monitores C. En el capítulo 7 se presenta una característica añadida a la herramienta para anticipar algunos resultados durante la ejecución de los monitores. El capítulo 8 presenta un

caso de aplicación proveniente de UAVs, mediante el cual se explica cómo extender el lenguaje con nuevas teorías de datos. Finalmente, en el capítulo 9 se discuten las conclusiones y trabajos futuros que se derivan de esta tesina.

Capítulo 2

Conceptos Previos y Estado del Arte

Este capítulo presenta un repaso del marco teórico en el que se sitúa esta tesina. Concretamente, se realiza una breve introducción a *Runtime Verification* y a *Stream Runtime Verification* y se discute brevemente el estado del arte en el área.

2.1. Runtime Verification

La *verificación en tiempo de ejecución* (*RV*, *Runtime Verification*) [3, 50] es un método formal liviano que analiza el comportamiento de un sistema procesando una única traza de ejecución. En la actualidad, se aplican técnicas de RV, tanto durante como después del desarrollo de sistemas, para llevar a cabo diferentes tareas, incluyendo verificación, control, testing y depuración.

En RV se genera un monitor a partir de una especificación formal que describe el comportamiento a analizar, para luego evaluarlo con una traza del sistema, obteniendo así un veredicto, como ilustra la Figura 2.1.

Las técnicas de RV comprenden tres tareas fundamentales:

1. Especificar las propiedades que describen el comportamiento a analizar usando un *lenguaje de especificación*.
2. Generar (automáticamente) un *monitor* a partir de la especificación.
3. Conectar el monitor al sistema, lo que se conoce como *instrumentación* y comprende el manejo y comunicación de eventos para que el monitor reciba la traza a analizar.

A diferencia de las técnicas de verificación estática, como *model checking* [10], que se centran en demostrar que toda ejecución del sistema satisface una determinada propiedad, RV sólo se ocupa de analizar una única traza finita. Este enfoque resigna completitud pero provee un mecanismo formal práctico

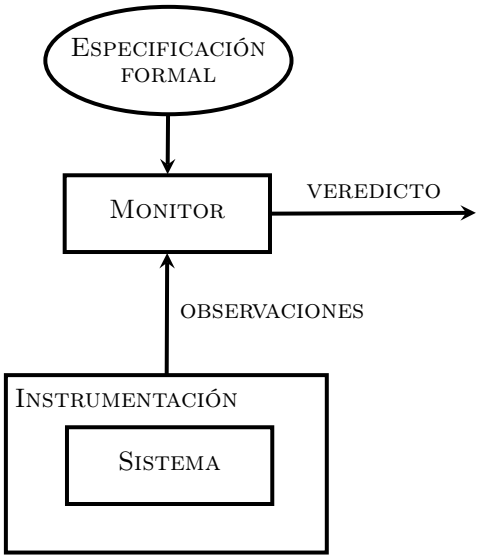


Figura 2.1: Esquema general de RV

para obtener información precisa del comportamiento del sistema en tiempo de ejecución, que puede combinarse con técnicas de testing y depuración [3, 50].

Típicamente, una especificación describe propiedades de correctitud. Las primeras técnicas de RV se basan en lógicas temporales como LTL [53], seguidas por otros enfoques basados en expresiones regulares [70], lenguajes basados en reglas [2], o reescritura [68]. Todas estas técnicas provienen de la verificación estática donde la decidibilidad es crucial para obtener soluciones algorítmicas a problemas de decisión y, por lo tanto, tienen la limitación de que las observaciones y los veredictos deben ser exclusivamente valores booleanos.

Posteriormente han surgido enfoques más expresivos para contemplar otros análisis más complejos, como es el caso de los análisis estadísticos. En lo que sigue se discuten brevemente algunos de ellos.

2.2. Stream Runtime Verification

La *verificación en tiempo de ejecución por secuencias o flujos* (*SRV*, *Stream Runtime Verification*) [19, 69] es un formalismo de RV que parte de la observación de que la mayoría de los algoritmos de monitoreo provenientes de las técnicas tradicionales de RV pueden ser generalizados a datos más complejos, si se provee una separación estricta entre el cómputo de las dependencias temporales y de las operaciones puntuales que manipulan los datos en cada paso. Concretamente, SRV propone especificar monitores mediante dependencias explícitas entre los valores del flujo de entrada (que representan las observaciones

del sistema) y los valores del flujo de salida (que representan los veredictos del monitor), manteniendo por separado los cómputos temporales de los de datos. De esta manera, SRV permite especificar análisis cuantitativos y estadísticos e incluso incorporar tipos de datos arbitrarios.

Lola [19] es el lenguaje de especificación para sistemas síncronos [1] pionero en SRV. Una especificación Lola es un conjunto de ecuaciones que relacionan flujos de entrada y flujos de salida mediante un constructor que provee acceso explícito a instantes de tiempo pasados y futuros. Si bien originariamente Lola fue usado para testing de hardware síncrono [19], su uso se ha hecho extensivo a otras áreas de aplicación [9, 82, 20].

2.2.1. Estado del Arte

Una de las primeras herramientas más similares a Lola es Eagle [2]. Eagle permite describir monitores usando el menor y mayor punto fijo de definiciones recursivas. Difieren en su naturaleza descriptiva y en el hecho de que Eagle está restringido a fórmulas lógicas, mientras que Lola también trabaja con valores numéricos.

TeSSLa [15] y Striver [34] son dos herramientas modernas de SRV, diseñadas para sistemas de tiempo real. TeSSLa ofrece un conjunto reducido de primitivas para expresar dependencias entre secuencias pero con la particularidad de permitir eventos con etiquetas temporales. Su dominio de aplicación es el monitoreo de sistemas ciberfísicos. Striver ofrece un lenguaje similar a Lola, que generaliza los supuestos temporales para incluir secuencias de eventos de tiempo real, manteniendo las referencias explícitas a instantes de tiempo. Está dirigido a testing y monitoreo de sistemas en la nube y monitoreo de hardware multinúcleo. Estos lenguajes, sin embargo, soportan un número limitado de tipos de datos.

hLola [9, 36] es una implementación reciente de Lola, en la forma de lenguaje de dominio específico embebido en Haskell. Si bien SRV se fundamenta en la separación estricta entre el cómputo de dependencias temporales y el de teorías de datos, las herramientas de SRV suelen proveer una implementación *ad-hoc* de un conjunto reducido de tipos de datos, requiriendo demasiados cambios para incorporar nuevas teorías. Por el contrario, hLola es fácilmente extensible a tipos de datos arbitrarios, lo cual, combinado con otras características que obtiene directamente del lenguaje anfitrión, como la parametrización estática y el sistema de módulos, permite escribir fácilmente librerías para distintos dominios de aplicación. Por ejemplo, hLola ha sido usado con éxito para el monitoreo del vuelo de drones y UAVs [82]. Sin embargo, en el campo de aplicación de los UAVs, algunas regulaciones aéreas prohíben el uso de garbage collectors, característica intrínseca de Haskell.

[1] Los sistemas síncronos son aquellos cuyas operaciones son coordinadas o controladas por un reloj global.

Copilot [62] implementa SRV mediante un lenguaje de dominio específico embebido en Haskell que genera monitores C que operan en tiempo y espacio constante. Sin embargo, Copilot no ofrece acceso explícito a instantes de tiempo y, en particular, es un lenguaje *causal*, es decir que no permite referencias futuras.

RV y SRV comparten algunas características con otras áreas de investigación afines. Tal es el caso de los lenguajes síncronos como Lustre [35], Esterel [6] y Signal [29]. Estos lenguajes también se basan en flujo de datos mediante dependencias funcionales entre las secuencias de entrada y salida, pero a diferencia de SRV son lenguajes causales ya que su objetivo es describir comportamientos del sistema, no observaciones ni monitores. En el paradigma de *Programación Funcional Reactiva* (*FRP*, *Functional Reactive Programming*), los programas describen un paso de cómputo como una reacción ante nuevos eventos de entrada, estableciendo de esta manera una dependencia implícita entre valores de entrada y salida. Nuevamente, la mayor diferencia con SRV es que FRP no provee referencias explícitas al tiempo ni permite especificaciones no causales.

Capítulo 3

El lenguaje LOLA

En este capítulo se describe la sintaxis y semántica de LOLA [19]. Además, se explica un algoritmo de monitoreo online para sistemas síncronos y se identifica sintácticamente una clase de especificaciones para la cual los requerimientos espaciales del algoritmo son independientes de la longitud de la traza.

Este capítulo está basado en el paper *Online and Offline Stream Runtime Verification of Synchronous Systems* [69]. Para más información, incluyendo las demostraciones de teoremas y corolarios que no se incluyen en este informe, se sugiere consultar dicha publicación.

3.1. Sintaxis

Una especificación LOLA describe la computación de un conjunto de *streams de salida* en términos de un conjunto de *streams de entrada*.

Un *stream* de tipo T es una secuencia finita de valores de tipo T, que representa los valores de una variable a lo largo del tiempo. Dado un stream σ de longitud N, denotamos con $\sigma(i)$, para $0 \leq i < N$, al valor de σ en la posición i, que representa el valor de σ en el instante de tiempo i.

Para describir las computaciones correspondientes a los streams de salida, se utilizan teorías interpretadas de primer orden para representar dominios de datos. Una teoría se define por una colección finita $\mathcal{T}$ de tipos y una colección finita $\mathcal{F}$ de símbolos de función u operadores. Las teorías son interpretadas en el sentido que cada tipo $T \in \mathcal{T}$ tiene asociado un dominio D_T de valores y cada símbolo $f \in \mathcal{F}$ tiene asociada una función computable que, dado elementos en el dominio de los tipos de sus argumentos, computa un valor en el dominio del tipo del resultado. Por ejemplo:

- La teoría *Boolean* utiliza el tipo $Bool$, asociado con el dominio $\{\top, \bot\}$, y los símbolos de función $true$ y $false$ constantes de tipo $Bool$, $\neg$ de tipo $Bool \to Bool$, $\wedge$ y $\vee$ de tipo $Bool \times Bool \to Bool$, con sus interpretaciones usuales.

- La teoría *Naturals* está conformada por dos tipos: Nat y $Bool$, asociados a los dominios $\{0, 1, ...\}$ y $\{\top, \bot\}$, respectivamente. A los símbolos de función de la teoría *Boolean* se suman las constantes 0, 1, ... de tipo Nat, los símbolos $+$, $*$, etc. de tipo $Nat \times Nat \to Nat$ y $\leq$, $<$, etc. de tipo $Nat \times Nat \to Bool$, todos ellos con sus interpretaciones usuales.

En particular, asumimos que todas las teorías usan el tipo $Bool$ y están provistas con los símbolos $=$ de tipo $T \times T \to Bool$ e $if \cdot then \cdot else$ de tipo $Bool \times T \times T \to T$, correspondientes a la comparación por igualdad y a la estructura condicional *if-then-else*.

En lo que sigue se asume implícitamente una teoría $(\mathcal{T}, \mathcal{F})$ definida por tipos $T_1, ..., T_n$ y operadores $f_1, ..., f_m$.

Definición 3.1 (Expresiones de streams). *Dado un conjunto finito Z de variables de streams, el conjunto de expresiones de streams sobre Z se define como el menor conjunto $Expr(Z)$ tal que:*

- *(function app) Si $f : T_1 \times T_2 \times ... \times T_k \to T$ es un símbolo de función k-ario y $e_i \in Expr(Z)$ es una expresión de stream de tipo T_i, para $1 \leq i \leq k$, entonces $f(e_1, e_2, ..., e_k) \in Expr(Z)$ y decimos que $f(e_1, e_2, ..., e_k)$ es una expresión de stream de tipo T.*

- *(now) Si $s \in Z$ es una variable de stream de tipo T, luego $s \in Expr(Z)$ y decimos que s es una expresión de stream de tipo T.*

- *(offset) Si $s \in Z$ es una variable de stream de tipo T, c es una constante de tipo T y k es un valor entero, entonces $s[k, c] \in Expr(Z)$ y decimos que $s[k, c]$ es una expresión de stream de tipo T.*

Mientras que la regla *(function app)* permite introducir en el lenguaje las teorías de datos, *(now)* y *(offset)* introducen referencias temporales. Informalmente, una expresión $s[k, c]$ refiere al valor del stream s con un corrimiento de k posiciones respecto a la posición o instante de tiempo actual i (es decir, $s(i + k)$), tomando a c como valor por defecto cuando $i + k$ se exceda del final del stream o se anticipe al principio; mientras que s refiere al valor del stream s en la posición o instante de tiempo actual. Por ejemplo, $u[-1|true]$ refiere a la posición previa del stream u, tomando el valor $true$ cuando u no tiene una posición previa, es decir cuando $u[-1|true]$ se evalúa en la posición 0.

Una especificación LOLA utiliza expresiones de streams para describir la relación entre streams de entrada y de salida.

Definición 3.2 (Especificación). *Una especificación* LOLA $\varphi = \langle I, O, E \rangle$ *consiste de:*

- *Un conjunto finito I de variables de streams tipadas, que llamamos variables de streams de entrada.*

- *Un conjunto finito O de variables de streams tipadas, que llamamos variables de streams de salida.*
- *Un conjunto E de expresiones de streams, con exactamente una expresión $E_y \in Expr(I \cup O)$ para cada $y \in O$, donde E_y e y son del mismo tipo. Decimos que E_y es la expresión que define a y.*

Las variables de streams de entrada representan las entradas del sistema, por lo cual se modelan como variables libres. Las variables de streams de salida representan los veredictos del sistema, con lo cual sus valores se computan a partir de las expresiones que los definen. De esta manera, computar veredictos no es otra cosa que resolver un sistema de ecuaciones.

En lo que sigue, nos referiremos a las variables de streams y expresiones de streams simplemente como variables y expresiones, respectivamente. Asimismo, en algunas ocasiones usaremos la palabra stream para denotar a la variable que representa a un stream. Por último, para simplificar la escritura de especificaciones, introducimos la siguiente notación:

- Para indicar que x es un stream de entrada de tipo T escribimos:

  ```
  input T x
  ```

- Para indicar que y es un stream de salida de tipo T definido por la expresión e escribimos:

  ```
  output T y = e
  ```

De esta manera, una especificación Lola se define por un conjunto de declaraciones **input** y un conjunto de declaraciones **output**.

Ejemplo 3.3
A continuación se presenta como ejemplo una especificación Lola para la teoría *Integer* (conformada por los tipos *Bool* e *Int* correspondientes a los valores booleanos y enteros) con streams de entrada x_1, x_2, x_3 y streams de salida $y_1, y_2, ..., y_7$.

```
input  int  x1
input  bool x2
input  int  x3
output bool y1 = true
output int  y2 = x1
output int  y3 = x3 + 3
output bool y4 = if x1 < y3 then x2 else ¬ x2
output int  y5 = x1[-1|5]
output bool y6 = x2[1|false]
output int  y7 = y7[-1|0] + if (x1 > 0) then 1 else 0
```

La variable y_1 denota un stream cuyo valor es *true* en todas las posiciones, mientras que y_2 denota un stream cuyo valor coincide con el de x_1 en todas las

posiciones, e y_3 coincide en cada posición con el valor de x_3 incrementado en 3. El valor de y_4 en la posición i es igual al de x_2 si en la posición i el valor de x_1 es menor que el de y_3, y es la negación de x_2 en caso contrario. La variable y_5 se corresponde con un stream cuyo valor en la posición i coincide con el valor de x_1 en la posición $i-1$, a excepción de la primera posición ($i=0$) donde vale 5. Análogamente, el valor y_6 en la posición i coincide con el valor de x_2 en la posición $i+1$, salvo en la última posición donde vale $false$. Por último, y_7 cuenta la cantidad de valores positivos en x_1. ◇

3.2. Semántica

Para dar semántica al lenguaje LOLA, primero se requiere definir cómo evaluar expresiones.

Definición 3.4 (Interpretación). *Sea $\varphi = \langle I, O, E\rangle$ una especificación* LOLA *y $N \in \mathbb{N}$. Decimos que σ es una interpretación de φ de tamaño N si σ es un mapeo que asigna a cada variable $v \in I \cup O$ un stream σ_v de tamaño N y mismo tipo que v.*

Definición 3.5 (Valuación). *Dada una interpretación σ de tamaño N, una valuación es un mapeo $[\![\cdot]\!]$ que asigna a cada expresión de stream un stream de tamaño N y del mismo tipo que la expresión según se describe a continuación:*

$$
\begin{array}{lcl}
[\![f(e_1,...,e_k)]\!](i) & = & f([\![e_1]\!](i),...,[\![e_k]\!](i)) \\
[\![v]\!](i) & = & \sigma_v(i) \\
[\![v[k,c]]\!](i) & = & \begin{cases} [\![v]\!](i+k) & si\ 0 \le i+k < N \\ c & en\ otro\ caso \end{cases}
\end{array}
$$

Ahora, podemos definir cuándo una interpretación σ es un modelo de evaluación para φ, lo que nos permite describir una semántica denotacional para especificaciones LOLA.

Definición 3.6 (Modelo de evaluación). *Una interpretación σ de una especificación $\varphi = \langle I, O, E\rangle$ es un modelo de evaluación para φ siempre que para todo $y \in O$ se verifique que:*

$$[\![y]\!] = [\![E_y]\!]$$

En este caso, notamos $\sigma \models \varphi$.

Como se muestra en el siguiente ejemplo, una vez fijada una interpretación para las variables de entrada, una especificación LOLA puede tener cero, uno o múltiples modelos de evaluación.

Ejemplo 3.7
Considérense las siguientes especificaciones LOLA, todas con un único stream

de entrada de tipo **int** y un único stream de salida de tipo **bool**:

φ_1:
```
input  int  x
output bool y = x > 0
```

φ_2:
```
input  int  x
output bool y = y
```

φ_3:
```
input  int  x
output bool y = ¬ y
```

Para cualquier stream de entrada σ_x, φ_1 tiene un único modelo de evaluación: $\{x \mapsto \sigma_x, y \mapsto \sigma_y\}$, donde $\sigma_y(i) = true$ si y sólo si $\sigma_x(i) > 0$ para $0 \leq i < N$. Por su parte, la especificación φ_2 tiene múltiples modelos de evaluación para cualquier stream de entrada σ_x. En particular, para cualquier σ_x, tanto $\sigma_y = \langle true, true, ..., true \rangle$ como $\sigma_y = \langle false, false, ..., false \rangle$ hacen de $\{x \mapsto \sigma_x, y \mapsto \sigma_y\}$ un modelo de evaluación para φ_2. Por último, no existen modelos de evaluación para la especificación φ_3, ya que no existe solución para la ecuación $\sigma_y(i) = \neg\sigma_y(i)$. ◇

El propósito del lenguaje de especificación Lola es definir monitores que analizan las observaciones del sistema en estudio (streams de entrada) para computar una *única* respuesta o veredicto (streams de salida). De esta forma, el objetivo es representar mediante una especificación a una función que mapea valores de entrada en valores de salida, para lo cual se requiere que para cualquier instancia de streams de entrada exista un único modelo de evaluación. La siguiente definición formaliza este concepto.

Definición 3.8 (Especificación bien definida). *Una especificación* Lola $\varphi = \langle I, O, E \rangle$ *está bien definida si para cualquier mapeo* σ_I *de variables en* I *a streams de tamaño* $N > 0$*, existe un único mapeo* σ_O *de variables en* O *a streams de tamaño* N *tal que* $\sigma_I \cup \sigma_O \models \varphi$*.*

La propiedad de ser bien definida es una condición semántica difícil de verificar en el caso general. Por ello, se define a continuación una condición sintáctica más restrictiva, pero fácilmente verificable.

Definición 3.9 (Grafo de dependencias). *Sea* $\varphi = \langle I, O, E \rangle$ *una especificación* Lola*. El grafo de dependencias de* φ *es un multigrafo dirigido ponderado* $D = (V, A)$ *donde:*

- *El conjunto* V *de vértices está dado por* $I \cup O$*.*
- *Para cada* $s \in O$ *y* $v \in I \cup O$*, el conjunto* A *contiene una arista* $s \xrightarrow{0} v$ *si* v *ocurre en* E_s *y una arista* $s \xrightarrow{k} v$ *si* $v[k|c]$ *ocurre en* E_s*, para algún* c*.*

El grafo de dependencias de una especificación describe sus dependencias temporales. Una arista $s \xrightarrow{k} v$ indica que el valor de $s(i)$ depende del valor de $v(i + k)$ [1]. Por transitividad, $s(i)$ depende de $v(i + k)$ si existe un camino de peso k de s a v. De esta manera, si s forma parte de un ciclo de peso k

[1] Salvo que el corrimiento corresponda a una posición que exceda los límites del stream.

entonces los valores de $s(i)$ dependen de $s(i+k)$, y si $k = 0$ tendremos valores que dependen de sí mismos, con lo cual puede ocurrir que no exista un único modelo de evaluación. El siguiente teorema garantiza que en ausencia de ciclos de peso 0, la especificación está bien definida.

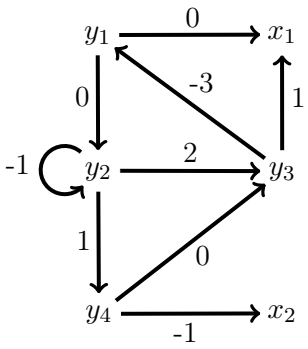
Definición 3.10 (Especificación bien formada). *Una especificación* LOLA *está bien formada si su grafo de dependencias no tiene ciclos de peso cero.*

Teorema 3.11. *Toda especificación* LOLA *bien formada está bien definida.*

Observar que es posible verificar eficientemente si una especificación está bien formada. En particular, basta con aplicar un algoritmo DFS sobre su grafo de dependencias.

Ejemplo 3.12
Considérese la siguiente especificación LOLA y su grafo de dependencias.

```
input  int  x1
input  bool x2
output int  y1 = x1 + y2
output int  y2 = y2[-1|0] + if y3[2|true] then y4[1|0] else 0
output bool y3 = y1[-3|0] < x1[1|1]
output int  y4 = if x2[-1|false] then y3 else y3+1
```



Los únicos ciclos del grafo son $y_1 \to y_2 \to y_3 \to y_1$, $y_1 \to y_2 \to y_4 \to y_3 \to y_1$ e $y_2 \to y_2$, con pesos -1, -2 y -1, respectivamente. Entonces, la especificación está bien formada y, por el Teorema 3.11, está bien definida. ◇

Observar que, en general, no vale el recíproco del Teorema 3.11. Basta considerar, por ejemplo, la siguiente especificación:

```
input  bool x
output bool y = y ∨ true
```

3.3. Algoritmo de monitoreo

Es posible distinguir dos situaciones diferentes para el monitoreo de sistemas: el monitoreo *online* y el monitoreo *offline*. En el primero, las trazas correspondientes a las observaciones del sistema (los streams de entrada) se reciben y procesan mientras el sistema está en ejecución, mientras que en el segundo las entradas se almacenan para ser analizadas una vez finalizada la ejecución del sistema. En lo que sigue, se describe un algoritmo de monitoreo online para especificaciones Lola, en el que se basa la implementación de este proyecto. Las modificaciones requeridas para trabajar en sistemas offline no fueron incluídas en este trabajo, pero pueden consultarse en [69].

3.3.1. Algoritmo

Sea $\varphi = \langle I, O, E \rangle$ una especificación Lola y sea i la posición o instante de tiempo actual, hasta el cual se encuentran disponibles todos los valores de entrada. Notamos con v^j a la variable posicional que denota al valor correspondiente al stream representado por la variable v en la posición j, es decir $v(j)$.

El algoritmo de monitoreo mantiene dos conjuntos de ecuaciones:

- El conjunto de *ecuaciones resueltas* R, formado por pares (v^j, c) que indican que el valor de la variable v en la posición j ya fue determinado y es c, es decir $v(j) = c$.

- El conjunto de *ecuaciones no resueltas* U, formado por pares (y^j, e) que indican que a y^j le corresponde la expresión e (posiblemente parcialmente resuelta).

Notar que si $(v^j, c) \in R$ es porque o bien $v \in I$ (con $j \leq i$) o bien $v \in O$ y ya se pudo resolver su correspondiente ecuación. Por otra parte, si $(v^j, e) \in U$, necesariamente $v \in O$ (de otro modo su valor estaría determinado) y e contiene alguna referencia temporal a un stream (de otro modo sólo tendría constantes y hubiese sido posible resolverlo).

El algoritmo de monitoreo online para especificaciones Lola se exhibe en el Algoritmo 1.

Algoritmo 1 Algoritmo de monitoreo online - Parte 1

```
1: procedure MONITOR
2:     R, U ← ∅, ∅
3:     i ← 0
4:     while nuevo input do
5:         R ← R ∪ {(x^i, σ_x(i)) | para cada x ∈ I}
6:         U ← U ∪ {(y^i, INST(E_y, i)) | para cada y ∈ O}
7:         PROPAGATE
8:         PRUNE(i)
9:         i ← i + 1
10:    N ← i
11:    FINALISE(N)
12: end procedure
```

En primer lugar se inicializan los conjuntos R y U como conjuntos vacíos y la variable i con 0 para marcar el inicio de la ejecución. Seguidamente, se repite el bucle principal del algoritmo (líneas 4-9) hasta que ya no se reciban nuevos valores de entrada.

En cada iteración del bucle principal se procede de la siguiente manera:

1. Se reciben los valores correspondientes a los streams de entrada en la posición actual y se agregan a R.

2. Se instancian en la posición actual las expresiones que definen a los streams de salida y se agregan a U, para lo cual se utiliza el procedimiento INST (Algoritmo 2).

3. Se propagan los nuevos valores (v^j, c) conocidos de R, substituyendo por c todas las ocurrencias de v^j en las expresiones no resueltas de U y simplificando la expresión resultante, como especifica el procedimiento PROPAGATE (Algoritmo 2). En algunos casos estas expresiones se resuelven (línea 34), actualizándose R y U en consecuencia (líneas 36-37).

4. Se invoca el procedimiento PRUNE (Algoritmo 3) para eliminar de R los valores que ya no serán necesarios para computar nuevos valores. La definición de este procedimiento se explica más adelante.

5. Se incrementa en 1 el valor de i.

Para concluir, el algoritmo MONITOR invoca al procedimiento FINALISE (Algoritmo 2), que determina si una expresión offset que aún no ha sido resuelta referencia a una posición que excede la longitud de la traza, en cuyo caso la substituye por su valor por defecto. Notar que el valor por defecto a utilizar es almacenado por INST (línea 25), ya que en el momento de la instanciación no es posible determinar si $k + j$ estará en rango; esto se determina luego de

k pasos cuando el valor aparece (línea 32) o bien cuando FINALISE detecta que se excede del límite (línea 43). Finalmente, se realiza un último llamado a PROPAGATE (línea 46) para terminar de computar los valores que quedaron en U. El Teorema 3.14 enunciado más adelante, garantiza que si φ está bien formada el algoritmo MONITOR termina y computa todos los valores de salida.

Algoritmo 2 Algoritmo de monitoreo online - Parte 2

```
13: procedure INST(e, j)
14:     switch e do
15:         case c :
16:             return c
17:         case f(e_1, ..., e_n) :
18:             return f(INST(e_1, j), ..., INST(e_n, j))
19:         case v :
20:             return v^j
21:         case v[k|c] :
22:             if k + j < 0 then
23:                 return c
24:             else
25:                 return v_c^{k+j}          ▷ Eventualmente retorna v^{k+j} o c
26: end procedure
27:
28: procedure PROPAGATE
29:     repeat
30:         change ← false
31:         for all (v^k, e) ∈ U do            ▷ Intenta resolver todo v^k ∈ U
32:             e' ← simplify(subst(e, R))     ▷ Aplica nuevos valores y simplifica
33:             U.replace(v^k, e')             ▷ Actualiza v^k
34:             if e' es valor then            ▷ Resolvió v^k
35:                 change ← true
36:                 R ← R ∪ {(v^k, e')}
37:                 U ← U \ {(v^k, e')}
38:     until ¬change
39: end procedure
40:
41: procedure FINALISE(N)
42:     for all (v^k, e) ∈ U do
43:         for all u_c^l subtérmino de e con l ≥ N do
44:             e ← subst(e, {(u^l, c)})       ▷ Substituye u_c^l por c (en e)
45:         U.replace(v^k, e)
46:     PROPAGATE
47: end procedure
```

Para completar la definición de MONITOR queda pendiente definir el procedimiento PRUNE. A continuación se muestra cómo eliminar de R información que ya no resulta necesaria para computar nuevos valores, mediante la introducción del concepto de *back-reference*.

Definición 3.13 (Back-reference). *Dada una especificación φ con grafo de dependencias $D = (V, E)$, el back-reference de un vértice $v \in V$, denotado ∇v, se define como:*

$$\nabla v \;=\; max\left(\{0\} \cup \left\{k \;\middle|\; u \xrightarrow{-k} v \in E\right\}\right)$$

Intuitivamente, el back-reference de un stream representa la máxima cantidad de instantes de tiempo que sus valores necesitan ser recordados, lo que permite definir el procedimiento PRUNE como se muestra en el Algoritmo 3.

Algoritmo 3 Algoritmo de monitoreo online - Parte 3

```
48: procedure PRUNE(j)
49:     for all (v^k, c) ∈ R do
50:         if ∇v + k ≤ j then                ▷ v^k ya no es necesario
51:             R ← R \ {(v^k, c)}
52: end procedure
```

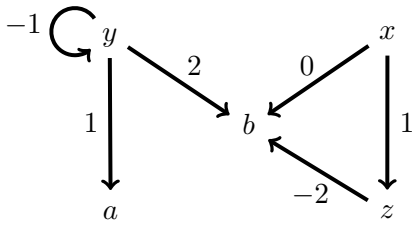
Teorema 3.14. *Sea $\varphi = \langle I, O, E\rangle$ una especificación* LOLA *y sea σ_I una instancia de streams de entrada de longitud N. Si φ está bien formada, entonces el Algoritmo 1 computa el único modelo de evaluación para φ correspondiente a σ_I. Es decir que el algoritmo finaliza, computando correctamente el valor de y^j para cada $y \in O$, $0 \leq j < N$.*

Ejemplo 3.15

Considérese la siguiente especificación LOLA:

```
input int a
input int b
output bool x = b < z[1|5]
output int  y = y[-1|0] + a[1|1] + b[2|0]
output int  z = b[-2|0] + 1
```

con su grafo de dependencias y valores de back-reference:



s	∇s
a	0
b	2
x	0
y	1
z	0

Para los streams de entrada $\sigma_a = \langle 5, 2, 1, 4\rangle$ y $\sigma_b = \langle 3, 1, 2, 8\rangle$, el Algoritmo 1 computará en cada paso los siguientes conjuntos R y U, donde se muestran tachados los valores eliminados por PRUNE:

	0	1	2	3
R	~~$(a^0, 5)$~~ $(b^0, 3)$ ~~$(z^0, 1)$~~	$(b^0, 3)$ ~~$(a^1, 2)$~~ $(b^1, 1)$ ~~$(z^1, 1)$~~ ~~$(x^0, false)$~~	~~$(b^0, 3)$~~ $(b^1, 1)$ ~~$(a^2, 1)$~~ $(b^2, 2)$ ~~$(z^2, 4)$~~ ~~$(y^0, 4)$~~ ~~$(x^1, true)$~~	~~$(b^1, 1)$~~ $(b^2, 2)$ ~~$(a^3, 4)$~~ $(b^3, 8)$ ~~$(z^3, 2)$~~ ~~$(y^1, 13)$~~ ~~$(x^2, false)$~~
U	$(x^0, 3 < z_5^1)$ $(y^0, 0 + a_1^1 + b_0^2)$	$(y^0, 0 + 2 + b_0^2)$ $(x^1, 1 < z_5^2)$ $(y^1, y_0^0 + a_1^2 + b_0^3)$	$(y^1, 4 + 1 + b_0^3)$ $(x^2, 2 < z_5^3)$ $(y^2, y_0^1 + a_1^3 + b_0^4)$	$(y^2, 13 + 4 + b_0^4)$ $(x^3, 8 < z_5^4)$ $(y^3, y_0^2 + a_1^4 + b_0^5)$

Como $\nabla a = \nabla x = \nabla z = 0$, los valores de a, x y z pueden ser eliminados de R en la misma iteración en la que son calculados, mientras que los valores de y y b deben conservarse por 1 y 2 instantes, respectivamente. En particular, los valores de y son eliminados en la iteración en la que son computados porque se resuelven con dos instantes de retraso.

Por último, el procedimiento FINALISE reemplaza por sus respectivos valores por defecto aquellas variables que refieran a posiciones mayores o iguales a la longitud de los streams (en este caso, 4) e invoca a PROPAGATE para resolver los cómputos pendientes:

	FINALISE	
	valores por defecto	PROPAGATE
R	$(b^2, 2)$ $(b^3, 8)$	$(b^2, 2)$ $(b^3, 8)$ $(y^2, 17)$ $(x^3, false)$ $(y^3, 18)$
U	$(y^2, 13 + 4 + 0)$ $(x^1, 8 < 5)$ $(y^3, y_0^2 + 1 + 0)$	$\emptyset$

$\diamond$

3.3.2. Especificaciones eficientemente monitorizables

En el caso general, la complejidad temporal y espacial del algoritmo MONITOR (Algoritmo 1) son lineales en la longitud de la traza[2], como ocurre en el siguiente ejemplo.

Ejemplo 3.16
La siguiente especificación LOLA computa un stream de salida y cuyo valor en todas las posiciones coincide con el valor correspondiente a la última posición del stream de entrada x.

```
input  int  x
output bool f    = false
output bool last = f[1|true]
output int  y    = if last then x else y[1|0]
```

Para cualquier instancia de valores de entrada σ_x de tamaño N, las ecuaciones:

$$(y^j, \texttt{if } last^j \texttt{ then } x^j \texttt{ else } y_0^{j+1})$$

se mantendrán en U hasta el instante N, momento en el cual FINALISE reemplazará en la expresión asociada a y^{N-1} la subexpresión y_0^N por 0, a partir de lo cual PROPAGATE podrá resolver y^{N-1}, y^{N-2}, ..., y^0. Por lo tanto, los requerimientos espaciales del Algoritmo 1 son lineales en la longitud de la traza. ◇

Al tener que analizar cada una de las posiciones de los streams, no es posible mejorar la complejidad lineal para el caso temporal. Sin embargo, no siempre es necesario almacenar en memoria todos los valores de los streams al mismo tiempo y, en particular, hacerlo trae problemas a la hora de implementar un monitor ejecutable. Si asumimos que las entradas serán de gran tamaño, resulta fundamental asegurar que los requerimientos espaciales sean independientes de la longitud de la traza. La siguiente definición captura este concepto.

Definición 3.17 (Especificación eficientemente monitorizable). *Una especificación* LOLA *es eficientemente monitorizable si el espacio en memoria requerido por el algoritmo* MONITOR *(Algoritmo 1) es independiente de la longitud de la traza.*

Al igual que ocurre con el concepto de especificación bien definida, la propiedad de ser eficientemente monitorizable es una condición semántica difícil de verificar en el caso general. En lo que sigue se presenta una condición sintáctica, basada en el grafo de dependencias, que garantiza que una especificación es eficientemente monitorizable.

Definición 3.18 (Especificación acotada a futuro). *Una especificación* LOLA *bien formada es acotada a futuro si su grafo de dependencias no contiene ciclos de peso positivo.*

[2] Asumiendo que cualquier valor puede almacenarse en un sólo registro y que el costo de toda función está acotado por una constante.

Observar que la propiedad de ser acotada a futuro, al igual que la de ser bien formada, puede verificarse eficientemente mediante un algoritmo DFS.

Definición 3.19 (Latencia). *La latencia de un stream s, que notamos Δs, se define como el máximo peso no negativo de un camino en el grafo de dependencias que comience en el vértice s:*

$$\Delta s \;=\; max\left(\{0\} \cup \left\{ \sum_{i=1}^{n} k_i \;\middle|\; s \xrightarrow{k_1} u_1, u_1 \xrightarrow{k_2} u_2, \ldots, u_{n-1} \xrightarrow{k_n} u_n \in E \right\}\right)$$

Notar que la latencia sólo está bien definida si la especificación es acotada a futuro. Intuitivamente, la latencia de un stream s nos da una cota superior para la cantidad de instantes de tiempo que deben transcurrir desde i para asegurar que la variable s^i pueda ser resuelta. El siguiente teorema formaliza esta noción.

Teorema 3.20. *Sea $\varphi = \langle I, O, E\rangle$ una especificación* Lola. *Para todo stream $y \in O$, la ecuación $(y^j, \text{Inst}(E_y, j))$ será resuelta por* Monitor *(Algoritmo 1) a lo sumo en el instante $j + \Delta y$.*

Ejemplo 3.21
Considérese la especificación del Ejemplo 3.15. La siguiente tabla muestra los valores de latencia de cada uno de sus streams:

s	Δs
a	0
b	0
x	1
y	2
z	0

Mientras que a^i, b^i y z^i fueron computados en el instante i, x^i fue computado en el instante $i + 1$ e y^i en el instante $i + 2$ (salvo y^3 calculado en el instante 4 por Finalise). ◇

Observar que si φ es una especificación acotada a futuro, su grafo de dependencias no tiene ciclos de peso positivo y, por tanto, todo stream s de φ tendrá latencia finita. Por lo tanto, sólo una cantidad acotada de ecuaciones (s^k, e) estarán en U al mismo tiempo, con lo cual los requerimientos espaciales resultan independientes del tamaño de la traza.

Corolario 3.22. *Toda especificación* Lola *acotada a futuro es eficientemente monitorizable.*

De esta manera, si una especificación Lola es acotada a futuro, entonces para cualquier instancia de valores de entrada existe un único modelo de evaluación y el algoritmo Monitor podrá computarlo eficientemente.

Capítulo 4

Introducción a mcLola

Este capítulo presenta una descripción general de mcLola. Para empezar, se explican brevemente sus componentes principales. Luego, se muestra cómo utilizar la herramienta a partir de una especificación sencilla. Finalmente, mediante distintos ejemplos, se ilustran algunas de sus principales características.

4.1. Descripción general del proyecto

El objetivo general del proyecto es la construcción de un prototipo, que llamamos mcLola, para la generación automática de monitores escritos en C a partir de especificaciones Lola. Para ello hemos implementado:

- Un lenguaje de especificación basado en Lola, llamado mcLola.
- Un mecanismo para compilar especificaciones mcLola a código C99.

Concretamente, hemos diseñado un lenguaje de dominio específico embebido en Haskell que permite al usuario escribir una especificación como una lista de streams, los que a su vez se describen a partir de expresiones, siguiendo la definición de Lola.

A partir de la especificación escrita por el usuario, mcLola procede a la generación de código C99, que se divide en tres etapas:

- Análisis estático de la especificación.
- Generación de tipos de datos en C99.
- Generación del engine temporal en C99.

Como muestra la Figura 4.1, el primer paso hacia la generación de código C99 consiste en analizar la especificación escrita por el usuario. Esta etapa tiene una doble funcionalidad: valida la especificación, mientras que extrae información útil para la síntesis de código. A partir de esta información y de

una noción de *traducción* de expresiones, se genera el código C99 correspondiente al monitor en cuestión que, como muestra la Figura 4.1, se organiza en dos etapas independientes. Por un lado, se genera el código correspondiente a la implementación de los tipos de datos, organizado en los archivos `data.h` y `data.c`; y por el otro se generan los archivos `engine.h` y `engine.c` con la implementación del engine temporal de la especificación.

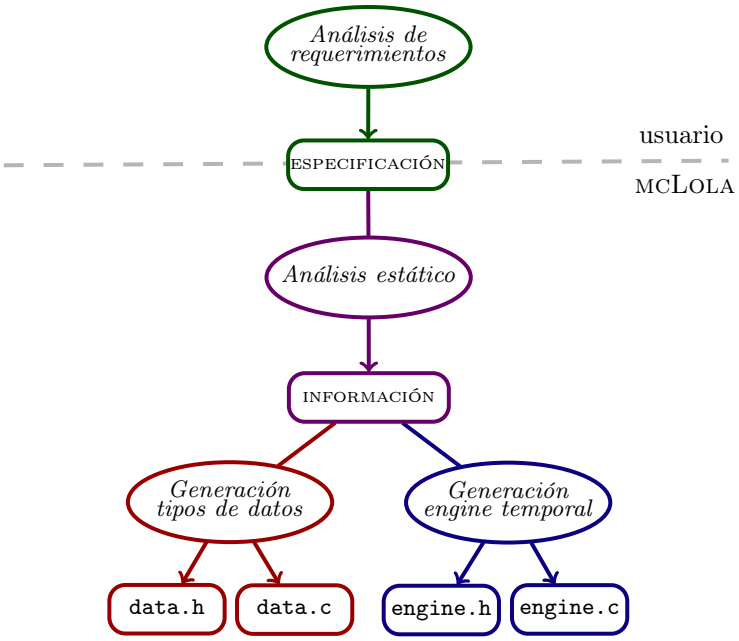


Figura 4.1: Proceso de generación de código C.

En lo que sigue se provee una breve descripción de los principales componentes del proyecto.

Lenguaje

Una especificación LOLA está definida por un conjunto de streams tipados de entrada y salida. Los streams de entrada se representan mediante un identificador, mientras que los streams de salida asocian a un identificador una expresión que lo define (Definición 3.2). El Ejemplo 4.2 muestra una especificación LOLA definida por un stream de entrada de tipo **int** llamado a y dos streams de salida: x de tipo **bool** e y de tipo **int**.

Ejemplo 4.2

```
input  int  a
output bool x = (a > a[-1|0]) ∨ x[-1|false]
output int  y = y[-1|1] + a[1|1]
```
◇

El objetivo principal del diseño del lenguaje es proveer un tipo de datos `Specification` para representar especificaciones Lola, que permita al usuario escribir especificaciones como la anterior. Para ello, primero es necesario contar con una implementación de expresiones y de streams parametrizados por el tipo.

Para definir expresiones, mcLola provee un tipo de datos **Exp** parametrizado por el tipo de la expresión. Así, un `e :: Exp a` representa una expresión Lola de tipo `a`[1]. Este tipo de datos provee un conjunto de constructores que permiten representar tanto a los operadores temporales de Lola, *now* y *offset*, como a los pertenecientes a las teorías de datos.

Definido **Exp**, se define el tipo **Stream** a partir de dos constructores: **In** para representar streams de entrada y **Out** para streams de salida. Así, podemos definir valores `a :: Stream Int`, `x :: Stream Bool` e `y :: Stream Int` para representar a los streams `a`, `x` e `y` de la especificación del Ejemplo 4.2. Para definir streams de entrada sólo se debe proveer un identificador, por ejemplo `a = In "a"`; mientras que para definir streams de salida se requiere además la expresión que lo define, por ejemplo si `e :: Exp Bool` es la representación de la expresión que define al stream `x`, entonces podemos definir a `x` como `x = Out ("x", e)`.

Este enfoque utiliza el sistema de tipos de Haskell para tipar los streams y las expresiones, permitiéndonos aprovechar el chequeo estático del lenguaje para realizar la verificación de tipos de las especificaciones. Sin embargo, esta solución presenta un problema a la hora de definir el tipo `Specification`. Dado que una especificación está conformada por uno o más streams, resulta razonable construir el tipo `Specification` a partir del tipo **Stream**, por ejemplo, representando una especificación como una lista de streams. En el Ejemplo 4.2, podríamos definir la especificación como `spec = [a, x, y]`. Sin embargo, esto no es posible puesto que en Haskell las listas son homogéneas y no todos los streams tienen el mismo tipo: continuando con el ejemplo anterior, `a :: Stream Int` mientras que `x :: Stream Bool`.

Para solucionar este inconveniente hemos seguido la estrategia empleada por hLola, que consiste en implementar listas heterogéneas. Intuitivamente, una vez definidos todos los streams usando el tipo **Stream** presentado anteriormente, debemos unificar el tipo de los mismos para poder reunirlos en una misma especificación. Concretamente, se provee un tipo **DynStream** para representar streams de cualquier tipo válido, junto con una función `toDynStrm` para transformar elementos de tipo `Stream a` en **DynStream**.

De esta manera, se implementa el tipo `Specification` simplemente como:

```
Specification = [DynStream]
```

pudiéndose definir la especificación del Ejemplo 4.2 de la siguiente manera:

[1] El tipo `a` no es un tipo arbitrario de Haskell, sino que debe cumplir algunas propiedades para que pueda ser traducido a un tipo C equivalente. Este punto será discutido en mayor detalle en el Capítulo 5.

```
spec :: Specification
spec = [toDynStrm a, toDynStrm x, toDynStrm y]
```

Análisis estático

Durante el análisis estático de la especificación se estudian los aspectos temporales de la misma para implementar dos tareas fundamentales: validar la especificación y extraer información relevante a la implementación del engine temporal.

Como se explicó en el Capítulo 3, para que sea posible generar un monitor ejecutable, es necesario que la especificación esté bien definida y sea eficientemente monitorizable. El análisis estático de MCLOLA verifica, en primer lugar, que la especificación sea acotada a futuro, lo que garantiza ambas propiedades (Teorema 3.11 y Corolario 3.22). Si la especificación no es acotada a futuro, se la rechaza y en caso contrario se continúa con el análisis temporal.

Nótese que la presencia de referencias temporales a instantes pasados y futuros introduce la necesidad de almacenar algunos valores de los streams durante la ejecución del monitor para poder completar cómputos futuros. El análisis temporal se ocupa de cuantificar los requerimientos espaciales necesarios para el monitoreo de la especificación.

Finalmente se completa el análisis estático con un análisis de tipos para identificar los tipos de datos que intervienen en la especificación, con el objetivo de definir, en una etapa posterior, tipos C equivalentes.

Engine

MCLOLA se enmarca en sistemas síncronos, donde el tiempo es discreto y lineal. Como se describió en el Algoritmo 1 de la Sección 3.3.1, en cada instante de tiempo i se envían al monitor los valores correspondientes a todos los streams de entrada en dicho instante. Inmediatamente, el monitor los procesa para computar nuevos valores de salida, lo que llamamos una *iteración* del engine, luego de lo cual vuelve a estar disponible para recibir los próximos valores de entrada, correspondientes al instante $i+1$. Este proceso se repite hasta que se acaban los eventos de entrada, momento en el cual el engine procede a finalizar los cómputos pendientes.

La implementación del engine temporal en MCLOLA está basada en el Algoritmo 1. La mayor diferencia reside en el tratamiento de las ecuaciones no resueltas. En lugar de trabajar con expresiones parcialmente evaluadas, MCLOLA computa los valores de los streams cuando dispone de toda la información necesaria. Esto facilita la implementación del engine en C, teniendo como contrapartida la necesidad de almacenar valores resueltos por más instantes de tiempo aunque, como se explicará en el Capítulo 6, para las especificaciones acotadas a futuro los requerimientos espaciales siguen siendo independientes de la longitud de la traza. Asimismo, en cada iteración en lugar de propagar

los nuevos valores encontrados hasta que no se registren cambios (como describe el procedimiento PROPAGATE del Algoritmo 1), los streams se procesan en un orden lineal fijo precomputado por el análisis estático que asegura que se computarán en una sola pasada todos los valores que se pueden calcular con la información disponible hasta el momento.

Por ejemplo, considérese la especificación del Ejemplo 4.2:

```
input  int  a
output bool x = (a > a[-1|0]) ∨ x[-1|false]
output int  y = y[-1|1] + a[1|1]
```

En este caso, en cada iteración del engine se procesarán los streams en el orden: a, x, y. Supongamos que queremos evaluar el monitor correspondiente a esta especificación con la traza $a = \langle 3, 5, 8, 2, 4 \rangle$. El engine procederá de la siguiente forma:

- Instante 0.
  Recibe $a(0)$ y computa $x(0)$ (usando $a(0)$).

instante	a	x	y
0	3	*true*	–

  No puede computar $y(0)$ porque aún no se conoce $a(1)$. En lugar de almacenar una expresión parcialmente evaluada (en este caso, $1 + a_1^1$) como se describe en el Algoritmo 1, simplemente se posterga el cómputo de $y(0)$.

- Instante 1.
  Recibe $a(1)$ y computa $x(1)$ (usando $a(1), a(0)$ y $x(0)$) y $y(0)$ (usando $a(1)$).

instante	a	x	y
0	3	*true*	6
1	5	*true*	–

  No puede computar $y(1)$ pues aún no se conoce $a(2)$.

- Instante 2.
  Recibe $a(2)$ y computa $x(2)$ (usando $a(2), a(1)$ y $x(1)$) y $y(1)$ (usando $y(0)$ y $a(2)$).

instante	a	x	y
0	3	*true*	6
1	5	*true*	14
2	8	*true*	–

  No puede computar $y(2)$ pues aún no se conoce $a(3)$.

- Instante 3.
  Recibe $a(3)$ y computa $x(3)$ (usando $a(3), a(2)$ y $x(2)$) y $y(2)$ (usando $y(1)$ y $a(3)$).

instante	a	x	y
1	5	*true*	14
2	8	*true*	16
3	2	*true*	–

  No puede computar $y(3)$ pues aún no se conoce $a(4)$. Además, elimina los valores de $a(0), x(0)$ e $y(0)$, puesto que ya no serán necesarios para computar nuevos valores.

- Instante 4.
  Recibe $a(4)$ y computa $x(4)$ (usando $a(4), a(3)$ y $x(3)$) y $y(3)$ (usando $y(2)$ y $a(4)$).

instante	a	x	y
2	8	*true*	16
3	2	*true*	20
4	4	*true*	–

  No puede computar $y(4)$ pues aún no se conoce $a(5)$. Además, elimina los valores de $a(1), x(1)$ e $y(1)$, puesto que ya no serán necesarios para computar nuevos valores.

- Finalización.
  Computa $y(4)$, que había quedado pendiente por referencias a futuro. Para ello usa $y(3)$ y el valor por defecto 1.

instante	a	x	y
2	8	*true*	16
3	2	*true*	20
4	4	*true*	21

Para implementar el comportamiento del engine temporal en C, MCLOLA genera tres funciones:

- `initialise`, que inicializa las estructuras de datos necesarias para dar comienzo a la ejecución del engine.
- `step`, que implementa una iteración del engine.
- `finish`, que concluye los cálculos pendientes una vez finalizados los eventos de entrada.

Datos

La etapa de generación de tipos de datos se ocupa de la implementación en C99 de los tipos de datos que utiliza la especificación. Puntualmente, cada tipo de datos que participa en la especificación se representa con un tipo C99 equivalente y se definen unas pocas funciones sobre ellos. Por ejemplo, para la especificación del Ejemplo 4.2 se utiliza el tipo `int` de C99 para representar el tipo de los streams a e y y el tipo `_Bool` para x.

Además, se declara un tipo `input` para encapsular los valores de entrada, que deberán pasarse como argumento a la función `step`. Para ello, se utiliza una estructura con tantos campos como streams de entrada tenga la especificación, cada uno con el mismo nombre y tipo. Por ejemplo, para la especificación del Ejemplo 4.2, MCLOLA declara el siguiente tipo:

```
struct input
{ int a;
};
typedef struct input input;
```

Por último, se provee una implementación de *buffers* para poder almacenar los valores de los streams durante la ejecución del monitor. Concretamente, para cada tipo `T` que sea el tipo de un stream de la especificación, se implementa un tipo `bufferT` para almacenar valores de tipo `T`.

4.2. Usando MCLOLA

A continuación se muestra cómo utilizar MCLOLA mediante un ejemplo sencillo. Primero se explica cómo escribir la especificación, luego se muestra cómo generar los archivos correspondientes al monitor y, finalmente, se ilustran los últimos pasos a seguir para completar la implementación del monitor.

A lo largo de esta sección trabajaremos con la especificación del Ejemplo 4.2:

```
input  int  a
output bool x = (a > a[-1|0]) ∨ x[-1|false]
output int  y = y[-1|1] + a[1|1]
```

Escritura de la especificación

Para definir una nueva especificación se debe crear un archivo Haskell que defina un elemento de tipo **Specification** que describa la especificación de interés. Para ello, se requiere importar:

- El módulo MCLola, que define los tipos **Specification** y **Stream**, la función toDynStrm para unificar los tipos de los streams y los operadores temporales del lenguaje: iOffset e iNow.

- Los módulos correspondientes a las teorías de datos que la especificación utiliza. En particular, MCLOLA implementa cada teoría de datos en un módulo separado, donde define los constructores de expresiones correspondientes a la misma. Por ejemplo, en el caso de nuestra especificación, que trabaja con valores booleanos y enteros, tendremos que importar los módulos `Theories.Bool` y `Theories.Numeric`. El primero define, entre otros, el constructor `iOr` que implementa la disyunción booleana, mientras que el segundo incluye los constructores `iIAdd` e `iIGt` para la suma y el predicado $>$ sobre expresiones enteras.

De esta manera, para escribir la especificación del Ejemplo 4.2, creamos un archivo Haskell `Spec.hs` (por ejemplo, en el directorio `Examples/Ex4_2` del código fuente), comenzando con las siguientes líneas de código:

```
module Examples.Ex4_2.Spec (spec) where

import MCLola
import Theories.Bool
import Theories.Numeric
```

donde en la primera línea indicamos que sólo se exportará el elemento `spec`.

Para definir `spec`, primero necesitamos definir valores `a :: Stream Int`, `x :: Stream Bool` e `y :: Stream Int` para representar a los streams de la especificación. En el caso de los streams de entrada simplemente usamos el constructor **In** que toma como único parámetro un elemento de tipo **String** para representar el nombre del stream, por ejemplo:

```
a :: Stream Int
a = In "a"
```

Observar que es necesario incluir la signatura de tipo de `a`, ya que de lo contrario el sistema de tipos de Haskell no podría inferirlo.

Para definir un stream de salida, usamos el constructor **Out** que toma como único argumento un par cuya primer componente es el nombre del stream, y cuya segunda componente es la expresión que lo define. Para mejorar la legibilidad del código podemos usar el constructor sintáctico **where** de Haskell:

```
x :: Stream Bool
x = Out ("x", expr)
  where
    expr = undefined
```

En el caso del stream `x`, la expresión que lo define es la disyunción lógica de dos subexpresiones, con lo cual usamos el constructor `iOr` definido por la teoría `Theories.Bool`:

```
x :: Stream Bool
x = Out ("x", expr)
  where
```

```
    expr = iOr e1 e2
    e1   = undefined
    e2   = undefined
```

Como la expresión `e2` se corresponde con el operador *offset* de Lola, para implementarla utilizamos el constructor `iOffset` definido por MCLola. Además, para especificar el valor por defecto usamos el constructor `iValBool` definido por `Theories.Bool` que permite definir constantes booleanas:

```
x :: Stream Bool
x = Out ("x", expr)
  where
    expr = iOr e1 e2
    e1   = undefined
    e2   = iOffset x (-1) (iValBool False)
```

Para definir la expresión `e1`, usamos el constructor `iIGt` definido por la teoría `Theories.Numeric` que implementa el predicado $>$ sobre enteros; mientras que para escribir sus argumentos, usamos los operadores temporales `iNow` e `iOffset` y el constructor `iValInt` definido también en `Theories.Numeric` para definir una constante entera a utilizar como valor por defecto:

```
x :: Stream Bool
x = Out ("x", expr)
  where
    expr = iOr e1 e2
    e1   = iIGt (iNow a) (iOffset a (-1) (iValInt 0))
    e2   = iOffset x (-1) (iValBool False)
```

Análogamente, damos una definición para el stream `y`:

```
y :: Stream Int
y = Out ("y", expr)
  where
    expr = iIAdd (iOffset y (-1) dflt) (iOffset a 1 dflt)
    dflt = iValInt 1
```

Finalmente, se define `spec` aplicando la función `toDynStrm` a los streams para unificar sus tipos:

```
spec :: Specification
spec = [toDynStrm a, toDynStrm x, toDynStrm y]
```

En el Apéndice B.1, como así también en el directorio `Examples/Ex4_2` del repositorio del proyecto, puede encontrarse el código correspondiente a este ejemplo.

Generación de código C

Una vez escrita la especificación, para poder generar el código C99 correspondiente al monitor debemos indicar en el archivo `Compile.hs` (Código 4.3) sobre qué especificación se debe trabajar. Para nuestro ejemplo, importamos

el archivo `Spec.hs` e indicamos que la especificación a utilizar es `spec`, como muestran las líneas 3 y 10:

```
1  module Compile (analyse, codegen) where
2
3  import Examples.Ex4_2.Spec              (spec)
4  import StaticAnalysis.StaticAnalysis (analysis, outputAnalysis)
5  import Data.Data                        (compileData)
6  import Engine.Engine                    (compileEngine)
7
8  -- // Especificación
9
10 importedSpec     = spec
11 externalIncludes = ["lib.h", "math.h"]
12
13 -- // Análisis estático
14
15 (dgraph, latMap, size, ordSpec, typeMap, types, input) =
       analysis importedSpec
16
17 analyse :: IO ()
18 analyse = outputAnalysis dgraph latMap
19
20 -- // Generación de código
21
22 codegen :: IO ()
23 codegen =    compileData input typeMap types
24           >> compileEngine ordSpec size typeMap latMap
                externalIncludes
```

Código 4.3: `Compile.hs`

El archivo `Compile.hs` permite identificar claramente las tres etapas descriptas en la Sección 4.1. Primero se realiza el análisis estático sobre la especificación (línea 15) y a partir de la información obtenida se procede a la generación del código C que implementa el monitor; puntualmente se generan los archivos `data.h` y `data.c` que implementan los tipos de datos (línea 23) y los archivos `engine.h` y `engine.c` para el engine temporal (línea 24).

De esta manera, para generar estos archivos basta con cargar el archivo `Compile.hs` en el intérprete GHCi [32] y evaluar `codegen`. Asimismo, si se evalúa `analyse` (línea 18) se genera un archivo `analysis.txt` con la información relativa al análisis temporal de la especificación. Todos los archivos generados se almacenan en el directorio `monitor`.

```
$ ghci
GHCi, version 8.10.4: https://www.haskell.org/ghc/ :? for help
Loaded package environment from ..
Prelude> :l Compile.hs
[ 1 of 30] Compiling Engine.FreeVars ...
...
```

```
[29 of 30] Compiling Engine.Engine ...
[30 of 30] Compiling Compile ...
Ok, 30 modules loaded.
*Compile> codegen
*Compile> analyse
```

Completando la implementación

MCLOLA no genera un programa C99 completo para que los usuarios puedan ejecutar directamente, sino que genera los archivos `engine.h` y `engine.c` que proveen la interfaz necesaria para implementar el monitor, como se describió en la Sección 4.1:

- `void initialise(void)`, que inicializa las estructuras de datos necesarias para dar comienzo a la ejecución del engine.
- `void step(input *)`, que implementa una iteración del engine.
- `void finish(void)`, que concluye los cálculos pendientes una vez finalizados los eventos de entrada.

Es responsabilidad del usuario escribir una función `main` que implemente el comportamiento completo del monitor. Por ejemplo, para nuestro caso podemos escribir un archivo `monitor.c` como muestra el Código 4.4.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "engine.h"
4
5  int main ()
6  {
7      input *s = malloc(sizeof(input));
8
9      initialise();
10
11     while (scanf("%d", &(s->a)) != EOF)
12         step(s);
13
14     finish();
15
16     return 0;
17 }
```

Código 4.4: `monitor.c`

Observar que en la línea 3 se incluye la cabecera `engine.h`, ya que en `engine.c` se definen las funciones `initialise`, `step` y `finish` que implementan la interfaz del engine temporal. En la línea 7 se declara y reserva memoria para una variable de tipo `input *` que se usará para pasar a la función `step` los nuevos valores de entrada. Notar que el tipo `input` ha sido generado por la herramienta

y está definido en `data.h`, incluído en `engine.h`. Las líneas 9 y 14 se ocupan, respectivamente, de iniciar y finalizar las tareas del engine, mientras que el bucle while de las líneas 11-12 se encarga de manejar los eventos de entrada y de invocar una iteración de los cómputos del engine.

De esta forma, con `monitor.c` se completa el código del monitor y estamos en condiciones de compilarlo con GCC [30]:

```
$ gcc -std=c99 -o monitor monitor.c engine.c data.c
```

Por último, podemos ejecutar el monitor pasando por entrada estándar un archivo llamado `input.csv` con los valores de entrada:

```
$ ./monitor < input.csv
```

La Figura 4.5 muestra la salida generada por el monitor para la entrada analizada en la Sección 4.1, dada por $a = \langle 3, 5, 8, 2, 4 \rangle$. Para cada instante de tiempo, se muestran cuáles fueron los valores computados mediante tuplas (i, s, val), indicando que el valor del stream s en el instante i es val.

```
Instant 0:      Instant 2:      (2, y, 16)
(0, a, 3)       (2, a, 8)       Instant 4:
(0, x, true)    (2, x, true)    (4, a, 4)
Instant 1:      (1, y, 14)      (4, x, true)
(1, a, 5)       Instant 3:      (3, y, 20)
(1, x, true)    (3, a, 2)       Finish:
(0, y, 6)       (3, x, true)    (4, y, 21)
```

Figura 4.5: Salida generada para la especificación del Ejemplo 4.2 para la traza $a = \langle 3, 5, 8, 2, 4 \rangle$.

Requerimientos

Para usar MCLOLA se necesita tener instalado GHC [75, 54], como así también las siguientes librerías:

- language-c99 [48], language-c99-simple [47] y pretty [65]
  Utilizadas para generar código C99.
- compdata [12]
  Provee facilidades para escribir y manipular árboles de sintaxis abstracta, utilizadas para definir el lenguaje MCLOLA.
- base [4]
  El módulo `Type.Reflection` permite obtener una representación del tipo de valores Haskell, que hemos utilizado para identificar el tipo de los streams.

- mtl [59] y containers [14]
  Utilizadas para trabajar con mónadas de estado (**State**) y diccionarios (**Map**), respectivamente.

Para facilitar el uso de McLola se incluye un Dockerfile que provee el entorno necesario para ejecutar el proyecto. Para más información, puede consultarse el Apéndice A.2.

4.3. Características adicionales

En lo que resta del capítulo se presentan ejemplos que permiten ilustrar algunas características de McLola. El código completo de los mismos puede consultarse en el directorio `Examples/Pinescript` del código fuente, como así también en el Apéndice B.1 de este informe.

Tipos y teorías de datos

McLola implementa algunas teorías de datos, mientras que permite al usuario extender el lenguaje escribiendo sus propias teorías. La implementación base de McLola provee dos teorías básicas:

- `Theories.Bool`, que implementa el tipo **Bool**,
- `Theories.Numeric`, que implementa los tipos **Int** y **Double**,

e incluye teorías de datos para incorporar tipos compuestos:

- `Theories.Product` implementa el tipo `(,)`,
- `Theories.Maybe` implementa el tipo **Maybe**,
- `Theories.Either` implementa el tipo **Either**,
- `Theories.List` implementa el tipo `[]`, que se implementa en C como listas acotadas.

En el Capítulo 8 se mostrará cómo agregar nuevas teorías de datos.

Ejemplo de aplicación

Si bien en la introducción de este informe se explicó el rol que ocupan los sistemas de monitoreo en el desarrollo de UAVs, el ámbito de aplicación de las técnicas de RV incluye múltiples (y muy diversas) áreas. El siguiente ejemplo muestra un caso de uso de la herramienta enmarcado en el análisis de activos financieros.

Ejemplo 4.6

Pine Script [63] es un lenguaje de programación utilizado para escribir indicadores y estrategias empleados en análisis técnicos de datos financieros. Los usuarios del lenguaje escriben scripts usando funciones y variables para calcular resultados que luego serán renderizados en gráficas.

Dos de las funciones provistas por Pine Script son *sma* y *crossover* [64]. La operación *media móvil simple* (*sma*, *simple moving average*) calcula el promedio de los últimos n datos de una serie, mientras que *crossover* toma como argumentos dos series de datos x e y y devuelve *true* si en el instante actual el valor de x es mayor o igual que el valor de y pero en el instante anterior el valor de y era mayor o igual que el de x. En particular, suele ser de interés calcular el *crossover* de la media móvil simple de dos activos financieros.

Veamos cómo escribir una especificación MCLOLA para calcular el *crossover* del *sma* con $n = 5$ (o $sma5$) de la acción x respecto al $sma5$ de la acción y.

Para empezar, como sólo se requiere manipular datos booleanos y numéricos, importamos las teorías `Theories.Bool` y `Theories.Numeric`, además del módulo `MCLola`:

```
module Examples.Pinescript.Ex1.Spec (spec) where
import MCLola
import Theories.Bool
import Theories.Numeric
```

Luego, definimos los streams de entrada `x` e `y`, correspondientes a los valores de las acciones x e y que representamos con el tipo **Double**:

```
x :: Stream Double
x = In "x"

y :: Stream Double
y = In "y"
```

Para calcular el *crossover* definimos un stream `crov :: Stream Bool`, para lo cual primero necesitaremos definir streams `sma5x :: Stream Double` y `sma5y :: Stream Double` que computen el $sma5$ de las acciones x e y, respectivamente.

Para computar el $sma5$ se deben sumar los últimos 5 valores de la acción y dividirlos entre 5, con la excepción de los primeros 4 instantes de tiempo, cuando aún no se dispone de 5 datos. Para resolver esos casos, podemos contar la cantidad de instantes i transcurridos: si $i < 5$, dividimos por i, sino dividimos por 5. Para contar la cantidad de instantes de tiempo podemos definir un stream `instant :: Stream Int` como se muestra a continuación:

```
instant :: Stream Int
instant = Out("instant", expr)
  where
    expr = iIAdd (iOffset instant (-1) (iValInt 0)) (iValInt 1)
```

Luego, podemos definir sma5x y sma5y de la siguiente manera:

```
sma5x :: Stream Double
sma5x = Out("sma5x", expr)
  where
    num  = foldr (\off e → iDAdd (iOffset x off (iValDouble 0)
        ) e) (iNow x) [(−4)..(−1)]
    den  = iIntToDouble (iIMin (iNow instant) (iValInt 5))
    expr = iDDiv num den

sma5y :: Stream Double
sma5y = Out("sma5y", expr)
  where
    num  = foldr (\off e → iDAdd (iOffset y off (iValDouble 0)
        ) e) (iNow y) [(−4)..(−1)]
    den  = iIntToDouble (iIMin (iNow instant) (iValInt 5))
    expr = iDDiv num den
```

Para computar el *sma*5 simplemente computamos el numerador y denominador y dividimos usando el constructor iDDiv, definido en Theories.Numeric, que implementa la división de expresiones de tipo **Double**. El denominador se calcula como el mínimo entre el valor actual de instant y 5. Nótese el uso del constructor iIntToDouble para transformar un valor entero en uno de punto flotante.

El numerador está dado por la suma de los últimos 5 valores del stream. Para definir num hemos usado el operador de alto orden **foldr** de Haskell. Si bien la siguiente definición es equivalente, resulta menos legible y, peor aún, menos escalable para valores más grandes de n.

```
num = iDAdd (iOffset x (−4) (iValDouble 0))
            (iDAdd (iOffset x (−3) (iValDouble 0))
                   (iDAdd (iOffset x (−2) (iValDouble 0))
                          (iDAdd (iOffset x (−1) (iValDouble 0))
                                 (iNow x))))
```

Una de las ventajas de los lenguajes de dominio específico embebidos en un lenguaje anfitrión es poder utilizar las abstracciones que este último provee. En ese sentido, MCLOLA no necesita implementar un operador *fold* para construir expresiones porque puede usar directamente la función **foldr** de Haskell.

Por último, definimos el stream crov usando los operadores iAnd e iDLeq:

```
crov :: Stream Bool
crov = Out ("crov", expr)
 where
  dflt = iValDouble 0
  expr = iAnd (iDLeq (iNow sma5y) (iNow sma5x))
    (iDLeq (iOffset sma5x (−1) dflt) (iOffset sma5y (−1) dflt))
```

De esta manera, podemos definir la especificación de la siguiente forma:

```
spec :: Specification
spec = [ toDynStrm x, toDynStrm y, toDynStrm instant
       , toDynStrm sma5x, toDynStrm sma5y, toDynStrm crov]
```

Observar que en la definición de la especificación se incluyen todos los streams definidos, incluso los streams intermedios o auxiliares como sma5x, sma5y e instant. Esto es necesario puesto que MCLOLA traduce a C cada stream individualmente, asumiendo que todo stream referenciado forma parte de la especificación. Si en la definición de un stream se referencia a otro que no se incluye en la lista que define la especificación, entonces la herramienta reportará el error indicando que dicho stream no se encuentra definido. ◇

Ejemplo 4.7

En el ejemplo anterior, el *sma5* se computa en los primeros 4 instantes de tiempo con menos de 5 datos. Si, por el contrario, sólo se quisiera calcular el valor cuando estén los 5 datos disponibles se podría usar un tipo opcional para indicar que el cómputo puede fallar.

En Haskell, el constructor de tipos **Maybe** encapsula valores opcionales. Un valor de tipo **Maybe** a es un valor x de tipo a (que se representa como **Just** x) o bien es vacío (representado con **Nothing**). Usualmente, se usa **Maybe** para manejar errores o casos excepcionales, como ocurre con *sma* cuando no existen suficientes datos.

MCLOLA incluye la teoría de datos Theories.Maybe para trabajar con tipos opcionales. El tipo **Maybe** a de Haskell se traduce a un tipo struct de C99 con dos campos: `just` del tipo C99 correspondiente a a y `nothing` de tipo `int` que indica si existe o no un valor. La teoría **Maybe** ofrece dos constructores de valores (iJustM e iNothingM) y un observador (iIsNothing). Además, provee el operador iFromMaybe, análogo a la función **fromMaybe** de Haskell definida en Data.Maybe.

Para computar el *sma5* únicamente cuando se dispone de al menos 5 datos, podemos importar la teoría Theories.Maybe y reimplementar el stream sma5x de la siguiente manera:

```
sma5 :: Stream (Maybe Double)
sma5 = Out("sma5", expr)
  where
    num  = foldr (\off e → iDAdd (iOffset x off (iValDouble 0)
        ) e) (iNow x) [(-4)..(-1)]
    den  = iValDouble 5
    expr = iIte (iDLeq den (iIntToDouble (iNow instant)))
                (iJustM (iDDiv num den))
                (iNothingM (iValDouble 0))
```

Notar que para decidir si el cómputo es exitoso se usa el constructor iIte, definido en Theories.Bool, que implementa el operador **if . then . else** de LOLA. El constructor iNothingM, a diferencia del constructor **Nothing** de Haskell, requiere de un argumento que se usará como valor por defecto para el campo `just` cuando se realice la traducción a C. ◇

Parametrización estática

La *parametrización estática* es una característica provista por algunos sistemas de SRV que permite escribir especificaciones abstractas para luego instanciarlas con valores específicos. Esto permite reusar especificaciones repetitivas, capturando la semántica del stream y abstrayendo valores puntuales, evitando así la duplicación de código.

Como ilustra el siguiente ejemplo, la parametrización estática puede obtenerse en MCLOLA usando directamente características de Haskell, sin necesidad de extender la implementación del lenguaje.

Ejemplo 4.8

Las definiciones MCLOLA para las fórmulas *sma* y *crossover* del Ejemplo 4.6 pueden parametrizarse como se muestra a continuación. Mientras que *crossover* está parametrizado por dos streams de tipo **Double**, *sma* abstrae un stream de tipo **Double** y un valor n de tipo **Int** que determina la cantidad de datos a considerar.

```
sma :: Stream Double → Int → Stream Double
sma x n = Out(name, expr)
  where
    name = "sma_" ++ show n ++ "_" ++ ident x
    num  = foldr (\off e → iDAdd (iOffset x off (iValDouble 0)
         ) e) (iNow x) [((-1)*(n-1))..(-1)]
    den  = iIntToDouble (iIMin (iNow instant) (iValInt n))
    expr = iDDiv num den

crossover :: Stream Double → Stream Double → Stream Bool
crossover x y = Out (name, expr)
  where
    name = ident x ++ "_crossover_" ++ ident y
    dflt = iValDouble 0
    expr = iAnd (iDLeq (iNow y) (iNow x))
                (iDLeq (iOffset x (-1) dflt) (iOffset y (-1)
                    dflt))
```

De esta manera, para definir `crov` sólo se deben instanciar adecuadamente `crossover` y `sma`:

```
sma5x :: Stream Double
sma5x = sma x 5

sma5y :: Stream Double
sma5y = sma y 5

crov :: Stream Bool
crov = crossover sma5x sma5y
```

◇

Librerías

Otro de los beneficios de implementar un lenguaje de dominio específico embebido en Haskell es el de poder hacer uso de su sistema de módulos, lo cual no sólo permite usar librerías externas, sino también escribir librerías propias.

Este hecho, sumado a la parametrización estática, facilita la creación de librerías para un dominio específico.

Ejemplo 4.9

Para escribir una librería con funciones de Pine Script, podemos agrupar las definiciones vistas anteriormente en un archivo `Pinescript.hs` en el directorio `Lib`, como muestra el siguiente código.

```
module Lib.Pinescript where

import MCLola
import Theories.Bool
import Theories.Numeric

-- Instant stream
instant :: Stream Int
instant = Out("instant", expr)
  where
    expr = iIAdd (iOffset instant (-1) (iValInt 0)) (iValInt 1)

-- Simple Moving Average (needs instant)
sma :: Stream Double → Int → Stream Double
sma x n = Out(name, expr)
  where
    name = "sma_" ++ show n ++ "_" ++ ident x
    num  = foldr (\off e → iDAdd (iOffset x off (iValDouble 0)
        ) e) (iNow x) [((-1)*(n-1)) .. (-1)]
    den  = iIntToDouble (iIMin (iNow instant) (iValInt n))
    expr = iDDiv num den

-- Crossover
crossover :: Stream Double → Stream Double → Stream Bool
crossover x y = Out (name, expr)
  where
    name = ident x ++ "_crossover_" ++ ident y
    dflt = iValDouble 0
    expr = iAnd (iDLeq (iNow y) (iNow x))
                (iDLeq (iOffset x (-1) dflt) (iOffset y (-1)
                    dflt))
```

Para usar la librería simplemente se debe importar el módulo `Lib.Pinescript`. En el Apéndice B.1 se muestra un ejemplo de uso. ◇

Capítulo 5

Diseño del lenguaje

En este capítulo se aborda el diseño de un lenguaje para especificaciones Lola. Primero se introducen los fundamentos de los lenguajes de dominio específico y, en particular, los lenguajes de dominio específico embebidos en Haskell. Luego se comentan los principales desafíos encontrados durante el proceso de diseño del lenguaje junto con las soluciones adoptadas.

5.1. Lenguajes de Dominio Específico

Un *lenguaje de dominio específico* (*DSL*, *Domain Specific Language*) es un lenguaje de programación diseñado para resolver problemas de un dominio en particular, a diferencia de los lenguajes de propósito general como C o Haskell. LaTeX, SQL y R son algunos de los lenguajes de dominio específicos más conocidos, que resuelven, respectivamente, el problema de editar texto científico, manejar bases de datos y realizar análisis estadísticos.

La principal ventaja de los DSLs frente a los lenguajes de programación de propósito general es que, al centrarse en un dominio de aplicación específico, proveen un mayor nivel de abstracción. Por ejemplo, un usuario de SQL puede escribir queries sin tener que preocuparse por el manejo de las estructuras internas donde se almacenan los datos. Como consecuencia, estos programas resultan más fáciles de escribir, analizar y modificar si se los compara con programas equivalentes escritos en lenguajes de propósito general, lo que los hace mucho más accesibles a no expertos.

La principal desventaja de los DSLs es que resulta muy costoso diseñar e implementar un lenguaje de programación desde cero. El proceso requiere desarrollar un compilador o intérprete, lo que en general involucra el desarrollo de analizadores lexicográficos, analizadores semánticos y optimizadores, entre otros. Además, se deben proveer herramientas de soporte para el desarrollo de software como editores con resaltado de sintaxis, librerías y generadores de documentación.

Un enfoque que reduce considerablemente los costos es el de los *lenguajes de dominio específico* ***embebidos*** dentro de un lenguaje de propósito general (*EDSLs*, *Embedded Domain Specific Languages*) [38, 37]. Un EDSL aprovecha la infraestructura de un lenguaje pre-existente, llamado *lenguaje anfitrión*, para implementar un lenguaje de dominio específico. Esta técnica permite reutilizar la sintaxis, sistema de tipos, compiladores, librerías y demás herramientas del lenguaje anfitrión, facilitando enormemente la tarea de implementar el DSL. Además, los EDSLs permiten a los usuarios acceder a las abstracciones propias del lenguaje subyacente, como así también combinar DSLs de diferentes dominios que comparten el lenguaje anfitrión.

5.1.1. Lenguajes de Dominio Específico Embebidos

Al construir un EDSL, se diseña un lenguaje con ciertas primitivas que permiten manipular elementos con el objetivo de computar resultados de interés en el dominio de aplicación. Concretamente, un EDSL se construye manipulando uno o más tipos de datos: se provee un conjunto de *constructores del lenguaje* para construir elementos de esos tipos, y *funciones de interpretación* para hacer observaciones semánticas sobre ellos.

Por ejemplo, supongamos que queremos diseñar un lenguaje de dominio específico embebido en Haskell para trabajar con expresiones aritméticas sobre enteros. Más precisamente, el lenguaje estará conformado únicamente por la suma y la resta de enteros. Como ilustra la interfaz del Código 5.1, debemos definir un tipo `Expr` para representar expresiones enteras junto con constructores de expresiones (`lit`, `add` y `sub`) y una función de interpretación para evaluar expresiones (`eval`).

```
type Expr

lit :: Int → Expr
add :: Expr → Expr → Expr
sub :: Expr → Expr → Expr

eval :: Expr → Int
```

Código 5.1: Interfaz del lenguaje de expresiones aritméticas.

En general, para implementar un EDSL se puede optar entre dos alternativas diferentes [33]:

- *Embebimiento superficial*, donde los elementos del lenguaje se representan con un tipo de datos que explicita su semántica en el lenguaje anfitrión.

- *Embebimiento profundo*, donde los elementos del lenguaje se representan como constructores en un *árbol de sintaxis abstracta* (*AST*, *Abstract Syntax Tree*), a partir del cual se computa su semántica.

Para ejemplificar los dos enfoques, en el Código 5.2 se muestra cómo implementar este pequeño lenguaje de expresiones aritméticas en cada caso.

```
-- Tipo de datos
newtype Expr = E Int

-- Constructores del lenguaje
lit :: Int → Expr
lit n = E n

add :: Expr → Expr → Expr
add (E n) (E m) = E (n + m)

sub :: Expr → Expr → Expr
sub (E n) (E m) = E (n − m)

-- Función de interpretación
eval :: Expr → Int
eval (E n) = n
```

(a) EDSL superficial.

```
-- Tipo de datos
data Expr = Lit Int
          | Add Expr Expr
          | Sub Expr Expr

-- Constructores del lenguaje
lit :: Int → Expr
lit = Lit

add :: Expr → Expr → Expr
add = Add

sub :: Expr → Expr → Expr
sub = Sub

-- Función de interpretación
eval :: Expr → Int
eval (Lit n)   = n
eval (Add x y) = eval x + eval y
eval (Sub x y) = eval x − eval y
```

(b) EDSL profundo.

Código 5.2: Embebido superficial y profundo para el lenguaje de expresiones aritméticas.

En el embebido superficial los constructores del lenguaje son los responsables de dar semántica al lenguaje, con lo cual la definición de la función de interpretación es trivial. En cambio, en el embebido profundo se utiliza un *tipo de datos algebraico* (*ADT*, *Algebraic Data Type*) para representar a las expresiones. De esta manera, los constructores del lenguaje se definen simplemente dando el constructor de valores adecuado, mientras que la función de interpretación realiza todo el trabajo semántico.

Analizando este ejemplo, podemos notar que si se sigue el enfoque superficial resulta muy sencillo extender el lenguaje con nuevos constructores. Por ejemplo, si quisiéramos agregar a nuestro lenguaje de expresiones aritméticas la multiplicación de enteros, sólo tendríamos que agregar una función:

```
mul :: Expr → Expr → Expr
mul (E n) (E m) = E (n ∗ m)
```

Como contrapartida, dado que el dominio semántico está fijo, no es posible agregar otras interpretaciones sin tener que reimplementar completamente el lenguaje. Por ejemplo, si además de evaluar expresiones quisiéramos contar la cantidad de sumas tendríamos que cambiar la definición del tipo **Expr** y, por lo tanto, la implementación de cada uno de los constructores y funciones de interpretación.

Por el contrario, para agregar una nueva interpretación en el enfoque profundo basta con definir otra función que dé semántica a los distintos constructores del árbol de sintaxis abstracta. Por ejemplo:

```
nSum :: Expr → Int
nSum (Lit _)     = 0
nSum (Add x y) = nSum x + nSum y + 1
nSum (Sub x y) = nSum x + nSum y
```

Sin embargo, si se requiere extender el lenguaje con un nuevo constructor se debe modificar el tipo **Expr** y, por lo tanto, todas las funciones de interpretación ya definidas.

En este sentido, ambos enfoques son duales: mientras que el embebido superficial es extensible a nuevos constructores del lenguaje, el embebido profundo es extensible a nuevas interpretaciones. No obstante, estos son dos puntos extremos dentro de un amplio espectro. De hecho, uno de los mayores desafíos del diseño de EDSLs consiste en poder combinar las ventajas de ambos enfoques en una misma implementación. A menudo los EDSLs se implementan con un enfoque híbrido, generalmente más cercano a uno de estos dos extremos, dependiendo de las características del dominio de aplicación.

En el caso de los EDSLs diseñados para generar código, como mcLola, lo más natural es seguir un enfoque profundo. La generación automática de código requiere de múltiples análisis sobre las expresiones del lenguaje, por lo cual resulta apropiado disponer de un AST para realizar distintas interpretaciones.

5.1.2. Haskell como lenguaje anfitrión

Al desarrollar un EDSL, el diseño queda limitado por las capacidades del lenguaje anfitrión, por lo que resulta fundamental que el mismo sea lo suficientemente flexible, tanto en términos de abstracción y semántica como de sintaxis.

Haskell posee varias características que lo hacen un buen lenguaje anfitrión, como su sistema de tipos, las funciones de alto orden y la evaluación perezosa. Su sintaxis concisa, el llamado a función sin paréntesis, la sobrecarga de operadores y los operadores definidos por el usuario contribuyen enormemente a definir un lenguaje con su propio *look and feel*. Además, la amplia disponibilidad de librerías y extensiones del lenguaje facilitan considerablemente la implementación.

El buen desempeño de Haskell como lenguaje anfitrión se evidencia en el gran número de implementaciones exitosas de EDSLs que abarcan diversos dominios, incluyendo análisis de regiones geométricas [40], generación de parsers [61], animación [27], diseño de hardware [7] y composición musical [39].

Teniendo en cuenta las características que hacen de Haskell un buen lenguaje anfitrión y la experiencia positiva del desarrollo de hLola, se decidió implementar mcLola como un lenguaje de dominio específico embebido en Haskell.

Árboles de Sintaxis Abstracta en Haskell

Al implementar un EDSL con un enfoque profundo resulta fundamental disponer de facilidades para trabajar con árboles de sintaxis abstracta. A continuación se mencionan algunas características de Haskell relevantes para la construcción y manipulación de ASTs.

Como se exhibe en el Código 5.2b correspondiente al ejemplo del lenguaje de expresiones aritméticas, un AST puede implementarse fácilmente en Haskell utilizando un tipo de datos algebraico:

```
data Expr = Lit Int
          | Add Expr Expr
          | Sub Expr Expr
```

Otra de las ventajas de Haskell es el uso extensivo de pattern matching, que contribuye a mejorar la legibilidad del código, como puede apreciarse en la función de interpretación del Código 5.2b.

Los ADTs se pueden combinar con otras características del sistema de tipos de Haskell, como el polimorfismo y las clases de tipos, para lograr tipos más generales y expresivos. Si quisiéramos disponer de un lenguaje de expresiones aritméticas análogo al del ejemplo pero sobre valores reales en lugar de enteros, podríamos repetir la implementación usando **Float** o **Double** en lugar de **Int**. Sin embargo, es posible evitar esta duplicación de código definiendo un tipo algebraico polimórfico y agregando una restricción de clase para indicar que debe tratarse de un tipo numérico:

```
data Expr a = Lit a
            | Add (Expr a) (Expr a)
            | Sub (Expr a) (Expr a)

eval :: Num a ⇒ Expr a → a
eval (Lit n)   = n
eval (Add n m) = eval n + eval m
eval (Sub n m) = eval n − eval m
```

No obstante, los ADTs tienen sus limitaciones. Supongamos que se necesita extender el lenguaje de expresiones aritméticas del ejemplo con expresiones booleanas. En particular, se requiere agregar los valores *true* y *false* y una construcción condicional $if \cdot then \cdot else$. De esta manera, el lenguaje estará formado por expresiones enteras y expresiones booleanas. Una primera alternativa para diseñar el nuevo AST sería extender el tipo **Expr** del Código 5.2b con los nuevos constructores:

```
data Expr = ILt Int
          | Add Expr Expr
          | Sub Expr Expr
          | BLt Bool
          | Ite Expr Expr Expr
```

Sin embargo, esta implementación resulta deficiente. Observamos que no es posible diferenciar a nivel de tipos las expresiones enteras de las booleanas, en efecto, tanto `Add (ILt 5) (ILt 2)` como `BLt False` tienen tipo `Expr`. Como consecuencia, existen expresiones sintácticamente correctas que no tienen sentido semántico, como por ejemplo `Ite (ILt 15) (ILt 15) (BLt True)` o `Add (BLt False) (ILt 1)`, con lo cual no se aprovecha el sistema de tipos de Haskell para rechazar estáticamente expresiones inválidas. Además, la signatura de tipos de `eval` debe cambiar, ya que las expresiones del lenguaje pueden computar un valor de tipo `Int` o `Bool`. La alternativa de utilizar un coproducto (como `Either Int Bool`) es insatisfactoria, puesto que en la definición de `eval` se deben considerar todos los posibles casos, afectando la legibilidad del código y haciéndolo poco escalable para incorporar nuevos tipos al lenguaje.

Para poder diferenciar a nivel de tipos las expresiones enteras de las booleanas, se busca parametrizar el tipo `Expr` con el tipo de la expresión, por ejemplo: `Add (ILt 5) (ILt 2) :: Expr Int` y `BLt False :: Expr Bool`. Si además queremos que sólo sea posible construir expresiones que tengan sentido semántico, se deben restringir los tipos de los constructores. Por ejemplo, para el constructor `Add` se necesita explicitar que ambos argumentos sean de tipo `Expr Int` mientras que para `Ite` el primero debe ser de tipo `Expr Bool` y el segundo y tercero deben ser del mismo tipo (en este caso, `Expr Bool` o `Expr Int`). El problema reside en que los tipos algebraicos de Haskell no son lo suficientemente expresivos para explicitar el tipo de sus constructores de esta manera.

Sin embargo, existe una extensión de Haskell llamada *Generalized Algebraic Data Types (GADTs)* [31] que resuelve esta limitación. Las extensiones se utilizan para habilitar características específicas que no forman parte de la definición estándar de Haskell pero que resultan útiles en determinados casos. En general, se usan para relajar restricciones del sistema de tipos o para agregar nuevas construcciones. La extensión GADTs provee una generalización de los tipos algebraicos tradicionales que permite anotar los tipos de los constructores explícitamente. Utilizando esta extensión, podemos definir el tipo `Expr` de la siguiente manera:

```
data Expr a where
  ILt :: Int → Expr Int
  Add :: Expr Int → Expr Int → Expr Int
  Sub :: Expr Int → Expr Int → Expr Int
  BLt :: Bool → Expr Bool
  Ite :: Expr Bool → Expr a → Expr a → Expr a
```

Notar que, a diferencia de los tipos algebraicos tradicionales, es posible especificar el tipo de los elementos que construye cada constructor. Por ejemplo, mientras que `Add` construye un elemento de tipo `Expr Int`, `BLt` construye uno de tipo `Expr Bool` e `Ite` uno del mismo tipo que su segundo (o tercer) argumento. Esta generalidad permite definir un tipo de datos composicional para construir expresiones de distintos tipos y, en particular, permite definir fácil-

mente una función `eval` que computa elementos de diferentes tipos, en este caso **Int** o **Bool**:

```
eval :: Expr a → a
eval (ILt n)      = n
eval (Add n m)    = eval n + eval m
eval (Sub n m)    = eval n − eval m
eval (BLt b)      = b
eval (Ite b x y) = if (eval b) then eval x else eval y
```

El punto clave de GADTs es que el pattern matching induce inferencia de tipos [77]. Por ejemplo, en la ecuación `eval (ILt n) = n` se infiere que el tipo `a` es **Int** por el uso del constructor **ILt**. De esta manera, se logra aprovechar el sistema de tipos de Haskell para definir un AST que permite construir expresiones de distintos tipos y definir funciones de evaluación de forma concisa y elegante.

5.2. Lenguaje para especificaciones Lola

Una *especificación* Lola está definida por un conjunto de *streams*, los que a su vez se describen a partir de *expresiones* sobre un determinado conjunto de teorías de datos. Por lo tanto, para desarrollar un lenguaje para especificaciones Lola como un lenguaje de dominio específico embebido en Haskell, se requiere representar tres elementos de interés principales:

- Expresiones
- Streams
- Especificaciones

En lo que resta del capítulo se explica cómo hemos representado estos conceptos, comentando los principales desafíos encontrados y las soluciones adoptadas. La implementación de los mismos puede consultarse en el directorio `Language` del código fuente.

5.2.1. Streams y expresiones

Lola trabaja con streams tipados que se definen a partir de expresiones sobre distintas teorías de datos. Para aprovechar el sistema de tipos de Haskell para hacer el chequeo de tipos, el primer objetivo del diseño del lenguaje es definir tipos de datos paramétricos:

- **Exp** `a`, para representar expresiones de tipo `a`.
- **Stream** `a`, para representar streams de tipo `a`.

La definición de **Exp** constituye el mayor desafío del diseño del lenguaje. Recordemos que uno de los principios de SRV es mantener por separado los cómputos temporales de los de datos, de manera tal que incorporar nuevas teorías no implique cambios significativos en la implementación. Esto significa que el lenguaje de expresiones debería ser extensible a nuevos constructores, lo que se logra fácilmente si se implementa un embebido superficial. Al mismo tiempo, la generación automática de código requiere de varios análisis relativamente complejos sobre las expresiones del lenguaje, con lo cual resulta más apropiado trabajar con un enfoque profundo. En la Sección 5.3 se explica en detalle cómo hemos abordado este problema.

Asumiendo definida una representación de las expresiones en Haskell, la definición de un tipo de datos para representar streams resulta inmediata. En Lola todo stream es, o bien un stream de entrada, o bien un stream de salida. En el primer caso, el stream queda definido por un identificador, mientras que en el segundo además de un nombre se requiere la expresión que lo define. Por lo tanto, se podría definir el tipo **Stream** a usando un GADT con un constructor para cada caso como se muestra a continuación.

```
type Identifier = String

data Stream a where
  In  :: Identifier           → Stream a
  Out :: (Identifier, Exp a) → Stream a
```

Sin embargo, esta definición resulta demasiado general. Puntualmente, cada tipo Haskell que constituya el tipo de un stream deberá ser traducido a un tipo C equivalente; y, por cuestiones de seguridad (safety) que serán discutidas en la Sección 6.2.1, buscamos trabajar con tipos C construidos únicamente a partir de tipos built-in simples y structs, sin usar memoria dinámica. Por lo tanto, debemos garantizar que los tipos Haskell a usar en la especificación tengan una representación en C que respete esta restricción. Para ello, hemos definido una clase de tipos llamada **Streamable** para describir a los tipos soportados por mcLola. Las instancias definidas para esta clase son unos pocos tipos básicos más algunos otros tipos compuestos como (,), **Maybe** o **Either**, pero no, por ejemplo, tipos funcionales o tipos recursivos arbitrarios.

Finalmente, podemos definir el tipo **Stream** a agregando esta restricción de tipos:

```
data Stream a where
  In  :: Streamable a ⇒ Identifier           → Stream a
  Out :: Streamable a ⇒ (Identifier, Exp a) → Stream a
```

Traducción de tipos

Para traducir tipos Haskell a tipos C equivalentes, primero se necesita identificar cuáles son esos tipos. Para ello, hemos definido un tipo de datos, llamado **TypeRepr**, para representar a los tipos soportados por MCLOLA, como muestra el Código 5.3.

```
data TypeRepr where
  TBool     :: TypeRepr
  TInt      :: TypeRepr
  TDouble   :: TypeRepr
  TMaybe    :: TypeRepr → TypeRepr
  TProd     :: TypeRepr → TypeRepr → TypeRepr
  TEither   :: TypeRepr → TypeRepr → TypeRepr
  TList     :: TypeRepr → TypeRepr
  TNAProd   :: String → [(String, TypeRepr)] → TypeRepr
  deriving (Eq , Ord)
```

Código 5.3: Representación de tipos soportados por MCLOLA.

Los primeros tres constructores se corresponden con los tipos básicos de MCLOLA: booleanos, enteros y reales; mientras que los restantes con los tipos compuestos: tipos opcionales, productos, coproductos, listas [1] y records. Si bien es cierto que en comparación con HLOLA se ha restringido notablemente el conjunto de los tipos soportados, estos resultan suficientemente expresivos para escribir la mayoría de las especificaciones que surgen en la práctica en SRV.

Notar que **TypeRepr** constituye una representación intermedia de los tipos, en el sentido de que es independiente del lenguaje objetivo (C99). Como consecuencia, el análisis de tipos puede reutilizarse para generar monitores escritos en otro lenguaje objetivo.

De esta manera, un tipo de Haskell podrá ser traducido a un tipo C99 equivalente si se corresponde con un elemento de **TypeRepr**. Para expresar esta caracterización, hemos definido una clase de tipos llamada **C99able**:

```
class C99able a where
  typeRepr :: TypeRepr
```

Para agregar instancias a esta clase hemos utilizado las extensiones de Haskell `ScopedTypeVariables` y `TypeApplications` [51, 76] para poder acceder con `typeRepr @a` a la representación de un subtipo `a` de un tipo compuesto, por ejemplo:

```
instance C99able Bool where
  typeRepr = TBool

instance (C99able a, C99able b) ⇒ C99able (a, b) where
  typeRepr = TProd (typeRepr @a) (typeRepr @b)
```

[1] Debido a las restricciones relativas al uso de memoria dinámica mencionadas anteriormente, y que serán discutidas en mayor profundidad en el Capítulo 6, la implementación de MCLOLA se restringe a listas acotadas.

Por otro lado, para poder asociar a cada stream un elemento de tipo **TypeRepr**, es necesario acceder dinámicamente a información de tipos de términos de Haskell. Para esto, hemos utilizado la clase **Typeable** definida en Data.Typeable [21], que permite calcular una representación concreta para los tipos que constituyen sus instancias. En particular, todos los tipos Haskell que se corresponden con un elemento de **TypeRepr** son instancias de **Typeable**.

Por lo tanto, mcLola podrá traducir un tipo de Haskell a C99 cuando éste sea instancia de las clases **Typeable** y **C99able**, lo que nos permite definir la clase **Streamable** de la siguiente manera:

```
type Streamable a = (Typeable a, C99able a)
```

5.2.2. Especificaciones

Suponiendo definidos los tipos **Exp** a y **Stream** a, sólo resta definir un tipo de datos **Specification** para representar especificaciones Lola. Considerando que una especificación es, sencillamente, un conjunto de streams y que a partir del análisis temporal se define un orden lineal para el procesamiento de los streams, resulta natural representar a las especificaciones como listas de streams. Sin embargo, dado que en Haskell las listas son homogéneas, no será posible construir una lista con streams de distintos tipos. Para resolver este problema, hemos dado una implementación de listas heterogéneas basada en la solución adoptada en hLola. La idea principal consiste en transformar cada stream, de tipo **Stream** a, en un elemento equivalente respecto a la información relevante para la generación de código, pero que oculte el tipo concreto del stream, como se muestra a continuación:

```
data DynStrm where
  DIn  :: Identifier           → DynStrm
  DOut :: (Identifier, Exp a)  → DynStrm
```

De esta manera, el sistema de tipos de Haskell *olvida* el tipo del stream, permitiéndonos armar una lista de streams que originariamente tenían tipos distintos. Sin embargo, para generar código C necesitamos conocer el tipo de cada stream y, en particular, si se usa el tipo **DynStrm** se pierde cualquier referencia al tipo de los streams de entrada. Para solucionarlo, basta con agregar un elemento de tipo **TypeRepr** para indicar el tipo del stream:

```
type DynStream = (DynStrm, TypeRepr)
```

Definido el tipo de datos **DynStream** con toda la información relevante para la generación de código, sólo resta definir una función toDynStrm que realice la transformación requerida:

```
toDynStrm :: Streamable a ⇒ Stream a → DynStream
toDynStrm x@(In s)       = (DIn s, streamType x)
toDynStrm x@(Out (s, e)) = (DOut (s, e), streamType x)
```

La función `streamType :: Streamable a ⇒ Stream a → TypeRepr` computa la representación del tipo de un stream. Para su implementación se ha utilizado el módulo `Type.Reflection` que ofrece una representación para los tipos que sean instancia de **Typeable**.

Finalmente, la definición del tipo **Specification** es inmediata:

```
type Specification = [DynStream]
```

Una desventaja de este enfoque es que se pierden los tipos originales de los streams. Sin embargo, este paso se realiza luego de que Haskell haya realizado el chequeo de tipos, con lo cual se olvidan tipos de términos que han sido chequeados anteriormente y, por lo tanto, se sabe que están bien tipados.

5.3. Lenguaje para expresiones Lola

En Lola, las expresiones se construyen a partir de dos elementos:

- *Referencias temporales*, que introducen al lenguaje las dependencias temporales entre los streams.
- *Aplicación de funciones*, que introducen al lenguaje las teorías de datos.

La tarea de desarrollar un lenguaje para expresiones Lola consiste, esencialmente, en encontrar una representación adecuada para estos dos tipos de construcciones. En particular, buscamos que nuestro lenguaje de expresiones verifique tres requerimientos principales:

1. Ser tipado.
   Como Lola trabaja con streams y expresiones tipadas, el tipo de datos que se use para representar a las expresiones debe estar parametrizado por el tipo de la expresión. Puntualmente, queremos aprovechar el sistema de tipos de Haskell para realizar el chequeo de tipos de las expresiones.

2. Ser extensible a nuevas interpretaciones semánticas.
   La generación automática de código es una tarea compleja que requiere de múltiples análisis, entre ellos:

   - El análisis temporal de las expresiones, para determinar las dependencias temporales entre los streams.
   - El análisis de tipos de las expresiones, para determinar todos los tipos que intervienen en la definición de los streams.
   - La traducción de las expresiones al lenguaje objetivo.

   Si bien estas interpretaciones semánticas están fijas, es deseable que el lenguaje permita modificarlas libremente, sin alterar otras partes del código, lo que facilita considerablemente la implementación. Además, si el

lenguaje es extensible a nuevas interpretaciones se puede generar código para otros lenguajes objetivo simplemente agregando nuevas traducciones, reutilizando los análisis temporal y de tipos.

3. Ser extensible a nuevos constructores.
   Uno de los principios de SRV reside en mantener por separado los cómputos temporales de los de datos, de manera tal que incorporar nuevas teorías no implique cambios significativos en la implementación. Por lo tanto, el lenguaje de expresiones debería ser extensible a nuevos constructores.

En esta sección se explica cómo hemos abordado el desarrollo de un lenguaje para expresiones Lola, siguiendo estos tres requerimientos.

5.3.1. Diseñando un lenguaje para expresiones

El objetivo del diseño del lenguaje para expresiones Lola es definir un tipo de datos para representar expresiones que respete, dentro de lo posible, las tres propiedades discutidas anteriormente.

Como se puede observar en el ejemplo del lenguaje de expresiones aritméticas presentado en la Sección 5.1, el uso de un GADT permite cubrir satisfactoriamente las primeras dos propiedades. Por un lado, provee un mecanismo para construir ASTs, haciendo que el lenguaje sea extensible a interpretaciones semánticas; y por el otro, permite explicitar el tipo de los constructores. Por lo tanto, nos propusimos como primera aproximación definir un tipo de datos **Exp** a utilizando un GADT, poblándolo con constructores adecuados que permitan representar tanto a los operadores temporales de Lola como a los correspondientes a las teorías de datos.

Respecto al primer grupo, Lola provee dos operadores temporales: *now* para referenciar el valor de un stream en el instante actual y *offset* para referenciar el valor de un stream en un instante pasado o futuro. Utilizando el tipo **Stream** definido anteriormente se pueden implementar ambos operadores como constructores de **Exp**, tal como ilustra el Código 5.4. En el caso de **Now**, sólo se requiere conocer el stream al que se hace referencia, mientras que en **Offset** además del stream se necesita conocer el corrimiento, que representamos con un valor de tipo **Int**, y la expresión a usar como valor por defecto, cuyo tipo debe coincidir con el tipo del stream.

```
data Exp a where
  Now    :: Stream a → Exp a
  Offset :: Stream a → Int → Exp a → Exp a
  ...
```

Código 5.4: Añadiendo constructores temporales.

Para añadir al lenguaje teorías de datos, extendemos el AST agregando nuevos constructores al tipo **Exp**, que representen a los distintos operadores de las teorías. Por ejemplo, para agregar la negación, disyunción, conjunción e implicación de valores booleanos; la comparación por igualdad; la construcción condicional $if \cdot then \cdot else$; y la suma, resta, multiplicación y división de valores enteros, podemos definir:

```
data Exp a where
  Now    :: Stream a → Exp a
  Offset :: Stream a → Int → Exp a → Exp a
  BLit   :: Bool → Exp Bool
  Not    :: Exp Bool → Exp Bool
  Or     :: Exp Bool → Exp Bool → Exp Bool
  And    :: Exp Bool → Exp Bool → Exp Bool
  Impl   :: Exp Bool → Exp Bool → Exp Bool
  Eq     :: Exp a → Exp a → Exp Bool
  Ite    :: Exp Bool → Exp a → Exp a → Exp a
  ILit   :: Int → Exp Int
  IAdd   :: Exp Int → Exp Int → Exp Int
  ISub   :: Exp Int → Exp Int → Exp Int
  IMul   :: Exp Int → Exp Int → Exp Int
  IDiv   :: Exp Int → Exp Int → Exp Int
```

Código 5.5: Añadiendo teorías de datos.

No obstante, el problema de este diseño es que el lenguaje no resulta extensible a nuevos constructores. En efecto, si quisiéramos agregar una nueva teoría de datos o extender alguna teoría existente con un nuevo operador, tendríamos que extender el tipo **Exp** a y, por lo tanto, modificar todas las funciones ya definidas sobre **Exp**. Afortunadamente, este problema ha sido estudiado en profundidad, conociéndose en la actualidad varias soluciones [49, 11, 28, 73, 79, 60]. En lo que sigue se citará y explicará brevemente la solución que hemos utilizado para este trabajo.

5.3.2. Expression Problem y Data types à la Carte

Recordemos la implementación como EDSL profundo del lenguaje de expresiones aritméticas de la Sección 5.1 provista en el Código 5.2b:

```
data Expr = Lit Int
          | Add Expr Expr
          | Sub Expr Expr

eval :: Expr → Int
eval (Lit n)   = n
eval (Add x y) = eval x + eval y
eval (Sub x y) = eval x − eval y
```

Una vez definido el tipo de datos **Expr**, podemos agregar nuevas funciones de interpretación libremente. Por ejemplo, podemos definir una función para mostrar expresiones:

```
render :: Expr → String
render (Lit n)   = show n
render (Add x y) = "(" ++ render x ++ " + " ++ render y ++ ")"
render (Sub x y) = "(" ++ render x ++ " − " ++ render y ++ ")"
```

Sin embargo, si queremos agregar nuevos operadores al lenguaje tendremos que extender el tipo **Expr**, lo que requerirá agregar los casos adicionales en todas las funciones ya definidas. En 1998, Philip Wadler introdujo el término *Expression Problem* para referirse a este problema [78]:

> *El objetivo es definir un tipo de datos por casos, de manera que sea posible agregar nuevos casos al tipo de datos y nuevas funciones sobre el tipo de datos, sin recompilar el código existente y conservando las garantías del tipado estático.*

Como ilustra el ejemplo anterior, el uso de ADTs (o GADTs) permite definir en Haskell tipos de datos por casos de manera tal que resulta muy sencillo agregar nuevas funciones, pero agregar nuevos constructores requiere modificar el código existente. En 2008, Wouter Swierstra publicó una solución al expression problem para Haskell titulada *Data types à la Carte* (*DTC*) [73]. A continuación se muestra cómo rediseñar el tipo **Expr** siguiendo esta técnica.

Tipo de datos para `Expr`

Si definimos **Expr** como un tipo de datos algebraico con un constructor de valores para cada constructor del lenguaje, no será posible añadir nuevos constructores sin tener que modificar el código existente. En lugar de tener que elegir de antemano un conjunto fijo de constructores, podemos definir un tipo de datos parametrizado por los constructores:

```
data Expr f = In (f (Expr f))
```

donde el parámetro de tipo `f` representa la signatura de los constructores. Más concretamente, `f` es un constructor de tipos cuyo parámetro de tipo se corresponde con las expresiones que aparecen como subárboles de los constructores. El tipo de datos **Expr** completa la definición recursiva usando a `Expr f` como argumento de `f`.

Por ejemplo, para representar constantes enteras se define un constructor de tipos **Lit**:

```
data Lit e = Lit Int
```

que no usa el parámetro `e` ya que no se define a partir de subexpresiones. Si quisiéramos un lenguaje que admita solamente constantes enteras basta con definir:

```
type ConstExpr = Expr Lit
```

donde las únicas expresiones válidas son de la forma `In (Lit x)` para algún `x :: Int`.

Análogamente, para representar la suma de expresiones se define otro constructor de tipos:

```
data Add e = Add e e
```

En este caso, como la suma se construye a partir de dos subexpresiones, el constructor **Add** toma dos argumentos de tipo `e`.

Para construir un lenguaje que contenga a ambos constructores, Swierstra propone tomar el coproducto de sus signaturas. El coproducto de dos signaturas es análogo al tipo **Either**, sólo que en lugar de combinar dos tipos base, i.e. de kind $*$, combina dos constructores de tipos de kind $* \to *$:

```
data (f :+: g) e = Inl (f e) | Inr (g e)
```

Ahora, podemos usar el tipo de datos **Expr** (**Lit** **:+:** **Add**) para representar a las expresiones aritméticas construidas a partir de la suma de enteros. Por ejemplo, el término $8 + 5$ se puede representar de la siguiente manera:

```
example1 :: Expr (Lit :+: Add)
example1 = In (Inr (Add (In (Inl (Lit 8))) (In (Inl (Lit 5)))))
```

Agregando funciones de interpretación

Supongamos que queremos definir una función `eval :: Expr f → Int` para evaluar expresiones del lenguaje. Debido a la naturaleza recursiva del tipo **Expr** `f`, necesitamos proveer un mecanismo para consumir expresiones, que nos permita especificar el comportamiento de `eval` para cada constructor del lenguaje, combinando recursivamente los resultados.

Sin embargo, los constructores del lenguaje no están fijos, sino que se determinan dinámicamente: por ejemplo, podemos tener un lenguaje dado por **Expr** (**Lit**) o **Expr** (**Lit** **:+:** **Add**). Para manejar este comportamiento dinámico, podemos usar el sistema de clases de Haskell. En particular, si:

- `f` es instancia de la clase **Functor**; y
- se tiene una función α `:: f a → a`, conocida como *álgebra* en el área de teoría de categorías, que determina cómo computar composicionalmente un resultado de tipo `a` correspondiente a una expresión construida con la signatura `f`

entonces podemos definir una función `foldExpr` para consumir expresiones de la siguiente manera:

```
foldExpr :: Functor f ⇒ (f a → a) → Expr f → a
foldExpr α (In t) = α (fmap (foldExpr α) t)
```

En particular, observamos que los constructores de tipos definidos para representar a los constructores del lenguaje son, efectivamente, instancias de la clase **Functor**:

```
instance Functor Lit where
  fmap f (Lit n) = Lit n

instance Functor Add where
  fmap f (Add x y) = Add (f x) (f y)
```

y que el coproducto preserva las instancias de **Functor** (es decir, el coproducto de dos funtores es un funtor):

```
instance (Functor f, Functor g) ⇒ Functor (f :+: g) where
  fmap f (Inl e) = Inl (fmap f e)
  fmap f (Inr e) = Inr (fmap f e)
```

Por lo tanto, para poder implementar la función `eval` usando `foldExpr`, sólo resta proveer un álgebra α `:: f a → a` que determine cómo evaluar cada constructor del lenguaje. Nuevamente, como los constructores del lenguaje se determinan dinámicamente, recurrimos al sistema de clases de Haskell. Puntualmente, podemos representar un álgebra utilizando una clase de tipos:

```
class Functor f ⇒ Eval f where
  evalAlgebra :: f Int → Int
```

Luego, para especificar el comportamiento de `evalAlgebra` definimos una instancia de la clase **Eval** para cada constructor del lenguaje:

```
instance Eval Lit where
  evalAlgebra (Lit n) = n

instance Eval Add where
  evalAlgebra (Add x y) = x + y
```

y para el coproducto:

```
instance (Eval f, Eval g) ⇒ Eval (f :+: g) where
  evalAlgebra (Inl x) = evalAlgebra x
  evalAlgebra (Inr y) = evalAlgebra y
```

Finalmente, para definir la función de evaluación sólo debemos consumir una expresión usando `evalAlgebra`:

```
eval :: Eval f ⇒ Expr f → Int
eval = foldExpr evalAlgebra
```

Esta técnica permite agregar nuevas funciones de interpretación sin tener que modificar el código existente. Simplemente se debe definir una clase de tipos que implemente un álgebra adecuada, definir las instancias pertinentes de esta clase y utilizar `foldExpr` para computar el resultado. Por ejemplo, para mostrar una expresión basta con agregar el siguiente código:

```
class Functor f ⇒ Render f where
  renderAlgebra :: f String → String

instance Render Lit where
  renderAlgebra (Lit n) = show n

instance Render Add where
  renderAlgebra (Add x y) = "(" ++ x ++ " + " ++ y ++ ")"

instance (Render f, Render g) ⇒ Render (f :+: g) where
  renderAlgebra (Inl x) = renderAlgebra x
```

```
  renderAlgebra (Inr y) = renderAlgebra y

render :: Render f ⇒ Expr f → String
render = foldExpr renderAlgebra
```

Agregando constructores

Para agregar un nuevo constructor al lenguaje se requiere definir un constructor de tipos adecuado, dar su instancia de **Functor** y extender el coproducto. Por ejemplo, para extender el lenguaje de expresiones aritméticas con la resta agregamos:

```
data Sub e = Sub e e

instance Functor Sub where
  fmap f (Sub x y) = Sub (f x) (f y)
```

pudiendo, por ejemplo, representar la expresión $5 + (9 - 8)$ de la siguiente manera:

```
example2 :: Expr ((Lit :+: Add) :+: Sub)
example2 = In ((Inl . Inr) (Add (In ((Inl . Inl) (Lit 5))) (In
    (Inr (Sub (In ((Inl . Inl) (Lit 9))) (In ((Inl . Inl) (Lit
    8))))))))
```

Como se aprecia en este ejemplo, el problema de esta representación es que para escribir expresiones se necesita explicitar la inyección de cada constructor en el coproducto, lo que rápidamente se vuelve inmanejable al extender el lenguaje.

Para solucionarlo, se introduce una clase de tipos `(:<:)` para automatizar la inyección de un funtor en un coproducto, lo que permite definir un *constructor inteligente* para cada constructor del lenguaje, evitando definiciones ilegibles como `example1` o `example2`. Se define la clase `(:<:)` de la siguiente forma:

```
class (Functor sub, Functor sup) ⇒ sub :<: sup where
  inj :: sub a → sup a
```

El objetivo de esta clase es poder escribir restricciones `sub :<: sup` para indicar que existe una inyección de `sub a` a `sup a`. De esta manera, en lugar de escribir las inyecciones a mano usando **Inl** e **Inr**, las mismas se infieren usando la clase `(:<:)`, que consta de tres instancias:

```
instance Functor f ⇒ f :<: f where
  inj = id

instance {-# OVERLAPPING #-} (Functor f, Functor g) ⇒ f :<: (f
    :+: g) where
  inj = Inl

instance (Functor f, Functor g, Functor h, f :<: g) ⇒ f :<: (h
    :+: g) where
  inj = Inr . inj
```

Observamos que el solapamiento existente entre la segunda y tercera instancia no genera un comportamiento impredecible si no se hace uso de la información relativa al orden de los constructores de una expresión. Asimismo, si bien la tercera instancia sólo busca la inyección del lado derecho, podemos garantizar que la encontrará siempre que exista haciendo que `(:+:)` asocie a derecha. Usando esta clase es posible definir constructores inteligentes de la siguiente manera:

```
infixr 1 :+:

inject :: (g :<: f) ⇒ g (Expr f) → Expr f
inject = In . inj

lit :: (Lit :<: f) ⇒ Int → Expr f
lit n = inject (Lit n)

add :: (Add :<: f) ⇒ Expr f → Expr f → Expr f
add x y = inject (Add x y)

sub :: (Sub :<: f) ⇒ Expr f → Expr f → Expr f
sub x y = inject (Sub x y)
```

Ahora, podemos reescribir `example1` y `example2` de manera mucho más legible:

```
example1 :: Expr (Lit :+: Add)
example1 = (lit 8) `add` (lit 5)

example2 :: Expr (Lit :+: Add :+: Sub)
example2 = (lit 5) `add` ((lit 9) `sub` (lit 8))
```

Regresando al objetivo original de añadir nuevos constructores al lenguaje, una vez definido el constructor de tipos correspondiente, dada su instancia de **Functor** y su constructor inteligente, sólo resta agregar las instancias correspondientes a las álgebras ya definidas, en este caso:

```
instance Eval Sub where
  evalAlgebra (Sub x y) = x − y

instance Render Sub where
  renderAlgebra (Sub x y) = "(" ++ x ++ " − " ++ y ++ ")"
```

De esta manera observamos que es posible extender el lenguaje con nuevos constructores sin tener que modificar el código existente. Más aún, como ilustran las signaturas de tipos de los ejemplos `example1` y `example2`, una vez definido un *menú* (una colección) de constructores, podemos combinarlos libremente para crear tipos de datos *à la carte*.

Comentarios

En síntesis, DTC resuelve el expression problem definiendo un tipo de datos recursivo usando el punto fijo de un funtor, que podemos definir como el

coproducto de uno o más funtores. Esto permite agregar tanto nuevos casos al tipo de datos como nuevas interpretaciones semánticas sin tener que modificar el código existente.

Sin embargo, esta solución tiene sus limitaciones. En particular, si bien es posible desde un punto de vista algebraico extender esta técnica para usarse con GADTS [43, 45], se requiere utilizar una representación de alto orden de los tipos de datos que puede resultar un tanto complicado e incómodo para programar en Haskell. Más aún, en ciertos casos puede requerirse el uso de clases multiparamétricas y otras extensiones de Haskell 98. Para facilitar la implementación de este proyecto, hemos utilizado una librería que implementa las ideas de DTC con una extensión para trabajar con GADTs.

5.3.3. Lenguaje para expresiones Lola

Para implementar el lenguaje para expresiones Lola construimos un AST equivalente al del Código 5.5, pero utilizando DTC para que el lenguaje resulte extensible a nuevos constructores.

Con el objetivo de facilitar la implementación, hemos usado la librería *compdata: Compositional Data Types* [12] que implementa las ideas de DTC según se explica en el paper "Compositional data types" [1]. Esta librería provee múltiples facilidades para manipular y analizar árboles de sintaxis abstracta, entre ellas:

- Facilidades para derivar código.
  Una gran parte de la implementación de DTC consiste en dar definiciones un tanto repetitivas que se podrían automatizar, como las instancias de functores de los constructores del lenguaje o sus constructores inteligentes. La librería *compdata* ofrece primitivas para derivar automáticamente algunas definiciones, facilitando así la implementación del lenguaje.

- Extensión para trabajar con GADTs.
  Para implementar DTC en el ejemplo de la Sección 5.3.2 se representó a cada constructor del lenguaje con un ADT, instancia de **Functor**. Sin embargo, en el caso de las expresiones Lola, un ADT no resulta lo suficientemente expresivo para representar a los constructores del lenguaje, puesto que trabajaremos con expresiones de distintos tipos y necesitamos explicitar los tipos de cada constructor. La librería *compdata* provee una extensión para trabajar con GADTs que extiende la noción de funtor a funtor de alto orden [44]. Concretamente, define una clase de tipos **HFunctor** que representa un mapeo entre mapeos de funtores, es decir un mapeo entre transformaciones naturales:

```
-- | Transformaciones naturales
type f :→ g = forall i . f i → g i

-- | Funtores de alto orden
```

```
class HFunctor h where
    hfmap :: (f :→ g) → h f :→ h g
```

De esta manera, para representar operadores de LOLA definiremos GADTs de kind $(* \to *) \to * \to *$, que sean instancias de la clase **HFunctor**.

Operadores temporales

Para representar a los operadores temporales de LOLA, definimos un GADT con dos constructores:

```
data Temp (e :: * → *) (a :: *) where
  Now    :: Streamable a ⇒ Stream a → Temp e a
  Offset :: Streamable a ⇒ Stream a → Int → e a → Temp e a
```

Mientras que el parámetro `e :: * → *` se corresponde con la signatura de las expresiones que aparecen como subárboles de los constructores, el parámetro `a :: *` determina el tipo de la expresión. Así, el tercer argumento del constructor **Offset** es de tipo `e a` ya que se corresponde con el árbol de una expresión de tipo `a`.

Para concluir, usamos el mecanismo ofrecido por *compdata* para automatizar algunas definiciones [13]. Puntualmente, la librería define una serie de funciones para derivar instancias de ciertas clases. Por ejemplo, la función `makeHFunctor` permite derivar una instancia de **HFunctor** para un constructor de tipos, mientras que `smartConstructors` deriva sus constructores inteligentes. Asimismo, provee la función `derive` para generar una lista de instancias para una lista de constructores. De esta forma, para derivar una instancia de **HFunctor** para **Temp** y sus constructores inteligentes, `iNow` e `iOffset`, incluimos el siguiente código:

```
$(derive [makeHFunctor, smartConstructors]
         [''Temp])
```

De esta manera, los usuarios de MCLOLA podrán usar los constructores `iNow` e `iOffset` para introducir referencias temporales en sus especificaciones. Observamos que para derivar código, la librería utiliza Template Haskell [74], por lo cual fue necesario agregar dicha extensión.

Teorías de datos

De la misma manera que se ha definido **Temp**, se pueden definir otras signaturas para cada teoría de datos que se requiera agregar al lenguaje. Por ejemplo, para definir una teoría **BoolTheory** conformada por los constructores booleanos del Código 5.5, definimos el siguiente GADT:

```
data BoolTheory (e :: * → *) (a :: *) where
  ValBool :: Bool → BoolTheory e Bool
  Not     :: e Bool → BoolTheory e Bool
  Or      :: e Bool → e Bool → BoolTheory e Bool
  And     :: e Bool → e Bool → BoolTheory e Bool
```

```
  Impl    :: e Bool → e Bool → BoolTheory e Bool
  Eq      :: e a → e a → BoolTheory e Bool
  Ite     :: e Bool → e a → e a → BoolTheory e a

$(derive [makeHFunctor, smartConstructors]
         [''BoolTheory])
```

Notar que mientras **Not**, **Or**, **And** e **Impl**, correspondientes a la negación, disyunción, conjunción e implicación lógica, se definen exclusivamente para expresiones de tipo **Bool**; **Eq** e **Ite**, correspondientes a la comparación por igualdad y a la expresión condicional $if \cdot then \cdot else$, utilizan la variable de tipos `a` para dar una definición polimórfica.

Del mismo modo podemos definir otras teorías, como **IntTheory**:

```
data IntTheory (e :: * → *) (a :: *) where
  ValInt :: Int → IntTheory e Int
  IAdd   :: e Int → e Int → IntTheory e Int
  ISub   :: e Int → e Int → IntTheory e Int
  IMul   :: e Int → e Int → IntTheory e Int
  IDiv   :: e Int → e Int → IntTheory e Int

$(derive [makeHFunctor, smartConstructors]
         [''IntTheory])
```

Una vez definidas las teorías de datos, las combinamos usando el operador `(:+:)` definido por la librería, que implementa el coproducto de signaturas de alto orden:

```
type Data = BoolTheory :+: IntTheory
```

Para incorporar nuevas teorías sólo se requiere extender el coproducto con nuevos operadores. De la misma manera, es posible extender teorías existentes. Por ejemplo, si se quiere extender la teoría booleana con un operador xor, podemos agregar el siguiente código:

```
data Xor (e :: * → *) (a :: *) where
  Xor :: e Bool → e Bool → Xor e Bool

$(derive [makeHFunctor, smartConstructors]
         [''Xor])

type ExtBoolTheory = BoolTheory :+: Xor
```

Asimismo, si se quiere tener más libertad para elegir operadores, se podría definir la teoría **BoolTheory** utilizando un GADT para cada operador, de manera análoga a la definición de **Xor**. Sólo se requiere incluir en el coproducto que define a **Data** aquellos operadores que queremos que formen parte del lenguaje.

Para consultar la implementación completa de las teorías de datos ofrecidas por mcLola, referirse al directorio `Theories` del código fuente.

Expresiones Lola

Finalmente, definimos el tipo **Exp** a para representar expresiones Lola de tipo a de la siguiente manera:

```
type Sig    = Temp :+: Data

type Exp a = Term Sig a
```

donde el constructor **Term** está definido por la librería *datacomp* e implementa el punto fijo de las signaturas definidas por **Sig**, de manera análoga a la definición de **Expr** vista en la implementación de DTC para el ejemplo de las expresiones aritméticas de la Sección 5.3.2.

Capítulo 6

Generación de Monitores

En este capítulo se explica el proceso de generación de monitores C a partir de una especificación mcLola escrita por el usuario. Primero se describe la implementación del análisis estático, donde se extrae información de la especificación necesaria para las etapas posteriores de la generación de código. Luego, se discuten cuestiones generales relativas a la síntesis de código C, como las herramientas utilizadas y las propiedades del código generado. Finalmente, se explica cómo se implementó el engine temporal y, en particular, la traducción de expresiones mcLola a código C99.

6.1. Análisis estático

El análisis estático constituye el primer paso hacia la generación de código. En esta etapa se aplican diversos análisis sobre la especificación escrita por el usuario con el objetivo de cubrir dos tareas fundamentales:

- Validar la especificación.
- Extraer información necesaria para la generación del monitor.

Estos análisis se basan principalmente en cuestiones temporales del lenguaje Lola, lo que denominamos *análisis temporal* de la especificación, aunque se realiza, también, un *análisis de tipos*.

La implementación de los análisis descriptos en esta sección puede consultarse en el directorio `StaticAnalysis` del código fuente.

6.1.1. Validación de la especificación y grafo de dependencias

No toda especificación Lola gramáticamente correcta es válida. Al escribir una especificación mcLola, algunos errores, como errores sintácticos, de tipos o de referencias, pueden ser chequeados por el compilador de Haskell; pero para garantizar que sea posible generar un monitor ejecutable, la especificación debe estar *bien definida* y ser *eficientemente monitorizable*.

Una especificación está bien definida si para cada posible conjunto de entradas existe un único modelo de evaluación válido (Definición 3.8). Esta condición permite garantizar que para cualquier entrada válida el monitor producirá una única respuesta. Sin embargo, se trata de una condición semántica difícil de verificar en el caso general. Por este motivo, MCLOLA valida, en su lugar, una propiedad sintáctica más restrictiva: que la especificación esté bien formada, es decir que su grafo de dependencias no tenga ciclos de peso 0 (Teorema 3.11).

Además de que exista una única respuesta para cada posible entrada, debemos asegurar que el monitor pueda calcularla, en particular que los requerimientos espaciales para computar las salidas sean acotados. Formalmente, una especificación es eficientemente monitorizable si el espacio en memoria que necesita el monitor es independiente de la longitud de los streams (Definición 3.17). Una vez más, no es fácil verificar esta propiedad para el caso general, con lo cual verificamos que la especificación sea acotada a futuro, es decir que su grafo de dependencias no tenga ciclos de peso positivo (Corolario 3.22).

De esta manera, para validar una especificación sólo se necesita construir su grafo de dependencias para determinar si tiene ciclos de peso mayor o igual a 0. Como se explica más adelante, la utilidad del grafo de dependencias no se limita únicamente a la validación de la especificación; por el contrario, constituye el punto de partida del resto de los análisis temporales de la especificación.

Grafo de dependencias

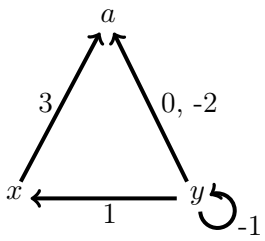
Recordemos que el grafo de dependencias de una especificación (Definición 3.9) es un multigrafo dirigido ponderado dado por un conjunto V de vértices y E de aristas, donde:

- V está dado por el conjunto de los nombres de los streams de la especificación.
- E contiene una arista $x \xrightarrow{0} v$ si x es un stream de salida cuya definición cuenta con una ocurrencia de v; es decir, una referencia a v en el instante actual.
- E contiene una arista $x \xrightarrow{k} v$ si x es un stream de salida cuya definición cuenta con una ocurrencia de v[k|d] para algún d; es decir, una referencia a v con un corrimiento de k instantes de tiempo.

Observamos que podemos representar a todo grafo de dependencias utilizando un grafo simple, en lugar de un multigrafo, si etiquetamos sus aristas con listas de enteros en lugar de enteros, como se muestra en el siguiente ejemplo.

Ejemplo 6.1

```
input  int a
output int x = a[3|0] + 5
output int y = y[-1|0] + (a + a[-2|1] - x[1|1]) * 2
```



◇

Basándonos en esta última caracterización hemos definido un tipo de datos **DGraph** para representar grafos de dependencias mediante listas de adyacencia:

```
type Vert   = String
type DGraph = Map Vert (Map Vert [Int])
```

donde **Map** es el tipo de datos definido en la librería Data.Map que provee una implementación eficiente de diccionarios.

De esta manera, podemos representar al grafo de dependencias del Ejemplo 6.1 como sigue:

```
fromList [("a",fromList []),
          ("x",fromList [("a",[3])]),
          ("y",fromList [("x",[1]), ("a",[0,-2]), ("y",[-1])])]
```

Construcción del grafo de dependencias

Para construir el grafo de dependencias de una especificación se requiere conocer sus dependencias temporales. En particular, se necesita computar para cada stream `s` una lista de tipo `[(Vert, Int)]` con todas las referencias temporales de `s` a otros streams. Por ejemplo, al stream `y` del Ejemplo 6.1 le corresponde la lista `[("x", 1), ("a", 0), ("a", -2), ("y", -1)]`.

Observamos que si `s` es un stream de entrada, dicha lista será la lista vacía puesto que, por definición, no depende de otros streams. En cambio, si `s` es un stream de salida tendremos que recorrer la expresión que lo define para identificar todas las ocurrencias de los operadores **Now** y **Offset**. Hemos dado, entonces, con la primera función de interpretación del EDSL. Concretamente, se requiere definir una función computeTempRef que dada una expresión MCLOLA, compute una lista `[(Vert, Int)]` con las referencias temporales que ocurren en ella.

Para implementar una función de interpretación se debe definir una clase de tipos *Alg* que implemente un álgebra α adecuada, cuyo comportamiento está definido por las instancias de *Alg* para cada constructor del lenguaje; para luego consumir las expresiones del lenguaje con α.

En la Sección 5.3.2 se representó un álgebra como una función $\alpha :: \mathtt{f\ a} \to \mathtt{a}$ para computar composicionalmente un resultado de tipo `a` correspondiente a una expresión construida con una signatura `f`, donde `f` es un funtor. En este caso, dado que las signaturas que definen al lenguaje de expresiones Lola constituyen funtores de alto orden, se requiere de una representación de alto orden para las álgebras. Para ello, *compdata* provee un tipo de datos **Alg** para computar resultados sobre expresiones construidas con una signatura de alto orden `f :: (* → *) → * → *`:

```
type Alg f e = f e :→ e
```

Siguiendo esta representación, para computar las referencias temporales de una expresión necesitamos:

1. Definir una clase de tipos **TempRefAlg** que implemente un álgebra para computar referencias temporales.

   Para ello, primero se define un constructor de tipos **TempRef** `:: * → *` para representar el resultado a computar:

   ```
   data TempRef a where
     TR :: [(Vert, Int)] → TempRef a

   unTR (TR x) = x
   ```

   Luego, usamos el constructor **Alg** para definir la clase **TempRefAlg**:

   ```
   class TempRefAlg f where
     tempRefAlg :: Alg f TempRef
   ```

2. Dar las instancias de la clase **TempRefAlg** para cada constructor del lenguaje y para el coproducto.

   Los casos interesantes corresponden a los constructores **Now** y **Offset**:

   ```
   instance TempRefAlg Temp where
     tempRefAlg (Now s)          = TR $ [(ident s, 0)]
     tempRefAlg (Offset s k x) = TR $ [(ident s, k)] ++ unTR x
   ```

   mientras que para el resto de los constructores, simplemente se devuelve el resultado (recursivo) de sus subexpresiones, por ejemplo:

   ```
   instance TempRefAlg BoolTheory where
     tempRefAlg (ValBool _) = TR []
     tempRefAlg (Or x y)      = TR $ unTR x ++ unTR y
     ...
   ```

   Para el coproducto, podemos derivar la instancia de **TempRefAlg** con la función `liftSum` provista por la librería:

```
$(derive [liftSum] ["TempRefAlg])
```

3. Finalmente, para computar la lista de referencias temporales de una expresión se aplica recursivamente sobre ella el álgebra `tempRefAlg`. Para ello, se usa la función `cata` provista por *compdata*, que construye un catamorfismo para un álgebra dada, el equivalente de alto orden a la función `foldExpr` definida para el ejemplo de expresiones aritméticas:

```
computeTempRef :: (HFunctor e, TempRefAlg e) ⇒
    Term e a → [(Vert, Int)]
computeTempRef t = unTR (cata tempRefAlg t)
```

Para concluir se define `createDGraph :: Specification → DGraph`, una función que construye el grafo de dependencias de una especificación a partir del resultado de aplicar `computeTempRef` sobre las expresiones que definen a cada stream de salida.

Validación del grafo de dependencias

Construido el grafo de dependencias, para validar la especificación sólo resta determinar si el mismo contiene un ciclo de peso mayor o igual a 0. Para ello, se define la función `checkDGraph :: DGraph → Bool` que implementa un algoritmo DFS para detectar ciclos de peso no negativo.

Si existe un ciclo de peso no negativo, entonces se rechaza la especificación, interrumpiéndose la generación de código. En caso contrario, se continúa con el análisis temporal.

6.1.2. Información para el engine temporal

El grafo de dependencias de una especificación resume su información temporal. A partir del mismo pueden computarse indicadores temporales como el back-reference y la latencia de cada stream, los que a su vez permiten extraer información relevante a la implementación del engine temporal.

A continuación se explica cómo se computan en MCLOLA estos indicadores y cómo se calcula a partir de ellos el tamaño de los buffers que almacenan los valores de los streams y el orden en que éstos deben procesarse en cada iteración del engine.

Back-reference y latencia

Como se enuncia en la Definición 3.13, el back-reference de un stream s, denotado ∇s, es el máximo $k \geq 0$ tal que $s(i)$ es utilizado para computar $u(i+k)$ para algún stream u. Formalmente:

$$\nabla s \;=\; max\left(\{0\} \cup \left\{k \;\middle|\; u \xrightarrow{-k} s \in E\right\}\right)$$

donde E es el conjunto de aristas del grafo de dependencias de la especificación. El back-reference de un stream provee una cota para la cantidad de instantes de tiempo que el valor de un stream debe ser recordado para un uso posterior, y por ende ofrece una cota inferior para la cantidad de memoria necesaria.

Recordamos, también, que la latencia de un stream s (Definición 3.19), que notamos Δs, se define como el máximo peso de un camino en el grafo de dependencias que comience en el vértice s, o 0 si no existen tales caminos o si todos ellos son de peso negativo:

$$\Delta s \;=\; max\left(\{0\} \cup \left\{ \sum_{i=1}^{n} k_i \;\middle|\; s \xrightarrow{k_1} u_1, u_1 \xrightarrow{k_2} u_2, \ldots, u_{n-1} \xrightarrow{k_n} u_n \in E \right\}\right)$$

La latencia de s indica la máxima cantidad de instantes de tiempo que deben transcurrir para asegurar que el algoritmo de monitoreo compute el valor de s en una posición dada (Teorema 3.20). En otras palabras, para todo stream s e instante i, $s(i)$ será calculado antes del instante $i + \Delta s + 1$.

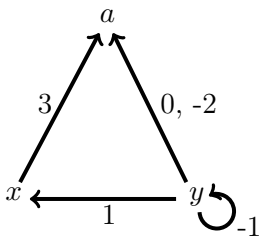
Una vez construido el grafo de dependencias de la especificación, resulta sencillo calcular el back-reference y la latencia de los streams. En el primer caso basta con comparar el peso de las aristas entrantes en cada vértice del grafo de dependencias; mientras que para calcular la latencia de s se puede realizar un recorrido DFS para computar el máximo peso de un camino con inicio en s. Concretamente, se han definido las funciones `createRefMap` y `createLatMap` que construyen, a partir del grafo de dependencias de la especificación, un mapeo de nombres de streams a su back-reference y a su latencia, respectivamente.

Tamaño de buffers

El uso de referencias temporales en una especificación Lola introduce la necesidad de almacenar el valor de algunos streams para poder completar cálculos futuros. Por ejemplo, en la especificación del Ejemplo 6.1, el valor de $a(0)$, que se recibe en el instante 0, debe conservarse por al menos dos instantes de tiempo, ya que en el instante 2 será necesario para calcular $y(2)$.

mcLola provee una implementación de buffers para almacenar una determinada cantidad de valores temporalmente contiguos para cada stream. Notar que la elección del tamaño no es una cuestión menor: idealmente, debe elegirse el menor tamaño que permita recuperar todos los valores que el algoritmo de monitoreo necesita para completar los cómputos. En particular, hemos simplificado el problema utilizando el mismo tamaño para todos los streams.

Dado que nuestra implementación utiliza una versión modificada del algoritmo de monitoreo, que en lugar de trabajar con expresiones parcialmente evaluadas resuelve expresiones cuando todos los valores de los que depende están disponibles, el back-reference provee sólo una cota inferior para la cantidad



	back-ref	latencia
a	2	0
x	0	3
y	1	4

Figura 6.2: Grafo de dependencias, back-reference y latencia de la especificación del Ejemplo 6.1.

de instantes de tiempo que deben conservase los valores. Considérese, nuevamente, el grafo de dependencias de la especificación del Ejemplo 6.1, junto al back-reference y la latencia de sus streams que se ilustran en la Figura 6.2. Si bien el back-reference de a es 2, notar que un buffer de tamaño 2 no resulta suficiente para almacenar sus valores. En particular, para computar $y(i)$ además de necesitarse los valores de $a(i-2)$ y $a(i)$, se requiere conocer $x(i+1)$, que a su vez se define a partir de $a(i+4)$. Por lo tanto, para computar $y(i)$ necesitaremos acceder tanto al valor de $a(i-2)$ como de $a(i+4)$, con lo cual el tamaño del buffer no puede ser menor a 7.

De esta forma, observamos que el tamaño de los buffers depende tanto del back-reference como de la latencia de los streams. El siguiente teorema formaliza esta noción.

Teorema 6.3. *Sea size el tamaño de los buffers que almacenan los valores de los streams de una especificación φ acotada a futuro. Si*

$$size \;=\; max\{\nabla s \,|\, s \in \varphi\} + max\{\Delta s \,|\, s \in \varphi\} + 1$$

entonces, la versión modificada del algoritmo de monitoreo podrá recuperar todos los valores necesarios para completar los cómputos de streams.

Demostración. Sean $B = max\{\nabla s \,|\, s \in \varphi\}$, $L = max\{\Delta s \,|\, s \in \varphi\}$ y (V, E) el grafo de dependencias de la especificación φ.

Sea $s \in \varphi$. El algoritmo de monitoreo podrá calcular $s(i)$ en un instante i' tal que $i \leq i' \leq i + \Delta s$ (Teorema 3.20). Veamos que si se usan buffers de tamaño $size = B + L + 1$ es posible recuperar en el instante i' todos los valores de streams necesarios para computar $s(i)$:

- Si s no tiene referencias temporales, vale trivialmente.
- En caso contrario, existe $s \xrightarrow{k} u \in E$. Veamos que es posible recuperar el valor de $u(i+k)$ en el instante i' usando un buffer de tamaño *size*. Procedemos por casos en k:
  - $k \leq 0$

Tenemos que $-\nabla u \leq k \leq 0$ y, por lo tanto, $i - \nabla u \leq i + k \leq i$.
Entonces, para recuperar $u(i + k)$ en el instante i' el tamaño del buffer debe ser al menos $i' - (i - \nabla u) + 1$, lo cual vale puesto que:

$$\begin{aligned} i' - (i - \nabla u) + 1 &\leq (i + \Delta s) - (i - \nabla u) + 1 \\ &= \Delta s + \nabla u + 1 \\ &\leq L + B + 1 \\ &= size \end{aligned}$$

- $k > 0$
  Tenemos que $0 < k \leq \Delta s$ y, por lo tanto, $i < i + k \leq i + \Delta s$.
  Entonces, para recuperar $u(i + k)$ en el instante i' el tamaño del buffer debe ser al menos $i' - i$, lo cual vale puesto que:

$$i' - i \leq (i + \Delta s) - i = \Delta s \leq L \leq L + B + 1 = size$$

□

Observamos, además, que si el tamaño de los buffers es menor al valor dado por $max\{\nabla s \,|\, s \in \varphi\} + max\{\Delta s \,|\, s \in \varphi\} + 1$ no está garantizado que se puedan computar todos los valores de los streams. Basta considerar la especificación del Ejemplo 6.1, donde $max\{\nabla s \,|\, s \in \varphi\} + max\{\Delta s \,|\, s \in \varphi\} + 1 = 2 + 4 + 1 = 7$, y como se explicó anteriormente, las dependencias temporales del stream y requieren que el tamaño del buffer correspondiente al stream a sea, al menos, 7. Por lo tanto, elegimos como tamaño de buffer:

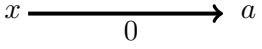
$$size \;=\; max\{\nabla s \,|\, s \in \varphi\} + max\{\Delta s \,|\, s \in \varphi\} + 1$$

Ordenamiento de streams

Para implementar el engine temporal, MCLOLA genera un monitor que procesa cada stream una única vez por iteración, siempre en un mismo orden secuencial previamente elegido. Por ejemplo, para la especificación del Ejemplo 6.1 se utiliza el orden: a, x, y.

Observar que al procesarse cada stream una única vez por iteración, el orden de procesamiento resulta relevante para garantizar que cada stream s sea calculado en, a lo sumo, Δs instantes de tiempo como indica el Teorema 3.20. Considérese como ejemplo la siguiente especificación, con su grafo de dependencias:

```
input  int a
output int x = a + 2
```



Notamos que $\Delta x = \Delta a = 0$, con lo cual $x(i)$ y $a(i)$ deberían poder computarse en el instante i. Sin embargo, si en las iteraciones del engine se procesa

x antes que a no será posible computar $x(i)$ en el instante i puesto que $a(i)$ aún no estará disponible.

De esta manera, observamos que las dependencias temporales de una especificación inducen un orden parcial para el procesamiento de los streams. Concretamente, para que un stream u necesite procesarse antes que s debe ocurrir que:

- s referencie a u, i.e. existe k tal que $s \xrightarrow{k} u$.
- $u(i+k)$ sea calculado en el mismo instante que $s(i)$, es decir que $i+\Delta s = i + k + \Delta u$ o, equivalentemente, $\Delta s - \Delta u = k$.

Por lo tanto, para representar las restricciones en el orden de procesamiento necesarias para garantizar que todo stream s sea calculado en a lo sumo Δs instantes de tiempo, podemos construir un grafo como se describe a continuación.

Definición 6.4. *El grafo de ordenamiento de una especificación es un grafo simple dirigido (V, E) donde:*

- *V está dado por el conjunto de los nombres de los streams de la especificación.*
- *E contiene una arista $u \to s$ si $s \xrightarrow{k} u$ es una arista del grafo de dependencias de la especificación con $\Delta s - \Delta u = k$.*

La Figura 6.5 muestra el grafo de ordenamiento correspondiente a la especificación del Ejemplo 6.1.

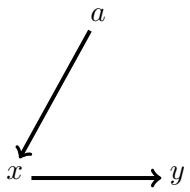


Figura 6.5: Grafo de ordenamiento de la especificación del Ejemplo 6.1.

En particular, el grafo de ordenamiento verifica la siguiente propiedad:

Teorema 6.6. *El grafo de ordenamiento de una especificación bien formada es acíclico.*

Demostración. Sean $DG = (V, E_D)$ y $OG = (V, E_O)$, respectivamente, los grafos de dependencias y de ordenamiento de una especificación bien formada.

Supongamos que existe un ciclo $s_0 \to s_1 \to \cdots \to s_{n-1} \to s_0$ en OG. Luego, existen $\{k_i\}_{i=0}^{n-1}$ tales que $s_{mod(i+1,n)} \xrightarrow{k_i} s_i \in E_D$ y $\Delta s_{mod(i+1,n)} - \Delta s_i = k_i$.

Por lo tanto, existe en DG un ciclo de peso

$$\sum_{i=0}^{n-1} k_i = \sum_{i=0}^{n-1} \Delta s_{mod(i+1,n)} - \Delta s_i = 0$$

lo que contradice la hipótesis de que la especificación está bien formada.

$\therefore OG$ es acíclico. □

Por el Teorema 6.6, el grafo de ordenamiento de una especificación válida es un grafo acíclico dirigido y, por lo tanto, se puede aplicar un ordenamiento topológico sobre él. De esta manera, para obtener un orden de procesamiento que garantice que todo stream s sea calculado en a lo sumo Δs instantes de tiempo, simplemente se construye el grafo de ordenamiento de la especificación a partir del grafo de dependencias y de los valores de latencia previamente computados; y se implementa un algoritmo DFS para calcular una ordenación topológica del mismo.

6.1.3. Información de tipos

Para completar el análisis estático, se identifican los tipos de datos utilizados en la especificación con el objetivo de traducirlos a tipos C equivalentes en una etapa posterior en el proceso de generación de código. Concretamente, el *análisis de tipos* se ocupa de:

- Identificar el tipo de cada stream.
- Identificar el conjunto de todos los tipos que participan en la definición de la especificación.

Como se explicó en la Sección 5.2, se utiliza el tipo **TypeRepr** para representar los tipos de datos soportados por MCLOLA. Usando esta representación, hemos definido dos funciones para implementar el análisis de tipos:

- `createTypeMap :: Specification → Map Identifier TypeRepr`, que dada una especificación construye un mapeo que asocia al nombre de cada stream el elemento de tipo **TypeRepr** que representa su tipo.
- `computeTypes :: Specification → [TypeRepr]`, que construye una lista con todos los tipos de datos que aparecen en una especificación.

Mientras que la implementación de `createTypeMap` es inmediata, para definir `computeTypes` se requiere de un análisis más sofisticado. Notar que los tipos de los streams no son, necesariamente, los únicos tipos que intervienen en la especificación. En particular, las subexpresiones que definen a los streams de salida pueden introducir nuevos tipos. Por ejemplo, en la siguiente especificación, todos los streams son de tipo **int**, pero el uso del constructor $if \cdot then \cdot else$ introduce el tipo **bool**:

```
input  int p
output int q = if (p < p[-1|0]) then 1 else 2
```

Por lo tanto, para determinar el conjunto de todos los tipos que intervienen en la especificación, se requiere determinar el tipo de todas las subexpresiones que conforman la expresión que define a cada stream de salida. Formalmente, dada una expresión Lola e, si $tipo(e)$ denota el tipo de e, es posible calcular el conjunto de todos los tipos que aparecen en e de la siguiente manera:

$$tipos(e) = \begin{cases} \{tipo(e)\} & \text{si } e \text{ es una constante} \\ \bigcup_{i=1}^{n} tipos(e_i) \cup \{tipo(e)\} & \text{si } e = op\ e_1\ e_2\ \ldots\ e_n \end{cases}$$

Para implementar este cómputo, debemos definir una función de interpretación sobre el conjunto de expresiones del lenguaje. Para ello, definimos una nueva clase de tipos para representar un álgebra de alto orden adecuado:

```
class TypesAlg e where
  typesAlg :: Alg e Types
```

donde `Types :: * → *` representa el resultado a computar.

Como queremos calcular una lista con todos los tipos que aparecen en la especificación, resulta razonable utilizar el tipo `[TypeRepr]` para definir `Types`:

```
data Types e where
  Types :: [TypeRepr] → Types e
```

Sin embargo esta representación presenta un problema. En particular, necesitamos computar el tipo concreto de la expresión (que hemos notado con $tipo(e)$), lo que muchas veces depende del tipo de sus argumentos, como en el caso del constructor $if \cdot then \cdot else$ cuyo tipo es el tipo del segundo (o tercer) argumento. Si elegimos el tipo `[TypeRepr]` para representar el resultado del cómputo, sabremos cuáles son todos los tipos usados en cada argumento, pero no el tipo de los argumentos. Esto se soluciona fácilmente agregando un elemento de tipo `TypeRepr` para indicar el tipo de la expresión:

```
data Types e where
  Types :: ([TypeRepr], TypeRepr) → Types e
```

No obstante, en esta representación persiste otro problema: decidir el tipo de una expresión construida con el operador `Now`, ya que para eso se requiere conocer el tipo del stream. Para lograrlo basta con usar la mónada `Reader` [67], que permite consultar los valores de un entorno dado:

```
type TypeMap = Map Identifier TypeRepr

data Types e where
  Types :: Reader TypeMap ([TypeRepr], TypeRepr) → Types e

unTypes (Types x) = x
```

Definida la clase `TypesAlg`, procedemos a dar las instancias para cada constructor del lenguaje, por ejemplo:

```
instance TypesAlg BoolTheory where
  typesAlg (ValBool _) = Types $ return ([TBool], TBool)
  typesAlg (Ite b x y) =
    Types (do (tsb, _) ← unTypes b
              (tsx, t) ← unTypes x
              (tsy, _) ← unTypes y
              return (tsb ++ tsx ++ tsy, t))
  typesAlg (Eq x y) =
    Types (do (tsx, _) ← unTypes x
              (tsy, _) ← unTypes y
              return ([TBool] ++ tsx ++ tsy, TBool))
  ...

instance TypesAlg Temp where
  typesAlg (Now s) =
    Types (do tm ← ask
              let t = getType (ident s) tm
              return ([t], t))
  typesAlg (Offset s k x) = x
```

Por último, definimos la función `exprTypes` para computar recursivamente los tipos de una expresión:

```
exprTypes :: (HFunctor e, TypesAlg e) ⇒ Term e a → Reader
    TypeMap ([TypeRepr], TypeRepr)
exprTypes t = unTypes (cata typesAlg t)
```

De esta manera, para obtener la lista de los tipos correspondientes a una expresión `e` simplemente ejecutamos la mónada `exprTypes e` utilizando como entorno el mapeo de streams a tipos calculado por `createTypeMap`.

Por otro lado, teniendo en cuenta que utilizaremos la lista de identificadores de tipos de una especificación para generar un tipo C adecuado para cada uno de sus elementos, debemos garantizar algunas propiedades:

- Que la lista no tenga elementos repetidos. De lo contrario, se duplicará una definición de tipos C.
- Que la lista contenga todas las definiciones necesarias para la traducción, en un orden apropiado para generar definiciones C válidas. Por ejemplo, si la lista contiene el tipo **TProd TInt TDouble**, para generar un tipo C equivalente se requieren tipos C para **TInt** y **TDouble**. Entonces, **TInt** y **TDouble** deben incluirse en la lista antes que **TProd TInt TDouble**.
- Si `t` es alguno de los tipos computados originariamente o agregado para satisfacer la propiedad anterior, entonces **TMaybe** `t` también debe ser incluido en la lista. Al trabajar con referencias a futuro se necesita expresar que un valor puede o no ser conocido, para lo cual usamos tipos opcionales.

Por lo tanto, para definir `computeTypes` primero calculamos los tipos que intervienen en la definición de cada stream, utilizando la función `exprTypes`

para el caso de los streams de salida, y luego reunimos esos resultados en una única lista, haciendo los ajustes necesarios para satisfacer las tres propiedades enunciadas.

6.2. Generación de código C

El proceso de generación de código C correspondiente al monitor de una especificación MCLOLA se organiza en dos etapas independientes:

- Implementación de tipos de datos.
- Implementación del engine temporal.

A continuación se aborda el desarrollo de estas dos etapas, cuya implementación puede consultarse en los directorios `Data` y `Engine` del código fuente, respectivamente.

6.2.1. Herramientas para la generación de código C

Con el objetivo de facilitar la implementación del backend, hemos considerado diferentes herramientas para la generación de código C desde Haskell, que pudiesen adaptarse satisfactoriamente a las necesidades puntuales del proyecto. A continuación se discuten brevemente las principales alternativas analizadas.

Ivory

Ivory [42] es un lenguaje de dominio específico embebido en Haskell diseñado para la programación de sistemas críticos, provisto de un backend para producir código C.

La principal característica de Ivory es que garantiza por construcción ciertas propiedades relativas a la seguridad de tipos y el acceso a memoria, a la vez que permite al usuario especificar otras propiedades de seguridad que pueden ser verificadas por demostradores externos, lo que lo hace particularmente adecuado para el desarrollo de sistemas críticos. Para alcanzar estas garantías, Ivory elimina varias fuentes de comportamiento dinámico, centrándose fundamentalmente en la prohibición del uso de memoria dinámica.

Tomando en consideración las ventajas en materia de seguridad, hemos utilizado la herramienta para implementar algunos ejemplos sencillos con el objetivo de familiarizarnos con el entorno y evaluar la viabilidad de su uso. Durante este proceso hemos notado que, para satisfacer muchas de las garantías de seguridad, Ivory demanda mucho conocimiento estático del programa, volviéndose inadecuado para la generación automática de monitores para especificaciones arbitrarias, en contraposición con el uso de la herramienta para generar código C específico. Por otra parte, el código C que la herramienta genera resulta un tanto lejano a lo que un programador escribiría debido, principalmente, a la falta de uso de nombres significativos para las variables.

Language-c99

La librería *language-c99* [22] de Haskell, desarrollada en el marco de Copilot [62], provee una implementación del AST de C99, según se describe en el estándar ISO/IEC 9899:TC3 [41] publicado en 2007, junto con un pretty printer para generar código C99.

Además, provee una extensión llamada *language-c99-simple* [23], que implementa una versión simplificada del AST de C99 para facilitar la escritura de programas, junto con un mecanismo de traducción del AST simplificado al AST de *language-c99*, que permite utilizar el pretty printer para la generación de código. Si bien este AST constituye un subconjunto propio del original, incluye las características más usadas de C99, que resultan suficientemente expresivas para la mayoría de las aplicaciones.

Trabajar con el AST de programas C99 provee una mayor libertad para elegir cómo implementar los monitores, permitiendo, en particular, generar código más cercano a lo que un programador escribiría. Por este motivo, y considerando la experiencia exitosa en el desarrollo de Copilot, hemos decidido utilizar *language-c99-simple* como herramienta para la generación de código C. Sin embargo, esta flexibilidad para escribir programas C arbitrarios trae como contrapartida la necesidad de tomar ciertos recaudos para evitar potenciales errores de programación que podrían ocasionar fallas en la seguridad. A continuación se explica cómo hemos abordado este problema.

Misra C

Desde la publicación de la primera versión de *The C Programming Language* en 1978, el lenguaje C fue ganando popularidad hasta convertirse en uno de los lenguajes de programación más usados. La eficiencia de sus programas y la experiencia de uso en sistemas críticos son algunos de los factores a los que se atribuye su popularidad. Sin embargo, algunos de sus aspectos pueden contribuir a cometer errores e introducir comportamientos indeseados, en particular:

- La definición del lenguaje.
  El estándar ISO no especifica el lenguaje por completo, dejando algunas áreas con comportamiento indefinido, inespecificado o a elección de la implementación.

- El uso incorrecto del lenguaje.
  Algunos aspectos de la sintaxis de C facilitan la escritura de código ofuscado y contribuyen a errores de programación difíciles de detectar.

- La incorrecta interpretación del lenguaje.
  Por ejemplo, algunas reglas de precedencia y de tipado pueden resultar confusas para programadores familiarizados con otros lenguajes.

- Chequeos en tiempo de ejecución.
  Los programas C generalmente no proveen chequeos en tiempo de ejecución para problemas comunes como división por cero, overflow, validez de punteros o errores de límites de arreglos.

Misra C [58] es un conjunto de recomendaciones y reglas para el desarrollo de software escrito en C confeccionado por *Motor Industry Software Reliability Association* (*MISRA*) [56], cuyo principal objetivo es contribuir a la seguridad, fiabilidad y portabilidad del código en el contexto del software embebido. En particular, Misra C es utilizado en el desarrollo de software crítico, como en el caso de los sectores automovilístico, ferroviario, aeroespacial y médico.

En general, estas reglas buscan evadir problemas frecuentes de la programación en C relacionados con los aspectos mencionados anteriormente, mediante una serie de buenas prácticas de programación. Las mismas se centran, fundamentalmente, en evitar el uso de funciones y estructuras propensas a error, restringir el uso de memoria dinámica y evadir el uso de características cuyo comportamiento pueda diferir entre distintos compiladores.

La tercera (y más reciente) versión de Misra C es MISRA-C:2012, definida por 143 reglas y 16 directivas clasificadas en tres categorías: *obligatorias*, *requeridas* y *recomendables*. El primer grupo concierne a aquellas normas que deben verificarse sin excepción para que el código cumpla con MISRA-C:2012; mientras que las directrices del segundo grupo pueden no cumplirse si se provee una justificación adecuadamente documentada, conocida como *desviación*. La última categoría agrupa directrices que no revisten carácter obligatorio, pero cuyo cumplimiento se alienta siempre que sea razonable.

Para el desarrollo de este proyecto hemos decidido seguir, dentro de lo posible, los lineamientos de MISRA-C:2012. Los esfuerzos se dirigieron, principalmente, a evitar el uso de memoria dinámica y a definir y usar adecuadamente las estructuras de datos.

Verificar el cumplimiento de las directrices es una tarea compleja, más aún cuando se trata de código generado automáticamente. Mientras muchas de ellas pueden chequearse con herramientas de análisis estático, algunas requieren analizar el comportamiento dinámico.

En este proyecto se ha analizado el cumplimiento de los lineamientos de MISRA-C:2012 en el código generado para algunas especificaciones relativamente complejas. Para ello se utilizó *Cppcheck* [17], una herramienta *open source* para el análisis estático de código C que provee una extensión [18] para verificar el cumplimiento de buena parte de las reglas de MISRA-C:2012. Si bien se encuentra en proceso de desarrollo, a la fecha permite chequear 105 de las 143 directrices. Aquellas reglas no implementadas por la herramienta fueron inspeccionadas manualmente. En el Apéndice A.3 se detallan las desviaciones a las reglas utilizadas en mcLola.

6.2.2. Generación de tipos de datos

La generación de tipos de datos se ocupa de la implementación en C99 de los tipos de datos utilizados por la especificación, como así también de la implementación de los buffers requeridos para almacenar los valores de los streams durante la ejecución del monitor.

El punto de partida de esta etapa es el resultado del análisis de tipos llevado a cabo durante el análisis estático:

- El mapeo que asocia a cada stream el identificador correspondiente a su tipo (`typeMap :: Map Identifier TypeRepr`).
- La lista de todos los tipos de datos que participan en la definición de la especificación (`types :: [TypeRepr]`).

En lo que sigue se explica la implementación de la generación de tipos de datos a partir de estos valores.

Implementación de tipos de datos en C

A continuación se describe cómo se representa en C99 cada uno de los tipos de datos de la lista `types`. Concretamente, se da una traducción recursiva sobre el tipo **TypeRepr**, donde los tipos simples se representan con tipos C99 built-in equivalentes, mientras que para los tipos compuestos se definen structs cuyos campos varían según su estructura.

- **TBool**, **TInt** y **TDouble**

  Se traducen a C99 como `_Bool`, `int` y `double`, respectivamente.

- **TMaybe**

  Para representar a los tipos opcionales, se define un struct con dos campos: `nothing`, de tipo `int`, que indica si el valor es desconocido, y `just` para almacenar el valor, si existe. Por ejemplo, para el tipo **TMaybe TDouble**, MCLOLA genera la siguiente declaración de tipos:

  ```
  struct MaybeMdoubleM
  { double just;
    int nothing;
  };
  typedef struct MaybeMdoubleM MaybeMdoubleM;
  ```

- **TProd**

  Para representar al producto de tipos, se define un struct con dos campos, `c1` y `c2`, para almacenar la primera y segunda componente del par, respectivamente. Por ejemplo, para el tipo **TProd TInt TDouble**, MCLOLA genera la siguiente declaración de tipos:

```
struct PRintXdoubleRP
{ int c1;
  double c2;
};
typedef struct PRintXdoubleRP PRintXdoubleRP;
```

- **TEither**

  Para representar al coproducto de tipos, se define un struct con tres campos: `left` correspondiente al primer tipo, `right` correspondiente al segundo, y `opt`, de tipo `int`, que indica con cuál de las dos proyecciones se corresponde el valor actual. Por ejemplo, para **TEither TInt TDouble**, mcLola genera la siguiente declaración de tipos:

```
struct ETHintORdoubleHTE
{ int left;
  double right;
  int opt;
};
typedef struct ETHintORdoubleHTE ETHintORdoubleHTE;
```

- **TList**

  Las listas se implementan en C99 como listas acotadas, definiéndose globalmente un tamaño máximo para su longitud. Para representar este tipo de listas, se define un struct con tres campos: `data`, un arreglo con los elementos de la lista; `n`, que representa la cantidad de elementos que contiene la lista; y `size`, la máxima cantidad de elementos que la lista puede almacenar. Por ejemplo, para el tipo **TList TBool** mcLola genera la siguiente declaración de tipos:

```
struct LsboolsL
{ _Bool data[(30)];
  int n;
  int size;
};
typedef struct LsboolsL LsboolsL;
```

  donde 30 corresponde al tamaño máximo para la longitud de una lista.

- **TNAProd**

  Para representar a los tipos records, se define un struct con tantos campos como campos tenga el tipo original, respetando su nombre y tipo. Por ejemplo, para **TNAProd** `"Point2"` `[("x", TDouble), ("y", TDouble)]`, mcLola genera la siguiente declaración de tipos:

```
struct Point2
{ double x;
  double y;
};
typedef struct Point2 Point2;
```

Una de las reglas de Misra C establece que todos los campos de un struct deben inicializarse con algún valor. Por este motivo, se utilizan valores por defecto para completar campos irrelevantes, como el campo `just` de un valor de un tipo opcional con `nothing` = 1.

Para completar la implementación de los tipos de datos, se definen dos funciones para cada tipo `T`:

- `_Bool equalsT(T, T)` que compara por igualdad dos elementos de tipo `T`.
- `void printT(T)` para imprimir un valor de tipo `T`.

Mientras que la primera función tiene por objeto la implementación del predicado **Eq** incluido en la teoría de datos Theories.Bool, la segunda es utilizada por el engine temporal para comunicar al usuario los resultados a medida que los computa.

Por último, se define un tipo de datos para representar a los valores de entrada de la especificación. Como se describió anteriormente, en cada iteración del engine se reciben los valores de los streams de entrada correspondientes al instante actual. Para simplificar el manejo de estos valores, se decidió encapsularlos en un struct llamado `input`. Esta estructura tiene un campo por cada stream de entrada de la especificación, con el mismo nombre y tipo. Por ejemplo, para la especificación del Ejemplo 6.1, MCLOLA declarará la siguiente estructura `input`:

```
struct input
{ int a;
};
typedef struct input input;
```

Implementación de buffers

Para almacenar los valores de los streams durante la ejecución del monitor, MCLOLA incluye una implementación de buffers basada en arreglos circulares. En particular, el cálculo del tamaño efectuado durante el análisis estático asegura que el algoritmo de monitoreo nunca necesitará un valor que fue sobrescrito.

Concretamente, para cada tipo `T` que sea el tipo de un stream, se declara un tipo `bufferT` para almacenar valores de tipo `T`, que se implementa con un struct con dos campos: un arreglo para almacenar los valores de los streams y un entero que indica el tamaño del arreglo. Por ejemplo, para la especificación del Ejemplo 6.1 se genera la siguiente declaración de tipos:

```
struct bufferint
{ MaybeMintM * arr;
  int size;
};
typedef struct bufferint bufferint;
```

Nótese que al trabajar con referencias a futuro, algunos valores de los streams pueden no conocerse en un determinado instante. Por este motivo se utilizan arreglos de tipos opcionales.

La implementación de buffers se completa con la definición de las siguientes funciones:

- `bufferT initT(bufferT b, MaybeMTM * arr, int size)`

  Inicializa un buffer de tipo `T`. Concretamente, `initT` se ocupa de:

  - inicializar el campo `size`;
  - inicializar en `1` el campo `nothing` de cada posición del arreglo, para indicar que todos los valores son desconocidos;
  - completar con un valor por defecto el campo `just` de cada posición del arreglo para verificar Misra C.

- `bufferT putT(bufferT b, T val, int i)`

  Almacena un valor en un buffer para un instante de tiempo dado.

- `T getT(bufferT b, int i)`

  Recupera de un buffer el valor correspondiente a un instante dado.

- `_Bool askT(bufferT b, int i)`

  Determina si se conoce el valor correspondiente a un instante dado.

- `MaybeMTM offsetT(bufferT b, int k, MaybeMTM df, int i, int now, int end)`

  Accede al elemento de un buffer correspondiente a un determinado desplazamiento respecto de una posición dada, devolviendo un valor por defecto (pasado como argumento) en caso de que el desplazamiento en cuestión esté fuera de rango.

6.2.3. Generación del engine temporal

El manejo del tiempo es uno de los conceptos fundamentales del engine temporal. Para implementar en C el reloj del engine hemos declarado una variable entera global, de nombre `now`, que se inicializa en 0 al principio de la ejecución y se incrementa en 1 al concluir cada iteración del engine.

Asimismo, se declaran variables globales correspondientes al buffer de cada stream y su respectivo arreglo. Por ejemplo, para la especificación del Ejemplo 6.1, se generan las siguientes declaraciones:

```
bufferint a_buff;              MaybeMintM a_arr[(7)];
bufferint x_buff;              MaybeMintM x_arr[(7)];
bufferint y_buff;              MaybeMintM y_arr[(7)];
```

La función `initint` presentada más arriba se ocupa de asignar a cada buffer su arreglo correspondiente, i.e. `a_buff` apunta a `a_arr`, etc.

Definidas las estructuras de datos pertinentes, se definen las tres funciones que implementan el engine temporal:

- `void initialise(void)`, que inicializa las estructuras de datos necesarias para dar comienzo a la ejecución del engine.
- `void step(input * s)`, que implementa una iteración del engine.
- `void finish(void)`, que concluye los cálculos pendientes una vez finalizados los eventos de entrada.

Inicialización del engine

La función `initialise` se ocupa de inicializar el reloj y los buffers, como así también de marcar el inicio de los eventos de entrada. Esto último permite a las funciones `offsetT` determinar, en el caso de las referencias a futuro, si se debe devolver el valor por defecto por haber llegado al final de la traza.

El siguiente código muestra la definición de la función `initialise` generada por MCLOLA para la especificación del Ejemplo 6.1:

```
void initialise(void) {
  (now) = (0);
  (inputEnd) = (0);
  (a_buff) = ((initint)((a_buff), (a_arr), (7)));
  (x_buff) = ((initint)((x_buff), (x_arr), (7)));
  (y_buff) = ((initint)((y_buff), (y_arr), (7)));
  return;
}
```

Iteraciones del engine

Para implementar la función `step` se procesan los streams en el orden computado por el análisis estático. Para ello, se encapsulan las instrucciones correspondientes al procesamiento de un stream s en una función llamada `eval_s`, cuya signatura de tipo y definición dependerá según se trate de un stream de entrada o salida.

Si s es un stream de entrada de tipo `T`, se define una función `void eval_s(T)` que recibe como único argumento el valor de s en el instante actual. La función `eval_s` simplemente almacena dicho valor en el buffer correspondiente a s y comunica al usuario el resultado. En particular, para reportar un resultado se imprime una tupla (i, s, val) que indica que el valor del stream s en el instante i es val.

Por ejemplo, para el stream a de la especificación del Ejemplo 6.1 se define la siguiente función:

```
void eval_a(int val) {
  (a_buff) = ((putint)((a_buff), (val), (now)));
  (printf)(("(%d, %s, "), (now), ("a"));
  (printint)((val));
```

```
  (printf)((")\n"));
}
```

Si s es un stream de salida, se define una función `void eval_s(void)`. Notar que, a diferencia de los streams de entrada, no es suficiente con (intentar) computar el valor de s en el instante actual, puesto que debido a la presencia de referencias temporales futuras, algunos valores anteriores pueden no haber sido computados aún. Concretamente, se debe verificar si los últimos $\Delta s + 1$ valores han sido resueltos o no. En el primer caso, simplemente se omite el cómputo; mientras que en el segundo se ejecutan las instrucciones correspondientes a la expresión que define al stream, cuya implementación se discutirá más adelante. Si el cálculo pudo completarse, se guarda el valor en el buffer correspondiente y se comunica el resultado.

El siguiente código muestra un fragmento de la implementación del procesamiento del stream x de la especificación del Ejemplo 6.1:

```
void eval_x(void) {
  // declaración de variables auxiliares
  ...
  int i = ((now) - (3));
  for (i; (i) <= (now); ++(i)) {
    if (((i) < (0)) || ((askint)((x_buff), (i)))) {
      continue;
    } else {
      // cómputo del valor
      ...
    };
  };
}
```

De esta forma, para la especificación del Ejemplo 6.1 se define la función `step` de la siguiente manera:

```
void step(input * s) {
  (printf)(("Instant %d:\n"), (now));
  (eval_a)(((s)→a));
  (eval_x)();
  (eval_y)();
  (moveClock)();
  return;
}
```

donde el llamado a `printf` tiene por fin informar al usuario en qué iteración del engine se encuentra la ejecución; mientras que la función `moveClock` incrementa en 1 el valor de `now` y sobrescribe el contenido del índice correspondiente al instante `now` de cada buffer para indicar que se desconocen los nuevos valores.

Finalización del engine

En el momento que terminan de recibirse los valores de entrada, puede haber valores correspondientes a streams de salida con latencia positiva que

aún no han sido calculados. Para finalizar los cómputos pendientes, se define la función `finish` que marca el fin de los eventos de entrada y agrega $max\{\Delta s\}$ iteraciones para procesar los streams de latencia positiva.

Para la especificación del Ejemplo 6.1 se genera la siguiente definición:

```
void finish(void) {
  int j = (0);
  (inputEnd) = (1);
  (undoClock)();
  (printf)(("Finish:\n"));
  for (j; (j) < (4); ++(j)) {
    (eval_x)();
    (eval_y)();
  };
  return;
}
```

donde se invoca a `undoClock` para deshacer la actualización del reloj y la sobreescritura del buffer correspondiente a la última ejecución de `step`.

Traducción de expresiones

Para completar la implementación del monitor resta traducir las expresiones MCLOLA que definen a los streams de salida de la especificación. Concretamente, se requiere definir una función de interpretación que asocie a cada expresión MCLOLA código C99 que implemente su cómputo.

En primer lugar, definimos un tipo de datos **C99Exp** para representar código C99 correspondiente a una expresión MCLOLA. Observamos que una expresión MCLOLA se puede representar con una expresión de C99, para la cual puede ser necesario declarar variables y ejecutar instrucciones. Por ejemplo, para representar a la expresión **ValInt** 5 podemos usar, sencillamente, la expresión `5` de C99; mientras que para la expresión **Pair** (**ValInt** 5) (**ValInt** 4) podemos usar la expresión `p` de C99 si se incluye previamente el siguiente código:

```
PRintXintRP p;
p.c1 = 5;
p.c2 = 4;
```

Siguiendo esta representación, podemos definir el tipo **C99Exp** de la siguiente manera:

```
type C99Exp = (Expr, [Decln], [Stmt])
```

donde los tipos **Expr**, **Decln** y **Stmt** son los tipos definidos por la librería `Language.C99.Simple` para representar expresiones, declaraciones e instrucciones C99, respectivamente.

Sin embargo, para representar algunas expresiones se necesita información relativa a los tipos. Por ejemplo, para **Eq** (**ValInt** 4) (**ValInt** 5) se requiere realizar un llamado a la función `equalsint`. En general, para traducir el operador **Eq** se necesita conocer el tipo de las expresiones que conforman sus argumentos

para invocar a la función de comparación adecuada. Para lograrlo, es suficiente añadir a **C99Exp** el tipo de la expresión:

```
type C99Exp = (Expr, [Decln], [Stmt], TypeRepr)
```

Por otro lado, recordamos que para traducir expresiones mcLola a un elemento de tipo **C99Exp** es necesario declarar algunas variables, como en el caso del constructor **Pair**; requiriéndose de un mecanismo para la generación automática de nombres que garantice la unicidad de los identificadores.

Para generar nombres únicos en la traducción de la expresión que define a un stream s, hemos optado por utilizar un prefijo, generalmente el nombre del stream s, seguido de un entero único, comenzando por 0 y eligiendo el sucesor del último usado cada vez que se requiere un nuevo nombre. Para implementarlo, definimos un tipo **C99** para representar la traducción de expresiones a código C99 utilizando una mónada de estado, cuyo estado está conformado por el nombre del stream que se está traduciendo y un entero que se incrementará en 1 conforme se necesiten nuevas variables:

```
type C99St = (Identifier, Int)
type C99   = State C99St C99Exp
```

No obstante, para traducir algunos constructores se necesita más información. Puntualmente, para el caso del operador **Offset** se debe invocar a la función `offsetT`, para el tipo `T` adecuado; por lo cual se requiere conocer el tipo de cada stream. Por ello, se agrega al estado el mapeo calculado durante el análisis estático que asocia a cada stream el identificador de su tipo:

```
type C99St = (TypeMap, Identifier, Int)
type C99   = State C99St C99Exp
```

Luego, a partir del tipo **C99** definimos un álgebra de alto orden para traducir expresiones mcLola de la siguiente manera:

```
data FC99 e where
  FC99 :: C99 → FC99 e

unC99 (FC99 x) = x

class C99Alg e where
  c99Alg :: Alg e FC99
```

Al definir las instancias de **C99Alg** para cada constructor se debe tener en cuenta, nuevamente, que debido a las referencias a futuro, el engine debe trabajar con valores opcionales para expresar que puede haber valores desconocidos. Veamos, por ejemplo, cómo definir la instancia de **C99Alg** correspondiente al operador **ValBool**. Para empezar, definimos una función que dado un valor booleano, construye un valor de tipo **C99** para representar dicho valor:

```
c99ValBool :: Bool → C99
c99ValBool b = return (LitBool b, [], [], TBool)
```

donde **LitBool** es el constructor de Language.C99.Simple para definir constantes booleanas. En este caso, no es necesario incluir declaraciones de variables ni ejecutar instrucciones.

Si bien siempre es posible computar el valor de una constante, para poder combinarla con otras expresiones que pueden tener valores desconocidos se requiere transformar la representación anterior en una que utilice un tipo opcional. En otras palabras, en lugar de definir la traducción directamente a partir la función c99ValBool:

```
instance C99Alg ValBool where
  c99Alg (ValBool b) = FC99 $ c99ValBool b
```

la componemos con la función toMaybeVal que encapsula un valor C99 conocido en un valor opcional:

```
instance C99Alg ValBool where
  c99Alg (ValBool b) = FC99 $ (toMaybeVal . c99ValBool) b
```

Análogamente, para implementar las instancias de operadores como **Or** o **Add**, se debe verificar si las expresiones sobre las que se aplican tienen valores conocidos. En principio, sólo podremos calcular el resultado de una operación cuando todos sus argumentos sean conocidos.

La definición completa de las instancias de **C99Alg** puede consultarse en el directorio `Theories` del código fuente, a excepción de los operadores temporales que se encuentran en `Language/Lola.hs`.

Vale la pena observar que, si bien en el ejemplo visto para el operador **ValBool** se utilizó un AST de C99 para implementar la traducción, también se puede usar, directamente, una función C99. La librería Language.C99.Simple provee un constructor **Funcall** para invocar funciones, y en MCLOLA se incluye un archivo `lib.c` para dar definiciones de funciones en C99. Si bien la gran mayoría de los operadores de MCLOLA se implementaron usando únicamente los constructores de la librería Language.C99.Simple, puede consultarse como ejemplos de esta segunda alternativa la implementación de los operadores **IntToDouble** y **DoubleToInt** de Theories.Numeric que invocan a las funciones `toDouble` y `toInt` definidas en `lib.c`.

Definidas las instancias para cada constructor del lenguaje de expresiones MCLOLA, se define la traducción a C99 de la siguiente manera:

```
trans :: (HFunctor e, C99Alg e) ⇒ Term e a → C99
trans t = unC99 (cata c99Alg t)
```

Por lo tanto, para obtener las declaraciones e instrucciones C99 correspondientes a la expresión e que define a un stream de salida s, necesarias para completar la definición de `eval_s` vista anteriormente, simplemente se evalúa la mónada `trans e` con el estado dado por `(typeMap, ident s, 0)`, donde typeMap es el mapeo que asocia a cada stream su tipo y `ident s` es el nombre del stream s.

Capítulo 7

Extendiendo MCLOLA con Anticipación

En el contexto de monitores que describen propiedades booleanas para sistemas con ejecuciones infinitas, el principio de *anticipación* plantea que una vez que toda continuación (infinita) de una traza finita evalúa al mismo veredicto, puede asegurarse que dicha traza finita evalúa a ese veredicto [26]. El objetivo de la anticipación es evaluar resultados tan pronto como sea posible, permitiendo, por ejemplo, detectar tempranamente violaciones del sistema para actuar en consecuencia.

Siguiendo las ideas introducidas por el concepto de anticipación y teniendo en cuenta que el uso de referencias futuras puede retrasar considerablemente la resolución de un stream, hemos estudiado cómo incorporar mecanismos a MCLOLA que permitan computar resultados tan pronto como sea posible.

Para anticipar resultados pueden adoptarse estrategias más o menos agresivas. Por ejemplo, si se considera la siguiente especificación:

```
input  bool a
output bool x = x[-1|false] ∨ a
```

y valores de entrada dados por $a = \langle false, false, true, ... \rangle$, observamos que en el instante 2 es posible anticipar el valor de x en todos los siguientes instantes de tiempo, ya que si se conoce que el valor de p es $true$ entonces el valor de $p \vee q$ también lo será, independientemente del valor de q. Esta estrategia de anticipación, que intenta computar en cada momento todos los valores desconocidos de los streams, constituye un extremo marcadamente *eager*.

Otra estrategia menos exhaustiva consiste en intentar computar todos los valores no conocidos de los streams *hasta* el instante actual. Si bien este enfoque no supone ningún cambio en el caso de streams sin referencias futuras, como el ejemplo anterior, puede servir para anticipar el resultado de un stream con latencia positiva. Por ejemplo considérese la siguiente especificación con valores de entrada dados por $a = \langle false, false, true, ... \rangle$:

```
input  bool a
output bool x = a[2|false] ∨ a
```

En este caso, en el instante 2, además de computarse $x(0)$, puede resolverse $a(2)$, que en la versión sin anticipación se calcularía en el instante 4.

En este capítulo se describen dos mecanismos que hemos incorporado a MCLOLA para anticipar resultados, siguiendo el enfoque de intentar computar los valores no resueltos *hasta* el instante actual.

7.1. Simplificadores

En algunos casos ciertas funciones pueden ser evaluadas con información parcial de sus argumentos, como ocurre en los ejemplos anteriores con la disyunción booleana. Esta propiedad puede utilizarse para computar resultados tan pronto como la información concluyente esté disponible. Para ello, se requiere utilizar una implementación alternativa del operador en cuestión que contemple cómo computar un resultado con información parcial de sus argumentos, lo que denominamos *simplificador*; en contraposición con la implementación *tradicional* o *estricta* que requiere conocer el valor de todos sus argumentos.

Los simplificadores pueden afectar drásticamente el desempeño de los monitores en términos del instante en el cual los valores de un stream son calculados. Por ejemplo, considérese la siguiente especificación que analiza el comportamiento de un sistema hasta que un determinado evento se registre tres veces:

```
input  bool s
input  bool a
input  bool b
output int  count = count[−1|0] + if ok then 1 else 0
output bool end   = count ≥ 3
output bool ok    = if   s[−2|false] ∨ s[2|false]
                  then a ∧ a[1|true]
                  else b ∨ b[7|true]
```

Para resolver el stream *ok* (de latencia 7) pueden utilizarse simplificadores para las operaciones $\vee$, $\wedge$ e $if \cdot then \cdot else$. En los primeros dos casos se simplifica cuando uno de los dos valores sea $true/false$; mientras que para el operador $if \cdot then \cdot else$ puede usarse el simplificador especificado en el siguiente pseudocódigo, donde el predicado known determina si se conoce el valor de un elemento y value indica su valor:

```
ite b x y =
  if   known b
  then if   value b
       then x                                          -- caso 1
       else y                                          -- caso 2
  else if   known x ∧ known y ∧ value x ≡ value y
       then x                                          -- caso 3
       else unknown                                    -- caso 4
```

Si se usan estos tres simplificadores para implementar el monitor correspondiente a la especificación anterior y se ejecuta con una traza dada por:

$$\begin{aligned} s &= \langle false, true, true, true, false, ... \rangle \\ a &= \langle true, true, false, true, true, ... \rangle \\ b &= \langle true, false, true, false, true, ... \rangle \end{aligned}$$

se obtienen los siguientes resultados:

- $ok(0) = true$ es calculado en el instante 1, usando el caso 3 del simplificador de $if \cdot then \cdot else$ y el simplificador de $\vee$.
  Por lo tanto, $count(0) = 1$ es calculado en el instante 1.

- $ok(1) = false$ es calculado en el instante 3, usando el caso 1 del simplificador de $if \cdot then \cdot else$.
  Por lo tanto, $count(1) = 1$ es calculado en el instante 3.

- $ok(2) = true$ es calculado en el instante 4, usando el caso 2 del simplificador de $if \cdot then \cdot else$ y el simplificador de $\vee$.
  Por lo tanto, $count(2) = 2$ es calculado en el instante 4.

- $ok(3) = true$ es calculado en el instante 4, usando el caso 1 del simplificador de $if \cdot then \cdot else$ y el simplificador de $\vee$.
  Por lo tanto, $count(3) = 3$ es calculado en el instante 4.

De esta manera, en el instante 4 podemos garantizar que se han registrado tres ocurrencias de *ok*, pudiendo finalizar la ejecución del sistema; cuando en la versión sin anticipación se hubiese llegado al mismo veredicto en el instante 10.

Sin embargo, si se los compara con las versiones estrictas, en general, los simplificadores presentan la desventaja de requerir más líneas de código para su implementación y ser más costosos computacionalmente. Por este motivo, es deseable utilizarlos únicamente cuando esto suponga un beneficio en términos de la anticipación de resultados, lo que potencialmente ocurre cuando un stream tiene latencia positiva. Siguiendo esta idea, hemos decidido implementar en MCLOLA dos traducciones a C99:

- La traducción *estricta* o *por defecto*, presentada en el capítulo anterior, que utilizamos para implementar el código correspondiente a los streams con latencia 0.

- La traducción *simplificada*, definida a partir de simplificadores, que utilizamos para implementar el código correspondiente a los streams con latencia positiva.

Nótese que para implementar la traducción simplificada se requiere extender el lenguaje con una nueva función de interpretación análoga a la función `trans` presentada en el capítulo anterior. De hecho, para explicitar que se trata de la traducción por defecto, hemos renombrado esta función como `transDef` y al álgebra que usamos para definirla como **C99DefAlg**; añadiendo la función `transSim` para implementar la traducción simplificada a partir del álgebra **C99SimAlg**:

```
class C99SimAlg e where
  c99SimAlg :: Alg e FC99
```

En particular, la instancia de **C99SimAlg** coincidirá con la instancia de **C99DefAlg** para aquellos operadores que requieren estrictamente conocer todos sus argumentos para computar el resultado, como la suma de enteros, el constructor de pares o la negación booleana. De esta manera, sólo se deben escribir las instancias de **C99SimAlg** para los constructores que admiten simplificadores, como la disyunción booleana o la multiplicación de enteros.

No obstante, el código C99 que implementa un simplificador suele ser considerablemente más complejo que el de las versiones estrictas. Con el objetivo de facilitar su implementación, hemos introducido un mecanismo para derivar simplificadores al nivel del AST del lenguaje. Partimos de la observación de que algunos simplificadores pueden expresarse a partir de otros, por ejemplo:

```
or x y   = ite x x y
and x y  = ite x y x
impl x y = ite x y true
```

De esta forma, dar una implementación en C99 para el simplificador del operador $if \cdot then \cdot else$ nos permite obtener simplificadores para la disyunción, conjunción e implicancia sin tener que escribir su correspondiente AST del código C99 que los implementa:

```
instance C99SimAlg BoolTheory where
  c99SimAlg (Ite b x y) = ...
  c99SimAlg (Or x y)    = c99SimAlg (Ite x x y)
  c99SimAlg (And x y)   = c99SimAlg (Ite x y x)
  c99SimAlg (Impl x y)  = c99SimAlg (Ite x y true)
    where true = c99SimAlg (ValBool True)
  c99SimAlg x           = c99DefAlg x
```

Observamos, además, que puede haber diferentes formas de computar un resultado. Por ejemplo, para la disyunción booleana, que es una operación conmutativa, tenemos:

```
or x y = ite x x y
or x y = ite y y x
```

En particular, dependiendo de los valores puntuales de las expresiones, en algunos casos se puede llegar a un veredicto más rápidamente con una computación, y en otros casos con la otra. Por ejemplo, para computar $x \vee x[1|false]$

conviene utilizar la primera opción, mientras que para $x[1|false] \vee x$ es preferible la segunda. De esta manera, considerar ambas computaciones nos provee de otro mecanismo para anticipar resultados. A partir de esta observación, hemos decidido extender el lenguaje con operadores que permiten expresar una computación de distintas maneras:

```
data Choice e a where
  Choice :: e a → e a → Choice e a

data IteK e a where
  IteK :: e a → e b → e b → IteK e b
```

Choice e1 e2 devuelve la expresión e1 si su valor es conocido y e2 en caso contrario; mientras que **IteK** b e1 e2 devuelve e1 si b es conocido y e2 sino. Si bien **Choice** es un caso especial de **IteK**, se decidió incluirlo separadamente por cuestiones de eficiencia y claridad en el código generado.

De esta manera, el operador **Choice** ofrece un mecanismo para expresar una computación de distintas maneras y, en particular, de aprovechar la conmutatividad de algunos constructores para anticipar un resultado. Por ejemplo, podemos definir:

```
comm f e1 e2 = Choice (c99SimAlg (f e1 e2))
                      (c99SimAlg (f e2 e1))

instance C99SimAlg BoolTheory where
  ...
  c99SimAlg (Or x y) = c99SimAlg (comm orOp x y)
    where
      orOp e1 e2 = Ite e1 e1 e2

  c99SimAlg (And x y) = c99SimAlg (comm andOp x y)
    where
      andOp e1 e2 = Ite e1 e2 e1
  ...
```

Nótese que a diferencia de los operadores temporales y de datos de McLola, los constructores **Choice** e **IteK** no se usan para describir especificaciones, sino que son constructores internos cuyo único propósito es derivar simplificadores, motivo por el cual no se derivan sus constructores inteligentes.

7.2. Reescritura de expresiones

El mecanismo de anticipación provisto por los simplificadores está basado en información específica de los datos, en particular, depende estrictamente de los valores puntuales de las expresiones. Por ejemplo, el valor *true* permite simplificar una disyunción lógica, mientras que $false$ no. En otros casos, en cambio, es posible anticipar el valor de una expresión a partir de su estructura, independientemente de los valores concretos de sus subexpresiones. Por ejemplo, en la siguiente especificación:

```
input  int a
output int x = fst (a[-1|5], a[3|5])
```

notamos que no se necesita conocer el valor de $a(i+3)$ para computar $x(i)$, ya que el valor de la segunda componente del par resulta irrelevante. En particular, si reescribimos la especificación de la siguiente manera:

```
input  int a
output int x = a[-1|5]
```

las mismas resultan equivalentes en cuanto a sus resultados, pero con la diferencia de que la segunda permite computar los valores correspondientes al stream x tres instantes antes que la primera.

Siguiendo esta observación, hemos implementado una reescritura de expresiones MCLOLA para eliminar cómputos innecesarios, basándonos en las siguientes reglas de reescritura sobre constructores y observadores de productos:

```
fst (pair a b) = a
snd (pair a b) = b
```

Estas reglas también aplican a tipos records. Por ejemplo, si tenemos un constructor de puntos bidimensionales `point` y observadores `x` e `y` para acceder a las componentes, entonces:

```
x (point a b)  = a
y (point a b)  = b
```

Para implementar esta característica, hemos definido una función de interpretación llamada `rewrite` que aplica estas reglas sobre una expresión MCLOLA. Para ello, hemos definido una clase **Rewrite** para implementar un álgebra de alto orden, dando instancias concretas de la misma únicamente para las expresiones que matchean con el lado izquierdo de las reglas presentadas anteriormente, usando como definición por defecto para el resto de las expresiones a la transformación identidad.

Por otro lado, para contemplar el caso de que una regla de reescritura introduzca una expresión donde puede aplicarse otra, se implementa la reescritura como el punto fijo de la función `rewrite`. Nótese que para poder definir el punto fijo de una función sobre expresiones MCLOLA se requiere definir una relación de igualdad sobre las expresiones del lenguaje. Para ello se utiliza la clase **EqHF**, provista por *compdata*, que implementa la noción de igualdad para funtores de alto orden. En general, sus instancias pueden ser derivadas automáticamente usando la función makeEqHF ofrecida por la librería[1].

Finalmente, se define la función de traducción simplificada de la siguiente manera:

[1]Las instancias de **EqHF** fueron derivadas automáticamente con makeEqHF en todos los casos, con la excepción de los operadores temporales, cuya instancia se define en el módulo `Language.Lola`.

```
transSim :: (HFunctor e, C99SimAlg e, Rewrite e e, EqHF e) ⇒
    Term e a → C99
transSim t = unC99 (cata c99SimAlg (fix rewrite t))
```

Observamos que si bien mcLola sólo incluye reglas de reescritura sobre los constructores y observadores de los tipos pares y records, pueden agregarse instancias para otras teorías que el usuario escriba. Es responsabilidad del programador garantizar que dichas reglas estén bien definidas, i.e. que no introduzcan ciclos de reescritura.

Nótese que esta técnica, además de proveer un mecanismo para anticipar resultados, puede emplearse para optimizar un monitor, eliminando cómputos innecesarios. Por este motivo hemos decidido incluirla también en la traducción estricta:

```
transDef :: (HFunctor e, C99DefAlg e, Rewrite e e, EqHF e) ⇒
    Term e a → C99
transDef t = unC99 (cata c99DefAlg (fix rewrite t))
```

Sin embargo, una desventaja de este método es que no permite anticipar resultados cuando la estructura a simplificar se encuentra definida en distintos streams, como ocurre en la siguiente especificación:

```
input  int          a
output int          x = fst y
output (int, int)   y = (a[-1|5], a[3|5])
```

Una alternativa posible para solucionar este problema consiste en utilizar tipos enriquecidos que permitan representar información parcial dentro de un mismo valor. Este punto constituye unas de las líneas de trabajo a futuro.

La implementación del mecanismo de reescritura puede consultarse en el archivo `Engine/Rewrite.hs` del código fuente, mientras que en el directorio `Theories` se encuentran las instancias de **Rewrite** definidas para cada teoría de datos.

Capítulo 8

Caso de estudio: Monitorización de UAVs

En este capítulo usaremos MCLOLA para escribir una especificación relativa al vuelo de un UAV. En particular, mediante este ejemplo, veremos cómo extender el lenguaje con nuevas teorías de datos.

8.1. Descripción de la especificación

En lo que sigue trabajaremos con una especificación proveniente del área de UAVs. La misma está basada en el ejemplo "UAV Specification 1" presentado en la página de HLOLA [71], donde puede encontrarse su implementación en HLOLA junto con una instancia de valores de entrada con su correspondiente salida.

La especificación con la que trabajaremos difiere de la original en la representación de los eventos de la cámara y en el uso de un único polígono en lugar de una lista de polígonos para describir ciertas zonas prohibidas, dado que la instancia de entrada del ejemplo sólo utiliza uno.

El Código 8.1 muestra la especificación en cuestión, escrita en lenguaje LOLA. La misma tiene por objetivo:

- Evaluar dos propiedades sobre el vuelo de un UAV:
  - Que el UAV no vuela sobre regiones prohibidas (flying_in_safe_zones).
  - Que el UAV está en una posición correcta cuando toma una foto (all_ok_capturing).
- Computar la distancia mínima para salir de la zona prohibida, en caso de que el UAV esté volando en una región prohibida (depth_into_poly).

Por su parte, los streams de entrada describen:

- El estado del UAV en cada instante de tiempo (attitude, velocity, position, altitude y target).
- El polígono que representa la zona prohibida para volar (nofly).
- Los eventos generados por la cámara (events_within).

Los tipos **Attitude**, **Point2**, **Position** y **Target** son tipos records que pueden implementarse en Haskell como muestra el Código 8.2. Además, se representa a los polígonos con listas de puntos bidimensionales correspondientes a sus vértices, y a los eventos de la cámara con códigos numéricos según se muestra en la Figura 8.3.

```
-- Streams de entrada

input Attitude attitude
input Point2   velocity
input Position position
input double   altitude
input Target   target
input [Point2] nofly
input [int]    events_within

-- Streams de salida

--- Streams auxiliares

output double filtered_pos_xp =
  if   instantN < 3
  then nowpos
  else (nowpos + 0.6 * (2 * prev - prev'/2)) / 1.9
    where nowpos = xp position
          prev   = filtered_pos_xp[-1|0]
          prev'  = filtered_pos_xp[-2|0]

output double filtered_pos_yp =
  if   instantN < 3
  then nowpos
  else (nowpos + 0.6 * (2 * prev - prev'/2)) / 1.9
    where nowpos = yp position
          prev   = filtered_pos_yp[-1|0]
          prev'  = filtered_pos_yp[-2|0]

output double filtered_pos_alt =
  if   instantN < 3
  then nowpos
  else (nowpos + 0.6 * (2 * prev - prev'/2)) / 1.9
    where nowpos = alt position
          prev   = filtered_pos_alt[-1|0]
          prev'  = filtered_pos_alt[-2|0]
```

```
output int instantN = instantN[-1|0] + 1

-- Stream de salida: all_ok_capturing

output bool roll_ok = (abs (roll attitude)) < 0.0523

output bool pitch_ok = (abs (pitch attitude)) < 0.0523

output bool height_ok = filtered_pos_alt > 0

output bool near = (distance filterpos targetpos) < 1
    where filterpos = point2 filtered_pos_xp filtered_pos_yp
          targetpos = point2 (xt target) (yt target)

output bool open_capture =
  if   isEmpty relevantEvents
  then open_capture[-1|false]
  else last relevantEvents == 45
    where is_rel cod      = isElem cod [45, 50, 51]
          relevantEvents = filter is_rel events_within

output bool capturing =
  (isElem 45 events_within) || open_capture[-1|false]

output bool all_ok_capturing =
  capturing -> (height_ok && near && roll_ok && pitch_ok)

-- Stream de salida: flying_in_safe_zones

output [Point2] no_fly = no_fly[-1 | nofly]

output bool flying_in_safe_zones =
  ! (pointInPoly filPos no_fly)
    where filPos = point2 filtered_pos_xp filtered_pos_yp

-- Stream de salida: depth_into_poly

output (maybe double) depth_into_poly =
  if   pointInPoly filPos no_fly
  then just shortestDist
  else nothing
    where filPos      = point2 filtered_pos_xp filtered_pos_yp
          polySides   = polygonSides no_fly
          shortestDist = listMin (map (\ seg . distancePointSeg
               filPos seg) polySides)
```

Código 8.1: Especificación UAV

```
data Attitude = Attitude {yaw :: Double, roll :: Double, pitch
    :: Double} deriving (Show, Eq)

data Point2 = P {x :: Double, y :: Double} deriving (Show, Eq)

data Position = Position {xp :: Double, yp :: Double, alt ::
    Double, zone :: Double} deriving (Show, Eq)

data Target = Target {xt :: Double, yt :: Double, mutex ::
    Double, num_wp :: Double} deriving (Show, Eq)
```

Código 8.2: Tipos de datos para representar el estado de un UAV.

código	significado
1	initial_config
5	takeoff
6	takeoff_ended
9	land
10	arrived
20	sort_locations
21	has_next
22	go_next
23	yes_next
24	no_next
25	remove_next
45	capture
50	yes_person
51	no_person
...	...

Figura 8.3: Representación de los eventos.

Para implementar en MCLOLA la especificación correspondiente al Código 8.1 necesitaremos de una teoría de datos para cálculos geométricos que incluya:

- Un tipo de datos para representar puntos bidimensionales (**Point2**), junto con constructores (point2) y observadores (x e y).
- Un operador distance para computar la distancia entre dos puntos.
- Un predicado pointInPoly que determine si un punto pertenece a un polígono.
- Un operador polygonSides que compute los lados de un polígono.

- Un operador distancePointSegment que compute la distancia de un punto a un segmento.

Además, necesitaremos representar los tipos **Attitude**, **Position** y **Target** que describen el estado del vuelo de un UAV. En lo que sigue se comenta brevemente cómo lograr estos objetivos mediante la incorporación de nuevas teorías de datos.

8.2. Agregando teorías de datos

Para escribir en mcLola la especificación del UAV presentada en la sección anterior hemos extendido el lenguaje con dos nuevas teorías de datos:

- Theories.Geometry2D, que implementa algunas operaciones geométricas en dos dimensiones.
- Theories.UAV, que implementa tipos de datos para representar el estado del vuelo de un UAV.

Una teoría de datos está formada por un conjunto finito de tipos de datos y un conjunto finito de símbolos de función sobre ellos, que representan los operadores de la teoría (Sección 3.1). Una vez identificados los elementos que conforman la teoría, para comenzar con su implementación creamos un nuevo archivo en el directorio `Theories`, en este caso `Geometry2D.hs` y `UAV.hs`.

En lo que sigue se explicará (parcialmente) la implementación de la teoría Theories.Geometry2D. Para consultar la implementación completa de esta u otras teorías, referirse al directorio `Theories` del código fuente del proyecto.

Agregando tipos de datos

Para trabajar con cálculos geométricos en dos dimensiones, necesitamos un tipo de datos para representar puntos bidimensionales. Para ello, agregamos la siguiente definición de tipo en el archivo `Geometry2D.hs`:

```
data Point2 = P {x :: Double, y :: Double} deriving (Show, Eq)
```

Para implementar la traducción de **Point2** a un tipo C99 equivalente procedemos de la siguiente manera:

1. Damos una instancia de la clase **C99able** (presentada en la Sección 5.2.1) para el tipo **Point2**. En este caso, usamos el constructor **TNAProd** que representa a los tipos records:

   ```
   instance C99able Point2 where
     typeRepr = TNAProd "Point2" [("x", TDouble), ("y", TDouble)]
   ```

2. Extendemos la definición de la función streamType (discutida en la Sección 5.2.2) que determina la representación del tipo de un stream, añadiendo el nuevo caso.

Agregando operadores

En primer lugar, debemos proveer mecanismos para construir y observar puntos. Siguiendo los lineamientos de Datatypes à la Carte, definimos un nuevo GADT para implementar un operador **Point2D** para construir puntos y operadores **X2D** y **Y2D** para acceder a las componentes:

```
data Point2D (e :: * → *) (a :: *) where
  Point2D :: e Double → e Double → Point2D e Point2
  X2D     :: e Point2 → Point2D e Double
  Y2D     :: e Point2 → Point2D e Double
```

Completamos la implementación dando las instancias del mismo para las clases **TempRefAlg**, **TypesAlg**, **C99DefAlg** y **Rewrite** (nótese que al tratarse de tipos records tiene sentido dar una instancia de **Rewrite**).

Finalmente, extendemos el coproducto que define a las teorías de datos del lenguaje:

```
type Data =      BoolTheory :+: NumericTheory
           :+: ProdTheory :+: MaybeTheory :+: EitherTheory
           :+: ListTheory :+: Point2D
```

Con esto ya podemos construir puntos y acceder a sus componentes. Sin embargo, para que la teoría resulte útil se requiere incluir otros operadores que implementen cómputos más complejos, como por ejemplo el producto escalar o la distancia entre dos puntos. En principio podemos repetir el proceso, extendiendo el AST con nuevos constructores como **ScalarMult** y **Distance**, tal como hicimos con **Point2D**, **X2D** y **Y2D**. No obstante, observamos que en general las funciones que queremos agregar a la teoría simplemente operan sobre las componentes de los puntos que son, sencillamente, valores numéricos. Es decir que las operaciones que queremos agregar al lenguaje son operaciones derivadas de los constructores y observadores de puntos y de los operadores numéricos.

De esta manera, para definir un operador que compute la distancia entre dos puntos no necesitamos agregar un nuevo constructor al AST, sino que podemos implementarlo en términos de otros constructores ya definidos:

```
type DistanceConstr e = (DBasicOp :<: e, Point2D :<: e)

iDistance :: DistanceConstr e ⇒ Term e Point2 → Term e Point2
     → Term e Double
iDistance a b = iSqrt (iDAdd difx dify)
  where
    difx = iPow (iDSub (iX2D a) (iX2D b)) (iValDouble 2)
    dify = iPow (iDSub (iY2D a) (iY2D b)) (iValDouble 2)
```

donde:

- Recordamos que `Op :<: e` significa que el operador `Op` es parte de la signatura `e`.

- El constructor **DBasicOp**, definido en Theories.Numeric, agrupa a los operadores básicos de los números de punto flotante, incluyendo constantes, suma, resta, potenciación y raíz cuadrada (**ValDouble**, **DAdd**, **DSub**, **Pow**, **Sqrt**).

Incluso podemos escribir operadores derivados a partir de otros operadores derivados, como es el caso del operador iNorm que calcula la distancia de un punto al origen:

```
type NormConstr e = (DistanceConstr e, Point2D :<: e)

iNorm :: NormConstr e ⇒ Term e Point2 → Term e Double
iNorm p = iDistance p (iPoint2D (iValDouble 0) (iValDouble 0))
```

En general, es una buena práctica mantener el AST de un lenguaje lo más acotado posible, extendiéndolo únicamente cuando no resulte lo suficientemente expresivo para añadir una determinada funcionalidad o cuando se pueda optimizar alguna implementación [72].

Algunos de los operadores geométricos que necesitamos para implementar la especificación del UAV involucran cálculos un poco más complejos que los vistos para iDistance e iNorm. Tal es el caso del operador iDistancePointSeg que computa la distancia de un punto a un segmento (que representamos como un par de puntos). El siguiente pseudocódigo indica cómo computarlo según se explica en [80].

```
1  distancePointSegment p seg =
2    if   0 ≤ rez ∧ rez ≤ 1
3    then norm (scalarMul bminusa imz)
4    else min (distance p a) (distance p b)
5      where
6        a       = fst seg
7        b       = snd seg
8        bminusa = minus b a
9        z       = complexdiv (minus p a) bminusa
10       rez     = x2d z
11       imz     = y2d z
```

Definidos los operadores iNorm, iScalarMul, iDistance, iMinus e iComplexDiv que implementan, respectivamente, la distancia al origen de un punto, el producto escalar, la distancia y la resta entre dos puntos y la división de números complejos (interpretados como puntos), podemos definir a iDistancePointSegment como un operador derivado. Sin embargo, el mecanismo de traducción de expresiones a código C que hemos implementado traduce expresiones composicionalmente, sin compartir subexpresiones comunes. Por lo tanto, en este caso, estaríamos repitiendo algunos cómputos. Por ejemplo, el cálculo correspondiente a la expresión minus b a se repetiría hasta cuatro veces: dos veces para resolver rez (línea 2), una vez en bminusa y una vez en imz (línea 3).

Para evitar esta repetición de cómputos sin modificar el mecanismo de traducción de expresiones a código C, hemos decidido extender el lenguaje con

operadores internos para introducir *let bindings* o *ligadura de variables*, cuya implementación puede consultarse en el módulo `Theories.LetBind`.

```
data LetBind (e :: * → *) (a :: *) where
  Let :: String → e a → e b → LetBind e b
  Ref :: String → LetBind e a
```

Observar que el primer argumento (de tipo **String**) de ambos constructores corresponde al nombre de la variable. Por ejemplo, a la expresión `let a = 5 in a` le corresponde la expresión MCLola dada por

```
iLet "a" (iValInt 5) (iRef "a")
```

Por lo tanto, considerando que usaremos la teoría **LetBind** para optimizar el código C99 generado para operadores derivados, resulta imprescindible elegir nombres de variables adecuados para evitar potenciales capturas de variables que puedan surgir al componer operadores derivados arbitrarios. Para ello, hemos extendido el lenguaje con una función de interpretación `strsIn` que determina los nombres que fueron usados en una expresión, junto con una función `fresh` que computa nombres frescos. Con estos recursos, podemos definir el operador `iDistancePointSegment` de la siguiente manera:

```
type DistancePointSegConstr e = (Point2D :<: e, LetBind :<: e,
    ProdTheory :<: e, BoolTheory :<: e, MinusConstr e,
    ComplexDivConstr e, ScalarMulConstr e, DistanceConstr e,
    NormConstr e, DBasicOp :<: e, DOrd :<: e)

iDistancePointSeg :: DistancePointSegConstr e ⇒ Term e Point2
    → Term e (Point2, Point2) → Term e Double
iDistancePointSeg p s =
    let fs = strsIn p ++ strsIn s
        [a, b, bmina, q, qx, qy] = fresh 6 fs
        ae     = iRef a
        be     = iRef b
        bminae = iRef bmina
        qe     = iRef q
        qxe    = iRef qx
        qye    = iRef qy

        cond   = iAnd (iDLeq (iValDouble 0) qxe)
                      (iDLeq qxe (iValDouble 1))
        exp1   = iNorm (iScalarMul qye bminae)
        exp2   = iDMin (iDistance p ae) (iDistance p be)

    in  iLet a (iFst s) (
        iLet b (iSnd s) (
        iLet bmina (iMinus be ae) (
        iLet q (iComplexDiv (iMinus p ae) bminae) (
        iLet qx (iX2D qe) (
        iLet qy (iY2D qe) (
           iIte cond exp1 exp2
        ))))))
```

8.3. Especificación mcLola

A partir de las teorías de datos Theories.Geometry2D y Theories.UAV escribimos en mcLola la especificación Lola presentada en la Sección 8.1 como se muestra en el Apéndice B.3. En particular, nótese que:

- Para implementar los streams filtered_pos_xp, filtered_pos_yp y filtered_pos_alt se utilizó parametrización estática para abstraer el cómputo del filtro.
- Para implementar el stream depth_into_poly, que computa un valor numérico sólo si el UAV se encuentra en la zona prohibida, se usó la teoría Theories.Maybe.
- La teoría Theories.List ha sido utilizada tanto en la implementación de la teoría geométrica para representar polígonos como listas de puntos y lados de un polígono como listas de pares de puntos; como así también en la propia especificación para representar a los eventos de la cámara. Notar, por ejemplo, el uso del operador foldrL en depth_into_poly para computar la mínima distancia del UAV a un lado del polígono o el uso del operador elemL en capturing para determinar si el UAV está tomando una foto.

En el directorio `Examples/UAV` del repositorio del proyecto puede encontrarse el archivo `UAV1.hs` que implementa esta especificación, junto con un archivo `monitor.c` que completa la implementación del monitor, una instancia de valores de entrada (`input.csv`) y la correspondiente salida producida por el monitor (`output.txt`).

Capítulo 9

Conclusiones y Trabajo Futuro

En este trabajo se desarrolló un lenguaje de dominio específico embebido en Haskell para la generación automática de monitores escritos en C99 para especificaciones Lola [19].

La elección de implementar el proyecto como un lenguaje de dominio específico embebido en Haskell se sustenta no sólo en las conocidas ventajas de embeber un lenguaje de dominio específico en un lenguaje anfitrión y en las características que hacen de Haskell un buen anfitrión, sino también en la experiencia positiva de otros lenguajes de SRV [19, 69], como hLola [9] y Copilot [62].

Concretamente, hemos desarrollado un lenguaje de especificación basado en Lola, que llamamos mcLola, junto con un mecanismo para generar monitores escritos en C99 a partir de especificaciones mcLola.

Uno de los mayores desafíos del proyecto fue el diseño del lenguaje. SRV parte de la observación de que los cómputos temporales son independientes de los cómputos de datos, con lo cual los lenguajes deberían ser fácilmente extensibles a nuevas teorías de datos, es decir a nuevos constructores. Por otro lado, la generación automática de código requiere de múltiples análisis sobre los términos del lenguaje, resultando apropiado que el lenguaje sea extensible a nuevas interpretaciones. Satisfacer ambas restricciones es un problema clásico en el diseño de lenguajes de dominio específico, conocido como *Expression Problem* [78]. Para este trabajo hemos optado por aplicar la solución ofrecida por *Datatypes à la Carte* [73] y, en particular, hemos utilizado la librería *compdata* [1, 12] para facilitar la implementación.

Para poder generar monitores C a partir de una especificación mcLola escrita por el usuario, en primer lugar se realiza un análisis estático. Durante esta etapa se analizan los aspectos temporales de la especificación, requeridos tanto para la validación de la misma como para el cómputo de los requerimientos espaciales del algoritmo de monitoreo; y se identifican los tipos de datos que participan en la especificación para poder traducirlos a tipos C99 equivalentes en una etapa posterior. Para implementar estos análisis hemos definido

funciones de interpretación adecuadas, siguiendo las técnicas de Datatypes à la Carte.

Para implementar la generación de código C99 hemos optado por utilizar la librería *language-c99-simple* [47] que permite manipular el AST de programas C99. Para contribuir a la seguridad del código generado, hemos seguido las directrices del estándar MISRA-C:2012 [58], que reúne una serie de buenas prácticas de programación en C para el desarrollo de sistemas críticos.

Además, hemos estudiado cómo introducir al lenguaje mecanismos para anticipar resultados en especificaciones que utilizan referencias a futuro. Concretamente, se incorporó al lenguaje el uso de simplificadores y se estudió cómo derivar simplificadores al nivel del AST del lenguaje. También se implementó una reescritura de expresiones para eliminar algunos cómputos innecesarios.

Hemos utilizado la herramienta para escribir especificaciones procedentes de diversas áreas, incluyendo lógicas temporales, análisis de datos financieros y UAVs. En este último caso, hemos implementado una especificación para la cual fue necesario extender el lenguaje con nuevas teorías de datos. La elección de implementar el proyecto como un lenguaje de dominio específico embebido en Haskell permitió facilitar considerablemente la tarea de incorporar nuevas teorías.

Trabajos futuros

Si bien la herramienta ya puede ser usada, existen varias líneas de trabajo para mejorarla y extenderla con nuevas características.

Los archivos de código C99 generados actualmente por la herramienta tienen, en general, muchas más líneas de código que las que escribiría normalmente un programador para resolver el mismo problema. Esto se debe, en parte, al uso de valores por defecto en los campos de los structs requerido por Misra C, combinado con el hecho de que la herramienta no implementa formas para reusar o compartir variables, por fuera del uso explícito de los operadores de let bindings. Asimismo, el uso de tipos opcionales para manejar referencias a futuro incrementa el número de líneas de código de la implementación, aún cuando se trata de streams con referencias exclusivamente pasadas. Para reducir el número de líneas de código generadas se pueden emplear técnicas similares a las usadas en compiladores para optimizar el uso de variables en una etapa posterior a la generación de código, como *Common Subexpression Elimination*; o bien se puede trabajar directamente en reducir el tamaño de la implementación durante la generación de código.

Otra línea de trabajo a futuro consiste en extender los mecanismos de anticipación para cubrir nuevos casos. Actualmente, cada valor puede ser (completamente) conocido o (completamente) desconocido. Considerar información parcial de un valor podría traer mejoras para anticipar resultados cuando se trabaja con tipos compuestos, como productos o records. Por otro lado, sería

interesante estudiar cómo incorporar dominios abstractos para aprovechar mejor la información estática. Un primer dominio abstracto a considerar podría ser el dado por $\{\top, ?, \bot\}$, donde, para un instante de tiempo i, $\top$ indica que un valor se conoce (o puede calcularse con seguridad), ? indica que el valor puede o no ser computado, dependiendo de los valores específicos con los que se instancien las variables, y $\bot$ indica que aún no puede computarse un valor. Conocer de antemano a qué categoría pertenece una expresión nos permitiría, por ejemplo, evitar cómputos innecesarios.

Otra extensión interesante para MCLOLA sería permitir el agregado de anotaciones en el código C generado, lo que resulta particularmente útil si se quiere utilizar herramientas externas para demostrar propiedades del código. Por ejemplo, en versiones anteriores de Copilot se ha utilizado *C Bounded Model Checking* [8, 46] para probar algunas propiedades de seguridad de memoria [16].

Otra línea de trabajo prometedora es la de producir *error-aware monitors*. El diseño que hemos dado para nuestro lenguaje de dominio específico, hace que el mismo sea fácilmente extensible a nuevas interpretaciones. Por ejemplo, podemos agregar funciones de interpretación que calculen los errores de redondeo en el caso de los streams numéricos. Poder computar automáticamente a partir de la especificación el error de su correspondiente monitor tiene implicancias muy útiles en la práctica. Por ejemplo, si tenemos tres monitores diferentes para una misma aplicación, podemos decidir cuál usar en cada situación específica, según la tolerancia al error que ofrezca cada uno.

Similarmente, puede estudiarse cómo introducir probabilidad al lenguaje para obtener monitores que manejen distintos niveles de confianza en los valores de los streams. Actualmente, asumimos como exactos a los valores dados como entradas, mientras que asumimos como completamente desconocidos a los valores futuros o aún no computados. Para incluir ciertas características que surgen con frecuencia en la práctica, como pérdidas de eventos o errores de medición, sería interesante introducir al lenguaje una noción de niveles de confianza en los datos.

Bibliografía

[1] Patrick Bahr y Tom Hvitved. "Compositional Data Types". En: *Proceedings of the Seventh ACM SIGPLAN Workshop on Generic Programming*. WGP '11. Tokyo, Japan: Association for Computing Machinery, 2011, págs. 83-94. ISBN: 9781450308618. DOI: 10.1145/2036918.2036930. URL: https://doi.org/10.1145/2036918.2036930.

[2] Howard Barringer, Allen Goldberg, Klaus Havelund y Koushik Sen. "Rule-Based Runtime Verification". En: *Verification, Model Checking, and Abstract Interpretation*. Ed. por Bernhard Steffen y Giorgio Levi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, págs. 44-57. ISBN: 978-3-540-24622-0.

[3] Ezio Bartocci, Yliès Falcone, Adrian Francalanza y Giles Reger. "Introduction to Runtime Verification". En: *Lectures on Runtime Verification: Introductory and Advanced Topics*. Ed. por Ezio Bartocci e Yliès Falcone. Cham: Springer International Publishing, 2018, págs. 1-33. ISBN: 978-3-319-75632-5. DOI: 10.1007/978-3-319-75632-5_1. URL: https://doi.org/10.1007/978-3-319-75632-5_1.

[4] *base: Basic libraries*. URL: https://hackage.haskell.org/package/base.

[5] Marco Benedetti y Alessandro Cimatti. "Bounded Model Checking for Past LTL". En: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. por Hubert Garavel y John Hatcliff. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, págs. 18-33. ISBN: 978-3-540-36577-8.

[6] Gérard Berry. "The Foundations of Esterel". En: *Proof, Language, and Interaction: Essays in Honour of Robin Milner*. Cambridge, MA, USA: MIT Press, 2000, págs. 425-454. ISBN: 0262161885.

[7] Per Bjesse, Koen Claessen, Mary Sheeran y Satnam Singh. "Lava: Hardware Design in Haskell". En: *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998*. Ed. por Matthias Felleisen, Paul Hudak y Christian Queinnec. ACM, 1998, págs. 174-184. DOI: 10.1145/289423.289440. URL: https://doi.org/10.1145/289423.289440.

[8] *C Bounded Model Checking for Software*. URL: http://www.cprover.org/cbmc/.

[9] Martin Ceresa, Felipe Gorostiaga y Cesar Sanchez. *Declarative Stream Runtime Verification (hLola)*. Feb. de 2020.

[10] E. M. Clarke, O. Grunberg y D. A. Peled. *Model Checking*. MIT Press, 1999.

[11] Curtis Clifton, Gary Leavens, Craig Chambers y Todd Millstein. "MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java". En: vol. 35. Oct. de 2000. DOI: 10.1145/354222.353181.

[12] *compdata: Compositional Data Types*. URL: https://hackage.haskell.org/package/compdata-0.12.1.

[13] *compdata: Compositional Data Types - Deriving boilerplate code*. URL: https://hackage.haskell.org/package/compdata-0.12.1/docs/Data-Comp-Multi-Derive.html.

[14] *containers: Assorted concrete container types*. URL: https://hackage.haskell.org/package/containers.

[15] Lukas Convent, Sebastian Hungerecker, Martin Leucker, Torben Scheffel, Malte Schmitz y Daniel Thoma. "TeSSLa: Temporal Stream-Based Specification Language". En: *Formal Methods: Foundations and Applications*. Ed. por Tiago Massoni y Mohammad Reza Mousavi. Cham: Springer International Publishing, 2018, págs. 144-162. ISBN: 978-3-030-03044-5.

[16] *Copilot: a DSL for Monitoring Embedded Systems*. URL: https://galois.com/blog/2010/09/copilot-a-dsl-for-monitoring-embedded-systems/.

[17] *Cppcheck*. URL: http://cppcheck.sourceforge.net/.

[18] *Cppcheck - Misra C 2012 Compliance*. URL: http://cppcheck.sourceforge.net/misra.php.

[19] B. D'Angelo, S. Sankaranarayanan, C. Sanchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra y Z. Manna. "LOLA: runtime monitoring of synchronous systems". En: *12th International Symposium on Temporal Representation and Reasoning (TIME'05)*. 2005, págs. 166-174. DOI: 10.1109/TIME.2005.26.

[20] Luis Miguel Danielsson y César Sánchez. "Decentralized Stream Runtime Verification". En: *Runtime Verification*. Ed. por Bernd Finkbeiner y Leonardo Mariani. Cham: Springer International Publishing, 2019, págs. 185-201. ISBN: 978-3-030-32079-9.

[21] *Data.Typeable*. URL: https://hackage.haskell.org/package/base-4.15.0.0/docs/Data-Typeable.html.

[22] Frank Dedden. *An implementation of the C99 AST that strictly follows the standard.* URL: `https://hackage.haskell.org/package/language-c99`.

[23] Frank Dedden. *language-c99-simple: C-like AST to simplify writing C99 programs.* URL: `https://hackage.haskell.org/package/language-c99-simple`.

[24] Kadir Demir, Halil Cicibaş y Nafiz Arica. "Unmanned Aerial Vehicle Domain: Areas of Research". En: *Defence science journal* 65 (jul. de 2015), págs. 319-329. DOI: `10.14429/dsj.65.8631`.

[25] *Docker.* URL: `https://www.docker.com`.

[26] Wei Dong, Martin Leucker y Christian Schallhart. "Impartial Anticipation in Runtime-Verification". En: vol. 5311. Oct. de 2008, págs. 386-396. ISBN: 978-3-540-88386-9. DOI: `10.1007/978-3-540-88387-6_33`.

[27] Conal Elliott y Paul Hudak. "Functional Reactive Animation". En: *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming.* ICFP '97. Amsterdam, The Netherlands: Association for Computing Machinery, 1997, págs. 263-273. ISBN: 0897919181. DOI: `10.1145/258948.258973`. URL: `https://doi.org/10.1145/258948.258973`.

[28] Jacques Garrigue. "Code Reuse Through Polymorphic Variants". En: nov. de 2000.

[29] Thierry Gautier, Paul Le Guernic y Löic Besnard. "SIGNAL: A Declarative Language for Synchronous Programming of Real-Time Systems". En: *Proc. of a Conference on Functional Programming Languages and Computer Architecture.* Portland, Oregon, USA: Springer-Verlag, 1987, págs. 257-277. ISBN: 0387183175.

[30] *GCC, the GNU Compiler Collection.* URL: `https://gcc.gnu.org`.

[31] *Generalised Algebraic Data Types.* URL: `https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/exts/gadt.html`.

[32] *GHCi.* URL: `https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/ghci.html`.

[33] Andy Gill. "Domain-Specific Languages and Code Synthesis Using Haskell: Looking at Embedded DSLs". En: *Queue* 12.4 (2014), págs. 30-43. ISSN: 1542-7730. DOI: `10.1145/2611429.2617811`. URL: `https://doi.org/10.1145/2611429.2617811`.

[34] Felipe Gorostiaga y César Sánchez. "Striver: Stream Runtime Verification for Real-Time Event-Streams: 18th International Conference, RV 2018, Limassol, Cyprus, November 10–13, 2018, Proceedings". En: nov. de 2018, págs. 282-298. ISBN: 978-3-030-03768-0. DOI: `10.1007/978-3-030-03769-7_16`.

[35] N. Halbwachs, P. Caspi, P. Raymond y D. Pilaud. "The synchronous data flow programming language LUSTRE". En: *Proceedings of the IEEE* 79.9 (1991), págs. 1305-1320. DOI: 10.1109/5.97300.

[36] *HLola Home Page*. URL: https://software.imdea.org/hlola/.

[37] P. Hudak. "Modular Domain Specific Languages and Tools". En: *Proceedings of the 5th International Conference on Software Reuse*. ICSR '98. USA: IEEE Computer Society, 1998, pág. 134. ISBN: 0818683775.

[38] Paul Hudak. "Building Domain-Specific Embedded Languages". En: *ACM Comput. Surv.* 28.4es (dic. de 1996), 196-es. ISSN: 0360-0300. DOI: 10.1145/242224.242477. URL: https://doi.org/10.1145/242224.242477.

[39] Paul Hudak. "Haskore Music Tutorial". En: *In Second International School on Advanced Functional Programming*. Springer Verlag, 1996, págs. 38-67.

[40] Paul Hudak y Mark P. Jones. *P.: Haskell vs. Ada vs. C++ vs. awk vs.. . . An experiment in software prototyping productivity*. Inf. téc. Department of Computer Science, Yale University, 1994.

[41] *ISO/IEC 9899:TC3*. URL: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf.

[42] *Ivory Language*. URL: https://ivorylang.org/.

[43] Patricia Johann y Neil Ghani. "Initial Algebra Semantics Is Enough!" En: *Typed Lambda Calculi and Applications*. Ed. por Simona Ronchi Della Rocca. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, págs. 207-222. ISBN: 978-3-540-73228-0.

[44] Patricia Johann y Neil Ghani. "Foundations for Structured Programming with GADTs". En: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '08. San Francisco, California, USA: Association for Computing Machinery, 2008, págs. 297-308. ISBN: 9781595936899. DOI: 10.1145/1328438.1328475. URL: https://doi.org/10.1145/1328438.1328475.

[45] Patricia Johann y Neil Ghani. "Foundations for structured programming with GADTs". En: vol. 43. Ene. de 2008, págs. 297-308. DOI: 10.1145/1328438.1328475.

[46] Daniel Kroening y Michael Tautschnig. "CBMC – C Bounded Model Checker". En: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. por Erika Ábrahám y Klaus Havelund. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, págs. 389-391. ISBN: 978-3-642-54862-8.

[47] *language-c99-simple: C-like AST to simplify writing C99 programs*. URL: https://hackage.haskell.org/package/language-c99-simple.

[48] *language-c99: An implementation of the C99 AST that strictly follows the standard.* URL: `https://hackage.haskell.org/package/language-c99`.

[49] Chambers Leavens, Craig Chambers y Gary Leavens. "Typechecking and Modules for Multi-Methods". En: *ACM SIGPLAN Notices* 29 (sep. de 1994). DOI: `10.1145/191081.191083`.

[50] Martin Leucker y Christian Schallhart. "A brief account of runtime verification". En: *The Journal of Logic and Algebraic Programming* 78.5 (2009). The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07), págs. 293-303. ISSN: 1567-8326. DOI: `https://doi.org/10.1016/j.jlap.2008.08.004`. URL: `https://www.sciencedirect.com/science/article/pii/S1567832608000775`.

[51] *Lexically scoped type variables.* URL: `https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/exts/scoped_type_variables.html`.

[52] Peter Liu, Albert Chen, Yin-Nan Huang, Jen-Yu Han, Jihn-Sung Lai, Shih-Chung Kang, Tzong-Hann Richard Wu, Ming-Chang Wen y Meng-Han Tsai. "A review of rotorcraft Unmanned Aerial Vehicle (UAV) developments and applications in civil engineering". En: *SMART STRUCTURES AND SYSTEMS* 13 (jun. de 2014), págs. 1065-1094. DOI: `10.12989/sss.2014.13.6.1065`.

[53] Zohar Manna y Amir Pnueli. *Temporal Verification of Reactive Systems: Safety.* Berlin, Heidelberg: Springer-Verlag, 1995. ISBN: 0387944591.

[54] Simon Marlow y Simon Peyton Jones. "The Glasgow Haskell Compiler". En: *The Architecture of Open Source Applications, Volume 2.* The Architecture of Open Source Applications, Volume 2. Lulu, ene. de 2012. URL: `https://www.microsoft.com/en-us/research/publication/the-glasgow-haskell-compiler/`.

[55] Muhammad Nadeem Mirza, Irfan Hasnain Qaisrani, Lubna Abid Ali y Ahmad Ali Naqvi. "Unmanned Aerial Vehicles: A Revolution in the Making". En: *South Asian Studies - A Research Journal of South Asian Studies* 31.2 (2016), págs. 243-256. URL: `https://halshs.archives-ouvertes.fr/halshs-02951743`.

[56] *MISRA.* URL: `https://www.misra.org.uk/`.

[57] *Misra-C:2004. Guidelines for the Use of the C Language in Critical Systems.* 2004. ISBN: ISBN 978-0-9524156-4-0.

[58] *Misra-C:2012. Guidelines for the Use of the C Language in Critical Systems.* 2013. ISBN: ISBN 978-1-906400-11-8.

[59] *mtl: Monad classes, using functional dependencies.* URL: `https://hackage.haskell.org/package/mtl`.

[60] Bruno Oliveira y William Cook. "Extensibility for the masses: practical extensibility with object algebras". En: jun. de 2012, págs. 2-27. ISBN: 978-3-642-31056-0. DOI: 10.1007/978-3-642-31057-7_2.

[61] *parsec: Monadic parser combinators*. URL: https://hackage.haskell.org/package/parsec.

[62] Ivan Perez y Alwyn Goodloe. *Copilot 3*. Abr. de 2020. DOI: 10.13140/RG.2.2.35163.80163.

[63] *Pine Script*. URL: https://www.tradingview.com/pine-script-docs/en/v4/index.html.

[64] *Pine Script language reference manual*. URL: https://www.tradingview.com/pine-script-reference/.

[65] *pretty: Pretty-printing library*. URL: https://hackage.haskell.org/package/pretty.

[66] Hailong Qin, Jin Q. Cui, Jiaxin Li, Yingcai Bi, Menglu Lan, Mo Shan, Wenqi Liu, Kangli Wang, F. Lin, Y. F. Zhang y Ben M. Chen. "Design and implementation of an unmanned aerial vehicle for autonomous firefighting missions". En: *2016 12th IEEE International Conference on Control and Automation (ICCA)*. 2016, págs. 62-67. DOI: 10.1109/ICCA.2016.7505253.

[67] *Reader Monad*. URL: https://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-Reader.html.

[68] Grigore Roşu y Klaus Havelund. "Rewriting-Based Techniques for Runtime Verification". En: *Automated Software Engineering* 12 (ene. de 2005), págs. 151-197. DOI: 10.1007/s10515-005-6205-y.

[69] César Sánchez. "Online and Offline Stream Runtime Verification of Synchronous Systems". En: *Runtime Verification*. Ed. por Christian Colombo y Martin Leucker. Cham: Springer International Publishing, 2018, págs. 138-163. ISBN: 978-3-030-03769-7.

[70] Koushik Sen y Grigore Roşu. "Generating Optimal Monitors for Extended Regular Expressions". En: *Electronic Notes in Theoretical Computer Science* 89.2 (2003). RV '2003, Run-time Verification (Satellite Workshop of CAV '03), págs. 226-245. ISSN: 1571-0661. DOI: https://doi.org/10.1016/S1571-0661(04)81051-X. URL: https://www.sciencedirect.com/science/article/pii/S157106610481051X.

[71] *Specification examples in HLola*. URL: https://software.imdea.org/hlola/specs.html.

[72] Josef Svenningsson y Emil Axelsson. "Combining Deep and Shallow Embedding for EDSL". En: *Trends in Functional Programming*. Ed. por Hans-Wolfgang Loidl y Ricardo Peña. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, págs. 21-36. ISBN: 978-3-642-40447-4.

[73] Wouter Swierstra. "Data Types à La Carte". En: *J. Funct. Program.* 18.4 (jul. de 2008), págs. 423-436. ISSN: 0956-7968. DOI: 10.1017/S0956796808006758.

[74] *Template Haskell*. URL: https://downloads.haskell.org/~ghc/7.0.3/docs/html/users_guide/template-haskell.html.

[75] *The Glasgow Haskell Compiler*. URL: https://www.haskell.org/ghc/.

[76] *Visible type application*. URL: https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/exts/type_applications.html.

[77] Dimitrios Vytiniotis, Stephanie Weirich y Simon Peyton Jones. "Simple unification-based type inference for GADTs". En: *International Conference on Functional Programming (ICFP'06)*. 2016 ACM SIGPLAN Most Influential ICFP Paper Award. ACM SIGPLAN. Abr. de 2006. URL: https://www.microsoft.com/en-us/research/publication/simple-unification-based-type-inference-for-gadts/.

[78] Philip Wadler. *Expression Problem*. URL: https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt.

[79] Stefan Wehr y Peter Thiemann. "JavaGI: The Interaction of Type Classes with Interfaces and Inheritance". En: *ACM Trans. Program. Lang. Syst.* 33 (ene. de 2011), pág. 12.

[80] *Wolfram Demonstration - Distance of a point to a segment*. URL: https://demonstrations.wolfram.com/DistanceOfAPointToASegment/.

[81] Chenchen Xu, Xiaohan Liao, Junming Tan, Huping Ye y Haiying Lu. "Recent Research Progress of Unmanned Aerial Vehicle Regulation Policies and Technologies in Urban Low Altitude". En: *IEEE Access* 8 (2020), págs. 74175-74194. DOI: 10.1109/ACCESS.2020.2987622.

[82] Sebastián Zudaire, Felipe Gorostiaga, César Sánchez, Gerardo Schneider y Sebastián Uchitel. "Assumption Monitoring Using Runtime Verification for UAV Temporal Task Plan Executions". En: *2021 IEEE International Conference on Robotics and Automation (ICRA)*. 2021, págs. 6824-6830. DOI: 10.1109/ICRA48506.2021.9561671.

Apéndice A

Referencia al código fuente

A.1. Hoja de ruta del código fuente

Actualmente este trabajo se encuentra alojado en un repositorio *Git* situado en el siguiente enlace: `https://github.com/imdea-software/McLola`. A continuación se expone un resumen general del contenido y organización del mismo, cuya estructura se visualiza en la Figura A.1.

El directorio `doc` contiene a este documento, mientras que en `src` se encuentra el código fuente correspondiente a la implementación del proyecto. Para facilitar su ejecución, se incluye un archivo `Dockerfile` (para más información consultar la Sección A.2).

Dentro del directorio `src`, el archivo `MCLola.hs` importa las definiciones básicas del lenguaje, requeridas para escribir cualquier especificación mcLola; mientras que en el archivo `Compile.hs`, presentado en la Sección 4.2, se debe indicar la especificación a analizar.

El directorio `Language` contiene la implementación del lenguaje mcLola, descripta en el Capítulo 5. Mientras que en el archivo `Lola.hs` se definen los tipos **Exp** y **Stream** y se implementan los operadores temporales del lenguaje, en el archivo `Spec.hs` se implementa la unificación de tipos de los streams para definir el tipo **Specification**. La implementación de los operadores del lenguaje se completa con el directorio `Theories` correspondiente a las teorías de datos.

El directorio `StaticAnalysis` contiene la implementación del análisis estático. Los archivos `TempRef.hs` y `Types.hs` definen las clases que implementan las álgebras requeridas para la construcción del grafo de dependencias y el cómputo de tipos, respectivamente, según se explica en las Secciones 6.1.1 y 6.1.3; mientras que el archivo `TypesComputation.hs` completa el análisis de tipos según se describe en el final de la Sección 6.1.3.

El directorio `Data` implementa la generación de tipos de datos en C99, que comprende la definición de tipos de datos usados por la especificación (`TypeGen.hs`) y la implementación de buffers para almacenar valores duran-

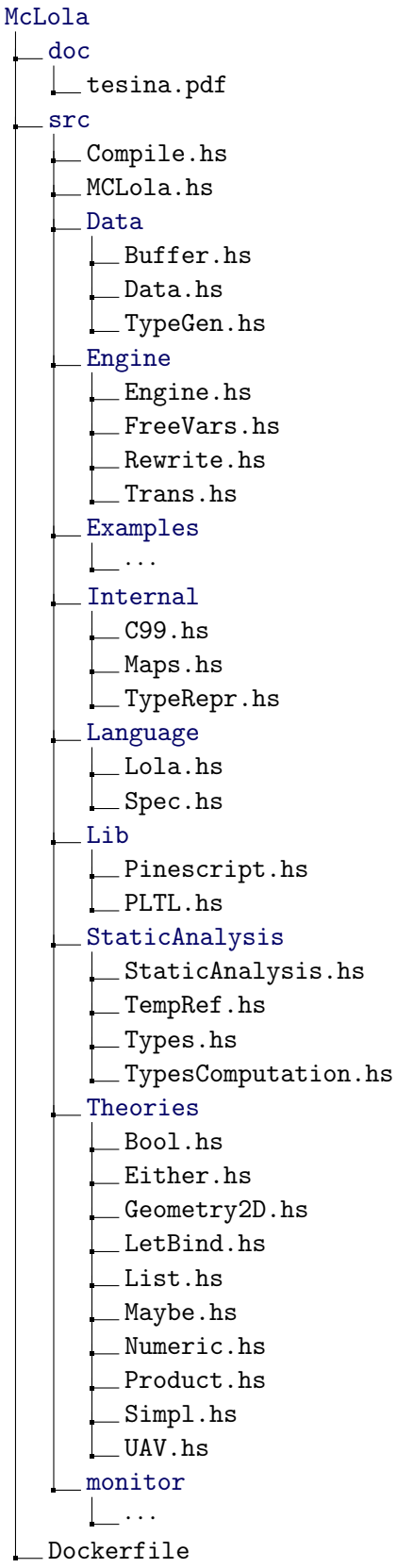


Figura A.1: Estructura del proyecto

te la ejecución del monitor (`Buffer.hs`), siguiendo las ideas explicadas en la Sección 6.2.2.

El directorio `Engine` implementa la generación del engine temporal, para lo cual se define en el archivo `Trans.hs` un álgebra para la traducción de expresiones según se explica en la Sección 6.2.3. El archivo `Rewrite.hs` define el álgebra para la reescritura de expresiones vistas en la Sección 7.2; mientras que `FreeVars.hs` implementa el cómputo de variables libres requerido para la generación de nombres frescos para evitar posibles capturas de variables introducidas por los operadores de let bindings.

El directorio `Internal` agrupa estructuras de datos usadas internamente por la herramienta incluyendo: la representación de los tipos vista en la Sección 5.2.1 (`TypeRepr.hs`), la representación de expresiones y tipos C99 introducida en la Sección 6.2.3 (`C99.hs`) y otras estructuras auxiliares para almacenar datos del análisis estático (`Maps.hs`).

El directorio `Examples` contiene varios ejemplos de especificaciones mcLola, incluyendo los vistos en este informe. Para cada ejemplo se provee:

- Un archivo de extensión `hs` con la especificación mcLola.
- Un archivo `monitor.c` con la definición de un monitor adecuado, usando la interfaz provista por el archivo `engine.h` generado por la herramienta.
- Un archivo `input.csv` con una instancia para los valores de entrada.
- Un archivo `output.txt` con la salida que genera el monitor para la instancia dada por `input.csv`.

Los archivos generados por la herramienta (`analysis.txt`, `data.h`, `data.c`, `engine.h` y `engine.c`) se almacenan en el directorio `monitor`.

Por último, el directorio `Lib` incluye librerías para escribir especificaciones mcLola. Al momento se ofrece una librería para *Past LTL* [5] (`PLTL.hs`) y una para *Pine Script* [63] (`Pinescript.hs`), esta última presentada como ejemplo en la Sección 4.3.

A.2. Imagen de Docker

Para facilitar el uso de la herramienta, se provee un archivo Dockerfile [25] para crear una imagen con todo el software necesario para ejecutar el proyecto.

A continuación se detallan los pasos a seguir para construir la imagen Docker, correr el container y utilizar la herramienta.

1. Descargar el repositorio del proyecto:

```
$ git clone https://github.com/imdea-software/McLola.git
```

2. Construir la imagen Docker del proyecto (`mclola`):

```
$ cd McLola/
$ docker build -t mclola .
```

Nota: dependiendo de la configuración de Docker, puede reportarse un error de acceso al correr este comando. En ese caso, si se usa un sistema Linux, prefijar los comandos docker con `sudo`.

3. Correr un container usando la imagen `mclola`:

```
$ docker run -it -v "$(pwd)":/MCLola --rm mclola bash
```

Este comando crea un container para la imagen `mclola`, monta el directorio local actual en el directorio `MCLola` y abre un shell interactivo. La opción `-rm` se agrega para borrar automáticamente el container una vez finalizada su ejecución.

4. Dentro del container, cambiarse al directorio `MCLola/src` para utilizar la herramienta:

```
user@contid:/# cd MCLola/src
user@contid:/MCLola/src# ghci
GHCi, version 8.10.4: https://www.haskell.org/ghc/ :? for help
Loaded package environment from /root/.ghc/x86_64-linux-8.10.4
/environments/default
Prelude> :l Compile.hs
[ 1 of 30] Compiling Engine.FreeVars
...
[30 of 30] Compiling Compile
Ok, 30 modules loaded.
*Compile> codegen
*Compile>
Leaving GHCi.
```

Incluso se puede compilar y ejecutar el monitor si se proveen los archivos complementarios descriptos en la Sección 4.2:

```
user@contid:/MCLola/src# cd monitor/
user@contid:/MCLola/src/monitor# make && ./monitor < input.csv
> output.txt
gcc -std=c99 -o monitor monitor.c engine.c data.c lib.c -lm
```

Al finalizar, presionar Ctrl + D para cerrar el container.

A.3. Desviaciones Misra C

Para analizar el cumplimiento de los lineamientos de MISRA-C:2012, hemos utilizado Cppcheck [17]. En el directorio `Examples` del código fuente del proyecto se incluye un script con los comandos necesarios para verificar las reglas Misra C soportadas por Cppcheck (archivo `checkMonitor`).

En particular, hemos utilizado una única desviación de las reglas *requeridas* de MISRA-C:2012, correspondiente a la regla que prohíbe el uso de funciones de entrada/salida de la librería estándar de C[1]. Si bien en este trabajo se invoca a la función `printf`, como sólo se la utiliza para comunicar los resultados computados por el engine, consideramos que no compromete la seguridad del código.

[1]Por motivos de licencias, no se indica su identificación.

Apéndice B

Ejemplos

A continuación se incluye el código completo correspondiente a las especificaciones MCLola presentadas como ejemplos en este informe.

B.1. Ejemplos Capítulo 4

Ejemplo 4.2

```
1  module Examples.Ex4_2.Spec (spec) where
2
3  import MCLola
4  import Theories.Bool
5  import Theories.Numeric
6
7  -- | Input
8
9  a :: Stream Int
10 a = In "a"
11
12 -- | Output
13
14 x :: Stream Bool
15 x = Out ("x", expr)
16   where
17     expr = iOr e1 e2
18     e1   = iIGt (iNow a) (iOffset a (-1) (iValInt 0))
19     e2   = iOffset x (-1) (iValBool False)
20
21 y :: Stream Int
22 y = Out ("y", expr)
23   where
24     expr = iIAdd (iOffset y (-1) dflt) (iOffset a 1 dflt)
25     dflt = iValInt 1
26
27 -- | Specification
28
29 spec :: Specification
```

```
30  spec = [ toDynStrm a
31         , toDynStrm x
32         , toDynStrm y]
```

Ejemplo 4.6

```
1   module Examples.Pinescript.Ex1.Spec (spec) where
2
3   import MCLola
4   import Theories.Bool
5   import Theories.Numeric
6
7   -- | Input
8
9   x :: Stream Double
10  x = In "x"
11
12  y :: Stream Double
13  y = In "y"
14
15  -- | Output
16
17  instant :: Stream Int
18  instant = Out("instant", expr)
19    where
20      expr = iIAdd (iOffset instant (-1) (iValInt 0)) (iValInt 1)
21
22  sma5x :: Stream Double
23  sma5x = Out("sma5x", expr)
24    where
25      num  = foldr (\off e → iDAdd (iOffset x off (iValDouble 0)
            ) e) (iNow x) [(-4)..(-1)]
26      den  = iIntToDouble (iIMin (iNow instant) (iValInt 5))
27      expr = iDDiv num den
28
29  sma5y :: Stream Double
30  sma5y = Out("sma5y", expr)
31    where
32      num  = foldr (\off e → iDAdd (iOffset y off (iValDouble 0)
            ) e) (iNow y) [(-4)..(-1)]
33      den  = iIntToDouble (iIMin (iNow instant) (iValInt 5))
34      expr = iDDiv num den
35
36  crov :: Stream Bool
37  crov = Out ("crov", expr)
38    where
39      dflt = iValDouble 0
40      expr = iAnd (iDLeq (iNow sma5y) (iNow sma5x))
41                  (iDLeq (iOffset sma5x (-1) dflt) (iOffset sma5y
                        (-1) dflt))
42
43  -- | Specification
44
```

```
45 spec :: Specification
46 spec = [ toDynStrm x, toDynStrm y
47         , toDynStrm instant, toDynStrm sma5x, toDynStrm sma5y
48         , toDynStrm crov]
```

Ejemplo 4.7

```
1  module Examples.Pinescript.Ex2.Spec (spec) where
2
3  import MCLola
4  import Theories.Bool
5  import Theories.Numeric
6  import Theories.Maybe
7
8  -- | Input
9
10 x :: Stream Double
11 x = In "x"
12
13 -- | Output
14
15 instant :: Stream Int
16 instant = Out("instant", expr)
17   where
18     expr = iIAdd (iOffset instant (-1) (iValInt 0)) (iValInt 1)
19
20 sma5 :: Stream (Maybe Double)
21 sma5 = Out("sma5", expr)
22   where
23     num  = foldr (\off e -> iDAdd (iOffset x off (iValDouble 0)
           ) e) (iNow x) [(-4)..(-1)]
24     den  = iValDouble 5
25     expr = iIte (iDLeq den (iIntToDouble (iNow instant)))
26                 (iJustM (iDDiv num den))
27                 (iNothingM (iValDouble 0))
28
29 -- | Specification
30
31 spec :: Specification
32 spec = [toDynStrm x, toDynStrm instant, toDynStrm sma5]
```

Ejemplo 4.8

```
1  module Examples.Pinescript.Ex3.Spec (spec) where
2
3  import MCLola
4  import Theories.Bool
5  import Theories.Numeric
6
7  -- | Input
8
9  x :: Stream Double
```

```
10 x = In "x"
11
12 y :: Stream Double
13 y = In "y"
14
15 -- / Output
16
17 instant :: Stream Int
18 instant = Out("instant", expr)
19   where
20     expr = iIAdd (iOffset instant (-1) (iValInt 0)) (iValInt 1)
21
22 sma :: Stream Double → Int → Stream Double
23 sma x n = Out(name, expr)
24   where
25     name = "sma_" ++ show n ++ "_" ++ ident x
26     num  = foldr (\off e → iDAdd (iOffset x off (iValDouble 0)
           ) e) (iNow x) [((-1)*(n-1)) .. (-1)]
27     den  = iIntToDouble (iIMin (iNow instant) (iValInt n))
28     expr = iDDiv num den
29
30 crossover :: Stream Double → Stream Double → Stream Bool
31 crossover x y = Out (name, expr)
32   where
33     name = ident x ++ "_crossover_" ++ ident y
34     dflt = iValDouble 0
35     expr = iAnd (iDLeq (iNow y) (iNow x))
36                 (iDLeq (iOffset x (-1) dflt) (iOffset y (-1)
                        dflt))
37
38 sma5x :: Stream Double
39 sma5x = sma x 5
40
41 sma5y :: Stream Double
42 sma5y = sma y 5
43
44 crov :: Stream Bool
45 crov = crossover sma5x sma5y
46
47 -- / Specification
48
49 spec :: Specification
50 spec = [ toDynStrm x, toDynStrm y
51        , toDynStrm instant, toDynStrm sma5x, toDynStrm sma5y
52        , toDynStrm crov]
```

Ejemplo 4.9

```
1 module Examples.Pinescript.Ex4.Spec (spec) where
2
3 import MCLola
4 import Lib.Pinescript
5
```

```
6  -- | Input
7
8  x :: Stream Double
9  x = In "x"
10
11 y :: Stream Double
12 y = In "y"
13
14 -- | Output
15
16 sma5x :: Stream Double
17 sma5x = sma x 5
18
19 sma5y :: Stream Double
20 sma5y = sma y 5
21
22 crov :: Stream Bool
23 crov = crossover sma5x sma5y
24
25 -- | Specification
26
27 spec :: Specification
28 spec = [ toDynStrm x, toDynStrm y
29        , toDynStrm instant, toDynStrm sma5x, toDynStrm sma5y
30        , toDynStrm crov]
```

B.2. Ejemplos Capítulo 7

Ejemplo de simplificadores

```
1  module Examples.Anticip.ExSimplChap7.Spec (spec) where
2
3  import MCLola
4  import Theories.Bool
5  import Theories.Numeric
6
7  -- | Input
8
9  s :: Stream Bool
10 s = In "s"
11
12 a :: Stream Bool
13 a = In "a"
14
15 b :: Stream Bool
16 b = In "b"
17
18 -- | Output
19
20 count :: Stream Int
21 count = Out ("count", expr)
22   where
```

```
23      expr = iIAdd prev curr
24      prev = iOffset count (-1) (iValInt 0)
25      curr = iIte (iNow ok) (iValInt 1) (iValInt 0)
26
27  end :: Stream Bool
28  end = Out ("end", expr)
29    where
30      expr = iIGeq (iNow count) (iValInt 3)
31
32  ok :: Stream Bool
33  ok = Out ("ok", expr)
34    where
35      expr = iIte cond e1 e2
36      cond = iOr (iOffset s (-2) (iValBool False)) (iOffset s 2 (
            iValBool False))
37      e1   = iAnd (iNow a) (iOffset a 1 (iValBool True))
38      e2   = iOr (iNow b) (iOffset b 7 (iValBool True))
39
40  -- | Specification
41
42  spec :: Specification
43  spec = [ toDynStrm s, toDynStrm a, toDynStrm b
44         , toDynStrm count, toDynStrm end, toDynStrm ok ]
```

Ejemplo de reescritura

```
1   module Examples.Anticip.ExRewriteChap7.Spec (spec) where
2
3   import MCLola
4   import Theories.Numeric
5   import Theories.Product
6
7   -- | Input
8
9   a :: Stream Int
10  a = In "a"
11
12  -- | Output
13
14  x :: Stream Int
15  x = Out ("x", expr)
16    where
17      expr = iFst pair
18      pair = iPair (iOffset a (-1) (iValInt 5)) (iOffset a 3 (
            iValInt 5))
19
20  -- | Specification
21
22  spec :: Specification
23  spec = [ toDynStrm a
24         , toDynStrm x ]
```

B.3. Ejemplo Capítulo 8

Especificación UAV 1 (Código 8.1)

```
1  module Examples.UAV.UAV1 (spec) where
2
3  import MCLola
4  import Theories.Bool
5  import Theories.Numeric
6  import Theories.Geometry2D
7  import Theories.UAV
8  import Theories.Maybe
9  import Theories.List
10
11 -- | Input streams
12
13 attitude :: Stream Attitude
14 attitude = In "attitude"
15
16 velocity :: Stream Point2
17 velocity = In "velocity"
18
19 position :: Stream Position
20 position = In "position"
21
22 altitude :: Stream Double
23 altitude = In "altitude"
24
25 target :: Stream Target
26 target = In "target"
27
28 nofly :: Stream [Point2]
29 nofly = In "nofly"
30
31 events_within :: Stream [Int]
32 events_within = In "events_within"
33
34 -- | Output streams
35
36 -- Auxiliar streams
37
38 filtered_pos :: (Exp Position → Exp Double) → String →
       Stream Double
39 filtered_pos field name = Out (name, expr)
40   where
41     nowcomp = field (iNow position)
42     this    = filtered_pos field name
43     calc    = iDDiv
44       (iDAdd nowcomp
45         (iDMul (iValDouble 0.6) (iDSub (iDMul (iValDouble 2) (
              iOffset this (-1) (iValDouble 0))) (iDDiv (iOffset
              this (-2) (iValDouble 0)) (iValDouble 2)))))
46       (iValDouble 1.9)
```

```
47      cond    = iILt (iNow instantN) (iValInt 3)
48      expr    = iIte cond nowcomp calc
49
50  filtered_pos_xp :: Stream Double
51  filtered_pos_xp = filtered_pos iXp "filtered_pos_xp"
52
53  filtered_pos_yp :: Stream Double
54  filtered_pos_yp = filtered_pos iYp "filtered_pos_yp"
55
56  filtered_pos_alt :: Stream Double
57  filtered_pos_alt = filtered_pos iAlt "filtered_pos_alt"
58
59  instantN :: Stream Int
60  instantN = Out ("instantN", expr)
61    where
62     expr = iIAdd (iValInt 1) (iOffset instantN (-1) (iValInt 0))
63
64  -- Output stream: all_ok_capture
65
66  roll_ok :: Stream Bool
67  roll_ok = Out ("roll_ok", expr)
68    where
69      eRoll = iDAbs (iRoll (iNow attitude))
70      expr  = iDLt eRoll (iValDouble 0.0523)
71
72  pitch_ok :: Stream Bool
73  pitch_ok = Out ("pitch_ok", expr)
74    where
75      ePitch = iDAbs (iPitch (iNow attitude))
76      expr   = iDLt ePitch (iValDouble 0.0523)
77
78  height_ok :: Stream Bool
79  height_ok = Out ("height_ok", expr)
80    where
81      expr = iDGt (iNow filtered_pos_alt) (iValDouble 0)
82
83  near :: Stream Bool
84  near = Out ("near", expr)
85    where
86      filPos = iPoint2D (iNow filtered_pos_xp)
87                        (iNow filtered_pos_yp)
88      tarPos = iPoint2D (iXt (iNow target)) (iYt (iNow target))
89      e_dist = iDistance filPos tarPos
90      expr   = iDLt e_dist (iValDouble 1)
91
92  open_capture :: Stream Bool
93  open_capture = Out ("open_capture", expr)
94    where
95      rel = foldr (\cod → iConsL (iValInt cod))
96              (iSinglL (iValInt 45)) -- 45 → "capture"
97              [50, 51] -- 50 → "yes_person", 51 → "no_person"
98      relEvents = filterElem (iValInt 0) rel (iNow events_within)
99      def       = iOffset open_capture (-1) (iValBool False)
100     lastIsCapt = iEq (iLast relEvents) (iValInt 45)
```

```
101     expr       = iIte (iEq (iValInt 0) (iLength relEvents))
102                       def
103                       lastIsCapt
104
105 capturing :: Stream Bool
106 capturing = Out ("capturing", expr)
107   where
108     hasCapt = elemL (iValInt 45) (iNow events_within)
109     expr    = iOr hasCapt
110                   (iOffset open_capture (-1) (iValBool False))
111
112 all_ok_capture :: Stream Bool
113 all_ok_capture = Out ("all_ok_capture", expr)
114   where
115     capt = iNow capturing
116     ok   = foldr (\ s → iAnd (iNow s))
117                  (iNow height_ok)
118                  [near, roll_ok, pitch_ok]
119     expr = iImpl capt ok
120
121 -- Output stream: flying_in_safe_zones
122
123 no_fly :: Stream [Point2]
124 no_fly = Out ("no_fly", expr)
125   where
126     expr = iOffset no_fly (-1) (iNow nofly)
127
128 flying_in_safe_zones :: Stream Bool
129 flying_in_safe_zones = Out ("flying_in_safe_zones", expr)
130   where
131     filPos = iPoint2D (iNow filtered_pos_xp)
132                       (iNow filtered_pos_yp)
133     poly   = iNow no_fly
134     expr   = iNot (pointInPoly filPos poly)
135
136 -- Output stream: depth_into_poly
137
138 depth_into_poly :: Stream (Maybe Double)
139 depth_into_poly = Out ("depth_into_poly", expr)
140   where
141     filPos       = iPoint2D (iNow filtered_pos_xp)
142                             (iNow filtered_pos_yp)
143     poly         = iNow no_fly
144     polySides    = polygonSides poly
145     shortestDist = foldrL
146             (\ seg n → iDMin n (iDistancePointSeg filPos seg))
147             (iDistancePointSeg filPos (iHead polySides))
148             (iTail polySides)
149
150     expr = iIte (pointInPoly filPos poly)
151                 (iJustM shortestDist)
152                 (iNothingM (iValDouble 0))
153
154 -- | Specification
```

```
155
156 spec :: Specification
157 spec = [ -- input streams
158           toDynStrm attitude
159         , toDynStrm velocity
160         , toDynStrm position
161         , toDynStrm altitude
162         , toDynStrm target
163         , toDynStrm nofly
164         , toDynStrm events_within
165
166           -- auxiliar streams
167         , toDynStrm filtered_pos_xp
168         , toDynStrm filtered_pos_yp
169         , toDynStrm filtered_pos_alt
170         , toDynStrm instantN
171
172           -- output stream: all_ok_capture
173         , toDynStrm roll_ok
174         , toDynStrm pitch_ok
175         , toDynStrm height_ok
176         , toDynStrm near
177         , toDynStrm capturing
178         , toDynStrm open_capture
179         , toDynStrm all_ok_capture
180
181           -- output stream: flying_in_safe_zones
182         , toDynStrm no_fly
183         , toDynStrm flying_in_safe_zones
184
185           -- output stream: depth_into_poly
186         , toDynStrm depth_into_poly ]
```