UNIVERSIDAD DE LA REPÚBLICA URUGUAY

Tesis de Maestría en Informática

Lenguaje de dominio específico embebido para programación estocástica multietapa en Scala

Germán Ferrari

Director de Tesis:

Carlos E. Testuri
Instituto de Computación
Universidad de la República
Uruguay

Director Académico y Director de Tesis:

Alberto Pardo
Instituto de Computación
Universidad de la República
Uruguay

Tesis presentada en cumplimiento con los requisitos para obtener el título de *Magister en Informática* otorgado por el Programa de Desarrollo de las Ciencias Básicas (PEDECIBA).
Ministerio de Educación y Cultura (MEC) – Universidad de la República (UDELAR)

Montevideo – Uruguay
Julio 2022

Tribunal:

Dr. Jordi Mateo-Fornés (Revisor)
Departament d'Informàtica i Enginyeria Industrial
Universitat de Lleida
España

Dr. Antonio Mauttone (Presidente)
Instituto de Computación
Universidad de la República

Dr. Daniel Calegari
Instituto de Computación
Universidad de la República

A Tabaré y Analisa

Agradecimientos

Quiero agradecer especialmente a mis tutores, Carlos Testuri y Alberto Pardo, por su apoyo y paciencia, por compartir su experiencia y conocimiento, y por la guía brindada durante el desarrollo de este trabajo.

Agradezco también a Bernardo Zimberg, con quien compartimos los proyectos entre UdelaR y ANCAP donde surgió este trabajo de tesis, por compartir su experiencia y su visión sobre como trabajar con estos problemas.

Quiero agradecer al revisor de esta tesis, Jordi Mateo-Fornés, por sus comentarios y sugerencias, y al resto del tribunal de tesis, Antonio Mauttone y Daniel Calegari, por su disponibilidad para evaluar este trabajo.

También quiero agradecer a mis compañeros de trabajo del InCo y de BCU, por su apoyo y comprensión, y por cubrir mis ausencias para poder dedicarme a la finalización del trabajo. Agradezco en particular a Marcos Viera por estar siempre dispuesto a responder mis dudas y a discutir soluciones.

Quiero agradecer a todos los que llevan adelante el Programa de Desarrollo de las Ciencias Básicas por permitir la realización de este trabajo.

Agradezco a mis familiares y seres queridos por su compañía y comprensión en el transcurso del trabajo.

Finalmente, quiero agradecer muy especialmente a Analisa y Tabaré, a quienes dedico este trabajo, por su cariñoso apoyo en todo momento.

(. . .)

que baje el puente y que se quede bajo

que entren la rabia y su ademán oscuro
que entren el mal y el bien
y lo que media
entre uno y otro
o sea
la verdad ese péndulo
que entre el incendio con o sin la lluvia
y las mujeres con o sin historia
que entre el trabajo y sobre todo el ocio
ese derecho al sueño
ese arco iris

que baje el puente y que se quede bajo

que entren los perros
los hijos de perra
las comadronas los sepultureros
los ángeles si hubiera
y si no hay
que entre la luna con su niño frío

que baje el puente y que se quede bajo

que entre el que sabe lo que no sabemos
y amasa pan
o hace revoluciones
y el que no puede hacerlas
y el que cierra los ojos

en fin
para que nadie se llame a confusiones
que entre mi prójimo ese insoportable
tan fuerte y frágil
ese necesario
ése con dudas sombra rostro sangre
y vida a término
ese bienvenido

que sólo quede afuera
el encargado
de levantar el puente

a esta altura
no ha de ser un secreto
para nadie

yo estoy contra los puentes levadizos.

"Contra los puentes levadizos", Mario Benedetti (fragmento).

Resumen

El soporte brindado por los lenguajes de modelado algebraico para trabajar con problemas de programación estocástica multietapa basada en escenarios es, en general, limitado y poco práctico. Las problemáticas incluyen excesiva verbosidad o redundancia, expresividad limitada, dificultad de integración con sistemas externos, ecosistemas cerrados, y dificultad para desarrollar extensiones. Los problemas de programación estocástica basados en escenarios suelen ser tratados como problemas de programación entera mixta a través de su forma extensiva, lo que permite utilizar todas las herramientas disponibles para ese tipo de problemas en la implementación computacional y resolución de los problemas estocásticos. Tratar a los problemas de programación estocástica como problemas enteros mixtos no es ideal. En primer lugar, exige la incorporación del control de la no anticipatividad de las decisiones en los modelos. Por otra parte, no brinda ninguna asistencia en la construcción del árbol de escenarios y la especificación de los valores de los parámetros estocásticos en forma consistente. Por último, no permite explotar la estructura particular de los problemas de programación estocástica en su resolución. En este trabajo se desarrolla un lenguaje de dominio específico (DSL), denominado `amphip`, en el que se explora una extensión al lenguaje de modelado algebraico para problemas de programación entera mixta, GNU MathProg, para incorporar soporte directo para programación estocástica multietapa basada en escenarios. El DSL es embebido en el lenguaje de programación Scala, permitiendo que los modelos puedan ser transformados y manipulados. El desarrollo es realizado utilizando íntegramente programación funcional, aprovechando el soporte provisto por Scala. El DSL desarrollado permite modelar los problemas de programación estocástica utilizando la formulación con escenarios separados. Las entidades del modelo son declaradas con una sintaxis que logra replicar con gran fidelidad la de MathProg. El DSL provee primitivas para la especificación del árbol de escenarios, las probabilidades, y los valores de los parámetros estocásticos en forma concisa y flexible. Aplicando transformaciones al modelo, se generan automáticamente las restricciones de no anticipatividad y versiones alternativas de los parámetros estocásticos sin escenarios separados, lo que permite construir una forma extensiva que es más eficiente que la se construiría manualmente. La funcionalidad provista puede ser extendida por el usuario, pudiendo definir, con pocas líneas de código, funciones para la construcción de árboles de escenarios con estructuras particulares, esquemas alternativos para definir las probabilidades de los escenarios, y funciones auxiliares para especificar los datos de los parámetros estocásticos tomando en cuenta las instancias particulares de los problemas.

Palabras clave: *programación estocástica multietapa basada en escenarios, lenguaje de dominio específico embebido, MathProg, Scala, programación funcional.*

Abstract

Support provided by algebraic modeling languages for scenario-based multistage stochastic programming is, in general, limited and impractical. Problems include excessive verbosity or redundancy, limited expressiveness, difficulties integrating with external systems, closed ecosystems, and difficulties developing extensions. Scenario-based stochastic programming problemas are usually treated as mixed integer programming problemas through their extensive form, which allows to use all the tools available for that type of problems in the computational implementation and resolution of stochastic problems. Treating stochastic programming problems as mixed integer problems is not ideal. In the first place, it requires the incorporation of control of the non-anticipativity of decisions in the models. Moreover, it does not provide any assistance in building the scenario tree and specifying the values of the stochastic parameters consistently. Finally, it does not allows to exploit the particular structure of stochastic programming problems in their resolution. In this work, a domain specific language (DSL), called `amphip`, is developed, in which an extension to the algebraic modeling language for mixed integer programming problems, GNU MathProg, is explored to incorporate direct support for scenario-based multistage stochastic programming. The DSL is embedded in the Scala programming language, allowing models to be transformed and manipulated. The development is carried out entirely using functional programming, taking advantage of the support provided by Scala. The developed DSL allows modeling stochastic programming problems using the formulation with separated scenarios. Model entities are declared with a syntax that replicates that of MathProg with great fidelity. The DSL provides primitives for specifying the scenario tree, probabilities, and stochastic parameter values in a concise and flexible way. Applying transformations to the model, the non-anticipativity constraints and alternative versions of the stochastic parameters without separated scenarios are automatically generated, which allows building an extensive form that is more efficient than the one that would be built manually. The functionality provided can be extended by the user, being able to define, with a few lines of code, functions for the construction of scenario trees with particular structures, alternative schemes to define the probabilities of the scenarios, and auxiliary functions to specify the data of the stochastic parameters taking into account the particular instances of the problems.

Keywords: *scenario-based multistage stochastic programming, embedded domain-specific language, MathProg, Scala, functional programming.*

Índice general

Capítulo 1

Introducción

La programación estocástica trata del modelado y resolución de problemas de optimización que involucran eventos aleatorios. Los modelos de programación estocástica incorporan en forma explícita la incertidumbre en sus parámetros, la que se propaga a las variables de decisión. Las soluciones en este tipo de problemas consisten en conjuntos de decisiones a tomar a medida que la incertidumbre es develada. El objetivo es encontrar una solución factible para todas las posibles realizaciones de la incertidumbre, que optimice, típicamente, el valor esperado de alguna función sobre los parámetros aleatorios y las variables de decisión. La programación estocástica es aplicada en diversas áreas como la energía, la logística, las finanzas, y la agricultura (Birge y F. Louveaux 2011; Shapiro y Philpott 2007).

En los problemas de programación estocástica los eventos aleatorios se intercalan temporalmente entre las decisiones, determinando *etapas*. Las decisiones de la primera etapa son deterministas y se toman antes de la ocurrencia de cualquier evento aleatorio, mientras que las decisiones de las etapas siguientes son aleatorias y deben tomarse luego de la ocurrencia de los eventos aleatorios de los que dependen. Las decisiones en etapas posteriores a la primera suelen denominarse *decisiones compensatorias*, porque permiten compensar las decisiones de las etapas anteriores un vez que la incertidumbre se devela.

Un caso particular y muy frecuente se da cuando los parámetros aleatorios tienen un conjunto finito de realizaciones, en cuyo caso los posibles valores que pueden tomar se denominan *escenarios*. En los modelos multietapa, los escenarios tienen una estructura arborescente. Dicha estructura se da porque las realizaciones de los eventos aleatorios están condicionadas a las realizaciones de los eventos aleatorios que los precedieron.

Al trabajar con escenarios, los modelos de programación estocástica pueden transformarse en problemas deterministas, aunque de gran tamaño, utilizando la denominada *forma extensiva*. En dicha forma se incorporan variables para decisiones compensatorias por cada posible realización de los eventos aleatorios. Como se optimizan todas las decisiones de todas las etapas en conjunto, debe tenerse especial cuidado que las decisiones en una etapa no se anticipen al futuro. En cada escenario, las variables de decisión deben tomar un único valor sopesando todos los posibles escenarios futuros descendientes. Según el tipo de formulación extensiva que se utilice, la *no anticipatividad* de las decisiones puede ser controlada por construcción, en forma implícita, o mediante el agregado de

restricciones adicionales, llamadas *restricciones de no anticipatividad*, en forma explícita.

Modelar la incertidumbre mediante un árbol de escenarios da gran flexibilidad en los tipos de decisiones que pueden representarse y en los métodos de resolución que pueden aplicarse (Birge y F. Louveaux 2011). Sin perjuicio de lo anterior, el modelado de la incertidumbre basado en escenarios tiene desafíos como el soporte limitado por parte de los *sistemas de modelado* y, en particular, por los *lenguajes de modelado algebraico* que estos disponen. Un encare usual al problema anterior consiste en aprovechar la forma extensiva del modelado basado en escenarios, y tratar a los problemas estocásticos como problemas de programación lineal o entera. Dicho encare presenta sus propios problemas. En primer lugar, el uso de la forma extensiva exige que se incorpore a los modelos el control de la no anticipatividad de las decisiones en forma implícita o explícita, lo que supone trabajo extra por parte del modelador. En segundo término, la construcción del árbol de escenarios y la especificación de los valores de los parámetros estocásticos en cada nodo del árbol es una tarea laboriosa, que puede ser simplificada con la ayuda de herramientas externas, pero que en todo caso implica desarrollos adicionales que pueden simplificarse con el soporte adecuado por parte del lenguaje. En el punto anterior es donde se da la mayor interacción entre los sistemas de modelado y los sistemas externos. Por último, al trabajar con el problema como si fuera determinista, se pierde la posibilidad de utilizar la estructura particular de los problemas de programación estocástica en su resolución mediante el uso de algoritmos especializados.

En este trabajo se explora una extensión a un lenguaje estándar para modelado algebraico de problemas de programación entera mixta, que incorpora soporte directo para programación estocástica multietapa basada en escenarios. La técnica fundamental utilizada para desarrollar la extensión es el *embedding* del lenguaje original en un lenguaje de programación de propósito general que actúa como anfitrión. Los lenguajes de modelado algebraico pueden considerarse lenguajes de dominio específico (*domain specific language*, DSL) (Fowler 2010; Gibbons 2015), siendo los dominios en este caso la programación entera mixta y la programación estocástica. El lenguaje de modelado algebraico que se extiende es *GNU MathProg* (Makhorin 2016b), que es una versión libre pero reducida de *AMPL* (Fourer et al. 2002), considerado este último un estándar de echo para problemas de optimización matemática. Como lenguaje anfitrión se selecciona *Scala* (Odersky et al. 2016), que es un lenguaje híbrido, orientado a objetos y funcional, con buen soporte para la construcción de DSL embebidos (*embedded* DSL, EDSL). La versatilidad de Scala hace que pueda ser utilizado exitosamente en una amplia variedad de dominios, donde potencialmente puede utilizarse también el EDSL propuesto.

Para la construcción del EDSL y la manipulación del lenguaje se utilizan distintas técnicas de programación funcional, evitando el estado mutable y funciones con efectos secundarios. El soporte para programación estocástica se provee como una capa sobre GNU MathProg embebido en Scala, aprovechando las capacidades del lenguaje anfitrión. El *embedding* permite la especificación del modelo de programación estocástica manejando en forma automática la no anticipatividad, y funciones para la construcción del árbol de escenarios y la especificación de los valores de los parámetros estocásticos en forma concisa y flexible. El EDSL es extensible quedando disponibles para los usuarios las mismas primitivas utilizadas internamente, no existiendo diferencias entre el código

provisto por la biblioteca con el desarrollado por terceros. Tanto el *embedding* como las funciones desarrolladas están disponibles como una biblioteca de código denominada `amphip` (Ferrari 2021).

1.1. Ejemplo

Para ilustrar algunas de las problemáticas planteadas y las soluciones propuestas, se presenta resumidamente un modelo estocástico para planificación financiera, basado en el presentado en Birge y F. Louveaux (2011).

Un inversor desea alcanzar un monto objetivo, G, durante un horizonte de planificación de tiempo discreto, $T = 1, \ldots, H$. Inicialmente, dispone de un monto base, b, y un conjunto de posibles inversiones, I. En cada período, el inversor puede cambiar los montos invertidos buscando alcanzar la meta.

En cada período $t \in T$, cada inversión $i \in I$ tiene un retorno incierto, ξ_i^t, que se asume con distribución de probabilidad conocida y espacio de probabilidades finito. En cada período $t = 1, \ldots, H-1$, se toman decisiones de inversión, variable x_i^t, con las que se invierte todo el dinero disponible. El dinero disponible en un período depende del retorno de las inversiones en los períodos anteriores, a excepción del primer período donde el dinero disponible es el monto base. El proceso de toma de decisiones de inversión en función de retornos inciertos, es un proceso estocástico que puede representarse mediante un árbol de H niveles, donde cada nivel se corresponde con un período de inversión y cada nodo con un posible escenario de realización de la incertidumbre (Figura 1.1). En dicho árbol, cada nodo en el nivel $t = 1, \ldots, H-1$ tiene tantos descendientes en el nivel $t+1$ como retornos posibles para las inversiones, y sus hojas, denominadas *escenarios finales*, determinan el conjunto total de posibles escenarios de inversión, S. Con la estructura anterior, la incertidumbre en los rendimientos de las inversiones y las decisiones que se toman una vez que se devela la incertidumbre, pueden representarse agregando el escenario como parámetro, pasando a ser $\xi_i^t(s)$ y $x_i^t(s)$.

Con las definiciones anteriores, el problema puede formularse de la siguiente manera. En el primer período se invierte completamente el monto base, b, mediante la restricción

$$\sum_{i \in I} x_i^1(s) = b, \quad s \in S.$$

Para cada escenario, en los períodos $t = 2, \ldots, H-1$, el dinero disponible para invertir es el producido por las inversiones al final del período anterior, y se invierte completamente

$$\sum_{i \in I} \xi_i^t(s) x_i^{t-1}(s) = \sum_{i \in I} x_i^t(s), \quad t = 2, \ldots, H-1, \;\; s \in S.$$

La anterior es una restricción de balance: todo el dinero que "entra" en un período debe "salir" en el siguiente. En el período final, H, el retorno de las inversiones realizadas en el período anterior constituyen el dinero disponible, el cual puede superar o estar por debajo de la meta, G. Puede suceder que en un escenario la meta sea superada mientras que en otro no se alcance. Las variables $y(s)$ y $w(s)$

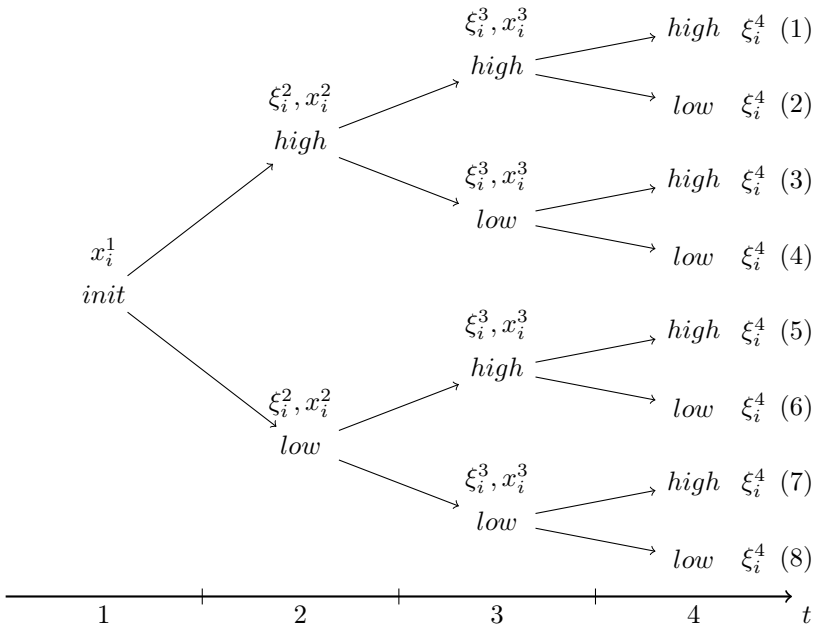


Figura 1.1: Árbol de escenarios con $H = 4$ y dos posibles rendimientos por etapa, $\{low, high\}$, lo que determina ocho escenarios finales

registran el exceso o faltante de dinero respecto al objetivo por cada escenario en el período final

$$\sum_{i \in I} \xi_i^H(s) x_i^{H-1}(s) - y(s) + w(s) = G, \quad s \in S.$$

Como función objetivo puede plantearse maximizar la utilidad esperada incorporando ponderadores, q y r, que representan la ganancia por superar la meta y el costo por no alcanzarla, respectivamente. Para cada escenario $s \in S$, $\pi(s)$ es la probabilidad de ocurrencia de dicho escenario. La función objetivo está definida por

$$\max \sum_{s \in S} \pi(s)(qy(s) - rw(s)).$$

Como se comentó anteriormente, las decisiones que se toman en un período no pueden cambiar según la realización específica de la incertidumbre en los períodos siguientes. Violar la condición anterior equivaldría a que las decisiones pudieran "anticiparse al futuro", adivinando lo que sucederá antes que suceda. Con la formulación planteada anteriormente, llamada con *escenarios separados*, no se restringe el "anticiparse al futuro" ya que para un mismo nodo del árbol las variables de decisión pueden tener distintos valores para cada posible escenario final. Por ejemplo, si se toma como referencia el árbol de la Figura 1.1, las variables $x_i^1(s), s = 1, \ldots, 8$ deberían tener todas el mismo valor, lo mismo sucede en el segundo período con $x_i^2(s'), s' = 1, \ldots, 4$ y $x_i^2(s''), s'' = 5, \ldots, 8$, y similarmente los cuatro "grupos" de decisiones del tercer período. Al trabajar con escenarios separados se deben incorporar restricciones de no anticipatividad explícitas para cada variable estocástica. Para definir dichas restricciones se requiere información

sobre la estructura del árbol de escenarios. Dado un escenario final s y un período t, $anc(s,t)$ denota al escenario "ancestro" de s en el nivel t del árbol de escenarios. Si dos escenarios finales s y s' identifican al mismo nodo del árbol en el período t, se debe cumplir que $anc(s,t) = anc(s',t)$. Con las definiciones anteriores, las restricciones de no anticipatividad se pueden expresar como

$$x_i^t(s) = x_i^t(s'), \quad t = 1,\ldots,H-1, \;\; i \in I, \;\; s,s' \in S, \;\; anc(s,t) = anc(s',t).$$

Para finalizar con el ejemplo, se considera una instancia de datos con cuatro períodos, $H = 4$, y un árbol de escenarios perfecto con aridad 2, equivalente al árbol de la Figura 1.1. Se toma monto objetivo $G = 80000$, monto base $b = 55000$, dos posibles inversiones $I = \{stock, bonds\}$, la ganancia por superar el objetivo $q = 1$, y el costo de no alcanzarlo $r = 4$. Se asume que los dos casos en que se abre cada escenario por período representan un escenario de "alto rendimiento" ($high$) y otro de "bajo rendimiento" (low) con rendimientos $\xi_{stock}^t(\ldots,high) = 1{,}25$, $\xi_{bonds}^t(\ldots,high) = 1{,}14$, $\xi_{stock}^t(\ldots,low) = 1{,}06$, $\xi_{bonds}^t(\ldots,low) = 1{,}12$, independientemente del período, t. Los escenarios finales se consideran equiprobables, es decir, $\pi(s) = 1/8$, para $s \in S$.

Este problema así planteado puede verse como un problema de programación entera mixta y como tal puede representarse en un lenguaje de modelado algebraico para ese tipo de problemas, como lo es MathProg.

```
s.t. budget{s in S}: sum{i in I} x[1,s,i] = b;

s.t. balance{t in 2..H-1, s in S}:
  sum{i in I} xi[t,s,i] * x[t-1,s,i] = sum{i in I} x[t,s,i];

s.t. goal{s in S}:
  sum{i in I} xi[H,s,i] * x[H-1,s,i] - y[s] + w[s] = G;

maximize utility: sum{s in S} pi[s] * (q * y[s] - r * w[s]);

s.t. na_x{
      t in T, s1 in S, s2 in S, i in I :
        t < H and anc[s1, t] == anc[s2, t]
    }: x[t,s1,i] = x[t,s2,i];
```

La instancia de datos también puede representarse en MathProg como se muestra a continuación.

```
param H := 4;
set I := stock bonds;

param G := 80000;
param b := 55000;

param q := 1;
param r := 4;

set S := 1 2 3 4 5 6 7 8;
param pi := 1 0.125, 2 0.125, 3 0.125, 4 0.125,
            5 0.125, 6 0.125, 7 0.125, 8 0.125;

param anc :
     1 2 3 4 :=
  1  1 1 1 1
  2  1 1 1 2
  3  1 1 2 3
  4  1 1 2 4
  5  1 2 3 5
  6  1 2 3 6
  7  1 2 4 7
  8  1 2 4 8 ;

param xi :=
[*,*,stock]:
        1     2     3     4     5     6     7     8 :=
  2  1.25  1.25  1.25  1.25  1.06  1.06  1.06  1.06
  3  1.25  1.25  1.06  1.06  1.25  1.25  1.06  1.06
  4  1.25  1.06  1.25  1.06  1.25  1.06  1.25  1.06

[*,*,bonds]:
        1     2     3     4     5     6     7     8 :=
  2  1.14  1.14  1.14  1.14  1.12  1.12  1.12  1.12
  3  1.14  1.14  1.12  1.12  1.14  1.14  1.12  1.12
  4  1.14  1.12  1.14  1.12  1.14  1.12  1.14  1.12 ;
```

Un problema que se evidencia al generar la instancia de datos en MathProg es que hay redundancia en los datos de los retornos de inversión, ξ, producto de utilizar la formulación con escenarios separados. Por otra parte, la estructura del árbol de escenarios está implícita en los datos. La consistencia entre la estructura del árbol de escenarios y los datos de los parámetros estocásticos debe mantenerse en forma manual o mediante un sistema externo, ya que no es chequeada por los sistemas de modelado porque ignoran que se trata de un problema de programación estocástica.

El mismo problema puede ser representado con el EDSL desarrollado en esta tesis, utilizando una sintaxis muy similar a la de MathProg pero dentro de Scala. Con el EDSL desarrollado no se requiere la introducción de las restricciones de no anticipatividad porque puede declararse que se trata de un problema de programación estocástica, indicando los conjuntos con los períodos y los escenarios finales, y el parámetro con las probabilidades de los escenarios.

```
val budget =
  st(s in S) {
    sum(i in I) { x(1,s,i) } === b
  }

val balance =
  st(t in (2 to H-1), s in S) {
    sum(i in I) { xi(t,s,i) * x(t-1,s,i) } ===
    sum(i in I) { x(t,s,i) }
  }

val goal =
  st(s in S) {
    sum(i in I) { xi(H,s,i) * x(H-1,s,i) } - y(s) + w(s) === G
  }

val utility =
  maximize {
    sum(s in S) { pi(s) * (q * y(s) - r * w(s)) }
  }

val stochModel =
  model(utility,
    budget, balance, goal
  ).stochastic(T, S, pi)
```

La instancia de datos puede especificarse también con el EDSL desarrollado. Para la especificación de los datos no se intenta simular la sintaxis de MathProg entendiendo que es mejor aprovechar las capacidades provistas directamente por Scala para este fin. El EDSL provee primitivas para la definición de árboles de escenarios con estructura arbitraria. En esta instancia particular de datos, el árbol de escenarios es un árbol binario perfecto con escenarios equiprobables, y los retornos de inversión no dependen del período. Utilizando las primitivas del EDSL, el árbol de escenarios puede definirse especificando las dos posibles alternativas en cada etapa junto con su probabilidad, y los valores de los retornos de inversión en cada alternativa por etapa.

```
val (stock, bonds) = ("stock", "bonds")

val stochModelDetData = stochModel
  .setData(I, List(stock, bonds))
  .paramData(G, 80000)
  .paramData(b, 55000)
  .paramData(q, 1)
  .paramData(r, 4)

val (t1, t2, t3, t4) = (Stage("1"), Stage("2"), Stage("3"), Stage("4"))
val (init, high, low) =
  (BasicScenario("init"), BasicScenario("high"), BasicScenario("low"))

val stochModelStages = stochModelDetData.stochStages(t1, t2, t3, t4)
val stochModelBS = stochModelStages
  .stochBasicScenarios(t1, init -> r"1")
  .stochBasicScenarios(t2, high -> r"1/2", low -> r"1/2")
  .stochBasicScenarios(t3, high -> r"1/2", low -> r"1/2")
  .stochBasicScenarios(t4, high -> r"1/2", low -> r"1/2")

val stochModelBSData = stochModelBS
  .stochBasicData(xi, t2, high, stock -> 1.25, bonds -> 1.14)
  .stochBasicData(xi, t2, low , stock -> 1.06, bonds -> 1.12)
  .stochBasicData(xi, t3, high, stock -> 1.25, bonds -> 1.14)
  .stochBasicData(xi, t3, low , stock -> 1.06, bonds -> 1.12)
  .stochBasicData(xi, t4, high, stock -> 1.25, bonds -> 1.14)
  .stochBasicData(xi, t4, low , stock -> 1.06, bonds -> 1.12)
```

El hecho de que el DSL sea embebido permite realizar otras computaciones no incluidas en el mismo, además de facilitar la interacción con el resto de la lógica del negocio. Utilizando la capacidad de abstracción del lenguaje anfitrión, es posible definir funciones para la construcción de árboles perfectos n-arios y para la especificación de valores de parámetros que no dependen de la etapa, y utilizarlas para definir `stochModelBS` y `stochModelBSData`, de forma de evitar la repetición existente por la instancia de datos utilizada.

```
val stochModelBS =
  stochPerfectTree(stochModelStages, init, List(high, low))

val stochModelBSData =
  stochStagesData(stochModelBS, xi, List(t2,t3,t4),
    high -> List(stock -> 1.25, bonds -> 1.14),
    low  -> List(stock -> 1.06, bonds -> 1.12)
  )
```

Con la información disponible, el EDSL desarrollado puede generar la forma extensiva del problema automáticamente, y construir un modelo MathProg igual o más eficiente que el que se escribiría manualmente en estos casos.

1.2. Contexto del trabajo

Este trabajo se desarrolló en el contexto de tres proyectos de colaboración entre la UdelaR y la Administración Nacional de Combustibles Alcohol y Portland (ANCAP) en los que se participó entre 2009 y 2016[1]. En los tres proyectos el

[1]Los proyectos fueron: *Modelo estocástico múltiple-etapa para apoyo a la toma de decisiones en la planificación de la producción* (proyecto CSIC 2008), *Modelo estocástico múltiple etapa*

producto principal fue un modelo de programación estocástica multietapa basado en escenarios para asistir en la toma de decisiones de provisión de combustible. Como resultado de los proyectos se produjeron dos publicaciones (Testuri et al. 2012; Zimberg et al. 2019). El EDSL propuesto en este trabajo toma en cuenta la experiencia obtenida y las dificultades encontradas al trabajar con modelos de programación estocástica de gran tamaño en los proyectos mencionados.

1.3. Organización del documento

El documento se encuentra organizado de la siguiente manera. En el Capítulo 2 se presenta el lenguaje MathProg y se detallan las técnicas utilizadas para embeberlo en el lenguaje de propósito general Scala. En el Capítulo 3 se describen con mayor detenimiento los principales conceptos de la programación estocástica, particularmente para el caso multietapa y basada en escenarios, y se detalla el soporte desarrollado en el EDSL propuesto. En el Capítulo 4 se describen otros DSL con soporte para programación estocástica multietapa basada en escenarios. Finalmente, en el Capítulo 5 se presentan las conclusiones y el trabajo a futuro.

*para la toma de decisiones en la provisión de fuel oil para generación térmica*s (convenio ANCAP-UdelaR 2011), y *Modelo estocástico múltiple etapa para la toma de decisiones en la provisión de combustibles para generación eléctrica* (proyecto FSE-ANII 2013).

Capítulo 2

Embedding de MathProg en Scala

GNU MathProg es un *lenguaje de modelado algebraico* (*algebraic modeling language*, AML) que forma parte del paquete de software libre de optimización *GNU Linear Programming Kit* (GLPK) (Makhorin 2016a; Makhorin 2016b). GLPK incluye un intérprete para el lenguaje GNU MathProg y un *solver* para problemas de programación entera mixta. MathProg es una versión libre pero reducida del lenguaje de modelado algebraico AMPL (Fourer et al. 2002), considerado un estándar de echo para problemas de optimización matemática. Los problemas que pueden representarse en MathProg están restringidos a problemas de optimización lineales con variables enteras o reales. Las capacidades de *scripting* de las que dispone MathProg están restringidas al despliegue de datos, por lo que cualquier tipo de algoritmia o manipulación programática del modelo debe realizarse por fuera del lenguaje.

El lenguaje MathProg puede entenderse como un lenguaje de dominio específico. Los lenguajes de dominio específico (*domain specific language*, DSL) son lenguajes de programación orientados a una tarea o dominio particular (Fowler 2010; Gibbons 2015). Según la definición anterior, los lenguajes de modelado algebraico son lenguajes de dominio específico para problemas de optimización matemática. En el caso de MathProg, se trata de un DSL *externo*. En los DSL externos, los modelos se escriben en forma independiente, disponiendo de un compilador o interprete propio. En contraposición con los DSL externos, los DSL *embebidos* (*embedded* DSL, EDSL) se codifican dentro de un lenguaje de propósito general que actúa como *anfitrión*, y del que heredan el sistema de tipos, el sistema de módulos y los tipos de datos básicos. Los EDSL pueden integrarse fácilmente en la lógica del negocio de las aplicaciones, y permiten que el usuario haga uso de las capacidades del lenguaje anfitrión, por ejemplo, para abstraer código repetido en funciones, acceder a bibliotecas de su ecosistema, o a herramientas asociadas como los entornos de desarrollo integrados.

Los EDSL pueden ser clasificados en dos categorías: *profundos* (*deep*) o *llanos* (*shallow*) (Gibbons 2015). En los *deep embeddings* las operaciones del lenguaje embebido son representadas como constructores del árbol de sintaxis abstracto del lenguaje (*abstract syntax tree*, AST), el cual puede ser luego recorrido para ser evaluado o transformado. En el caso de los *shallow embeddings* las expresiones

del lenguaje embebido son implementadas directamente mediante su semántica, sin pasar por la representación en términos de un AST.

El lenguaje de programación Scala es un lenguaje de propósito general que integra características de la programación funcional y la orientada a objetos (Odersky et al. 2016). Scala dispone de un conjunto de características que lo hacen particularmente apto para la construcción de EDSL como son una sintaxis flexible, la posibilidad de sobrecargar operadores, funciones de alto orden y los *implicits* que pueden utilizarse para implementar *type classes* entre otros usos (Oliveira et al. 2010). Los programas Scala son compilados por defecto a la máquina virtual de Java (Lindholm et al. 2018), pero pueden ser compilados también a JavaScript y a código nativo utilizando *plugins* del compilador (Doeraene 2018; Shabalin 2020). La versatilidad de Scala permite que sea utilizado en una amplia variedad de dominios como los sistemas distribuidos, desarrollo web y de *front-end*, procesamiento de grandes volúmenes de datos, ciencia de datos, y programación de sistemas.

En este capítulo se describe un *deep embedding* de MathProg en Scala 2.12. Como se mencionó en el Capítulo 1, el objetivo es soportar el trabajo con modelos de programación estocástica. Tener el lenguaje embebido hace que sea sencillo realizar transformaciones a los modelos y permite agregar funcionalidad extra, por ejemplo, por medio de capas de software. El código completo del EDSL se encuentra disponible en una biblioteca denominada `amphip` (Ferrari 2021).

2.1. Componentes de un modelo MathProg[1]

MathProg es un lenguaje de modelado para problemas de programación entera mixta (*mixed integer programming*, MIP) (Nemhauser y Wolsey 1988). Los problemas MIP pueden ser declarados en forma genérica de la siguiente manera:

minimizar (o maximizar) la función lineal

$$z = c_1 x_1 + c_2 x_2 + \ldots + c_n x_n + c_0, \tag{2.1}$$

sujeto a restricciones lineales

$$\begin{aligned} L_1 \;&\leq\; a_{11}x_1 + \;a_{12}x_2 + \ldots + \;a_{1n}x_n \leq U_1, \\ L_2 \;&\leq\; a_{21}x_1 + \;a_{22}x_2 + \ldots + \;a_{2n}x_n \leq U_2, \\ &\vdots \\ L_m &\leq a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n \leq U_m, \end{aligned} \tag{2.2}$$

y cotas para las variables

$$\begin{aligned} l_1 \;&\leq\; x_1 \leq u_1, \\ l_2 \;&\leq\; x_2 \leq u_2, \\ &\vdots \\ l_n &\leq x_n \leq u_n, \end{aligned} \tag{2.3}$$

donde, dados $i = 1, \ldots, n$ y $j = 1, \ldots, m$, x_i son las variables, z es la función objetivo, c_i y c_0 son los coeficientes y término constante de la función objetivo,

[1] Esta sección se basa fuertemente en la referencia del lenguaje GNU MathProg (Makhorin 2016b). En el Apéndice A se incluye una especificación formal de la gramática de las sentencias de GNU MathProg que son consideradas, la cual fue elaborada para este trabajo.

a_{ij} son los coeficientes de las restricciones, L_j y U_j son las cotas inferiores y superiores de las restricciones, y l_i y u_i son las cotas inferiores y superiores de las variables. Las variables, x_i, pueden ser reales, enteras o binarias. Los coeficientes, c_0, c_i, a_{ij}, y las cotas, l_i, u_i, L_j, U_j, son todos reales.

Las cotas de las variables y restricciones pueden ser finitas o infinitas, aunque en las restricciones alguna de las dos cotas debe ser finita. Cuando en una variable ambas cotas son infinitas, se dice que la variable es *libre*. Por otra parte, si ambas cotas de una variable coinciden ($l_i = u_i$) se dice que la variable está *fijada*.

Los modelos descritos en MathProg están compuestos por una serie de sentencias que se dividen en dos categorías: *sentencias de declaración* y *sentencias funcionales*. Las sentencias de declaración, o simplemente declaraciones, establecen las entidades del modelo (conjuntos, parámetros, y variables), las restricciones, y la función objetivo a maximizar o minimizar. Dichos componentes son referidos como *objetos del modelo* en la referencia del lenguaje (Makhorin 2016b). Las sentencias funcionales están vinculadas con la entrada/salida, el chequeo de datos, y la invocación del *solver*.

MathProg también permite escribir *bloques de datos* para especificar los valores de los parámetros y conjuntos. Estos bloques de datos pueden incluirse en una sección especial dentro de la descripción del modelo o en un archivo separado. Dicha separación entre la *sección de modelo* y la *sección de datos* ayuda a plantear el problema es forma más abstracta y permite que un mismo modelo pueda ser resuelto con distintas instancias de datos sin necesidad de modificarlo.

El *embedding* planteado en esta tesis abarca únicamente a las sentencias de declaración, que son las utilizadas para representar a los objetos del modelo. Para la especificación de los valores de los parámetros se utiliza una notación diferente a la provista por MathProg, considerando que el lenguaje anfitrión, Scala, tiene capacidades superiores para este fin (Sección 2.3).

Los objetos del modelo tienen un *nombre* que los identifica y que se utiliza para hacer referencia a ellos, y pueden ser opcionalmente indexados sobre conjuntos constituyendo arreglos multidimensionales de objetos. En el caso de ser arreglos multidimensionales, cada objeto individual se accede mediante *subíndices* que se escriben entre paréntesis rectos. Por ejemplo, si un parámetro a se indexa sobre los conjuntos I y J, cada parámetro elemental se referencia escribiendo $a[i, j]$, donde $i \in I$ y $j \in J$.

Las declaraciones de los objetos del modelo comienzan por una palabra reservada que indica el tipo de declaración, seguidas del nombre, un alias opcional, y un *dominio* opcional –que especifica si se trata de un objeto indexado o no–. Las entidades del modelo se declaran con las palabras reservadas `set`, `param` y `var`, y pueden tener adicionalmente una lista de *atributos* que sirven, por ejemplo, para asignarle un valor directamente en el modelo o establecer una cota. Las declaraciones siguientes son ejemplos de declaraciones de entidades:

Conjuntos:

```
set A;
set T := 1..H;
set C := A union P;
```

A
$T := 1, \ldots, H$
$C := A \cup P$

Parámetros:

```
param H;
param d{T};
param tau{A} in T;
param a{t in T} :=
  sum{c in A : tau[c] == t} q[c];
```

H
$d^t, \ t \in T$
$\tau_c, \ c \in A$
$a^t := \sum_{c \in A | \tau_c = t} q_c, \ t \in T$

Variables:

```
var y{T} >= 0;
var x{
  c in A,
  t in 1 .. tau[c] - 1
} binary;
var u{T} >= 0;
```

$y^t \geq 0, \ t \in T$
$x_c^t \in \{0,1\}, \ c \in A, \ t = 1, \ldots, \tau_c - 1$
$u^t \geq 0$

Las restricciones pueden declararse con las palabras reservadas `subject to`, `subj to` o `s.t.`. La definición de una restricción puede tomar básicamente dos formas: puede ser *simple* e involucrar solo dos expresiones relacionadas, o puede ser *doble* e involucrar tres expresiones relacionadas mediante una doble desigualdad. En el caso de ser simple, ambas expresiones pueden ser *expresiones lineales*, y, en el caso de ser dobles, la expresión del medio puede ser una expresión lineal y las expresiones de los extremos deben ser *expresiones numéricas*. Los siguientes son ejemplos de declaraciones de restricciones:

```
s.t. balance0:
  y0 + a[1] + u[1] =
    d[1] + w[1] + y[1];
```

$y^0 + a^1 + u^1 = d^1 + w^1 + y^1$

```
s.t. inventory{t in T}:
  ymin <= y[t] <= ymax;
```

$\underline{y} \leq y^t \leq \bar{y}, \ t \in T$

```
s.t. single_cancellation{c in A}:
  sum{t in 1 .. tau[c] - 1}
    x[c,t] <= 1;
```

$\sum_{t=1}^{\tau_c - 1} x_c^t \leq 1, \ c \in A$

La función objetivo se declara con las palabras reservadas `minimize` o `maximize`. La función a minimizar o maximizar es siempre una expresión lineal. En caso que un modelo tenga más de una declaración de función objetivo, se considera solo la primera. El siguiente es un ejemplo de declaración de función objetivo:

```
minimize cost:
sum{t in T} (
    sum{c in P : t <= H - gamma[c]}
      ca[c]  * q[c] * v[c,t]
  + sum{c in A : t <= tau[c] - 1}
      (cc[c] - ca[c]) * q[c] *
      x[c,t]
  + h[t] * y[t]
);
```

$$\min \sum_{t \in T} \Bigg[\sum_{c \in P | t \leq H - \gamma_c} ca_c q_c v_c^t + \sum_{c \in A | t \leq \tau_c - 1} (cc_c - ca_c) q_c x_c^t + h^t y^t \Bigg]$$

El dominio de cada una de las declaraciones anteriores, se representa mediante una *expresión de indexación* que define los conjuntos sobre los que están indexados los objetos. Una expresión de indexación está compuesta por una serie de *entradas* asociadas a conjuntos mediante *expresiones de conjuntos*. Opcionalmente, la expresión de indexación puede tener un *predicado* constituido por una *expresión lógica* que restringe las posibles tuplas que pueden utilizarse para indexar el objeto.

Cada entrada en la expresión de indexación puede introducir tuplas de *indices*, llamados *dummy indices* en la referencia del lenguaje, que sirven para hacer referencia a cada elemento individual de los conjuntos y son locales a la declaración. Dichos índices pueden ser utilizados, por ejemplo, en las expresiones de conjuntos en las entradas posteriores a su introducción, en el predicado de la expresión de indexación, en los atributos de las entidades o en las expresiones que definen las restricciones y función objetivo del problema. Por ejemplo, la expresión de indexación `{c in A, t in T : t <= tau[c] - 1}`, tiene dos entradas asociadas a los conjuntos A y T, e introduce un índice en cada una, c y t. Dicha expresión de indexación tiene además un predicado, por lo que solo se consideran como válidos los valores $c \in A$ y $t \in T$ que cumplen que $t \leq \tau_c - 1$. La expresión de indexación de la restricción `inventory{t in T}: ymin <= y[t] <= ymax`, tiene una única entrada asociada al conjunto T, e introduce un índice t para poder hacer referencia a cada valor individual del conjunto en la expresión y^t en su definición.

Los diferentes atributos que pueden tener las entidades del modelo permiten especificar su tipo (`integer`, `binary` o `symbolic` para parámetros, e `integer` o `binary` para variables) y otras restricciones como la pertenencia o inclusión en un conjunto (`in`, `within`), cotas para las variables (<=, >=) y relaciones para los parámetros (<, <=, =, ==, >=, >, <>, !=). En el caso de los conjuntos es posible especificar el tamaño que deben tener las tuplas que lo componen con el atributo `dimen`. Tanto para parámetros como para conjuntos es posible especificar su valor por defecto con el atributo `default`, o asignarles directamente un valor en el modelo con el atributo `:=`. En el caso de las variables es posible fijar su valor con el atributo =. Se observa que las expresiones en los atributos pueden hacer referencia a la propia entidad que se está declarando, admitiendo así definiciones recursivas. A continuación se muestran ejemplos adicionales de declaraciones de entidades con distintos tipos de atributos.

```
set arcs within nodes cross nodes;
set step{s in 1..maxiter} dimen 2, default arcs;

param N integer, >= 0, <= 100, default 20;
param month symbolic, in {"Mar", "Apr", "May"}, default "May";

var make{p in prd} integer, >= commit[p], <= market[p];
var z{i in I, j in J} >= i+j;
```

En MathProg existen cinco tipos de expresiones: numéricas, simbólicas, de conjuntos, lógicas, y lineales, siendo las expresiones de indexación mencionadas anteriormente un caso particular de expresiones de conjuntos. Las expresiones de conjuntos evalúan a secuencias de tuplas y las expresiones lineales a funciones lineales sobre las variables que las componen. Los otros tipos de expresiones tienen un comportamiento habitual.

Las expresiones numéricas están constituidas por los literales numéricos, índices de conjuntos, referencias a parámetros, llamadas a funciones predefinidas, *expresiones numéricas iteradas*, expresiones numéricas condicionales, y pueden ser combinadas mediante una serie de operadores aritméticos.

`time`	(parámetro)
`a["May␣2003",j+1]`	(parámetro con subíndices)
`abs(b[i,j])`	(función predefinida)
`sum{i in S diff T} alpha[i] * b[i,j]`	(expresión iterada)
`if i in I then 2 * p else q[i+1]`	(expresión condicional)
`(b[i,j] + 0.5 * c)`	(expresión parentizada)

Las expresiones simbólicas están constituidas por literales de texto, índices de conjuntos, referencias a parámetros, llamadas a funciones predefinidas, expresiones simbólicas condicionales y pueden ser combinadas mediante el operador de concatenación.

`substr(name[i],k+1,3)`	(función predefinida)
`"abc[" & i & "," & j & "]"`	(concatenación)
`if i in I then s[i,j] & "..." else t[i+1]`	(expresión condicional)

Las expresiones de conjuntos están constituidas por literales de conjuntos, referencias a conjuntos, *conjuntos aritméticos*, expresiones de indexación, *expresiones de conjuntos iteradas*, expresiones de conjuntos condicionales, y pueden ser combinadas mediante una serie de operadores de conjuntos.

`{(123,"aaa"), (i+1,"bbb"), (j-1,"ccc")}`	(literal de conjuntos)
`I`	(conjunto)
`S[i-1,j+1]`	(conjunto con subíndices)
`1 .. t-1 by 2`	(conjunto "aritmético")
`{t in 1..T, (t+1,j) in S : (t,j) in F}`	(expresión de indexación)
`setof{i in I, j in J} (i+1,j-1)`	(expresión iterada)
`if i < j then S[i,j] else F diff S[i,j]`	(expresión condicional)
`(A union B) inter (I cross J)`	(operadores de conjuntos)

Los conjuntos aritméticos representan secuencias de valores numéricos entre dos extremos con un "paso de incremento" opcional. Tienen la forma t_0 `..` t_1 `by` δt,

siendo t_0, t_1 y δt expresiones numéricas. En caso que el paso de incremento sea omitido se asume igual a 1.

Las expresiones lógicas están constituidas por las expresiones numéricas (cero se considera *falso*, otro valor numérico se considera *verdadero*), expresiones relacionales, *expresiones lógicas iteradas*, y pueden ser combinadas mediante operadores lógicos.

```
a[i,j] < 1.5                                  (expresión relacional)
s[i+1,j-1] <> "Mar" & year                    (expresión relacional)
(i+1,"Jan") not in I cross J                  (expresión relacional)
S union T within A[i] inter B[j]              (expresión relacional)
forall{i in I, j in J} a[i,j] < 0.5 * b[i]    (expresión iterada)
(a[i,j] < 1.5 or b[i] >= a[i,j]) and i in S   (operadores lógicos)
```

Las expresiones lineales están constituidas por referencias a variables, *expresiones lineales iteradas*, expresiones lineales condicionales, y pueden ser combinadas mediante algunos operadores aritméticos.

```
z                                             (variable)
x[i,j]                                        (variable con subíndices)
sum{j in J} (a[i,j] * x[i,j] + 3 * y[i-1])    (expresión iterada)
if i in I then x[i,j] else 1.5 * z + 3.25     (expresión condicional)
```

Las expresiones lineales pueden combinarse libremente con los operadores de suma y resta, pero solamente es posible utilizar los operadores de producto y división si alguno de los argumentos es numérico, ya que en otro caso la expresión resultante no sería lineal.

Las expresiones iteradas permiten calcular valores operando sobre conjuntos definidos a partir de expresiones de indexación. Tienen tres componentes: el *operador iterado* (`sum`, `prod`, `min`, `max`, `setof`, `forall`, `exists`), una expresión de indexación, y un *integrando* que es una expresión que puede depender de los índices introducidos en la expresión de indexación. El tipo de la expresión iterada depende del tipo del integrando y del operador iterado utilizado, pudiendo ser numérica, de conjuntos, lógica o lineal.

Por último, MathProg dispone de un conjunto de funciones predefinidas que pueden utilizarse como parte de las expresiones en la medida que el tipo de retorno coincida. El Apéndice A incluye gramática completa de las sentencias de declaración de MathProg, la que contiene la lista completa de operadores y funciones predefinidas.

2.2. Codificación del modelo en el EDSL

En esta sección se describe la codificación de los modelos MathProg en Scala. Por más detalles sobre el lenguaje Scala, en el Apéndice B se incluye una referencia rápida basada en el libro *Programming in Scala: A Comprehensive Step-by-Step Guide* (Odersky et al. 2019) y la referencia del lenguaje (Odersky et al. 2016).

2.2.1. Codificación del AST

Para representar el árbol de sintaxis abstracta (*abstract syntax tree*, AST) de MathProg se utilizan jerarquías de clases "selladas" (*sealed*), las que habilitan que el compilador de Scala realice chequeos de exhaustividad en los *pattern matchings*. Los tipos básicos se declaran con *traits* de Scala mientras que para los constructores se utilizan *case classes*. En el Apéndice C se lista el AST completo utilizado para el *deep embedding*.

El soporte para "herencia múltiple" de los *traits* mediante *mixins* es útil para representar la relación entre las expresiones numéricas y los otros tipos de expresiones. Las expresiones numéricas pueden ser utilizadas como expresiones simbólicas, lógicas o lineales. En el caso de las expresiones simbólicas, el valor numérico se convierte a texto utilizando un formato predefinido. En el caso de las expresiones lógicas, se considera que las expresiones numéricas son falsas si son cero y positivas en otro caso. Las expresiones numéricas se consideran expresiones lineales constantes. Por otra parte, múltiples construcciones del lenguaje soportan tanto expresiones numéricas como simbólicas, por lo que resulta de utilidad definir un tipo de expresión auxiliar, denominado *expresión simple*, para admitir cualquiera de los dos tipos de expresión sin aplicar conversiones de tipo. A continuación se muestra la jerarquía de clases sellada utilizada para representar los tipos de expresiones de MathProg. La Figura 2.1 muestra las relaciones de subtipo entre las expresiones.

```
sealed trait Expr
sealed trait SetExpr    extends Expr
sealed trait SimpleExpr extends Expr
sealed trait LogicExpr  extends Expr
sealed trait LinExpr    extends Expr
sealed trait SymExpr    extends SimpleExpr
sealed trait NumExpr    extends SimpleExpr with LogicExpr with LinExpr
```

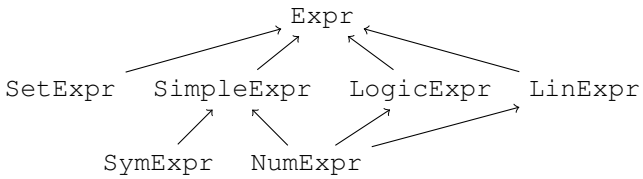


Figura 2.1: Relaciones de subtipo entre distintos tipos de expresiones en el AST de MathProg en Scala

Las *case classes* son adecuadas para la definición de estructuras de datos inmutables, que es la opción tomada para el AST. El código siguiente muestra la porción del AST correspondiente al "modelo completo" y los objetos del modelo (entidades, restricciones y función objetivo):

```
case class Model(statements: List[Stat])
sealed trait Stat
case class SetStat(
  name  : SymName,
  alias : Option[StringLit],
  domain: Option[IndExpr],
  atts  : List[SetAtt])     extends Stat
case class ParamStat(
  name  : SymName,
  alias : Option[StringLit],
  domain: Option[IndExpr],
  atts  : List[ParamAtt])   extends Stat
case class VarStat(
  name  : SymName,
  alias : Option[StringLit],
  domain: Option[IndExpr],
  atts  : List[VarAtt])     extends Stat
sealed trait ConstraintStat extends Stat
case class EqConstraintStat(/* ... */)  extends ConstraintStat
case class LTEConstraintStat(/* ... */) extends ConstraintStat
// ...
sealed trait ObjectiveStat extends Stat
case class Minimize(/* ... */) extends ObjectiveStat
case class Maximize(/* ... */) extends ObjectiveStat
```

Una vez definido el AST como una jerarquía de clases sellada, las funciones que procesan el AST pueden implementarse como recorridas sobre el árbol, utilizado "análisis de casos" aprovechando el vínculo entre los *case classes* y el *pattern matching* en Scala. Como ejemplo, se muestra a continuación parte de la definición de la instancia de la *type class* `Show` para las expresiones numéricas (Sección 2.6). La función `shows` toma como argumento una función que convierte un tipo `A` en un `String`, y devuelve la instancia de `Show[A]` correspondiente. Está especificada como un análisis de casos sobre los posibles subtipos de `NumExpr`.

```
implicit val NumExprShow: Show[NumExpr] = shows {
  case NumLit(num)          => num.shows
  case NumUnaryMinus(x)     => s"(-${x.shows})"
  case NumAdd(left, right)  => s"(${left.shows} + ${right.shows})"
  case NumMult(left, right) => s"(${left.shows} * ${right.shows})"
  case NumSum(indexing, integrand) =>
    s"(sum${indexing.shows} ${integrand.shows})"
  case CondNumExpr(test, ifTrue, otherwise) =>
    val otherwiseShows =
      otherwise.fold(")")(expr => s" else ${expr.shows})")
    s"(if ${test.shows} then ${ifTrue.shows}$otherwiseShows"

  // ...

}
```

En los lenguajes orientados a objetos las funciones asociadas a un tipo se suelen incorporar como métodos dentro de la clase que define el tipo. Los objetos integran *estado* y *comportamiento*. Para el desarrollo del *embedding*, se decide apartarse de la forma de trabajo habitual con lenguajes orientados a objetos, y definir los datos y las funciones por separado. Salvo escasas excepciones, las clases definidas para el AST no incluyen ningún método, representando únicamente datos que están determinados por los argumentos de su constructor. Las funciones

para operar con las distintas expresiones representadas con el AST se definen por separado en una "capa de sintaxis". Las funciones que se definen por separado actúan como *extension methods* del AST, lo que permite usarlos como si fueran métodos definidos en las clases. En la Sección 2.2.4 se fundamenta y se explican con más detalle los mecanismos utilizados.

2.2.2. Declaración de objetos del modelo

Para simular la sintaxis de las declaraciones de objetos del modelo, se utilizan métodos y funciones de Scala que actúan como *smart constructors* y permiten construir porciones del AST en forma más amigable. Para la construcción de las entidades del modelo se tienen tres *smart constructors* básicos: `set`, `param` y `xvar`, este último llamado así para no colisionar con `var` que es una palabra reservada en Scala. A continuación se muestran algunas declaraciones construidas con el EDSL y su equivalente en MathProg.

```
val T  = set := 1 to H              set T := 1..H;
val y0 = param                      param y0;
val v  = xvar(P, T).binary          var v{P, T} binary;
val u  = xvar(T) >= 0               var u{T} >= 0;
```

Se observa que en las llamadas a los *smart constructors* anteriores no es necesario especificar el nombre de los objetos del modelo que se están creando, los mismos son tomados del nombre de la variable ligada al resultado. Por ejemplo, en la declaración `val y0 = param`, la instancia de `ParamStat` que se está creando tiene como valor de `name` a la cadena `"y0"`. Para obtener el nombre se utiliza la biblioteca `sourcecode` (Haoyi 2017), la cual permite acceder en tiempo de compilación a información del código fuente de una expresión. Para dar dicha funcionalidad, la biblioteca se apoya fundamentalmente en dos características del lenguaje: *parámetros implícitos* y *macros*. Los métodos para construir los objetos del modelo, declaran al nombre del objeto como un *parámetro implícito*, el cual es determinado y pasado al método por el compilador de Scala, evitando que el programador tenga que especificarlo. Como ejemplo se muestra una de las definiciones sobrecargadas de `xvar`:

```
def xvar(entries: IndEntry*)(implicit name: sourcecode.Name): VarStat =
  xvar(name.value, entries: _*)
```

En la definición anterior, el *valor implícito* necesario para ser pasado como valor del parámetro implícito `name`, es provisto por la biblioteca `sourcecode`. Dicha biblioteca utiliza macros para determinar el nombre de la definición más cercana que rodea la expresión y construirlo. Los nombres de los objetos del modelo pueden ser especificados explícitamente en los métodos utilizando versiones sobrecargadas de los mismos, sin embargo, el disponer de las capacidades mencionadas del lenguaje permite evitar la redundancia que implica especificar el nombre explícitamente, lo que simplifica un poco la lectura del código y acerca más el EDSL al lenguaje MathProg.

Otra característica de la declaración de `xvar` anterior, es que el parámetro `entries` está declarado como *parámetro repetido*. Las llamadas a `xvar` pueden tener un número variable de argumentos en la medida que sean de un tipo `IndEntry`, como sucede en las declaraciones de `v` y `u` mostradas anteriormente, una con dos argumentos y otra con uno. El mecanismo anterior permite

representar cualquier expresión de indexación que no tenga predicado.

En los ejemplos de declaraciones de entidades anteriores se puede ver también la sintaxis utilizada para la definición de los atributos asignación (`:=`), cota inferior de variable (`>=`), y el que establece que una variable es binaria (`binary`). En todos los casos los atributos se especifican mediante métodos de Scala. Como el AST es inmutable, cada una de las llamadas a los métodos anteriores no modifica el objeto sino que crea uno nuevo con las diferencias necesarias para incorporar el cambio (por ejemplo la lista de atributos con un atributo más). El uso de *case classes* para los constructores del AST nuevamente resulta útil ya que puede aprovecharse el método `copy` del que disponen, permitiendo crear copias de objetos especificando únicamente los argumentos del constructor que deben cambiar. Por ejemplo, el siguiente código muestra un posible uso de `copy` para "actualizar" atributos:

```
val x1 = x0.copy(atts = x0.atts :+ Binary)
val x2 = x1.copy(atts = x1.atts :+ VarGTE(0))
```

Para las declaraciones de restricciones y la función objetivo se sigue un esquema similar que con las entidades del modelo. Se definen tres *smart constructors*, `st`, `minimize` y `maximize`, para definir restricciones y funciones objetivos de minimización o maximización. A continuación se muestran ejemplos de dichas declaraciones construidas con el EDSL y su equivalente en MathProg.

```
val singleAcquisition = st(c in P){
  sum(t in (1 to H - gamma(c))) {
    v(c,t)
  } <= 1
}
```

```
s.t. single_acquisition{c in P}:
  sum{t in 1 .. H - gamma[c]}
    v[c,t] <= 1;
```

```
val cost = minimize {
sum(t in T) {
  sum((c in P) | t <= H-gamma(c)) {
             ca(c)  * q(c) * v(c,t)
  } +
  sum((c in A) | t <= tau(c)-1) {
    (cc(c) - ca(c)) * q(c) * x(c,t)
  } +
  h(t) * y(t)
}}
```

```
minimize cost:
sum{t in T} (
  sum{c in P : t <= H - gamma[c]}
             ca[c]  * q[c] * v[c,t]
  +
  sum{c in A : t <= tau[c] - 1}
    (cc[c] - ca[c]) * q[c] * x[c,t]
  +
  h[t] * y[t]
);
```

La posibilidad que brinda Scala de que los métodos tengan *múltiples listas de parámetros* permite separar los parámetros que corresponden a la expresión de indexación de una restricción, del que tiene su definición. Por ejemplo, la siguiente es una de las declaraciones sobrecargadas de `st`:

```
def st(entries: IndEntry*)(ctr: ConstraintStat)
      (implicit name: sourcecode.Name): ConstraintStat =
  st(name.value, IndExpr(entries.toList).some, ctr)
```

Como en este caso la segunda lista de parámetros tiene únicamente un parámetro, es posible utilizar llaves (`{}`) en lugar de paréntesis para delimitar la lista al momento de aplicar la función, lo que puede usarse para distinguir el rol que tienen los parámetros. En otros *smart constructors*, como el utilizado para la sumatoria, `sum`, también se utilizan múltiples listas de parámetros con un fin similar al buscado en las restricciones.

En los ejemplos de restricción y función objetivos anteriores, se muestra también la sintaxis para hacer referencia a las entidades del modelo que son arreglos multidimensionales. En el *embedding* se utilizan paréntesis curvos en lugar de rectos, ya que esa es la sintaxis que usa Scala para la indexación en arreglos, al igual que para la aplicación de funciones. Scala provee de "azúcar sintáctico" que permite que cualquier método llamado `apply` pueda ser invocado sin nombrarlo, lo que permite utilizar la sintaxis para aplicación de funciones sobre construcciones definidas por el usuario.

Scala es un lenguaje estricto pero provee dos construcciones que habilitan que expresiones se evalúen con una semántica no estricta. Por un lado, es posible definir variables finales *perezosas* (*lazy vals*) en las cuales la expresión en el lado derecho de su definición solo es evaluada la primera vez que la variable es utilizada. Por otro lado, Scala dispone de parámetros pasados *por nombre*, los que solamente son evaluados cuando se hace referencia a ellos, recomputándose cada vez (*call-by-name*). Se observa que al soportar alto orden, siempre es posible obtener una semántica no estricta reemplazando una expresión de tipo `A` por una función `() => A` (una función del tipo `Unit` al tipo `A`) en la que su cuerpo sea la expresión que se está reemplazando (Paulson 1996, pág. 195; Wadler et al. 1998). Las características anteriores de Scala se utilizan en el EDSL para permitir definiciones recursivas de parámetros y conjuntos del modelo. Por ejemplo, los constructores de referencias a parámetros del modelo, clase `ParamRef`, toman como argumento una función `() => ParamStat` para tener control sobre el momento en el que se evalúa la declaración del parámetro.

```
class ParamRef(
  paramThunk: () => ParamStat,
  val subscript: List[SimpleExpr] = Nil) extends NumExpr
                                            with SymExpr
                                            with Product
                                            with Serializable {
  lazy val param: ParamStat = paramThunk()
  // ...
}
```

La clase internamente define una variable *lazy val*, llamada `param`, con el resultado de la función anterior, de modo que la primera vez que se acceda se guarde su resultado y no sea necesario volver a calcularlo. Todos los accesos al campo `param` de la clase `ParamRef` se realizan desde otras variables *lazy val* o mediante métodos (`def`), de modo de no forzar su evaluación prematuramente. La clase `ParamRef` no es una *case class* para tener control sobre la implementación de métodos como `equals` y `hashCode`, que Scala hereda de Java y tienen una implementación predefinida para las *case classes*. Para recuperar parte de la funcionalidad provista por las *case classes*, la clase `ParamRef` define un *objeto acompañante* con un *smart constructor*, `apply`, que recibe el `ParamStat` al que se hace referencia como un parámetro por nombre (indicado con el tipo `=> ParamStat`), el cual es pasado al constructor de `ParamRef` con una expresión *lambda*, de modo que no sea evaluado en ese momento.

```
object ParamRef {
  def apply(param: => ParamStat, subscript: List[SimpleExpr] = Nil) =
    new ParamRef(() => param, subscript)
}
```

Para la construcción de la instancia de `Model` correspondiente al modelo completo (Sección 2.2.1), se provee de un *smart constructor* adicional, `model`, que permite construir la instancia especificando la función objetivo y las restricciones a utilizar.

```
val mipModel =
  model(cost,
    balance0, balance, inventory,
    singleAcquisition, singleCancellation,
    acquiredFuel, cancelledFuel
  )
```

Esta función no tiene un análogo en MathProg, porque allí todas las funciones y restricciones declaradas implícitamente forman parte del modelo que se quiere construir.

2.2.3. Sobrecarga de igualdad e identificadores reservados

En Scala, al igual que en Java (Gosling et al. 2018), todas las clases heredan de una superclase común, llamada `Any` en Scala, que define una serie de métodos entre los que se encuentra el operador de igualdad, ==.

```
abstract class Any {
  final def ==(that: Any): Boolean = // ...
  // ... otros métodos y operadores ...
}
```

En MathProg, el operador de igualdad puede utilizarse como atributo de parámetros y variables del modelo, como operador relacional entre expresiones simples, o para definir restricciones del modelo. En ninguno de los casos anteriores el valor de retorno es un `Boolean`, por lo que en el *embedding* debe utilizarse una definición alternativa a la provista por la superclase `Any`. Como el argumento del operador de igualdad en Scala es de tipo `Any` (parámetro `that`), siempre es posible invocar el método == sobre cualquier valor y con cualquier argumento –la expresión `x == y` siempre es válida. Esto interfiere con el mecanismo de Scala para proveer *extension methods*, ya que solo se buscan posibles extensiones si la llamada al método no tiene éxito utilizando los métodos presentes en la clase. Por el problema anterior, en el *embedding*, para poder proveer las distintas sobrecargas del operador de igualdad requeridas mediante *extension methods*, se utiliza === con el fin de que no exista ninguna interacción con el heredado desde `Any`. El uso del operador === para representar la igualdad como alternativa sobre el heredado por todas las clases, está en línea con lo que se realiza en otras bibliotecas del ecosistema de Scala, como `Cats` y `Scalaz` (Cats 2018; Scalaz 2018).

Otros casos donde hay conflictos entre la sintaxis que se quiere proveer para el *embedding* y el lenguaje anfitrión, son los identificadores de MathProg que en Scala son palabras reservadas. En los casos en que el identificador en MathProg coincide con una palabra reservada en Scala, se decide utilizar un nombre alternativo consistente en prefijar la letra “x” al nombre original, como en el caso visto anteriormente del constructor de declaraciones de variables, en que se cambia `var` de MathProg por `xvar` en Scala. El otro caso en que se da el problema anterior es con la palabra `if` en las expresiones condicionales, utilizándose `xif` en el *embedding*.

2.2.4. Codificación de operadores

El *embedding* desarrollado es un *deep embedding* y utiliza el árbol de sintaxis abstracta como estructura de datos básica para representar los componentes del modelo (Gibbons 2015). Tener disponibles los componentes del modelo como una estructura de datos en memoria simplifica su manipulación y la aplicación de transformaciones a posteriori, las que pueden implementarse como recorridas sobre el árbol de sintaxis abstracta. El AST se implementa en Scala con una jerarquía de clases sellada utilizando *case classes* para representar los constructores siempre que sea posible (Sección 2.2.1).

Para la simular la sintaxis de MathProg se evita utilizar funciones declaradas directamente en las clases. La forma de trabajo anterior se separa de la práctica usual de los lenguajes orientados a objetos, donde los métodos relacionados con una clase (el "comportamiento") se define junto con la clase. Separar las funciones de los datos tiene dos objetivos: por un lado queda más clara visualmente la estructura del AST al no incluir las funciones en las clases, y, por otro, se consigue un grado mayor de extensibilidad del *embedding* evitando la prioridad que tienen en Scala los métodos definidos en un clase por sobre cualquier otro *extension method* definido por terceros (Odersky et al. 2019). Un problema del enfoque anterior es que hace más difícil encontrar las funciones asociadas a una clase, ya que pueden estar definidas en múltiples ubicaciones.

Muchos de los operadores y construcciones que constituyen la sintaxis de MathProg requieren de un gran número de sobrecargas. Por ejemplo, los siguientes son usos posibles del operador >= con los tipos de los operandos y del resultado:

```
param("p", J) >= 0      // : ParamStat => Int        => ParamStat
param("p", J) >= p2     // : ParamStat => SimpleExpr => ParamStat
 xvar("x", I) >= 0      // : VarStat   => Int        => VarStat
            i >= j      // : NumExpr   => NumExpr    => LogicExpr
       p(j1) >= p(j2)   // : NumExpr   => NumExpr    => LogicExpr
        x(i) >= 0       // : LinExpr   => Int        => ConstraintStat
          10 >= x(i)    // : Int       => LinExpr    => ConstraintStat
```

Por otra parte, se desea que sea posible que terceros definan nuevas sobrecargas, por ejemplo para otros tipos numéricos o tuplas de mayor aridad. La funcionalidad anterior puede ser lograda en Scala mediante el uso de *type classes* (Wadler y Blott 1989), las que son codificadas mediante un tipo abstracto paramétrico, e *instancias* que son definidas como valores implícitos del tipo paramétrico instanciado en un tipo concreto (Oliveira et al. 2010). La codificación de *type classes* en Scala usualmente incluye un componente adicional para poder utilizar las funciones en forma infija y tengan el mismo aspecto que la invocación de métodos en objetos. Esta codificación de *type classes* con tres componentes (llamados *ops*, *syntax* e *instances* en este trabajo) requiere de una cantidad significativa de *boilerplate*. Es posible reducir dicho *boilerplate* utilizando macros, por ejemplo utilizando la biblioteca `simulacrum` (Pilquist 2018), pero solamente soporta constructores de tipo con un solo argumento, por lo que no es utilizada.

Como ejemplo del uso de *type classes* para implementar la sintaxis, se comenta el caso del operador >=. El procedimiento es el mismo para todos los operadores o funciones que operan sobre el AST del lenguaje. La implementación del operador >= implica la definición de tres componentes: un *trait* paramétrico, `GTEOp[A,B,C]`, que representa la *type class*,

```
trait GTEOp[A, B, C] {
  def gte(lhe: A, rhe: B): C
}
```

instancias para los distintos tipos concretos en los que se quiere definir el operador, y una clase implícita, `GTESyntax[A]`, que agrega el método >= a cualquier tipo `A` para el que exista una instancia de `GTEOp[A,B,C]` apropiada.

```
implicit class GTESyntax[A](lhe: A) {
  def >=[B, C](rhe: B)(implicit GTEOp: GTEOp[A, B, C]): C =
    GTEOp.gte(lhe, rhe)
}
```

El objetivo es poder escribir expresiones de la forma `lhe >= rhe` que construyan el componente del modelo que corresponde en base a los tipos de `lhe` y `rhe`. La clase implícita `GTESyntax` establece que sobre un valor `lhe` de tipo `A` se puede invocar un método >= con un argumento `rhe` de tipo `B`, y devolver un valor de tipo `C`, siempre y cuando exista una instancia de `GTEOp[A,B,C]`. La restricción de que exista la instancia de `GTEOp` está dada por el parámetro implícito, de nombre `GTEOp`, declarado en la segunda lista de parámetros del método >=. La declaración del parámetro implícito a nivel del método en lugar de a nivel del constructor de la clase implícita, permite que ambos tipos, `A` y `B`, sean utilizados para determinar la instancia de la *type class*. Se observa que el tipo de retorno, `C`, no participa en la determinación de la instancia.

La *type class* `GTEOp[A,B,C]` define un método `gte` que es el encargado de implementar la operación >= para los tipos `A` y `B` retornando `C`. El método >= de la clase `GTESyntax` puede implementarse en términos de `GTEOp.gte` ya que tiene disponible una instancia de `GTEOp` en forma implícita. Clases como `GTESyntax` son llamadas *object-oriented forwarders* en `simulacrum` porque su único rol es permitir utilizar al método de la *type class*, `gte`, en forma infija, como si fuera la invocación a un método en el argumento de la izquierda.

Una vez definida la *type class* y el *extension method*, el problema se reduce a definir las instancias correctas con su prioridad asociada. Las instancias de la *type class* son valores implícitos, o métodos implícitos cuyos parámetros son todos implícitos. El uso de un método para definir una instancia permite que la instancia sea paramétrica, ya que en Scala los valores no son polimórficos. Por otra parte, declarar parámetros implícitos en los métodos que definen instancias permite condicionar la existencia de una instancia a la existencia de otra o a la existencia de cualquier valor implícito en forma general.

El *embedding* define cuatro instancias de `GTEOp`: para la especificación de la cota inferior en las declaraciones de parámetros del modelo, para la especificación de la cota inferior en las declaraciones de variables del modelo, para la comparación de expresiones numéricas, y para la construcción de restricciones de mayor o igual.

La instancia de la *type class* `GTEOp` que permite usar el operador >= para definir la cota inferior de un parámetro del modelo, `ParamStatGTEOp`, es un método implícito, paramétrico, que construye un valor de tipo `GTEOp[ParamStat, A, ParamStat]`.

```
implicit def ParamStatGTEOp[A]
  (implicit conv: A => SimpleExpr): GTEOp[ParamStat, A, ParamStat] =
  new GTEOp[ParamStat, A, ParamStat] {
    def gte(lhe: ParamStat, rhe: A) =
      lhe.copy(atts = lhe.atts :+ ParamGTE(rhe))
  }
```

El tipo del argumento de la derecha, en principio, puede ser cualquier tipo `A`. Sin otras restricciones, la instancia anterior es imposible de construir ya que el AST especifica tipos concretos para los valores en los atributos, en particular para la cota inferior se requiere un valor de tipo `SimpleExpr`. La instancia `ParamStatGTEOp` es válida porque el método requiere la existencia de un valor implícito, `conv`, de tipo `A => SimpleExpr`, es decir, una función que permita convertir cualquier valor de tipo `A` en un valor de tipo `SimpleExpr`. Como el parámetro `conv` de `ParamStatGTEOp` es implícito y es una función, define una *conversión implícita* que se aplica automáticamente cuando se usa el valor `rhe`, de tipo `A`, como argumento de `ParamGTE` en la definición del método `GTEOp.gte`.

Definir la instancia anterior en forma paramétrica en lugar de para el tipo concreto que se requiere, `SimpleExpr`, permite que el compilador de Scala la considere una instancia candidata para cualquier tipo "convertible" a `SimpleExpr`, sin que necesariamente exista una relación de subtipo entre ambos. En este caso, la búsqueda de la instancia de `GTEOp` se "encadena" con la búsqueda de conversiones implícitas entre el tipo en cuestión y `SimpleExpr`, lo que permite escribir expresiones como las siguientes:

```
/*
  Encadena búsqueda de instancia de `GTEOp[ParamStat, Int, ?]` con
  búsqueda de conversión implícita `Int => SimpleExpr`.
*/
param("a", I) >= 3

/*
  Encadena búsqueda de instancia de `GTEOp[ParamStat, String, ?]` con
  búsqueda de conversión implícita `String => SimpleExpr`.
*/
param("b", J) >= "A"
```

La instancia que permite usar el operador >= para definir la cota inferior de las variables del modelo, `VarStatGTEOp`, funciona en forma análoga a la usada para los parámetros, descrita en el párrafo anterior.

```
implicit def VarStatGTEOp[A]
  (implicit conv: A => NumExpr): GTEOp[VarStat, A, VarStat] =
  new GTEOp[VarStat, A, VarStat] {
    def gte(lhe: VarStat, rhe: A) =
      lhe.copy(atts = lhe.atts :+ VarGTE(rhe))
  }
```

La instancia `NumExprGTEOp` permite usar el operador >= para comparar dos expresiones numéricas y construir una expresión lógica. En forma similar al método `ParamStatGTEOp`, en el método `NumExprGTEOp` declara dos parámetros de tipo, a priori sin restricciones, pero que están condicionados a la existencia de valores implícitos declarados como parámetros implícitos de la función (`convA` y `convB`).

```
implicit def NumExprGTEOp[A, B]
  (implicit convA: A => NumExpr,
            convB: B => NumExpr): GTEOp[A, B, LogicExpr] =
  new GTEOp[A, B, LogicExpr] {
    def gte(lhe: A, rhe: B) = GTE(lhe, rhe)
  }
```

Los parámetros implícitos de `NumExprGTEOp` establecen que para poder usar el operador >= entre dos tipos `A` y `B` para construir una expresión lógica, ambos deben ser "convertibles" implícitamente al tipo `NumExpr`, que es el que requiere el constructor de expresiones lógicas de relaciones de "mayor o igual", `GTE`.

La instancia definida por `LinExprGTEOp` es análoga a la definida por `NumExprGTEOp`, en este caso los tipos `A` y `B` deben ser "convertibles" implícitamente a expresiones lineales y la expresión resultante es una restricción del modelo.

```
implicit def LinExprGTEOp[A, B]
  (implicit convA: A => LinExpr,
            convB: B => LinExpr): GTEOp[A, B, ConstraintStat] =
  new GTEOp[A, B, ConstraintStat] {
    def gte(lhe: A, rhe: B) =
      GTEConstraintStat(
        name  = gen.ctr.freshName, left  = lhe, right = rhe)
  }
```

Una dificultad que surge al definir las instancias anteriores es que algunas de ellas son ambiguas. La instancia para expresiones numéricas entra en conflicto con la instancia para declaraciones de parámetros y con la instancia para expresiones lineales. En el *embedding* se define una conversión implícita entre `ParamStat` y `NumExpr` que permite utilizar los parámetros del modelo como referencias a los mismos, para permitir expresiones como por ejemplo: `param("p") >= p2`. Esto provoca que en la expresión anterior, las instancias provistas por `ParamStatGTEOp` y `NumExprGTEOp` sean posibles candidatas. Por otra parte, las expresiones numéricas son subtipo de las expresiones lineales (Figura 2.1), por lo que para la comparación de dos expresiones numéricas podría utilizarse tanto la instancia provista por `NumExprGTEOp`, como la provista por `LinExprGTEOp`.

Para resolver las ambigüedades se utiliza el mecanismo de *priorización de implícitos* provisto por Scala. En Scala, las instancias definidas en una clase tienen mayor prioridad que las instancias definidas por una superclase. La instancia para expresiones lineales se define con la menor prioridad, de modo de no tratar a las expresiones numéricas como lineales innecesariamente.

```
trait GTEInstancesLowPriority2 {
  implicit def LinExprGTEOp[A, B]
    (implicit convA: A => LinExpr,
              convB: B => LinExpr): GTEOp[A, B, ConstraintStat] = /* */
}
```

La instancia para expresiones numéricas se define con prioridad intermedia

```
trait GTEInstancesLowPriority1 extends GTEInstancesLowPriority2 {
  implicit def NumExprGTEOp[A, B]
    (implicit convA: A => NumExpr,
              convB: B => NumExpr): GTEOp[A, B, LogicExpr] =  /* */
}
```

y la instancia para las declaraciones de parámetros con prioridad máxima, de modo que no se construya una expresión lógica cuando en realidad se quiere especificar un atributo.

```
trait GTEInstances extends GTEInstancesLowPriority1 {
  implicit def ParamStatGTEOp[A](implicit conv: A => SimpleExpr):
    GTEOp[ParamStat, A, ParamStat] = /* */

  implicit def VarStatGTEOp[A](implicit conv: A => NumExpr):
    GTEOp[VarStat, A, VarStat] = /* */
}
```

Las mismas técnicas utilizadas para el operador >= son utilizadas para codificar todos los operadores y funciones necesarias para simular la sintaxis de MathProg dentro de Scala. Por cada operador o función se define un *trait* análogo a `GTEInstances` que luego es "mezclado" en un *trait* general, `AllInstances`.

```
package amphip.model
trait AllInstances extends NumInstances
                      with SymInstances
                      with TupleInstances
                      // ...
```

Otro trait, `AllSyntax`, agrupa todas las clases análogas a `GTESyntax` y los diferentes *smart constructors* definidos.

```
package amphip.model
trait AllSyntax {
  // ...
  def set  (name: SymName, indexing: IndExpr): SetStat   = /* */
  def param(name: SymName, indexing: IndExpr): ParamStat = /* */
  def xvar (name: SymName, indexing: IndExpr): VarStat   = /* */
  // ...
  implicit class GTESyntax[A](lhe: A) { /* */ }
  // ...
}
```

Tener las definiciones en *traits*, permite aprovechar el sistema de módulos de Scala y dar flexibilidad al usuario para importar la funcionalidad provista por el EDSL en forma granular. Por ejemplo, para la "capa de modelo" definida bajo el paquete `amphip.model`, se tiene dos objetos `instances` y `syntax`, que incorporan `AllInstances` y `AllSyntax` respectivamente, y permiten trabajar con cada grupo de definiciones por separado.

```
package amphip.model
object instances extends AllInstances
object syntax    extends AllSyntax
```

Por otra parte, en la misma capa se tiene un *trait* `ModelDSL` que agrupa `AllInstances` y `AllSyntax` en un mismo *trait*, y que es incorporado en el objeto `amphip.model.dsl` para poder trabajar con la capa de modelo en forma separada, y en el objeto `amphip.dsl`, para el uso en conjunto con las otras capas de funcionalidad desarrolladas.

```
/*
  importar `amphip.model.dsl._` permite trabajar con la capa de modelo
  en forma independiente.
*/
package amphip.model
object dsl extends ModelDSL
trait ModelDSL extends AllInstances
                  with AllSyntax
                  with ShowInstances


/*
  importar `amphip.dsl._` permite trabajar con todas las capas del EDSL
  (modelo, datos, estocástica) al mismo tiempo.
*/
package amphip
object dsl extends model.ModelDSL
              with  data.DataDSL
              with stoch.StochDSL
```

Las *type classes* análogas a `GTEOp` para los otros operadores no se definen en un *trait* porque, de ser "mezclados" en diferentes módulos, definirían tipos distintos cada vez lo que sería inconveniente. Las *type classes* se definen todas en el objeto `amphip.model.ops`.

2.3. Soporte para especificación de datos

En MathProg, los datos de los conjuntos y parámetros del modelo pueden especificarse directamente en sus declaraciones mediante los atributos de asignación (`:=`) y valor por defecto (`default`), o mediante *bloques de datos* que pueden ser incluidos dentro de la especificación del modelo o en un archivo aparte (Sección 2.1). Los bloques de datos permiten establecer una separación entre la definición del problema (el modelo), y la instancia de datos particular que se utiliza para resolverlo.

Para la especificación de datos, en el *embedding* desarrollado es posible también especificar los valores de conjuntos y parámetros del modelo directamente en la declaración de los mismos utilizando atributos, pero no se dispone de una representación de los bloques de datos de MathProg. Para cumplir el rol de los bloques de datos se decide utilizar una sintaxis diferente de la que provee MathProg, aprovechando las capacidades del lenguaje anfitrión y su biblioteca estándar para la manipulación de datos.

Para brindar el soporte para la especificación de datos se define una nueva capa por sobre la capa de modelo descrita en la Sección 2.2. Esta nueva "capa de datos", ubicada bajo el paquete `amphip.data`, se define utilizando la misma estrategia que para la capa de modelo, con tres objetos: `ops` para las *type classes*, `instances` para las instancias predefinidas, `syntax` para los *extension methods* y *smart constructors*; y dos *traits*: `AllInstances` para agrupar las definiciones de instancias, y `AllSyntax` para agrupar las definiciones de sintaxis.

2.3.1. Estructuras de datos

Se definen tipos para representar los posibles valores que se pueden asignar a conjuntos y parámetros del modelo. En el caso de los parámetros, los valores

pueden ser solamente numéricos o simbólicos (*string*), por lo que se define un tipo `SimpleData` que representa la unión de esos dos tipos.

```
sealed trait SimpleData
case class SimpleNum(num: BigDecimal) extends SimpleData
case class SimpleStr(str: String)     extends SimpleData
```

En el caso de los conjuntos los valores son siempre listas de tuplas, y las tuplas son a su vez listas de valores numéricos o simbólicos, por lo que se definen una clase y un alias, `SetTuple` y `SetData`, que representan las tuplas y las listas de tuplas respectivamente.

```
type SetData = List[SetTuple]
case class SetTuple(values: List[SimpleData])
```

Para las tuplas se define una clase en vez de un alias para tener mayor control sobre las definiciones de instancias de *type classes*.

Para almacenar los datos en memoria, se define una clase `ModelData` que contiene todos los datos especificados hasta el momento.

```
case class ModelData(
    sets  : SetStatData   = LinkedMap.empty,
    params: ParamStatData = LinkedMap.empty,
    setsExpansion  : Expansion[SetStat]   = LinkedMap.empty,
    paramsExpansion: Expansion[ParamStat] = LinkedMap.empty) {

  // métodos auxiliares ...
}

type SetStatData   = LinkedMap[DataKey, SetData]
type ParamStatData = LinkedMap[DataKey, SimpleData]
type Expansion[A]  = LinkedMap[DataKey, A]

case class DataKey(name: String, subscript: List[SimpleData] = Nil)
```

La clase `ModelData` agrupa diferentes mapas que asocian los valores a los diferentes subíndices de cada conjunto y parámetro del modelo. Para las claves de los mapas donde se mantienen los valores, se define una tipo `DataKey` constituido por un nombre de declaración y un subíndice representado como una lista que puede ser vacía.

La clase también mantiene *expansiones* de declaraciones de conjuntos y parámetros del modelo, que también son mapas pero que asocian un `DataKey` con una declaración individual, sin un dominio que la indexe. Dichas expansiones son el producto de *evaluar* una declaración, que está potencialmente indexada, en un conjunto de declaraciones "atómicas", sin dominio (Sección 2.4). Tener disponible las declaraciones expandidas junto con los datos es útil para determinar el valor correspondiente a un parámetro o conjunto tomando en cuenta que puede haber sido definido con los atributos de asignación o valor por defecto.

Para representar un modelo MathProg con datos especificados total o parcialmente, se utiliza la clase `ModelWithData` que simplemente combina ambas estructuras:

```
case class ModelWithData(model: Model, data: ModelData)
```

2.3.2. Interfaz para la especificación de datos

La interfaz provista para trabajar con los datos consiste básicamente en dos *extension methods*, `paramData` y `setData`, que permiten especificar los valores para los parámetros y conjuntos. El código a continuación muestra distintas formas en las que se desea poder usar el método `paramData`.

```
val mipWithData1 = mipModel
  /* parámetro simple */
  .paramData(H, 3)

  /* parámetro con un conjunto índice, valores especificados como
   * pares (subíndice, valor)
   */
  .paramData(tau,
    "A1" -> 1,
    "A2" -> 2)

  /* parámetro con dos conjuntos índice, valores especificados como
   * pares (subíndice, valor). el subíndice es un par.
   */
  .paramData(yields,
    (1, "w") ->  3.0,
    (1, "c") ->  3.6,
    (1, "b") -> 24.0)

  /* parámetro con tres conjuntos índice, valores especificados como
   * pares (subíndice, valor). el subíndice es una lista.
   */
  .paramData(anc,
    List(3, 2, 3) -> 2,
    List(3, 2, 2) -> 1,
    List(3, 2, 1) -> 1)
}
```

El valor `mipModel` es un modelo construido mediante la función `model` (Sección 2.2.2) y contiene únicamente información sobre las declaraciones del modelo. El método `paramData` se invoca como un *extension method* sobre la clase `Model` y permite construir un "modelo con datos" representado con la *case class* `ModelWithData`, la que contiene además del modelo una instancia de `ModelData` (Sección 2.3.1). En cada llamada a `paramData` es posible especificar los valores asociados a un parámetro del modelo indicando, opcionalmente, el subíndice asociado a cada uno, en forma "extensiva".

Alternativamente, el EDSL permite especificar todos los valores de un vez, en forma "agregada", a partir de una lista o secuencia, la que puede ser calculada por separado haciendo uso de las capacidades del lenguaje anfitrión.

```
// datos generados en forma separada
val tauData = List("A1" -> 1, "A2" -> 2)
val yieldsData =
  List((1, "w") ->  3.0, (1, "c") ->  3.6, (1, "b") -> 24.0)
val ancData =
  List(List(3, 2, 3) -> 2, List(3, 2, 2) -> 1, List(3, 2, 1) -> 1)

val mipWithData2 = mipModel
  /* mismos parámetros anteriores con los valores especificados como
   * una lista de pares (subíndice, valor), manteniendo la forma de
   * especificar el subíndice utilizada anteriormente.
   */
  .paramData(tau   , tauData)
  .paramData(yields, yieldsData)
  .paramData(anc   , ancData)
```

Para los parámetros indexados se soporta especificar los datos sin indicar el subíndice al que corresponden, en cuyo caso los valores especificados se asignan en orden a los subíndices generados por la expresión de indexación del dominio del parámetro (Sección 2.4).

```
val yieldsDataSimple = List(3.0, 3.6, 24.0)

val mipWithData3 = mipModel
  /* método alternativo para espeficiar los valores de un parámetro
   * indexado sin indicar el subíndice que corresponde en cada caso.
   */
  .paramData(yields, yieldsDataSimple)
```

El caso de `setData` es muy similar a `paramData`. La diferencia principal que tiene `setData` respecto de `paramData`, es que cada valor individual a asignar a un conjunto es una *lista de tuplas*, las que pueden tener tamaño uno, el lugar de un valor simple.

```
val mipWithData1 = mipModel
  /* conjunto simple */
  .setData(I, List("stock", "bonds"))

  /* conjunto simple de dimensión dos. cada integrante es un par.
   */
  .setData(FB, List("FOB" -> "MFO", "DIL" -> "MFO"))

  /* mismo conjunto anterior utilizando listas para representar cada
   * tupla.
   */
  .setData(FB, List(List("FOB","MFO"), List("DIL","MFO")))

  /* conjunto indexado. valores especificados como pares (subíndice,
   * valor)
   */
  .setData(TS,
    1 -> List(1),
    2 -> List(1,2),
    3 -> List(1,2,3,4))
```

Al utilizar la forma agregada de especificar los valores, en lugar de especificar listas de valores simples opcionalmente indexados, en el método `setData` se especifican "listas de listas" de tuplas.

```
val TSData = List(1 -> List(1), 2 -> List(1,2), 3 -> List(1,2,3,4))
val TSDataSimple = List(List(1), List(1,2), List(1,2,3,4))

val mipWithData2 = mipModel
  /* mismo conjunto anterior con los valores especificados como
  * una lista de pares (subíndice, valor).
  */
  .setData(TS, TSData)

  /* método alternativo para espeficiar los valores de un conjunto
  * indexado sin indicar el subíndice que corresponde en cada caso
  */
  .setData(TS, TSDataSimple)
```

2.3.3. *Type classes* e instancias

De la misma forma que en el caso del *embedding* de los operadores de MathProg (Sección 2.2.4), el polimorfismo que admiten los métodos `paramData` y `setData` se consigue mediante el uso de *type classes* y la definición de instancias adecuadas. A continuación se muestra la definición de la *type class* utilizada, `DataOp`.

```
trait DataOp[A, B] {
  def data(decl: A, values: List[B])
    (implicit modelData: ModelData): ModelData
}
```

La *type class* `DataOp` tiene dos parámetros de tipo, `A` y `B`, y define un método `data` que toma una declaración de tipo `A`, una lista de valores de tipo `List[B]`, y un parámetro implícito de tipo `ModelData`, y devuelve otro `ModelData` con los datos que deberían agregarse al modelo para la declaración. La lista de valores representa a los parámetros repetidos que admiten tanto `paramData` como `setData` en la forma de uso extensiva. El método `data` de la *type class* `DataOp` recibe implícitamente una instancia de `ModelData` porque en su implementación puede ser necesaria la evaluación de alguna expresión (Sección 2.4), como por ejemplo el dominio de la declaración, y para ello puede requerirse datos definidos previamente como los valores asignados a un conjunto.

Las declaraciones de los *extension methods* son las siguientes:

```
implicit class ModelWithDataSyntax[M](m: M)
  (implicit conv: M => ModelWithData) {

  def paramData[B](decl: ParamStat, values: B*)
    (implicit DataOp: DataOp[ParamStat, B]) =
    declData(decl, values: _*)

  def setData[B](decl: SetStat, values: B*)
    (implicit DataOp: DataOp[SetStat, B]) =
    declData(decl, values: _*)

  // `declData` y otras declaraciones ...
}
```

Los métodos `paramData` y `setData` están definidos como *extension methods* para cualquier tipo `M` que sea "convertible" a `ModelWithData`, lo que habilita en particular que estén disponibles para la clase `Model`, ya que hay definida una

conversión implícita hacia `ModelWithData`. La clase implícita que se utiliza para dejar disponibles dichos *extension methods*, `ModelWithDataSyntax`, recibe en su constructor un valor `m` de un tipo `M`, convertible a `ModelWithData`, y es sobre el que se "invocan" los métodos.

La implementación de los métodos `paramData` y `setData` es idéntica, ya que no depende del tipo de la declaración en cuestión (conjunto o parámetro) sino que de la instancia de `DataOp` que se utiliza. Por lo anterior, ambos métodos delegan su implementación a otro método más general, `declData`, donde el tipo de la declaración es paramétrico y utiliza la instancia de `DataOp` para construir un `ModelData` con los datos que deberían agregarse al modelo.

```
def declData[A,B](decl: A, values: B*)
  (implicit DataOp: DataOp[A, B]): ModelWithData = {
  val newData = DataOp.data(decl, values.toList)(m.data)
  m.copy(data = m.data + newData)
}
```

En la llamada a `DataOp.data` en `declData`, el parámetro implícito de tipo `ModelData` es pasado explícitamente –se pasa el conjunto total de datos que tiene el modelo hasta el momento, obtenido de `m`. Una vez obtenidos los nuevos datos a agregar al modelo, se crea una copia del modelo con datos, combinando los datos que se tenían con los datos nuevos.

Los tipos concretos que pueden utilizarse en cada caso quedan determinados por las conversiones implícitas que estén en alcance. En base a los tipos definidos en la Sección 2.3.1, los valores que se utilizan para los parámetros del modelo deben ser convertibles al tipo `SimpleData`, y los que se utilizan para los conjuntos del modelo deben ser convertibles a `SetData`, mientras que los valores utilizados para los subíndices deben ser convertibles a `List[SimpleData]`. El EDSL define conversiones para tipos numéricos de Scala, Strings, tuplas y colecciones iterables, pudiéndose agregar nuevas conversiones en forma independiente de ser necesario.

El soporte para especificación extensiva de datos indexados es cubierto por una única instancia de `DataOp`, `ParamStatIndexedDataOp`.

```
implicit def ParamStatIndexedDataOp[B](
  implicit convB: B => (List[SimpleData], SimpleData)):
           DataOp[ParamStat, B] =
  new DataOp[ParamStat, B] {
    def data(decl: ParamStat, values: List[B])
      (implicit modelData: ModelData): ModelData = /* ... */
  }
```

Los distintos casos soportados quedan determinados por las conversiones implícitas que estén en alcance para pasar como valor implícito del parámetro `convB`. El tipo `B` de la instancia debe ser convertible al tipo de los datos de parámetros indexados, es decir, pares (*subíndice, valor*), que en el caso de los parámetros del modelo tienen tipo `(List[SimpleData], SimpleData)`. Por ejemplo, para soportar que se utilicen pares en lugar de listas como subíndices, se puede definir una conversión implícita como la siguiente:

```
implicit def Tuple2AsSubscript[A, B, C](t: ((A, B), C))
  (implicit convA: A => SimpleData,
            convB: B => SimpleData,
            convC: C => SimpleData): (List[SimpleData], SimpleData) =
  t match {
    case ((a, b), c) => List(convA(a), convB(b)) -> convC(c)
  }
```

Dicha conversión puede ser provista en forma separada del *embedding*, sin que haga falta modificar la instancia `ParamStatIndexedDataOp`. Como se ha comentado anteriormente, esta forma de trabajo permite que el EDSL sea más extensible, ya que no da prioridad a la funcionalidad predefinida sobre la desarrollada por terceros.

La instancia `ParamStatIndexedDataOp` primero analiza si la declaración en cuestión tiene un dominio o no. Si la declaración no tiene dominio, verifica que la lista de valores no sea vacía y que su primer elemento (`head`) tenga la lista vacía como subíndice, en ese caso interpreta los valores como si se tratara de un parámetro simple.

```
def data(decl: ParamStat, values: List[B])
  (implicit modelData: ModelData): ModelData = {
  decl.domain match {
    case None =>

      values.headOption.map(convB) match {
        case Some((Nil, head)) =>
          ModelData(params = LinkedMap(key(decl.name) -> head))
        case _ =>
          ModelData()
      }

    case Some(indexing) =>
      val evIndSet = eval(indexing).map(_.values.toList).toSet

      val valuesFilter = values
        .filter(x => evIndSet(x._1))
        .map(convB)

      val pairs = valuesFilter.map {
        case (subscript, value) =>
          key(decl.name, subscript) -> value
      }

      ModelData(params = LinkedMap(pairs: _*))
  }
}
```

Si la declaración del parámetro del modelo tiene un dominio, se utiliza el conjunto total de subíndices que pueden producirse con la expresión de indexación que constituye el dominio (variable `evIndSet`), para filtrar los valores especificados y descartar subíndices inválidos. Para obtener el conjunto de subíndices anterior, es necesario evaluar la expresión de indexación con la función `eval` (Sección 2.4), y para eso es requerido el parámetro `modelData` que se recibe implícitamente. Con los datos filtrados se construye una lista de pares y luego un mapa utilizando como clave un `DataKey` (Sección 2.3.1), y se construye el `ModelData` a ser devuelto.

La instancia para la especificación agregada de datos de parámetros indexados,

`ParamStatIndexedDataListOp`, tiene como única diferencia que el tipo `B` se espera que sea una lista de pares en lugar de un par individual.

```
implicit def ParamStatIndexedDataListOp[B]
  (implicit convB: B => List[(List[SimpleData], SimpleData)]):
            DataOp[ParamStat, B] =
  new DataOp[ParamStat, B] {
    def data(decl: ParamStat, values: List[B])
      (implicit modelData: ModelData): ModelData =
      ParamStatIndexedDataOp[(List[SimpleData], SimpleData)]
        .data(decl, values.flatten)
  }
```

Como en `paramData` el parámetro `values`, de tipo `B`, es un parámetro repetido, cuando se usa dicha instancia cada uno de los parámetros que se pasa debería ser una lista, aunque en este caso lo esperable es se especifique un único valor para la secuencia de parámetros repetidos. De todos modos, en la implementación de la instancia anterior se maneja el caso en que se especifique más de una lista, delegando la implementación a la instancia extensiva pasando como valores la concatenación de la listas de listas en una única lista de pares utilizando el método `flatten` de la biblioteca estándar de Scala.

Las otras instancias y conversiones implícitas necesarias para considerar todos los casos de especificación de datos de parámetros y conjuntos mostrados en la Sección 2.3.2 se definen siguiendo una lógica similar a la descrita en los párrafos anteriores.

2.4. Evaluación de componentes del modelo

Para la evaluación de expresiones y sentencias del modelo, se provee una función, `eval`, definida como un objeto con un método `apply` y una *type class*.

```
object eval {
  trait Eval[A, B] {
    def eval(expr: A)(implicit modelData: ModelData): B
  }

  def apply[A, B](expr: A)
    (implicit modelData: ModelData, Eval: Eval[A, B]): B =
    Eval.eval(expr)

  def apply[A, B](expr: A, modelData: => ModelData)
    (implicit Eval: Eval[A, B]): B =
    Eval.eval(expr)(modelData)

  // instancias de la type class ...
}
```

La *type class* `Eval` tiene dos parámetros de tipo, `A` y `B`, y define un método `eval` que recibe como parámetro un componente del modelo a evaluar, `expr`, de tipo `A`, y un parámetro implícito, `modelData`, de tipo `ModelData` (Sección 2.3.1), y devuelve el resultado de la evaluación de `expr`, de tipo `B`, en el contexto de `modelData`. El objeto `eval` define dos variantes para el método `apply` para poder especificar explícitamente la instancia de `ModelData` sin necesitar especificar también la instancia de `Eval` en forma explícita.

La Tabla 2.1 muestra los tipos a los que se evalúan los distintos componentes

del modelo. Algunos componentes se evalúan completamente a tipos predefinidos de Scala o a los tipos definidos para la especificación de datos (Sección 2.3.1).

Componente	Tipo destino
`Expr`	`Any`
`SimpleExpr`	`SimpleData`
`NumExpr`	`BigDecimal`
`SymExpr`	`String`
`SetExpr`	`SetData`
`LogicExpr`	`Boolean`
`LinExpr`	`LinExpr` (parcial)
`IndExpr`	`List[LinkedMap[DataKey, SimpleData]]`
`IndEntry`	`List[LinkedMap[DataKey, SimpleData]]`
`SetAtt`	`SetAtt` (parcial)
`ParamAtt`	`ParamAtt` (parcial)
`VarAtt`	`VarAtt` (parcial)
`Stat`	`Map[DataKey, Stat]` (parcial)
`SetStat`	`Map[DataKey, SetStat]` (parcial)
`ParamStat`	`Map[DataKey, ParamStat]` (parcial)
`VarStat`	`Map[DataKey, VarStat]` (parcial)
`ConstraintStat`	`Map[DataKey, ConstraintStat]` (parcial)
`ObjectiveStat`	`Map[DataKey, ObjectiveStat]` (parcial)

Tabla 2.1: Tipo al que se evalúan los distintos componentes de un modelo

Las expresiones de indexación se evalúan a listas de mapas donde cada mapa es un posible subíndice generado por la expresión. Las claves de los mapas anteriores son los los índices introducidos en la expresión de indexación y los valores constituyen el subíndice propiamente (Sección 2.1). A continuación se muestra un ejemplo de evaluación de expresión de indexación con la salida adaptada levemente para mejorar la presentación.

```
scala> val i = dummy
scala> val A = set := List(4,7,9)
scala> val j = dummy
scala> val k = dummy
scala> val B = set := List((1,"Jan"), (1,"Feb"), (2,"Mar"), (2,"Apr"),
     | (3,"May"), (3,"Jun"))
scala> val l = dummy
scala> val C = set := List("a","b","c")

scala> // Evaluación de expresión de indexación
scala> val ie = eval(ind(i in A, (j,k) in B, l in C), ModelData())
ie: List[LinkedMap[DataKey,SimpleData]] =
  List(
    Map(i -> num(4), j -> num(1), k -> str(Jan), l -> str(a)),
    Map(i -> num(4), j -> num(1), k -> str(Jan), l -> str(b)),
    Map(i -> num(4), j -> num(1), k -> str(Jan), l -> str(c)),
    Map(i -> num(4), j -> num(1), k -> str(Feb), l -> str(a)),
    Map(i -> num(4), j -> num(1), k -> str(Feb), l -> str(b)),
    Map(i -> num(4), j -> num(1), k -> str(Feb), l -> str(c)),
    ...
    Map(i -> num(9), j -> num(3), k -> str(Jun), l -> str(b)),
    Map(i -> num(9), j -> num(3), k -> str(Jun), l -> str(c))
  )
```

Las expresiones lineales y los objetos del modelo, solo pueden evaluarse parcialmente, manteniendo gran parte de su estructura. Las expresiones lineales se simplifican, evaluando cualquier subexpresión posible y las sumatorias se expanden a las sumas individuales en función de la evaluación de su expresión de indexación. Los objetos del modelo también son simplificadas y expandidas en base a su dominio. Cada declaración evalúa a un mapa que asocia cada posible subíndice de la declaración con la declaración "instanciada" en esos subíndices. Por ejemplo, dada la siguiente declaración de parámetro del modelo:

```
scala> val N = param := 10
scala> val n = dummy
scala> lazy val fib: ParamStat = {
     |   param(n in (0 to N)) :=
     |     xif (n === 0) {
     |       0
     |     } {
     |       xif (n === 1) {
     |         1
     |       } {
     |         fib(n-1) + fib(n-2)
     |       }
     |     }
     | }
```

su evaluación devuelve el siguiente mapa:

```
scala> val ps = eval(fib, ModelData())
ps: Map[DataKey,ParamStat] =
  Map(
    fib[0]  -> ParamStat(fib,None,None,List(ParamAssign(NumLit(0))))
    fib[1]  -> ParamStat(fib,None,None,List(ParamAssign(NumLit(1))))
    fib[2]  -> ParamStat(fib,None,None,List(ParamAssign(NumLit(1))))
    fib[3]  -> ParamStat(fib,None,None,List(ParamAssign(NumLit(2))))
    fib[4]  -> ParamStat(fib,None,None,List(ParamAssign(NumLit(3))))
    fib[5]  -> ParamStat(fib,None,None,List(ParamAssign(NumLit(5))))
    fib[6]  -> ParamStat(fib,None,None,List(ParamAssign(NumLit(8))))
    fib[7]  -> ParamStat(fib,None,None,List(ParamAssign(NumLit(13))))
    fib[8]  -> ParamStat(fib,None,None,List(ParamAssign(NumLit(21))))
    fib[9]  -> ParamStat(fib,None,None,List(ParamAssign(NumLit(34))))
    fib[10] -> ParamStat(fib,None,None,List(ParamAssign(NumLit(55))))
  )
```

Las sentencias de declaración introducen la dificultad adicional de que pueden ser recursivas, como es el caso del parámetro del modelo anterior (`fib`). Para manejar correctamente ese caso, las "expansiones" de las declaraciones en función de sus dominios son calculadas inicialmente en forma perezosa, y solo son *forzadas* en el momento que se precisa el valor para un subíndice particular, al evaluar una referencia a un parámetro del modelo (clase `ParamRef`). A continuación se muestra el cálculo de la expansión perezosa de un parámetro del modelo.

```
type LazyExpansion[A] = LinkedMap[DataKey, () => A]

@inline private def thunk[A](x: => A): () => A = { () => x }

def expand(param: ParamStat)
  (implicit modelData: ModelData): LazyExpansion[ParamStat] =
  param match {
    case ParamStat(name, alias, domain, atts) =>
      domain match {
        case None =>
          LinkedMap(
            key(name) -> thunk(ParamStat(
              name, alias, none,
              atts.map(eval(_)))))

        case Some(indExpr) =>
          val pairs =
            for {
              localData <- eval(indExpr)
            } yield {
              val k = key(name, localData.values.toList)
              k -> thunk(ParamStat(
                name, alias, none,
                atts.map(eval(_, modelData.plusParams(localData)))))
            }

          LinkedMap(pairs: _*)
      }
  }
```

El método `expand` se define básicamente mediante dos casos que discriminan si el parámetro del modelo `param` tiene un domino o no. El caso más interesante se da cuando la declaración que se pasa a `expand` tiene un dominio definido. Los atributos de un parámetro del modelo pueden hacer referencia a los índices introducidos en la expresión de indexación del domino del parámetro, por lo

que a la hora de evaluarlos es necesario que para cada índice exista un valor asociado en la instancia de `ModelData` que están en contexto. A su vez, el valor asociado a cada índice introducido por el dominio del parámetro, cambia con cada valor posible que puede generar la expresión de indexación. Por ejemplo, en la declaración del parámetro del modelo `fib` anterior, su definición se evalúa once veces, una por cada valor posible producido por su dominio, y cada vez con un valor diferente asociado al índice `n`. Para la expansión de declaraciones de parámetros con dominio, la expresión de indexación correspondiente es evaluada, y por cada elemento de la lista resultado, `localData`, se genera un par compuesto por un `DataKey`, `k`, para asociar el subíndice en `localData` con la declaración, y una función con el resultado de evaluar el parámetro, ahora sin dominio, incorporando `localData` el conjunto de datos en contexto de modo que estén disponibles al evaluar los atributos. Para construir la función anterior, se utiliza un método auxiliar, `thunk`, que convierte un parámetro pasado "por nombre" en una función desde `Unit`. Con la lista de pares `(DataKey, () => ParamStat)` se crea el mapa resultado de la expansión.

La expansión calculada con el método `expand` se utiliza en la evaluación de las referencias a parámetros del modelo. La instancia para referencias a parámetros, `ParamRefEval`, se define utilizando una función auxiliar `from` que permite crear una instancia de `Eval[A,B]` a partir de una función `ModelData => A => B` para evitar colisiones entre los nombres de los métodos `eval`.

```
private def from[A, B](f: ModelData => A => B): Eval[A, B] =
  new Eval[A, B] {
    def eval(a: A)(implicit modelData: ModelData): B =
      f(modelData)(a)
  }

implicit val ParamRefEval: Eval[ParamRef, SimpleData] =
  from(implicit modelData => {
    case ParamRef(param, subscript) =>
      val k = key(param.name, eval(subscript))

      val expParam =
        modelData.paramsExpansion.get(k)
        .orElse(expand(param).get(k).map(_()))
        .err(s"subscript ‘${subscript.shows}’ does not " +
          s"conform to parameter ‘${param.name}’ definition")

      val assignData  = evalAtt(expParam, PFParamAssing)
      val defaultData = evalAtt(expParam, PFParamDefault)

      assignData
        .orElse(modelData.params.get(k))
        .orElse(defaultData)
        .err(s"no data found for ‘$k’")
  })
```

El primer paso para evaluar una referencia a un parámetro del modelo es obtener la instancia de la expansión del parámetro que se corresponde con el subíndice utilizado por la referencia. Para evitar calcular la expansión cada vez, primero se la busca en la instancia de `ModelData` en contexto, en el campo `paramsExpansion` (Sección 2.3.1). Recién si la búsqueda en `ModelData` falla, se calcula y se busca la clave en la expansión. Si la búsqueda en la expasión calculada tiene éxito se calcula la evaluación que había quedado esperando

(llamada a la función `_()`). Si la búsqueda falla nuevamente, se asume que el subíndice de la referencia está mal especificado y aborta la ejecución devolviendo un error. Una vez que se tiene el valor correspondiente al parámetro del modelo en la expansión para el subíndice especificado, se extrae el posible valor asignado en la declaración o su valor por defecto (llamadas a `evalAtt`). El valor a usar para el parámetro puede provenir del atributo asignación (`:=`), del `ModelData` en contexto, o del atributo valor por defecto (`default`).

2.5. Caso de estudio

Para ilustrar el uso del *embedding* se presenta como caso de estudio un problema de planificación de adquisición de cargamentos de combustible, basado en el presentado en Testuri et al. (2019). A medida que se introduce el problema, se muestran los componentes del modelo matemático junto con sus representaciones en el EDSL. En las Tablas 2.2, 2.3 y 2.4 se muestran las entidades del problema, y en el Listado 2.1 su representación en el EDSL. Se observa que a diferencia que en MathProg, en el EDSL los índices (*dummy indices* en la referencia del lenguaje (Makhorin 2016b)) deben ser declarados en forma independiente. En el Apéndice D se muestra la implementación utilizando MathProg.

Para satisfacer la demanda de un combustible en un horizonte de planificación finito, H, con períodos T, se dispone de un conjunto de cargamentos de combustible, C, compuesto por cargamentos ya adquiridos, A, y posibles cargamentos a adquirir, P. Cada cargamento tiene un volumen asociado que no puede dividirse, q, y los cargamentos ya adquiridos tienen un período de recepción, τ. El volumen ya adquirido por período, a, puede calcularse totalizando los volúmenes asociados a los cargamentos que se prevé recibir en cada período

$$a^t := \sum_{c \in A | \tau_c = t} q_c, \quad t \in T$$

```
val a = param(t in T) :=
  sum((c in A) | tau(c) === t) {
    q(c)
  }
```

Los posibles cargamentos a adquirir tienen un tiempo de entrega, γ, que denota el número de períodos que pasan desde que el cargamento es comprado hasta que efectivamente puede ser utilizado. Las decisiones de compra de combustibles, v, deben ser tomadas con suficiente antelación para respetar dicho tiempo de entrega. Por otra parte, un cargamento de combustible no puede ser comprado más de una vez

$$\sum_{t=1}^{H-\gamma_c} v_c^t \leq 1, \quad c \in P$$

```
val singleAcquisition =
  st(c in P) {
    sum(t in (1 to H - gamma(c))) {
      v(c,t)
    } <= 1
  }
```

En este caso, la demanda de combustible, d, se resumen en un único valor por período que se asume conocido, sin embargo, es posible que adquirir un cargamento de P sea más beneficioso que alguno de los ya adquiridos en A, o que las decisiones de compra de cargamentos de A hayan sido realizados con una previsión diferente de la demanda. Por las razones anteriores, es posible

H	horizonte de planificación
T	períodos, $T := 1, \ldots, H$
A	cargamentos ya adquiridos
P	posibles cargamentos a adquirir
C	cargamentos totales, $C := A \cup P$

Tabla 2.2: Conjuntos, y parámetros de los conjuntos, del Caso de estudio

d^t	volumen demandado en el período $t \in T$
$y0$	volumen almacenado inicial
$\underline{y}, \bar{y}$	capacidades mínima y máxima de volumen almacenado
τ_c	período en que se recibe el cargamento ya adquirido $c \in A$
γ_c	tiempo de entrega del cargamento $c \in P$, tal que $0 \leq \gamma_c \leq H - 1$
q_c	volumen del cargamento $c \in C$
ca_c	costo unitario de adquisición del cargamento $c \in C$
cc_c	costo unitario de cancelación del cargamento $c \in A$
h^t	costo unitario de almacenamiento en el período $t \in T$
a^t	volumen ya adquirido que se recibe en el período $t \in T$, $a^t := \sum_{c \in A \mid \tau_c = t} q_c$ (resumen)

Tabla 2.3: Parámetros del Caso de estudio

y^t	volumen almacenado al final del período $t \in T$
v_c^t	decisión de adquirir el cargamento $c \in P$ en el período $t \in T$, tal que $1 \leq t \leq H - \gamma_c$ (binaria)
x_c^t	decisión de cancelar el cargamento $c \in A$ en el período $t \in T$, tal que $1 \leq t \leq \tau_c - 1$ (binaria)
u^t	volumen adquirido en el período $t \in T$ (resumen)
w^t	volumen cancelado en el período $t \in T$ (resumen)

Tabla 2.4: Variables del Caso de estudio

```
// Conjuntos (y parámetros de los conjuntos)
val t  = dummy
val tp = dummy
val H  = param
val T  = set := 1 to H

val c = dummy
val A = set
val P = set
val C = set := A | P

// Parámetros
val d    = param(T)
val y0   = param
val ymin = param
val ymax = param

val tau   = param(A) in T
val gamma = param(P) in (0 to H - 1)
val q     = param(C)

val ca = param(C)
val cc = param(A)
val h  = param(T)

val a = param(t in T) :=
  sum((c in A) | tau(c) === t) { q(c) }

// Variables
val y = xvar(T) >= 0
val v = xvar(c in P, t in (1 to H - gamma(c))).binary
val x = xvar(c in A, t in (1 to tau(c) - 1)  ).binary

val u = xvar(T) >= 0
val w = xvar(T) >= 0
```

Listado 2.1: Entidades del Caso de estudio representadas en el EDSL propuesto

cancelar un cargamento ya adquirido aplicando un costo unitario de cancelación. En forma similar a las decisiones de compra, las decisiones de cancelación de los cargamentos ya adquiridos, x, deben realizarse hasta un período antes del previsto en que sea recibidos, y no pueden tomarse más de una vez para un mismo cargamento

$$\sum_{t=1}^{\tau_c - 1} x_c^t \leq 1, \quad c \in A$$

```
val singleCancellation =
  st(c in A) {
    sum(t in (1 to tau(c) - 1)) {
      x(c,t)
    } <= 1
  }
```

Los volúmenes a adquirir y cancelar por período, u y w, pueden calcularse totalizando las decisiones de compra y cancelación respectivamente

$$u^t = \sum_{c \in P | \gamma_c \leq t-1} q_c v_c^{t-\gamma_c}, \quad t \in T$$

```
val acquiredFuel = st(t in T) {
  u(t) ===
    sum((c in P) | gamma(c) <= t-1) {
      q(c) * v(c, t-gamma(c))
    }
}
```

$$w^t = \sum_{c \in A | \tau_c = t} \left(q_c \sum_{t'=1}^{\tau_c - 1} x_c^{t'} \right), \quad t \in T$$

```
val cancelledFuel = st(t in T) {
  w(t) ===
    sum((c in A) | tau(c) === t) {
      q(c) *
      sum(tp in (1 to tau(c) - 1)) {
        x(c,tp)
      }
    }
}
```

Una vez atendida la demanda, el combustible restante por período, y^t, puede ser almacenado para ser utilizado en el período posterior respetando ciertas cotas, $\underline{y}$ e $\bar{y}$, con un costo asociado

$$\underline{y} \leq y^t \leq \bar{y}, \quad t \in T$$

```
val inventory = st(t in T) {
  dlte(ymin, y(t), ymax)
}
```

Una condición fundamental que debe cumplirse para mantener la consistencia del modelo, es el balance de volumen entre las entradas y las salidas en cada período, que debe tomar en cuenta la existencia de inventario de combustible al inicio de la planificación, $y0$

$$y^{t-1} + a^t + u^t = d^t + w^t + y^t, \quad t \in T, \; y^0 = y0$$

```
val balance0 = st {
  y0     + a(1) + u(1) === d(1) + w(1) + y(1)
}
val balance  = st((t in T) | t > 1) {
  y(t-1) + a(t) + u(t) === d(t) + w(t) + y(t)
}
```

Se observa que el "informalismo" que admite el modelo matemático en la expresión $y^0 = y0$, donde la variable y^t está indexada fuera de rango, no está permitido y debe ser representado de otra forma. La opción tomada en este caso es introducir dos variantes de la restricción de balance, una para el caso $t = 1$ y otra para el resto, cambiando la variable y^t por $y0$ cuando $t = 0$.

De todas las decisiones posibles de compra y cancelación que atiendan la demanda, se desea obtener la de menor costo, siendo ca_c, cc_c y h^t los costos unitarios de adquisición, cancelación y almacenamiento respectivamente

$$\min \sum_{t \in T} \Bigg[\sum_{c \in P | t \leq H - \gamma_c} ca_c q_c v_c^t + \sum_{c \in A | t \leq \tau_c - 1} (cc_c - ca_c) q_c x_c^t + h^t y^t \Bigg]$$

```
val cost = minimize {
sum(t in T) {
  sum((c in P) | t <= H-gamma(c)) {
              ca(c)  * q(c) * v(c,t)
  } +
  sum((c in A) | t <= tau(c)-1) {
    (cc(c) - ca(c)) * q(c) * x(c,t)
  } +
  h(t) * y(t)
}}
```

Como instancia de datos se considera una de las definidas en Testuri et al. (2019) reducida a tres períodos, $H = 3$, y seis cargamentos, $A = \{A1, A2\}$ y $P = \{P1, P2, P3, P4\}$. Para el inventario inicial se tiene $y0 = 20$, y para las cotas inferiores y superiores en cada período $\underline{y} = 0$ e $\bar{y} = 80$, respectivamente. Para cada cargamento el volumen, el costo de adquisición, y el costo de cancelación se muestrean con distribuciones uniformes $q_c \sim U[10, 50]$, $ca_c \sim U[150, 250]$ y $cc_c \sim U[30, 50]$, respectivamente. Los cargamentos ya adquiridos llegan en los períodos $\tau_{A1} = 1$ y $\tau_{A2} = 2$. Todos los cargamentos a adquirir tienen tiempo de entrega $\gamma_c = 1$. El costo unitario de almacenamiento por período se toma en $h^t = 1$. La demanda se muestrea con distribución uniforme $d^t \sim U[10, 50]$.

La instancia de datos puede especificarse en el EDSL desarrollado como se muestra a continuación.

```
val numStages = 3
val AData = List("A1", "A2")
val PData = List("P1", "P2", "P3", "P4")

val mipModelData = mipModel
  .paramData(H, numStages)
  .setData(A, AData)
  .setData(P, PData)
  .paramData(d, 1 -> 35.0, 2 -> 27.5, 3 -> 36.25)
  .paramData(y0  , 20)
  .paramData(ymin,  0)
  .paramData(ymax, 80)
  .paramData(tau, "A1" -> 1, "A2" -> 2)
  .paramData(gamma, PData.map(_ -> 1))
  .paramData(q , uniform(2)( 10,  50)(AData ::: PData))
  .paramData(ca, uniform(3)(150, 250)(AData ::: PData))
  .paramData(cc, uniform(4)( 30,  50)(AData))
  .paramData(h , (1 to numStages).map(_ -> 1))
```

Como se comentó anteriormente, para la especificación de los datos no se intenta simular la sintaxis de MathProg, en lugar de eso, se busca aprovechar las capacidades de Scala y su biblioteca estándar para la manipulación de datos (Sección 2.3). Al ser un lenguaje embebido, es posible generar directamente los datos muestreados con las distribuciones de probabilidad requeridas, utilizando

las capacidades del lenguaje anfitrión. La función `uniform` es una función auxiliar que toma una semilla, los extremos del rango sobre el cual muestrear y una lista, y devuelve una lista de pares asignando a cada valor de la lista un valor muestreado utilizando una distribución uniforme con los parámetros especificados.

El modelo de programación entera completo se muestra a continuación. En el Listado 2.2 se muestra su representación en el EDSL. Se observa que el *embedding* logra replicar en gran medida la sintaxis de las sentencias de declaración de MathProg dentro de Scala.

$$\begin{aligned}
\min \quad & \sum_{t\in T}\Bigg[\sum_{c\in P|t\le H-\gamma_c} ca_c q_c v_c^t \\
& \quad + \sum_{c\in A|t\le \tau_c-1} (cc_c - ca_c) q_c x_c^t \qquad (2.4)\\
& \quad + h^t y^t\Bigg] \\
\text{s. a} \quad & y^{t-1} + a^t + u^t = d^t + w^t + y^t, \qquad t\in T,\ y^0 = y0, \qquad (2.5)\\
& \underline{y} \le y^t \le \bar{y}, \qquad t\in T, \qquad (2.6)\\
& \sum_{t=1}^{H-\gamma_c} v_c^t \le 1, \qquad c\in P, \qquad (2.7)\\
& \sum_{t=1}^{\tau_c-1} x_c^t \le 1, \qquad c\in A, \qquad (2.8)\\
& u^t = \sum_{c\in P|\gamma_c\le t-1} q_c v_c^{t-\gamma_c}, \qquad t\in T, \qquad (2.9)\\
& w^t = \sum_{c\in A|\tau_c=t} \left(q_c \sum_{t'=1}^{\tau_c-1} x_c^{t'}\right), \qquad t\in T, \qquad (2.10)\\
& y^t, u^t, w^t \ge 0, \quad t\in T,\\
& v_c^t \in \{0,1\}, \qquad c\in P,\ t = 1,\dots,H-\gamma_c,\\
& x_c^t \in \{0,1\}, \qquad c\in A,\ t = 1,\dots,\tau_c - 1.
\end{aligned}$$

2.6. Generación de código MathProg

Para poder verificar los modelos e instancias de datos construidos y resolver los modelos utilizando el *solver* de MathProg, se provee dos funciones, `genModel` y `genData`, ambas en el paquete `amphip.sem.mathprog`, que devuelven el texto correspondiente a las secciones de modelo y datos de MathProg, a partir de una instancia de `Model` y de `ModelData`, respectivamente (Secciones 2.2.1 y 2.3.1).

La generación de la sección del modelo se obtiene mediante la definición de instancias para la *type class* `Show` (Scalaz 2018) para cada componente del AST utilizado en el *embedding* (Sección 2.2.1). Las instancias se definen de modo que la representación textual que establecen para cada componente del AST, coincida con su representación en MathProg. Como ejemplo, se muestra a continuación

```
// Función objetivo
val cost = minimize {
  sum(t in T) {
    sum((c in P) | t <= H - gamma(c)) {
      ca(c)  * q(c) * v(c,t)
    } +
    sum((c in A) | t <= tau(c) - 1) {
      (cc(c) - ca(c)) * q(c) * x(c,t)
    } +
    h(t) * y(t)
  }
}

// Restricciones
val balance0 = st {
  y0     + a(1) + u(1) === d(1) + w(1) + y(1)
}
val balance  = st((t in T) | t > 1) {
  y(t-1) + a(t) + u(t) === d(t) + w(t) + y(t)
}

val inventory = st(t in T) { dlte(ymin, y(t), ymax) }

val singleAcquisition  = st(c in P) {
  sum(t in (1 to H - gamma(c))) { v(c,t) } <= 1
}
val singleCancellation = st(c in A) {
  sum(t in (1 to tau(c) - 1)  ) { x(c,t) } <= 1
}

val acquiredFuel = st(t in T) {
  u(t) ===  sum((c in P) | gamma(c) <= t-1) {
              q(c) * v(c, t-gamma(c))
            }
}

val cancelledFuel = st(t in T) {
  w(t) ===  sum((c in A) | tau(c) === t) {
              q(c) * sum(tp in (1 to tau(c) - 1)) { x(c,tp) }
            }
}

// Modelo
val mipModel =
  model(cost,
    balance0, balance, inventory,
    singleAcquisition, singleCancellation,
    acquiredFuel, cancelledFuel
  )
```

Listado 2.2: Función objetivo, restricciones, y declaración de la instancia del modelo del Caso de estudio representadas en el EDSL propuesto

la instancia de la *type class* `Show` para las declaraciones de variables (clase `VarStat`):

```
implicit val VarStatShow: Show[VarStat] = shows {

  case VarStat(name, alias, domain, atts) =>
    val nameAliasDomainShows = {
      val nameShows   = name.some
      val aliasShows  = alias.map(x => (x: SymExpr).shows)
      val domainShows = domain.map(_.shows)

      if (alias.isDefined) {
        List(nameShows, aliasShows, domainShows).flatten.mkString("␣")
      } else {
        List(nameShows, domainShows).flatten.mkString("")
      }
    }

    val attsShows = atts
      .map {
        case VarLTE(expr) => s"<=␣${expr.shows}"
        case VarEq(expr)  => s"=␣${expr.shows}"
        case VarGTE(expr) => s">=␣${expr.shows}"
        case Integer      => "integer"
        case Binary       => "binary"
      }
      .toNel
      .map(_.toList.mkString(",␣"))

    val nameAliasDomainAttsShows =
      List(nameAliasDomainShows.some, attsShows)
        .flatten
        .mkString("␣")

    s"var␣$nameAliasDomainAttsShows;"

}
```

La función `shows` es provista por la biblioteca `Scalaz`, utilizándose dos sobrecargas de la misma. Utilizada como una función independiente, permite construir una instancia de `Show[A]` a partir de una función `A => String`. Utilizada como método en un objeto, devuelve un `String` usando la instancia de `Show` correspondiente que se encuentre en alcance. La instancia calcula las representaciones en MathProg de los componentes de la declaración de variable del modelo y las combina, contemplando que algunos componentes son opcionales. Las instancias de la *type class* `Show` para los componentes del modelo se encuentran en el *trait* `amphip.model.ShowInstances` en `amphip` (Ferrari 2021). La función `genModel` simplemente obtiene el `String` correspondiente al modelo utilizando las instancias anteriores.

La sección de datos se genera a partir de recorridas sobre los mapas con los datos de parámetros y conjuntos en los campos `sets` y `params` de una instancia de `ModelData`. La sintaxis que provee MathProg para la especificación de los bloques de datos de los parámetros del modelo, admite variantes que permiten especificar los valores en forma más compacta si los mismos tienen una estructura particular. En el EDSL solo se optimiza la generación de los bloques de datos si se detecta una estructura tabular en los datos de los parámetros del modelo. En el caso que el parámetro del modelo tenga tres o más subíndices, la estructura

tabular solo se intenta buscar en los primeros dos subíndices. Por más detalles, referirse a la implementación de la función `genData` en `amphip`.

El principal desafío para la generación de la sección del modelo está en "recolectar" todas las declaraciones de objetos del modelo, de modo de poder generar las sentencias de declaraciones correspondiente para cada uno. Como se comentó en la Sección 2.2.2, el valor correspondiente al modelo en el EDSL se construye con la función `model`, por ejemplo, mediante el código:

```
val mipModel =
  model(cost,
    balance0, balance, inventory,
    singleAcquisition, singleCancellation,
    acquiredFuel, cancelledFuel
  )
```

La función `model` recibe como argumentos únicamente a la función objetivo y a las restricciones que deben tomarse en cuenta.

```
def model(obj: ObjectiveStat, constr: ConstraintStat*): Model = {
  val refs = constr.flatMap(collectStat(_)) ++ collectStat(obj)
  val distinctRefs = refs.distinct.toList

  val (setOrParam, vars, ctrs) = group(distinctRefs)

  Model(setOrParam ::: vars ::: ctrs)
}
```

Para poder generar el modelo MathProg, es necesario identificar en dichas sentencias todas las referencias a declaraciones de entidades y obtener, a partir de estas, las declaraciones de conjuntos, parámetros y variables que deben aparecer también en la sección de modelo de MathProg para que sea válida. En la función `model` se "recolectan" todas las instancias de la clase `Stat` a las que se hace referencia en cada restricción y en la función objetivo, mediante la función `collectStat`.

```
private def collectStat[A](in: A)
    (implicit Collect: Collect[A]): List[Stat] = {
  val pf: PartialFunction[Any, Stat] = {case s: Stat => s}
  collect(in, pf)
}
```

Dicha función es aplicada sobre cada restricción utilizando la función `flatMap` sobre la secuencia de restricciones, `constr`, para obtener la lista de completa de instancias de `Stat` referenciadas en las mismas. `flatMap` es el nombre que se da en Scala a la función que encadena cómputos sobre mónadas (Wadler 1995), llamada `bind` en otros lenguajes como Haskell (Marlow et al. 2010). Luego de obtener la lista de todas las sentencias a las que se hace referencia, en la función `model` se eliminan los posibles repetidos y se reagrupan y ordenan las sentencias para que el modelo MathProg resultante quede más legible.

El grueso del trabajo realizado por la función `model` es realizado por la función `collectStat`, la que, por su parte, hace uso de la función `collect`. La función `collect` es implementada mediante un objeto con un método `apply` y una *type class*.

```
object collect {
  trait Collect[A] {
    def collect[B](in: A, matching: PartialFunction[Any, B])
      (implicit visited: Set[Stat]): List[B]
  }

  def apply[A, B](in: A, matching: PartialFunction[Any, B])
    (implicit Collect: Collect[A]): List[B] =
    Collect.collect(in, matching)(Set.empty)

  // instancias de la type class ...
}
```

La función `collect` permite recolectar en una lista todos los valores que cumplen determinada *función parcial* de Scala. El método `apply` toma un valor, `in`, del tipo paramétrico `A`, una función parcial desde `Any` al tipo paramétrico `B`, y devuelve una lista resultado de aplicar la función parcial a todos los componentes de `in` para los que la misma está definida. La *type class* `Collect` restringe los posibles tipos sobre los que se puede aplicar la función anterior.

La firma del método `apply` del objeto `collect` difiere de la del método `collect` de la *type class* `Collect` en que este último recibe, además, un conjunto de sentencias que representan las declaraciones que ya fueron visitadas hasta el momento. Dicho parámetro adicional se utiliza para evitar ciclos infinitos al recorrer declaraciones recursivas, y se recibe en forma implícita para evitar tener que pasarlo manualmente, ya que en la gran mayoría de los casos no hace falta consultarlo o actualizarlo.

Como ejemplo de la definición de las instancias para la *type class* `Collect`, se muestran las definiciones para las declaraciones de parámetros del modelo (clase `ParaStat`) y para las referencias a parámetros (clase `ParamRef`). Estas instancias permiten ver también el uso del conjunto de declaraciones visitadas.

```
implicit val ParamStatCollect: Collect[ParamStat] =
  new Collect[ParamStat] {
    def collect[B](x: ParamStat, pf: PartialFunction[Any, B])
      (implicit visited: Set[Stat]): List[B] = x match {

      case ParamStat(_, _, domain, atts) =>
        val newVisited = visited + x
        domain.toList.flatMap(collect_(_, pf, newVisited)) ++
          atts.flatMap(collect_(_, pf, newVisited)) ++
          pf.lift(x).toList

    }
  }
```

En el caso de la instancia para declaraciones de parámetros del modelo, `ParamStatCollect`, el conjunto de declaraciones visitadas, `visited`, es actualizado agregando la declaración que se está recorriendo, parámetro `x`, construyendo un nuevo conjunto, `newVisited`, el cual es pasado explícitamente a las llamadas a `collect_` en las subexpresiones que componen la declaración. Las subexpresiones de la declaración son las únicas que pueden generar ciclos si hacen referencia a la misma. La función `collect_` es una función auxiliar que permite invocar a `collect` especificado explícitamente al conjunto de declaraciones visitadas pero recibiendo implícitamente la instancia de la `Collect` requerida. La lista resultado se obtiene concatenando las listas obtenidas al aplicar la

función sobre los componentes de la declaración y sobre la propia declaración, `x`. En este último caso se aprovecha el método `lift` de las funciones parciales, que permite convertir funciones parciales de tipo `A => B` en funciones totales de tipo `A => Option[B]`.

El dominio opcional de la declaración de parámetros del modelo se convierte primero en una lista que tiene cero o un elementos, para luego aplicarle la función `flatMap` pasándole como argumento la función `collect_` *aplicada parcialmente* en la función parcial, `pf`, y la lista de declaraciones visitadas actualizada, obteniendo una función de tipo `A => List[B]` que es el esperado por `flatMap`. El esquema anterior es utilizado también con la lista de atributos, y en general para procesar listas o valores opcionales convertidos a listas de cero o un elemento.

En el caso de la instancia para referencias a parámetros, `ParamRefCollect`, el conjunto de declaraciones visitadas es utilizado para determinar si se debe recorrer el parámetro del modelo al que se hace referencia o no, y así evitar ciclos.

```
implicit val ParamRefCollect: Collect[ParamRef] =
  new Collect[ParamRef] {
    def collect[B](x: ParamRef, pf: PartialFunction[Any, B])
      (implicit visited: Set[Stat]): List[B] = x match {

      case ParamRef(param, subscript) =>
        val base =
          if (visited.contains(param)) Nil else collect_(param, pf)
        base ++ subscript.flatMap(collect_(_, pf)) ++ pf.lift(x).toList

    }
  }
```

El resultado final con todos los valores recolectados se obtiene siguiendo a grandes rasgos el mismo esquema que para la instancia de declaraciones de parámetros del modelo. Se observa que en este caso se utiliza una versión sobrecargada de `collect_` que recibe la lista de visitados implícitamente, ya que no es modificada en la implementación de la instancia.

Capítulo 3

Soporte para programación estocástica

La programación estocástica permite modelar problemas que involucran la toma de decisiones a lo largo del tiempo, intercaladas con eventos aleatorios. En el modelo básico, que considera un único evento aleatorio, las *decisiones de primera etapa* son deterministas y se toman antes de la ocurrencia del evento aleatorio, mientras que las *decisiones de segunda etapa* dependen del evento aleatorio y establecen decisiones compensatorias a tomar luego de la ocurrencia del evento. La formulación clásica del problema de dos etapas, originada en Beale (1955) y Dantzig (1955), se muestra en (3.1). En dicho modelo, ω representa el evento aleatorio, $\xi = \xi(\omega) = (q(\omega), T(\omega), W(\omega), h(\omega))$ representa los parámetros del modelo que son aleatorios, el vector x representa las decisiones de primera etapa, el vector $y(\omega)$ representa las decisiones de segunda etapa, la tupla (c, A, b) representa los parámetros del modelo que son deterministas, y $\mathbb{E}_\xi$ representa el valor esperado respecto a ξ.

$$\begin{aligned} \min \quad & c^T x + \mathbb{E}_\xi[\min q(\omega)^T y(\omega)] \\ \text{s. a} \quad & Ax = b, \\ & T(\omega)x + W(\omega)y(\omega) = h(\omega), \\ & x \geq 0, y(\omega) \geq 0. \end{aligned} \tag{3.1}$$

El modelo (3.1) puede verse como una extensión a los modelos de programación lineal a los que se agrega la posibilidad de que los parámetros del modelo sean aleatorios. En dicho modelo, la dependencia de las variables de decisión con el evento aleatorio indica que las decisiones no tienen por que ser las mismas según las distintas realizaciones de ω (Birge y F. Louveaux 2011). Se observa que el modelo puede incluir variables de decisión enteras manteniendo la misma formulación, cambiando el dominio de las variables que lo requieran.

El modelo anterior puede extenderse para considerar múltiples eventos aleatorios y etapas, manteniendo el mismo esquema. Las decisiones de la primera etapa son deterministas, mientras que las decisiones de las etapas siguientes son aleatorias y deben tomarse luego de la ocurrencia de los eventos aleatorios de los que dependen. La Figura 3.1 ilustra la secuencia de decisiones según la ocurrencia de los eventos aleatorios en el caso multietapa. En este caso la

notación usual varía levemente respecto del caso con dos etapas, utilizándose x^t para representar las decisiones en la etapa t en general, $\xi^1 = (c^1, W^1, h^1)$ para los parámetros deterministas y $\xi^t(\omega) = (c^t(\omega), T^t(\omega), W^t(\omega), h^t(\omega))$ para los parámetros aleatorios. En (3.2) se muestra el modelo (3.1) extendido a múltiples etapas considerando un horizonte temporal fijo, H (se omiten los transpuestos para evitar notación excesiva).

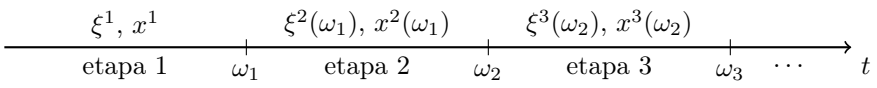
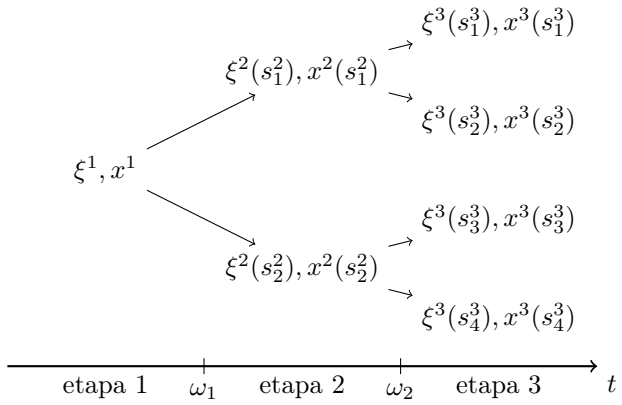


Figura 3.1: Secuencia de decisiones y etapas según la realización de los eventos aleatorios

$$
\begin{aligned}
\min \quad & c^1x^1 + \mathbb{E}_{\xi^2}[\min c^2(\omega_1)x^2(\omega_1) + \cdots + \mathbb{E}_{\xi^H}[\min c^H(\omega_{H-1})x^H(\omega_{H-1})]\cdots] \\
\text{s. a} \quad & W^1x^1 = h^1, \\
& T^2(\omega_1)x^1 + W^2(\omega_1)x^2(\omega_1) = h^2(\omega_1), \\
& T^3(\omega_2)x^2(\omega_1) + W^3(\omega_2)x^3(\omega_2) = h^3(\omega_2), \\
& \vdots \\
& T^H(\omega_{H-1})x^{H-1}(\omega_{H-2}) + W^H(\omega_{H-1})x^H(\omega_{H-1}) = h^H(\omega_{H-1}), \\
& x^1 \geq 0, x^t(\omega_{t-1}) \geq 0, \quad t = 2, \ldots, H.
\end{aligned}
\tag{3.2}
$$

Los eventos aleatorios suelen asumirse con un número finito de realizaciones, en cuyo caso los posibles valores que puede tomar ξ se denominan *escenarios*. Las propiedades que surgen de tener parámetros aleatorios con un conjunto finito de realizaciones simplifican el tratamiento teórico y computacional del problema y permiten aplicar algoritmos de resolución específicos (Birge y F. Louveaux 2011). Los escenarios se pueden obtener, por ejemplo, de la discretización de distribuciones de probabilidad continuas, en base a información histórica, o ser construidos a través del juicio de expertos (Birge y F. Louveaux 2011; Shapiro y Philpott 2007).

La ocurrencia de los eventos aleatorios discretos en el tiempo establece un *árbol de escenarios*. Para cada escenario s en la etapa t, se tiene un conjunto de escenarios, S_s^{t+1}, que representa las realizaciones de los parámetros aleatorios en la etapa $t+1$ condicionadas a la ocurrencia del escenario s en la etapa t. Los escenarios de la etapa t, se corresponden con los caminos posibles desde la raíz del árbol de escenarios hasta el nivel t del mismo. El árbol siguiente es un ejemplo de árbol de escenarios con tres etapas y dos alternativas por etapa, siendo $\xi^t(s_k^t)$ y $x^t(s_k^t)$ los valores de los parámetros aleatorios y las decisiones a tomar en el escenario k de la etapa t, respectivamente.



Se observa que en la tercera etapa se manejan dos alternativas pero constituyen cuatro escenarios distintos porque están condicionadas a las dos alternativas posibles consideradas para la etapa anterior.

El modelado basado en escenarios permite tratar al modelo multietapa (3.2) como determinista utilizando la denominada *forma extensiva*, en la que se incorporan variables para decisiones compensatorias para todas las posibles realizaciones de los eventos aleatorios. El modelo (3.3) muestra el modelo (3.2) utilizando la formulación extensiva y escenarios. En dicho modelo, el conjunto $S^t = s_1^t, \ldots, s_{K^t}^t$ contiene los escenarios de la etapa t, siendo K^t la cantidad de escenarios de la etapa, el parámetro $\pi(s)$ es la probabilidad del escenario s, y el conjunto S_s^t es el conjunto de escenarios del período t que son descendientes del escenario s del período $t-1$.

$$
\begin{aligned}
\min \quad & c^1x^1 + \sum_{t=2}^{H} \sum_{s \in S^t} \pi(s) c^t(s) x^t(s) \\
\text{s. a} \quad & W^1x^1 = h^1, \\
& T^2(s)x^1 + W^2(s)x^2(s) = h^2(s), \qquad s \in S^2, \\
& T^3(s)x^2(s') + W^3(s)x^3(s) = h^3(s), \qquad s' \in S^2, s \in S_{s'}^3, \\
& \vdots \\
& T^H(s)x^{H-1}(s') + W^H(s)x^H(s) = h^H(s), \quad s' \in S^{H-1}, s \in S_{s'}^H, \\
& x^1 \geq 0, x^t(s) \geq 0, \;\; t = 2, \ldots, H, s \in S^t.
\end{aligned}
\tag{3.3}
$$

Para la implementación computacional de los modelos de programación estocástica multietapa basada en escenarios, es beneficioso poder contar con lenguajes de dominio específico para facilitar la tarea de construcción del árbol de escenarios, la declaración de parámetros y variables estocásticas, y la definición de los valores de los parámetros estocásticos por cada escenario. Asimismo, el soporte del lenguaje de modelado puede asistir en el cálculo de medidas de valor sobre los modelos.

Este capítulo describe una extensión del EDSL descrito en el Capítulo 2, para trabajar en forma amigable con problemas de programación estocástica multietapa basada en escenarios. La Sección 3.1 profundiza en algunas características

adicionales de los modelos de programación estocástica multietapa basada en escenarios que son relevantes para este trabajo. La Sección 3.2 se concentra en el soporte del EDSL desarrollado para la especificación del modelo, los valores para los parámetros estocásticos, y la construcción del árbol de escenarios, incluyendo las transformaciones que se realizan al modelo previo a su resolución. En la Sección 3.3 se revisita el Caso de estudio (Sección 2.5), modelándolo como un problema de programación estocástica. Finalmente, en la Sección 3.4 se calculan medidas de valor para el Caso de estudio extendido, utilizando el soporte provisto por el EDSL para ese fin.

3.1. Programación estocástica multietapa basada en escenarios

En esta sección se describen algunos conceptos adicionales relacionados con el modelado basado en escenarios para problemas multietapa que son relevantes para el desarrollo del EDSL.

3.1.1. Formulación con escenarios separados

La formulación utilizada en el modelo (3.3) se denomina *extensiva*, porque incorpora variables para decisiones compensatorias por cada posible realización de los eventos aleatorios. Como los parámetros aleatorios tienen soporte finito, el problema obtenido utilizando la formulación extensiva es determinista.

Es posible definir una formulación extensiva alternativa a la utilizada en el modelo (3.3), en la que se trabaja solamente con escenarios finales, es decir, se trabaja con único conjunto de escenarios, S, equivalente a S^H. La formulación alternativa anterior se denomina con *escenarios separados* y se muestra a continuación (se omiten los transpuestos para evitar notación excesiva):

$$\begin{aligned} \min \quad & c^1x^1 + \sum_{t=2}^{H}\sum_{s\in S}\pi(s)c^t(s)x^t(s) \\ \text{s. a} \quad & W^1x^1 = h^1, \\ & T^2(s)x^1 + W^2(s)x^2(s) = h^2(s), \qquad s \in S, \\ & T^t(s)x^{t-1}(s) + W^t(s)x^t(s) = h^t(s), \quad t = 3,\dots,H, s \in S, \\ & x^1 \geq 0, x^t(s) \geq 0, \;\; t = 2,\dots,H, s \in S. \end{aligned} \tag{3.4}$$

El modelo (3.4) es más simple que el (3.3) y tiene la ventaja de que es sencillo "navegar" por los escenarios que están en un mismo camino del árbol de escenarios. Por ejemplo, dado un escenario $s \in S$ en la etapa $t \in T$, para acceder al escenario predecesor n etapas hacia atrás simplemente se indexa en $t - n$, manteniendo fijo el escenario. Se observa que en el modelo (3.4), las decisiones de una etapa solo pueden depender de las decisiones de la etapa inmediatamente anterior. La estructura de dicho modelo puede relajarse para que en cada etapa sea posible depender de las decisiones en cualquiera de las etapas anteriores, pudiéndose siempre recuperar la estructura original copiando decisiones de una etapa a la siguiente utilizando variables auxiliares, de modo que estén disponibles en la etapa adecuada. En la práctica es habitual depender de decisiones de

etapas anteriores más allá de la inmediatamente anterior, lo que es sencillo de modelar si se utiliza la formulación con escenarios separados, como se comentó anteriormente.

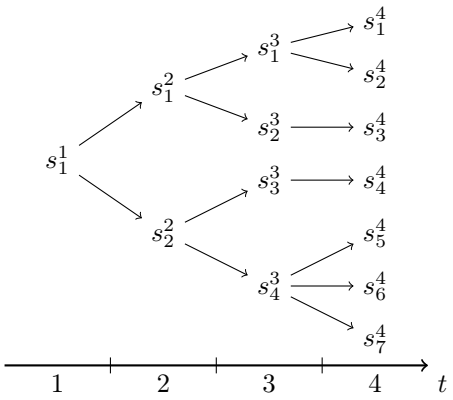


Figura 3.2: Ejemplo de árbol de escenarios con estructura arbitraria

3.1.2. No anticipatividad de las decisiones

La formulación con escenarios separados tiene el problema que, en cada etapa, cada variable de decisión puede tener tantos valores diferentes como escenarios finales existan, es decir, a priori se permite que en una etapa se tomen distintas decisiones para diferentes escenarios finales, cuando en realidad conceptualmente los escenarios son el mismo. Por ejemplo, si se toma como base el árbol de la Figura 3.2, en la formulación con escenarios separados en la etapa 2 las variables $x^2(s)$ pueden tomar valores diferentes para los escenarios finales s_1^4, s_2^4 y s_3^4, cuando en esa etapa dichos escenarios se corresponden con un único escenario, s_1^2, y por lo tanto las decisiones tomadas deberían ser las mismas. En el problema descrito anteriormente, las decisiones que se toman en una etapa pueden *anticiparse* al futuro, lo que viola la estructura general del problema de programación estocástica, donde una decisión solo puede depender de los eventos aleatorios pasados (Figura 3.1). Para corregir dicho problema, en la formulación con escenarios separados se introducen restricciones llamadas de *no anticipatividad*, que obligan a que las variables de decisión tomen el mismo valor para distintos escenarios finales si conceptualmente se corresponden con el mismo escenario en una etapa dada. La Figura 3.3 muestra el árbol de la Figura 3.2 utilizando escenarios separados, marcando los escenarios que deben tener el mismo valor por etapa para conservar la no anticipatividad.

Para poder especificar las restricciones de no anticipatividad es necesario disponer de información sobre la estructura del árbol de escenarios en el modelo, en general mediante la definición de parámetros o conjuntos auxiliares. Una opción es definir un parámetro $anc(s,t)$ que representa el escenario *ancestro* del escenario final $s \in S$ en la etapa $t \in T$, siendo $anc(s,H) = s$ y $anc(s,1) = 1$ para todo $s \in S$, es decir, todos los escenarios son "ancestros de si mismos" y en la primera etapa conceptualmente hay un único escenario, la raíz del árbol. A continuación se muestra una posible especificación del parámetro auxiliar anc

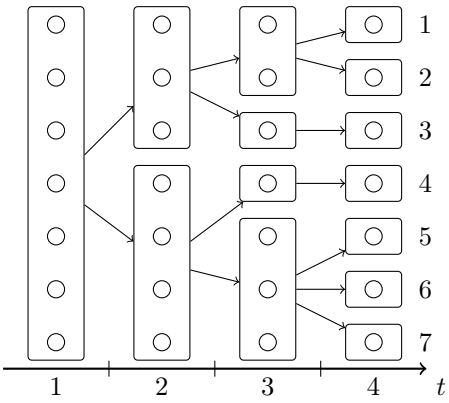


Figura 3.3: Árbol de la Figura 3.2 representado con escenarios separados. Los rectángulos indican los escenarios que deben tener el mismo valor por etapa para mantener la no anticipatividad

para el árbol de escenarios de la Figura 3.3:

$$anc(s,t) = \begin{cases} 1 & \text{si } t = 1, \\ 1 & \text{si } t = 2 \text{ y } s \in \{1,2,3\}, \\ 2 & \text{si } t = 2 \text{ y } s \in \{4,5,6,7\}, \\ 1 & \text{si } t = 3 \text{ y } s \in \{1,2\}, \\ 2 & \text{si } t = 3 \text{ y } s \in \{3\}, \\ 3 & \text{si } t = 3 \text{ y } s \in \{4\}, \\ 4 & \text{si } t = 3 \text{ y } s \in \{5,6,7\}, \\ s & \text{si } t = 4. \end{cases}$$

Utilizando el parámetro $anc(s,t)$ es posible definir las restricciones de no anticipatividad

$$x^t(s) = x^t(s'), \quad t = 2,\ldots,H, \;\; s,s' \in S, \;\; anc(s,t) = anc(s',t). \tag{3.5}$$

La formulación con escenarios separados también afecta a los parámetros estocásticos, ya que en forma similar a lo que sucede con la variables, a priori pueden tener valores diferentes para escenarios que conceptualmente son el mismo. En el caso de los parámetros del modelo no hay un equivalente a las restricciones de no anticipatividad, siendo responsabilidad del modelador, o del sistema de modelado utilizado, que los valores de los parámetros especificados sean consistentes con el árbol de escenarios que se busca representar.

3.1.3. Escenarios básicos y probabilidad

Los escenarios del árbol de escenarios son caminos que parten desde la raíz hasta un determinado nivel o etapa. En ese sentido, los escenarios pueden verse como secuencias de *escenarios básicos* que son los nodos del árbol que constituyen dichos caminos. Cada escenario básico tiene una probabilidad que está condicionada a la ocurrencia del escenario predecesor en el árbol de escenarios.

La probabilidad de un escenario es el producto de la *probabilidad condicional* de cada escenario básico que lo compone, la que se denomina también *probabilidad camino*. La probabilidad condicional de todos los escenarios básicos condicionados a un mismo escenario de la etapa anterior debe sumar uno, al igual que la suma de la probabilidad camino de todos los escenarios en una etapa. La figura 3.4 muestra el árbol de escenarios de la Figura 3.2 con una posible asignación de probabilidades camino y condicionales en cada escenario y escenario básico.

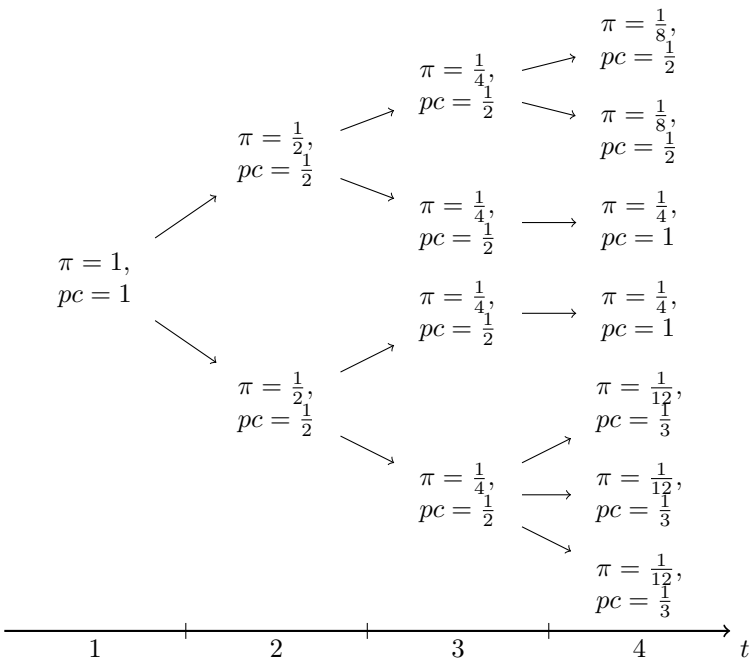


Figura 3.4: Posible asignación de probabilidades al árbol de escenarios de la Figura 3.2. Se indica con π la probabilidad camino de cada escenario, y con pc la probabilidad condicional de cada escenario básico

3.1.4. Medidas de valor para modelos multietapa

Se describen dos medidas de valor para los modelos de programación estocástica y sus soluciones: el *valor esperado de la información perfecta* (*expected value of perfect information*, EVPI) y el *valor de la solución estocástica* (*value of the stochastic solution*, VSS). Las definiciones utilizadas se basan en parte en las descritas en Birge y F. Louveaux (2011) y Escudero et al. (2007).

Valor esperado de la información perfecta (EVPI)

Sea ξ el parámetro aleatorio con soporte finito cuyas realizaciones, $\xi(s) = \xi^1(s), \ldots, \xi^H(s)$ con $\xi^t(s) = (c^t(s), T^t(s), W^t(s), h^t(s)), t = 1, \ldots, H$, se corres-

ponden con los escenarios $s \in S$, se define

$$\begin{aligned} RP = \min & \sum_{s \in S} \pi(s) \sum_{t=1}^{H} c^t(s) x^t(s) \\ \text{s. a} \quad & W^1(s) x^1(s) = h^1(s), \\ & T^t(s) x^{t-1}(s) + W^t(s) x^t(s) = h^t(s), \quad t = 2, \ldots, H, s \in S, \\ & x^t(s) \geq 0 \text{ no anticipativas}, \quad t = 1, \ldots, H, s \in S. \end{aligned} \tag{3.6}$$

El problema RP (con valor óptimo homónimo) es el problema estocástico, denominado en inglés *recourse problem*, y su solución óptima es x^*. El modelo (3.6) es equivalente al modelo con escenarios separados (3.4), la única diferencia es que las decisiones de primera etapa también dependen del escenario, lo que conceptualmente no es significativo en la medida que las restricciones de no anticipatividad estén bien definidas y los valores de los parámetros del modelo sean consistentes. Tener las decisiones de primera etapa "separadas" por escenario al igual que las decisiones de las otras etapas, permite simplificar un poco las definiciones.

Dado el problema RP, el valor "esperar y ver" (*wait and see*, WS) se define como la esperanza del valor óptimo del problema RP por cada escenario por separado, siendo $x^*(s)$ la solución óptima encontrada para cada escenario s de S. Al considerar cada escenario por separado, lo que sucede es que efectivamente se está permitiendo que las decisiones de las primeras etapas se anticipen al futuro, pudiéndose tomar distintas decisiones según el escenario final que se trate. Las soluciones óptimas por cada escenario, $x^*(s)$, pueden violar las restricciones de no anticipatividad. El valor WS puede ser calculado de diversas maneras, en particular, si se tiene las decisiones de primera etapa separadas por escenario, como en el modelo (3.6), el valor WS se corresponde con resolver el problema RP sin considerar las restricciones de no anticipatividad.

El valor esperado de la información perfecta se define como la diferencia entre el valor del problema RP y el valor WS

$$EVPI = RP - WS.$$

Como se está trabajando con problemas de minimización, siempre se cumple que $WS \leq RP$ ya que el problema WS es un problema relajado respecto a RP, la solución x^* toma en cuenta las restricciones para todos los escenarios mientras que las soluciones $x^*(s)$ consideran un único escenario a la vez.

El EVPI da una idea de cuanto se podría llegar a ganar si se tuviera mejor información (idealmente perfecta) sobre el futuro, o, considerándolo del lado opuesto, de cuanto es la pérdida de beneficio producto de la presencia de incertidumbre.

Valor de la solución estocástica (VSS)

Sea $\bar{\xi}^t = (\bar{c}^t, \bar{T}^t, \bar{W}^t, \bar{h}^t)$ el valor esperado de ξ en la etapa t, se define el *problema de valor esperado* (*expected value problem*, EV) en la etapa 1 como:

$$\begin{aligned} EV^1 = \min & \sum_{s\in S} \pi(s) \left[c^1(s)x^1(s) + \sum_{t=2}^{H} \bar{c}^t(s)x^t(s) \right] \\ \text{s. a} \quad & W^1(s)x^1(s) = h^1(s), \\ & \bar{T}^t(s)x^{t-1}(s) + \bar{W}^t(s)x^t(s) = \bar{h}^t(s), \quad t = 2,\dots,H, s \in S, \\ & x^t(s) \geq 0 \text{ no anticipativas}, \ t = 1,\dots,H, s \in S, \end{aligned} \tag{3.7}$$

donde $\bar{\xi}^t(s) = \bar{\xi}^t$ para todo $s \in S$.

El problema EV^1 conceptualmente reduce todos los escenarios a un único escenario en el que se utiliza el valor medio para cada parámetro estocástico. La dependencia de los parámetros y variables con los escenario en el modelo (3.7) no tiene efecto en la medida que las restricciones de no anticipatividad y los valores de los parámetros sean consistentes. Como ejemplo, en la Figura 3.5 se muestra el árbol correspondiente al problema EV^1 si el árbol de escenarios del problema RP es el de la Figura 3.3.

La solución óptima del problema EV se define como $\bar{x}$ y se denomina la *solución de valor esperado*. En el caso del problema EV^1 solo interesan las decisiones de la primera etapa, $\bar{x}^1$.

Para las etapas $t = 2,\dots,H-1$, el problema de valor esperado se define como:

$$\begin{aligned} EV^t = \min & \sum_{s\in S} \pi(s) \left[\sum_{t'=1}^{t-1} c^{t'}(s)\bar{x}^{t'}(s) + c^t(s)x^t(s) + \sum_{t'=t+1}^{H} \bar{c}_t^{t'}(s)x^{t'}(s) \right] \\ \text{s. a} \quad & W^1(s)\bar{x}^1(s) = h^1(s), \\ & T^{t'}(s)\bar{x}^{t'-1}(s) + W^{t'}(s)\bar{x}^{t'}(s) = h^{t'}(s), \quad t' = 2,\dots,t-1, s \in S, \\ & T^t(s)\bar{x}^{t-1}(s) + W^t(s)x^t(s) = h^t(s), \quad s \in S, \\ & \bar{T}_t^{t'}(s)x^{t'-1}(s) + \bar{W}_t^{t'}(s)x^{t'}(s) = \bar{h}_t^{t'}(s), \quad t' = t+1,\dots,H, s \in S, \\ & x^{t'}(s) \geq 0 \text{ no anticipativas}, \ t' = t,\dots,H, s \in S, \end{aligned} \tag{3.8}$$

donde $\xi^t(s) = (c^t(s), T^t(s), W^t(s), h^t(s))$ son los parámetros estocásticos del problema RP, $\bar{x}^t(s)$ es la solución en la etapa t del problema EV^t, y $\bar{\xi}_t^{t'}(s) = (\bar{c}_t^{t'}(s), \bar{T}_t^{t'}(s), \bar{W}_t^{t'}(s), \bar{h}_t^{t'}(s))$ es el valor esperado de los parámetros estocásticos en la etapa $t' > t$ a partir de los escenarios de la etapa t.

Cada problema EV^t utiliza las soluciones de valor esperado de las etapas anteriores. Los árboles de escenarios correspondientes a cada problema EV^t se corresponden con el árbol de escenarios del problema RP hasta la etapa t, y tienen un único escenario a partir de la etapa $t+1$, de modo de ser consistentes con los valores de los parámetros $\bar{\xi}_t^{t'}(s)$. Como ejemplo, en la Figura 3.5 se muestran los árboles de escenarios correspondientes a los problemas EV^2 y EV^3 para el árbol de la Figura 3.3.

Con las soluciones de valor esperado, puede calcularse el *resultado esperado de usar la solución de valor esperado* (*expected result of using the expected value*

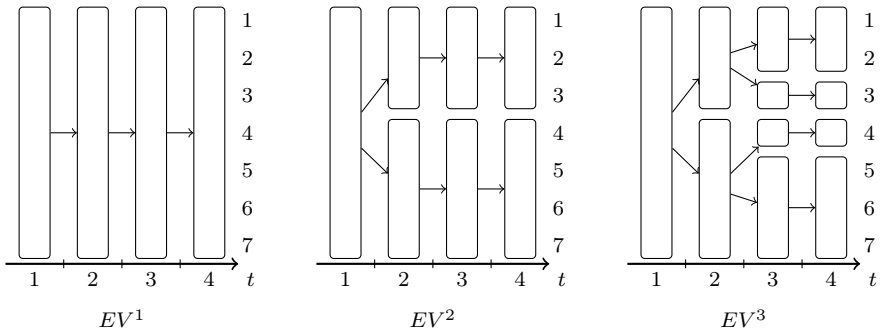


Figura 3.5: Árboles correspondientes a problemas de valor esperado para etapas 1, 2 y 3, tomando como base el árbol de la Figura 3.3

solution, EEV) para cada etapa $t = 1, \ldots, H-1$:

$$\begin{aligned} & EEV^t = RP \\ & \text{s. a} \quad x^{t'}(s) = \bar{x}^{t'}(s), \quad t' = 1, \ldots, t, \quad s \in S \end{aligned}$$

Los valores EEV^t permiten valorar las soluciones de valor esperado hasta la etapa t considerando la incertidumbre del problema y permitiendo que se tomen decisiones compensatorias a partir de la etapa $t+1$.

El valor de la solución estocástica se define como la diferencia entre el valor EEV^{H-1} y el valor del problema RP

$$VSS = EEV^{H-1} - RP.$$

En forma similar a lo que sucede con el EVPI, al trabajar en un problema de minimización siempre se cumple que $RP \leq EEV^t$ porque los problemas EEV^t tienen las restricciones adicionales de tener los valores de las decisiones hasta la etapa t fijadas en $\bar{x}^t$.

El VSS representa el beneficio que se consigue por utilizar la solución estocástica sobre la solución de valor esperado.

3.2. Programación estocástica en `amphip`

Como se comentó en la introducción, utilizar lenguajes de dominio específico para programación entera mixta con modelos de programación estocástica, presenta tres grandes problemas. En primer lugar es necesario incorporar el control de la no anticipatividad de las decisiones, ya sea en forma implícita o explícita, lo que implica un trabajo extra para el modelador. En segundo término, no se dispone de ningún soporte para la construcción del árbol de escenarios y la especificación de los valores de los parámetros estocásticos en cada nodo del árbol. La definición del árbol de escenarios es una tarea laboriosa que puede simplificarse con el soporte adecuando del lenguaje de modelado, y que muchas veces requiere interacción con otros componentes como paquetes de análisis estadístico o el uso de estructuras de datos y algoritmos específicos. Por último, en la resolución de los problemas no se pueden utilizar algoritmos que aprovechen la estructura particular de los problemas de programación estocástica.

Los dos primeros problemas mencionados en el párrafo anterior pueden resolverse, o aminorarse, con el soporte adecuado del lenguaje de modelado utilizado. El soporte para programación estocástica en el EDSL desarrollado, se incorpora como una "capa de programación estocástica" por encima de las capas de modelo y de datos, siguiendo un esquema de trabajo análogo al utilizado en esos casos (Secciones 2.2 y 2.3).

En el EDSL propuesto se trabaja con escenarios separados en forma similar a otros lenguajes, tales como SAMPL (Valente et al. 2009), StAMPL (Fourer y Lopes 2009), AIMMS (Bisschop 2006) y PySP (Watson et al. 2012). Para modelar los escenarios separados se manejan dos conjuntos que representan las etapas y los escenarios finales, los que pueden ser cualquier conjunto del modelo con la condición de que sus datos sean especificados únicamente utilizando la funcionalidad provista por la capa de programación estocástica (Sección 3.2.2). Es posible resolver el modelo estocástico convirtiéndolo primero en un modelo MathProg equivalente, para lo que se generan restricciones de no anticipatividad explícitas automáticamente (Sección 3.2.5).

La estructura del árbol de escenarios se define programáticamente ajustando progresivamente la estructura del árbol con diferente nivel de detalle (Sección 3.2.2). La probabilidad de los escenarios se especifica al mismo tiempo que se incorporan los escenarios en el árbol de escenarios. Como es un EDSL, es posible definir funciones auxiliares para tareas repetitivas como por ejemplo la definición de un árbol perfecto n-ario (Sección 3.2.2), o para especificar las probabilidades de los escenarios finales y propagarla automáticamente hacia los escenarios intermedios (Sección 3.2.3).

Los parámetros y variables estocásticas son aquellas que dependen de los conjuntos que representan las etapas y los escenarios finales. Los datos de los parámetros estocásticos se definen utilizando métodos específicos provistos por la capa de programación estocástica, que permiten asignarle valores con diferente nivel de granularidad, y reutilizan parte de la funcionalidad de la capa de datos (Sección 3.2.4).

Las probabilidades de los escenarios finales y las demandas pueden generarse con las distribuciones de probabilidad requeridas utilizando las capacidades del lenguaje anfitrión, y asignarse utilizando la funcionalidad provista por la capa de programación estocástica que, al igual que la capa de datos, permite que los datos se especifiquen desde una colección generada por separado (Sección 3.3).

En las secciones siguientes se describe cada componente de la capa de programación estocástica del EDSL en detalle.

3.2.1. Estructuras de datos

Para mantener los datos de los parámetros estocásticos en memoria, se define una clase `StochData` que contiene los datos estocásticos especificados hasta el momento, en forma análoga a `ModelData` en la capa de datos (Sección 2.3.1). La clase `StochData` incluye además la especificación del árbol de escenarios hasta el momento.

```
case class Stage(name: String)
case class BasicScenario(name: String)
type Scenario = List[BasicScenario]
// en amphip.data: type ParamStatData = LinkedMap[DataKey, SimpleData]

case class StochData private (
    stages: List[Stage],
    basicScenarios:
      LinkedMap[Stage, LinkedMap[BasicScenario, Rational]],
    customScenarios:
      LinkedMap[Scenario, LinkedMap[BasicScenario, Rational]],
    deletedScenarios: LinkedMap[Scenario, Set[BasicScenario]],
    defaults: LinkedMap[ParamStat, ParamStatData],
    basicData:
      LinkedMap[Stage,
        LinkedMap[BasicScenario, LinkedMap[ParamStat, ParamStatData]]],
    scenarioData:
      LinkedMap[Scenario, LinkedMap[ParamStat, ParamStatData]]) {

  // métodos auxiliares ...
}
```

Para la especificación de las probabilidades se utiliza el tipo `Rational` de `Spire` (Osheim 2012; Osheim y Switzer 2018), que implementa números racionales eficientes. Dicha biblioteca provee un *interpolador de strings*, `r`, para representar "literales de números racionales" en forma compacta utilizando *strings procesados* de Scala (Odersky et al. 2019), los que son simplificados en tiempo de compilación utilizando macros. Por ejemplo, el *string* procesado `r"2/6"` utiliza el interpolador `r` para construir el valor `Rational(1,3)`, simplificando el número en tiempo de compilación. Se utilizan números racionales para representar las probabilidades en lugar de punto flotante para evitar perder precisión prematuramente, siempre y cuando el costo en eficiencia a pagar se considere razonable.

Para la representación del modelo estocástico se crea un nuevo *trait* similar a `ModelWithData` (Sección 2.3.1), denominado `StochModel`, que agrupa un "modelo con datos" y un `StochData`, sobre el que se define funcionalidad mediante *extension methods*.

```
sealed trait StochModel {
  def model    : ModelWithData
  def stochData: StochData
  def S : SetStat
  def pi: ParamStat
}
case class MultiStageStochModel(
  model: ModelWithData,
  stochData: StochData,
  T: SetStat,
  S: SetStat,
  pi: ParamStat,
  naMode: NAMode) extends StochModel

sealed trait NAMode
case class STAdapter(T: SetStat, S: SetStat) extends NAMode
```

El modelo estocástico se define como un *trait* para habilitar la existencia de diferentes tipos de modelos estocásticos, por ejemplo de dos etapas, sobre los que la implementación de algunas funciones puede no estar definida o ser diferente.

En StochModel se provee acceso a las entidades que deben estar disponibles para cualquier modelo estocástico, y en la subclase MultiStageStochModel se agregan las entidades propias de los modelos multietapa, en particular el conjunto que representa las etapas y la estrategia con la que se debe manejar la no anticipatividad (*trait* NAMode). La estrategia a usar para manejar la no anticipatividad se define como un *trait* para permitir múltiples estrategias, siendo STAdapter la estrategia descrita en la Sección 3.2.5.

Para acceder a la funcionalidad de la capa de programación estocástica, se provee un *extension method*, stochastic, que permite convertir un modelo con datos en un modelo estocástico multietapa, provisto que se indiquen cuáles son los conjuntos de etapas y escenarios finales, y cuál es el parámetro del modelo que representa la probabilidad. A continuación se muestra la definición del *extension method* stochastic.

```
implicit class ModelWithDataStochSyntax[M](m: M)
  (implicit conv: M => ModelWithData) {
  def stochastic(
    T: SetStat, S: SetStat, prob: ParamStat): StochModel = {
    val newDataModel = List(S, T, prob) ++: m
    MultiStageStochModel(
      newDataModel, StochData(), T, S, prob, STAdapter(T, S))
  }
}
```

El *extension method* se define para cualquier tipo paramétrico M que sea "convertible" a ModelWithData para permitir que sea invocado también sobre modelos sin datos (clase Model, Sección 2.2.1). La función ++: crea un nuevo modelo con las sentencias especificadas agregadas al inicio. Los datos de los conjuntos de etapas y escenarios finales, y el parámetro del modelo para la probabilidad, pasan a ser manejados por amphip y determinados en función de la definición del árbol de escenarios (Sección 3.2.2).

3.2.2. Árbol de escenarios

En amphip el árbol de escenarios se define mediante una serie de pasos en los que se va refinando su estructura hasta alcanzar la deseada. Como todos los tipos de datos utilizados son inmutables, en cada paso de construcción del árbol se crea un nuevo modelo con el árbol de escenarios definido hasta el momento.

El primer paso para la construcción del árbol de escenarios consiste en definir la lista de etapas del problema estocástico, las que coinciden con los niveles del árbol. Para definir las etapas se provee el método stochStages que recibe un número variable de valores de tipo Stage (Sección 3.2.1), y actualiza los datos del modelo con los mismos. A continuación se muestra un ejemplo de uso de stochStages.

```
val (t1, t2, t3, t4) = (Stage("1"), Stage("2"), Stage("3"), Stage("4"))

val stochModelStages = stochModel.stochStages(t1, t2, t3, t4)
```

En el ejemplo se crean cuatro variables para las etapas con una única asignación de Scala utilizando *pattern matching*.

En el paso siguiente se define lo que en este trabajo se denomina el *generador*, que consiste en especificar *escenarios básicos* por cada etapa junto con su probabilidad condicional (Sección 3.1.3). Para definir el generador se provee el

método `stochBasicScenarios` que recibe una etapa y un número variable de pares `(BasicScenario, Rational)`, que constituyen los escenarios básicos asociados a la etapa en el generador junto con su probabilidad condicional. La siguiente es una posible asignación de escenarios básicos para el generador con sus probabilidades asociadas.

```
val (init, a, b, c) =
    (BasicScenario("init"),
     BasicScenario("a"), BasicScenario("b"), BasicScenario("c"))

val stochModelBS = stochModelStages
  .stochBasicScenarios(t1, init -> r"1")
  .stochBasicScenarios(t2, a -> r"1/2", b -> r"1/2")
  .stochBasicScenarios(t3, a -> r"1/2", b -> r"1/2")
  .stochBasicScenarios(t4, a -> r"1/3", b -> r"1/3", c -> r"1/3")
```

La Figura 3.6 muestra gráficamente el generador definido en el código anterior.

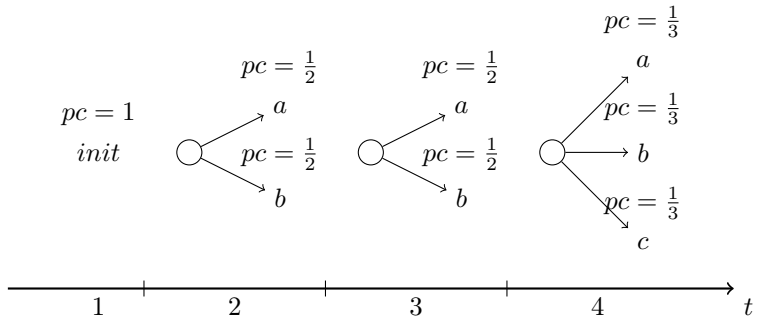


Figura 3.6: Generador con un escenario básico en la 1 inicial, dos en las etapas 2 y 3, y tres en la etapa 4

Se observa que los escenarios básicos del generador no están ligados a un camino particular del árbol de escenarios: los escenarios básicos en cada etapa del generador, pueden ser usados como "los escenarios básicos de la etapa" para cualquier camino posible que llegue a dicha etapa. La propiedad anterior determina que con el generador definido en el ejemplo mostrado anteriormente, el árbol de escenarios asociado al modelo `stochModelBS` sea el de la Figura 3.7.

Una primera ventaja de trabajar con el generador es que rápidamente se puede construir un árbol balanceado especificando una fracción de la información. En el ejemplo anterior se genera un árbol de diecinueve escenarios a partir de la información de ocho escenarios básicos.

El paso final para la construcción del árbol consiste en especificar los ajustes necesarios para conseguir la estructura deseada modificando escenarios específicos. Para definir escenarios en forma detallada se provee el método `stochCustomScenarios`, que recibe un escenario de la etapa t (representado como una lista de escenarios básicos de t elementos), y un número variable de pares `(BasicScenario, Rational)`, que representan los escenarios básicos que le corresponden a dicho escenario en la etapa $t+1$, junto con su probabilidad condicional.

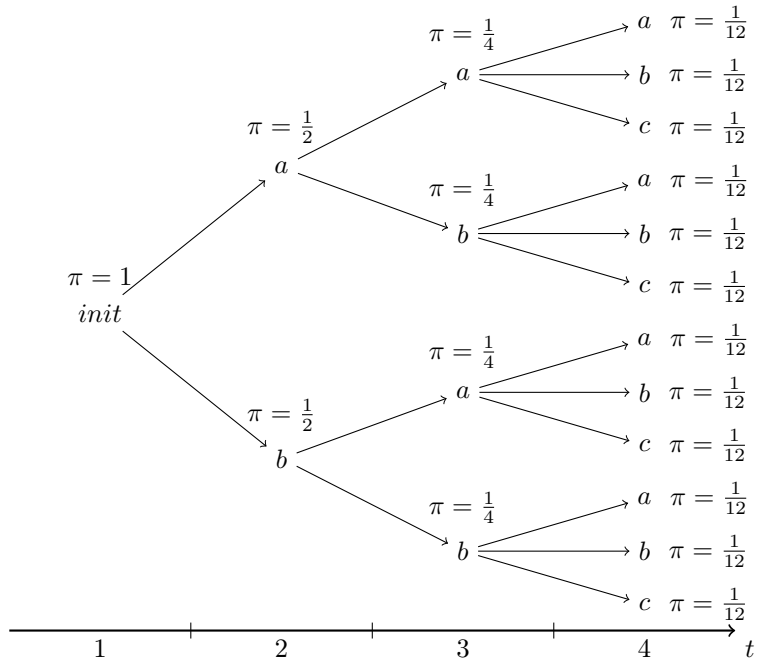


Figura 3.7: Árbol de escenarios generado con el generador de la Figura 3.6

```
val stochModelCS = stochModelBS
  .stochCustomScenarios(List(init, a, a), a -> r"1/2", b -> r"1/2")
  .stochCustomScenarios(List(init, a, b), a -> r"1")
  .stochCustomScenarios(List(init, b, a), a -> r"1")
```

Al utilizar `stochCustomScenarios`, los escenarios básicos del generador para la etapa que no aparecen en el conjunto de escenarios básicos especificado son marcados como eliminados en la rama del árbol correspondiente. A diferencia de `stochBasicScenarios`, en `stochCustomScenarios` los escenarios básicos que se especifican aplican únicamente a la rama correspondiente al escenario que se reciben como parámetro en lugar de a toda la etapa del generador.

El ejemplo anterior muestra los pasos necesarios para conseguir el árbol de la Figura 3.8 a partir del de la Figura 3.7. El código redefine algunos de los escenarios en la etapa 4 del árbol obtenido a partir del generador: la primera llamada a `stochCustomScenarios` borra el escenario básico c en la etapa 4 para el escenario de etapa 3 $[init, a, a]$, y redefine las probabilidades de los escenarios básicos a y b; las siguientes dos llamabas borran los escenarios básicos b y c en la etapa 4 para los escenarios de etapa 3 $[init, a, b]$ e $[init, b, a]$, dejando únicamente el escenario básico a con probabilidad condicional 1. Los escenarios básicos de la etapa 4 para el escenario de etapa 3 $[init, b, b]$ quedan incambiados.

Un segundo beneficio del generador se da al agregar escenarios básicos en una etapa, por ejemplo un escenario básico c en la etapa 2 del árbol de la Figura 3.8. Lo anterior puede conseguirse con una llamada

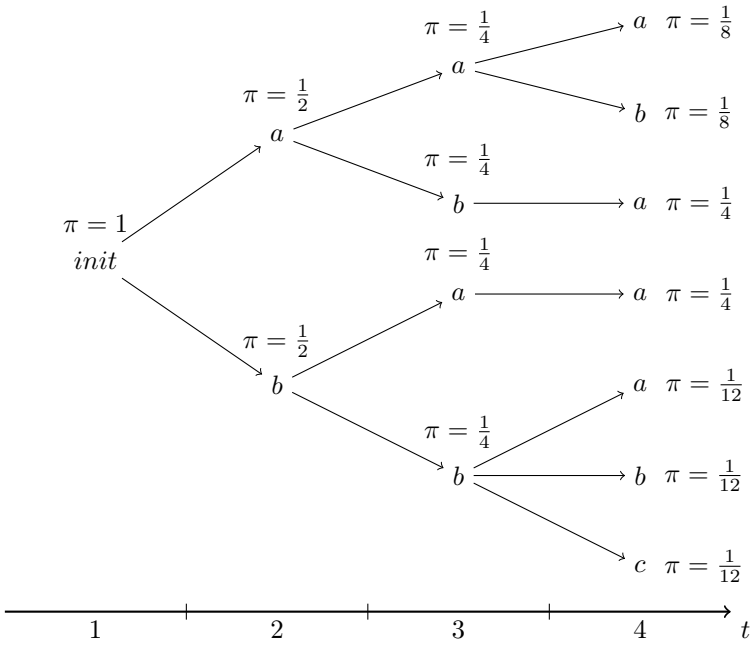


Figura 3.8: Árbol de la Figura 3.7 con modificaciones en algunos escenarios de la etapa 4

```
stochModelCS.stochCustomScenarios(
  List(init),
  a -> r"1/3", b -> r"1/3", c -> r"1/3")
```

donde se redefinen los escenarios básicos a continuación del escenario de primera etapa, $[init]$. En este caso no está claro como debe continuar el árbol de escenarios a partir del camino $[init, c]$. Por otro lado, para que el árbol de escenarios esté bien construido, todos los nodos hoja deben estar en el último nivel. La solución tomada en `amphip` para el problema anterior, es completar el árbol hasta las hojas utilizando el generador a partir del camino correspondiente al escenario agregado. En la Figura 3.9 se muestra el árbol de la Figura 3.8 con el escenario $[init, c]$, y el árbol generado a partir de allí, agregados.

Con los métodos descritos anteriormente, y aprovechando que el DSL está embebido en un lenguaje de propósito general, es posible definir métodos auxiliares para construir árboles con estructuras particulares. Por ejemplo, el constructor de árboles `nway(n)` provisto por SAMPL (Valente et al. 2009), que construye un árbol perfecto de aridad n, puede replicarse en `amphip` con una función `stochPerfectTree` como la siguiente:

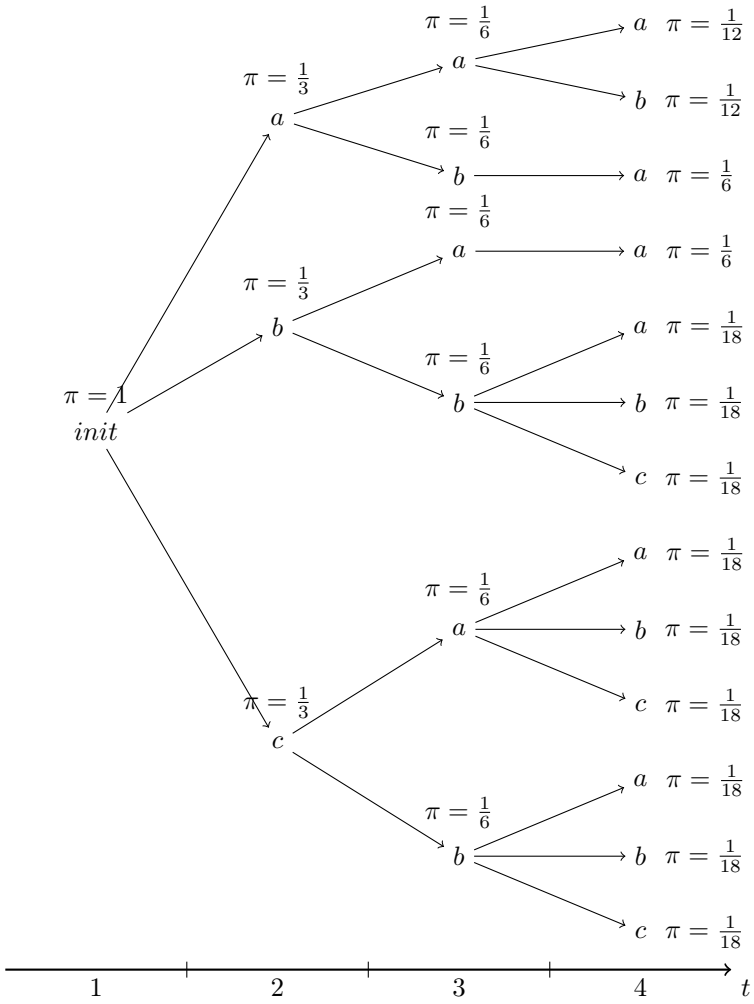


Figura 3.9: Árbol resultado de agregar el escenario $[init, c]$ al árbol de la Figura 3.8 utilizando el generador de la Figura 3.6

```
def stochPerfectTree(
  m           : StochModel,
  init        : BasicScenario,
  alternatives: Iterable[BasicScenario]): StochModel = {

  m.stages.toNel.fold(m) { nel =>
    val m1 = m.stochBasicScenarios(nel.head, init -> r"1")

    val den     = alternatives.size
    val prob    = r"1" / den
    val altProb = alternatives.map(_ -> prob)

    nel.tail.foldLeft(m1) { (model, t) =>
      model.stochBasicScenarios(t, altProb.toSeq: _*)
    }
  }
}
```

En la función `stochPerfectTree` los escenarios básicos son pasados como parámetros en lugar de especificar únicamente la aridad, para poder hacer referencia a los mismos posteriormente. La función trabaja con las etapas definidas en el modelo que recibe como parámetro, `m`, las que pueden ser una lista vacía. Para manejar los casos en que la lista de etapas esté vacía y no vacía al mismo tiempo, se utilizan las funciones `toNel`, provista por la biblioteca `Cats` (Cats 2018), y el `fold` de `Option`. La función `toNel` permite convertir una lista cualquiera en una lista que se asegura que es no es vacía, devolviendo el resultado "envuelto" en un `Option` que solamente tiene algún valor si se cumple que la lista es no vacía. El `fold` de `Option` recibe como argumentos el valor a devolver en el caso que no tenga ningún valor, y una función a aplicar en el caso que si lo tenga. En la función a aplicar sobre la lista no vacía de etapas es donde está propiamente la lógica de la función `stochPerfectTree`: se especifica el escenario básico a utilizar como raíz del árbol con probabilidad uno, luego se calcula la probabilidad de los escenarios de modo que sea equiprobable y se convierte la lista de alternativas en una lista de pares (*escenario básico, probabilidad condicional*), finalmente se "acumula" en `model` el resultado de asignar a cada etapa restante la lista de alternativas con las probabilidades calculadas, devolviéndose el modelo final con todas las etapas asignadas.

Para terminar con esta sección, se muestra el código con el que se genera la lista de escenarios finales a partir de la información especificada con los métodos anteriores.

```
lazy val finalScenarios: List[Scenario] = {
  val customTree =
    for {
      (history, bss) <- customScenarios.toList
      scen           <- bss.map(p => history :+ p._1)
      scenTree       <- balancedTree(scen, _._1)
        if scenTree.size == stages.size
    } yield {
      scenTree
    }

  val base = (customTree ::: balancedTree).distinct

  val target = base.filter { ss =>
    deletedScenariosList.forall(!ss.startsWith(_))
  }

  import Ordering.Implicits._
  target.sortBy(bsIndex)
}
```

En cada llamada a los métodos comentados anteriormente, `stochStages`, `stochBasicScenarios`, y `stochCustomScenarios`, se guarda la información sin realizar mayores cálculos. La lista de escenarios finales se mantiene en un campo *lazy*, `finalScenarios`, que solo es evaluado la primera vez que es accedido.

El campo `finalScenarios` básicamente calcula la lista de escenarios obtenida a partir de los escenarios especificados en forma detallada, la lista de escenarios correspondiente al árbol balanceado obtenido a partir del generador, y las junta, quitando los escenarios marcados como eliminados.

Para generar el primer árbol, `customTree`, se toman los escenarios especificados con `stochCustomScenarios` y se los completa utilizando el generador como se comentó anteriormente, utilizando la función `balancedTree`. La función `balancedTree`, sin argumentos, devuelve la lista de escenarios finales correspondientes al árbol balanceado producto de utilizar el generador. Luego de concatenar ambas listas y eliminar repetidos, se quitan todos los escenarios que tengan como prefijo alguno de los escenarios marcados como borrados. Finalmente se devuelve la lista producto de las operaciones anteriores, en orden. Para el ordenamiento de los escenarios se construye una lista de enteros para cada escenario de modo de tener un orden consistente con la función `bsIndex`, y se ordena según dicha lista de enteros utilizando la funcionalidad provista por Scala.

3.2.3. Probabilidades

Las probabilidades de los escenarios se definen al momento de construir el árbol de escenarios, como se muestra en la Sección 3.2.2. En esta sección se comentan brevemente detalles adicionales sobre el manejo de las probabilidades y métodos alternativos para especificarlas.

En los diferentes ejemplos mostrados, las probabilidades condicionales especificadas para el conjunto de escenarios básicos a utilizar en una etapa para un escenario, siempre suman uno. Lo anterior no tiene por qué ser siempre el caso, por lo que los diferentes métodos que aceptan listas de probabilidades como parámetros deben tomarlo en cuenta. En `amphip` las probabilidades son siem-

pre normalizadas para que sumen uno, de modo que los datos en `StochData` siempre sean consistentes (Sección 3.2.1).

Con los datos para las probabilidades especificados, puede construirse la lista con las probabilidades de cada escenario básico de los escenarios finales.

```
lazy val finalProbabilities: List[List[Rational]]
```

El algoritmo para determinar las probabilidades de los escenarios finales es más complejo que el algoritmo para determinar los escenarios finales (Sección 3.2.2). En la función `stochCustomScenarios`, se especifican los escenarios básicos del escenario en la etapa t que se está modificando, pero no sus probabilidades, solo se indican las probabilidades para los escenarios básicos en la etapa $t+1$. Por lo tanto, no es posible crear un "árbol con las probabilidades de los escenarios detallados" y así seguir la misma estrategia que para el cálculo de los escenarios finales. Para completar la información faltante, es necesario determinar las probabilidades de los escenarios del árbol balanceado obtenido con el generador (variable `balancedProbs`), tomando en cuenta, a su vez, los posibles escenarios eliminados, lo que se realiza como parte del cálculo de los escenarios por etapa (variable `balancedProbsByStage`).

```
val balancedProbs        = balancedTreeIdent
val balancedProbsByStage =
  for {
    (stage, bTree) <- byStage(balancedProbs)
  } yield {
    stage ->
      (for {
        (historyP, bss) <- bTree
        history         = historyP.unzip._1
        usedBS          =
          bss.filter { case (bs, _) =>
            val ss = history :+ bs
            deletedScenariosList.forall(!ss.startsWith(_))
          }
        if usedBS.nonEmpty
      } yield {
        history -> (LinkedMap() ++ usedBS)
      }).toMap
  }
```

Para determinar la lista final de probabilidades, por cada escenario básico de cada escenario final, se revisa si para el escenario predecesor, `history`, hay una especificación de escenarios detallados: si la hay, toma esa probabilidad; si no la hay, toma la del árbol balanceado calculado previamente. Lo anterior se consigue manteniendo cada probabilidad potencial en un valor de tipo `Option[Rational]` y combinándolos utilizando la función `orElse`.

```
val probs =
  for {
    scen <- finalScenarios
  } yield {
    for {
      p <- scen.zipWithIndex
      history = scen.take(p._2)
      customProb =
        for {
          cbss  <- customScenarios.get(history)
          cprob <- cbss.get(p._1)
        } yield {
          cprob
        }
      basicProb =
        for {
          t     <- stages.lift(p._2)
          bTree <- balancedProbsByStage.get(t)
          bss   <- bTree.get(history)
          prob  <- bss.get(p._1)
        } yield {
          prob
        }
      prob <- customProb.orElse(basicProb)
    } yield {
      prob
    }
  }
```

La variable `probs` tiene la lista de listas de probabilidades que se asigna a `finalProbabilities`.

Una vez obtenidas las probabilidades asociadas a cada escenario básico de cada escenario final, la probabilidad de los escenarios finales se calcula como la productoria de las probabilidades de sus escenarios básicos. El resultado de la productoria anterior, es utilizado como el valor asociado al parámetro del modelo que representa la probabilidad al generar el modelo MathProg correspondiente al equivalente determinista (Secciones 3.1). Los valores numéricos de MathProg solo pueden ser valores de punto flotante, por lo que las probabilidades deben ser convertidas previamente de `Rational` a `Double`. Por razones de performance, los valores individuales son convertidos a `Double` antes de calcular la productoria. El código a continuación muestra la función que calcula los valores a usar para el parámetro del modelo correspondiente a la probabilidad de los escenarios finales.

```
lazy val probabilityData: List[Double] = {
  for {
    path <- finalProbabilities
  } yield {
    path.map(_.toDouble).product
  }
}
```

La capa de programación estocástica brinda un mecanismo adicional para la especificación de las probabilidades, construido en base a las primitivas vistas anteriormente. El *extension method* `stochProbabilities`, permite especificar, dado un valor `m` convertible a `StochModel`, las *probabilidades camino* de los escenarios finales como una lista de pares (*escenario, probabilidad*). Utilizando este método, es posible especificar las probabilidades de los escenarios

finales siguiendo una distribución de probabilidad determinada, propagando las probabilidades hacia los escenarios de las etapas anteriores.

```
def stochProbabilities(
  sp: Iterable[(Scenario, Rational)]): StochModel
```

El método define escenarios detallados en todos los niveles del árbol, agrupando los escenarios según su predecesor en cada etapa (variable `hGroup`).

```
val hSize = m.stages.indices
val newStochData =
  hSize.foldLeft(m.stochData) { (stochData, th) =>
    val hGroup = sp.groupByLinked {
      case (scen, _) => scen.take(th)
    }

    // resto de la lógica
  }
update(newStochData)
```

A su vez, cada uno de los grupos anteriores es dividido en subgrupos según los escenarios de la etapa actual, calculando la probabilidad que le corresponde a cada grupo (variables `onStage` y `onStageProbs`).

```
val newStochData =
  hGroup.foldLeft(stochData) {
    case (stochData, (history, group)) =>
      val historyProb = group.unzip._2.qsum

      val onStage = group.groupByLinked {
        case (scen, _) => scen.take(th+1)
      }
      val onStageProbs = onStage.mapValues(_.unzip._2.qsum)

      // cálculo de probabilidades condicionales de escenarios básicos
  }
newStochData
```

Las probabilidades condicionales de los escenarios básicos por etapa se calculan dividiendo, por cada grupo de escenarios correspondiente a los escenarios de la etapa, la probabilidad del grupo (variable `prob`) entre la probabilidad del escenario predecesor (variable `historyProb`).

```
val bssProbs =
  for {
    (scen, prob) <- onStageProbs
    bs           <- scen.lastOption
  } yield {
    bs -> prob / historyProb
  }

stochData.customScenarios(history, bssProbs.toSeq:_*)
```

Finalmente el modelo es actualizado con una nueva instancia de `StochData` que tiene acumuladas todas las definiciones de escenarios detallados realizadas. En el Listado 3.1 se muestra el código completo del *extension method* `stochProbabilities`.

Se observa que si bien el método `stochProbabilities` se provee en forma predefinida, el mismo podría haber sido desarrollado por un tercero sin que

```
def stochProbabilities(
  sp: Iterable[(Scenario, Rational)]): StochModel = {
  val hSize = m.stages.indices
  val newStochData =
    hSize.foldLeft(m.stochData) { (stochData, th) =>
      val hGroup = sp.groupByLinked {
        case (scen, _) => scen.take(th)
      }

      val newStochData =
        hGroup.foldLeft(stochData) {
          case (stochData, (history, group)) =>
            val historyProb = group.unzip._2.qsum

            val onStage = group.groupByLinked {
              case (scen, _) => scen.take(th+1)
            }
            val onStageProbs = onStage.mapValues(_.unzip._2.qsum)

            val bssProbs =
              for {
                (scen, prob) <- onStageProbs
                bs           <- scen.lastOption
              } yield {
                bs -> prob / historyProb
              }

            stochData.customScenarios(history, bssProbs.toSeq:_*)
        }
      newStochData
    }

  update(newStochData)
}
```

Listado 3.1: Definición de método para especificación de probabilidades camino de los escenarios finales

existiera ninguna diferencia.

3.2.4. Parámetros estocásticos

La especificación de los valores para los parámetros estocásticos implica especificar un valor por cada nodo del árbol de escenarios para cada parámetro. Para alivianar el procedimiento anterior, en `amphip` los valores de los parámetros estocásticos pueden especificarse con diferentes niveles de detalle, buscando reducir en lo posible la cantidad de valores que deben especificarse para cubrir el conjunto total de escenarios. La especificación de valores de parámetros estocásticos sigue un esquema similar al utilizado para definir la estructura del árbol de escenarios (Sección 3.2.2). En el código a continuación se muestran ejemplos de asignaciones de valores de parámetros estocásticos utilizando cada nivel de detalle para un parámetro estocástico simple, *p1*.

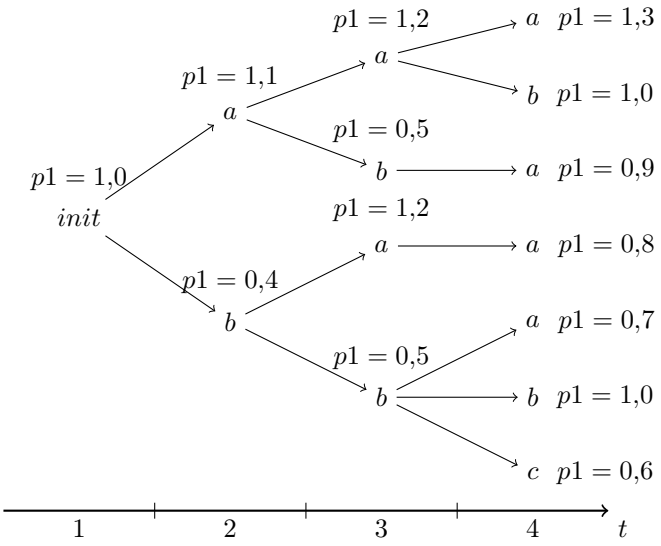
```
val stochModelData = stochModelCS
  .stochDefault(p1, 1.0)
  .stochBasicData(p1, t2, a, 1.1)
  .stochBasicData(p1, t2, b, 0.4)
  .stochBasicData(p1, t3, a, 1.2)
  .stochBasicData(p1, t3, b, 0.5)
  .stochBasicData(p1, t4, a, 1.3)
  .stochBasicData(p1, t4, c, 0.6)
  .stochScenarioData(p1, List(init, a, b, a), 0.9)
  .stochScenarioData(p1, List(init, b, a, a), 0.8)
  .stochScenarioData(p1, List(init, b, b, a), 0.7)
```

En el nivel de detalle más básico, es posible definir los valores por defecto a ser usados en cualquier escenario para el que no se disponga información para el parámetro estocástico, utilizando el método `stochDefault` que recibe el parámetro estocástico y los valores a utilizar.

En el siguiente nivel de detalle, los valores de los parámetros estocásticos pueden definirse para los escenarios básicos por etapa definidos en el generador (Sección 3.2.2). Para especificar los datos de los parámetros estocásticos según el generador, se provee del método `stochBasicData` que recibe el parámetro estocástico, la etapa y el escenario básico del generador, y los valores a asignar al parámetro.

Por último, en el nivel más detallado, los valores de los parámetros estocásticos pueden especificarse para un escenario individual de cualquier etapa, utilizando el método `stochScenarioData` que recibe el parámetro estocástico, el escenario, y los valores a asignarle.

El árbol a continuación muestra el árbol con estructura arbitraria construido en la Sección 3.2.2 (Figura 3.8), indicando en cada nodo el valor correspondiente del parámetro $p1$ si se usa la asignación de valores del ejemplo. En los escenarios $[init]$, $[init, a, a, b]$, e $[init, b, b, b]$ se utiliza el valor por defecto; en los escenarios $[init, a, b, a]$, $[init, b, a, a]$, e $[init, b, b, a]$ se utilizan los valores especificados por escenario individual; y en los escenarios restantes se utilizan los valores especificados para escenarios básicos del generador.



La implementación de los tres métodos `stochDefault`, `stochBasicData` y `stochScenarioData`, es similar: en todos los casos el parámetro estocástico se procesa como si fuera determinista, removiendo los conjuntos de etapas y escenarios de su dominio, se genera la instancia de `ModelData` correspondiente en forma análoga a lo que se realiza en la capa de datos (Sección 2.3), y se extraen los datos que corresponden específicamente al parámetro. El procedimiento seguido permite reutilizar toda la infraestructura desarrollada para la capa de datos, en la especificación de datos de parámetros estocásticos. Como referencia, se muestra a continuación la implementación de `stochScenarioData`.

```
def stochScenarioData[B](p: ParamStat, scen: Scenario, values: B*)
  (implicit ev: DataOp[ParamStat, B]): StochModel = {
  requireStochastic(p, m)
  val newData = ev.data(asDet(p, m), values.toList)(m.model.data)
  val pData   = filter(newData, p).params
  update(m.stochData.scenarioData(scen, p, pData))
}
```

El método `asDet` es el encargado de modificar el parámetro estocástico quitándole los conjuntos de etapas y escenarios de su dominio para que no se los tome en cuenta al evaluar la expresión de indexación, y el método `filter` construye un `ModelData` que tiene únicamente los datos asociados al parámetro (Sección 2.3.1). Con los datos obtenidos se actualiza la instancia de `StochData` del modelo en forma acorde.

Para el método `stochScenarioData` se provee, adicionalmente, una versión sobrecargada que recibe un parámetro estocástico y una lista de pares (*escenario, valor*), y permite especificar los valores del parámetro para múltiples escenarios en un solo paso. Dicha versión sobrecargada permite generar los valores para cada escenario en forma separada, por ejemplo siguiendo determinada distribución de probabilidad, y asignarlos todos de una vez.

Como en otros casos, es posible utilizar las capacidades de abstracción del lenguaje anfitrión para desarrollar métodos adicionales que simplifiquen la especificación de los datos en casos particulares. Por ejemplo, si los datos de un parámetro estocástico no dependen de las etapas, especificar los valores en

cada etapa es redundante. El código a continuación muestra la implementación de un método `stochStagesData` que replica los valores por escenario básico en un conjunto de etapas. El método acumula en el modelo que recibe como parámetro, `m`, el resultado de asignar los valores especificados para cada uno de los escenarios básicos en cada una de las etapas especificadas.

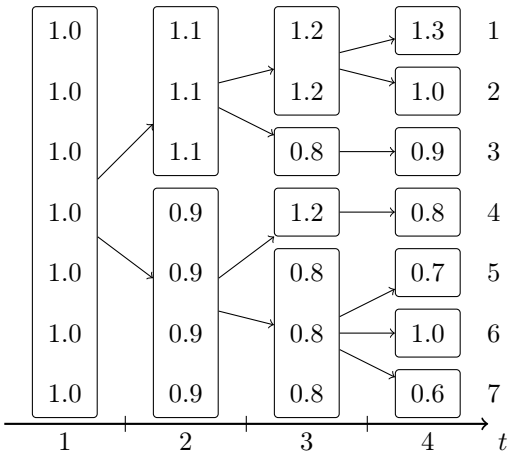
```
def stochStagesData[B](
  m          : StochModel,
  p          : ParamStat,
  stages     : List[Stage],
  bsDataList: (BasicScenario, B)*)
  (implicit ev: DataOp[ParamStat, B]): StochModel = {

  stages.foldLeft(m) { (model0, t) =>
    bsDataList.foldLeft(model0) { (model1, bsData) =>
      val (bs, data) = bsData
      model1.stochBasicData(p, t, bs, data)
    }
  }
}
```

Se observa que el método es paramétrico en los valores que se asignan a los parámetros estocástico, reutilizando la infraestructura de la capa de datos (Sección 2.3.3). Esto permite que el mismo método pueda utilizarse tanto para parámetros estocásticos "simples" como para parámetros estocásticos que tienen algún conjunto adicional en su dominio. Como ejemplo de uso de la función `stochStagesData`, se replica lo mostrado en la introducción para el problema de planificación financiera (Sección 1.1), para especificar los valores del retorno esperado por inversión, ξ_i, para las etapas $t_2, \ldots, t_4$, para las alternativas *high* y *low*.

```
val stochModelBasicData =
  stochStagesData(stochModelBasicScenarios, xi, List(t2,t3,t4),
    high -> List(stock -> 1.25, bonds -> 1.14),
    low  -> List(stock -> 1.06, bonds -> 1.12)
  )
```

Para generar el modelo MathProg a resolver, es necesario generar los valores a utilizar en cada subíndice posible de los parámetros estocásticos. Como se ha comentado anteriormente, utilizar la formulación con escenarios separados produce que la especificación de datos tenga mucha redundancia (Sección 3.1.1). Por ejemplo, en el árbol a continuación se muestran los datos del parámetro de ejemplo, $p1$, utilizando escenarios separados. Los rectángulos indican los escenarios que deben tener el mismo valor por etapa para mantener la no anticipatividad



La redundancia anterior a priori es transparente para `amphip` porque solo aparece en el momento de generar modelo MathProg y resolver el problema (Sección 2.6 y Sección 3.2.5). Sin embargo, dicha redundancia tiene impacto tanto en el tiempo de generación de las secciones de datos de MathProg como en el tamaño que ocupan las mismas, en particular para árboles de escenarios de gran tamaño. Para reducir los problemas anteriores, en `amphip` los datos de los parámetros estocásticos no se generan con escenarios separados, en lugar de eso, se crean versiones alternativas de cada parámetro estocástico sin utilizar escenarios separados, y se genera los valores solo para los parámetros alternativos, evitando la redundancia. Los parámetros estocásticos originales se redefinen en términos de los parámetros alternativos. Las transformaciones realizadas a los parámetros estocásticos están relacionadas fuertemente con la forma en que se maneja la no anticipatividad, por lo que se describen en la sección correspondiente (Sección 3.2.5).

El cambio principal a la hora de generar los valores de los parámetros estocásticos, es que en las versiones alternativas los escenarios están subordinados a las etapas. El conjunto S pasa a ser S^t. Para generar los valores de los parámetros estocásticos se trabaja con las etapas especificadas (valor `stage`) y con los escenarios por etapa (valor `scenariosByStage`), los que se determinan a partir de los escenarios finales agrupándolos según el prefijo correspondiente a cada etapa.

```
(stage, t_) <- stages.zipWithIndex
sts         <- scenariosByStage.get(stage).toList
(scen, s_)  <- sts.zipWithIndex
```

Con la etapa y los escenarios de la etapa, es posible determinar si el parámetro estocástico tiene un valor específico para el escenario,

```
cData =
  for {
    cbss <- scenarioData.get(scen)
    data <- cbss.get(param)
  } yield {
    data
  }
```

si hay un valor en el generador para el escenario básico correspondiente al último

componente del escenario,

```
bs <- scen.lastOption.toList
bData =
  for {
    bss  <- basicData.get(stage)
    ds   <- bss.get(bs)
    data <- ds.get(param)
  } yield {
    data
  }
```

o si hay un valor por defecto definido.

```
defaultData  = defaults.get(param)
```

Cada uno de los valores anteriores es de tipo `Option[ParamStatData]`. Con estos valores potenciales, es posible determinar el conjunto de datos a asignar al parámetro, combinándolos con la función `orElse`.

```
pData <- cData.orElse(bData).orElse(defaultData).toList
```

Una vez obtenido el `ParamStatData` a utilizar, se vuelven a introducir los subíndices para la etapa y el escenario a los subíndices del parámetro estocástico en la especificación de datos, y se devuelve la lista de pares $(subíndice, valor)$ con los valores del parámetro.

```
val scenariosData =
  for {
    // lógica descrita anteriormente
    t             = t_ + 1
    s             = s_ + 1
    (key, value) <- pData
  } yield {
    (List[SimpleData](t, s) ::: key.subscript) -> value
  }
```

En el Listado 3.2 se muestra el código completo de la función que genera los datos para un parámetro estocástico.

3.2.5. No anticipatividad y transformaciones al modelo

En la formulación extensiva *sin* escenarios separados (3.3), la no anticipatividad de las variables de decisión se consigue en forma *implícita*. La no anticipatividad está dada por construcción, ya que los escenarios están subordinados a la etapa y el modelo incorpora la relación "padre-hijo" entre los escenarios del árbol mediante el conjunto S_s^t, el que establece el conjunto de escenarios descendientes en la etapa t para un escenario s de la etapa $t-1$. Dicha relación "padre-hijo" puede especificarse en forma alternativa mediante un parámetro $pred(s), s \in S^t, t = 2, \ldots, H$, que especifica el escenario de S^{t-1} predecesor del escenario s. A continuación se muestra una variante de (3.3) que utiliza el

```
def paramDataST(param: ParamStat):
  List[(List[SimpleData], SimpleData)] = {
  val scenariosData =
    for {
      (stage, t_) <- stages.zipWithIndex
      sts         <- scenariosByStage.get(stage).toList
      (scen, s_)  <- sts.zipWithIndex
      cData =
        for {
          cbss <- scenarioData.get(scen)
          data <- cbss.get(param)
        } yield {
          data
        }
      bs <- scen.lastOption.toList
      bData =
        for {
          bss  <- basicData.get(stage)
          ds   <- bss.get(bs)
          data <- ds.get(param)
        } yield {
          data
        }
      defaultData   = defaults.get(param)
      pData        <- cData.orElse(bData).orElse(defaultData).toList
      t             = t_ + 1
      s             = s_ + 1
      (key, value) <- pData
    } yield {
      (List[SimpleData](t, s) ::: key.subscript) -> value
    }

  scenariosData.toList
}
```

Listado 3.2: Generación de datos sin escenarios separados asociados a un parámetro estocástico, a partir de los datos especificados utilizando la funcionalidad provista por `amphip`

parámetro $pred(s)$ en lugar del conjunto S_s^t:

$$\begin{aligned}
\min \quad & c^1x^1 + \sum_{t=2}^{H} \sum_{s \in S^t} \pi(s) c^t(s) x^t(s) \\
\text{s. a} \quad & W^1x^1 = h^1, \\
& T^2(s)x^1 + W^2(s)x^2(s) = h^2(s), \qquad s \in S^2, \\
& T^t(s)x^{t-1}(pred(s)) + W^t(s)x^t(s) = h^t(s), \quad t = 3, \dots, H, s \in S^t, \\
& x^1 \geq 0, x^t(s) \geq 0, \ \ t = 2, \dots, H, s \in S^t.
\end{aligned}$$

En la formulación *con* escenarios separados (3.4), la no anticipatividad debe conseguirse en forma *explícita*, introduciendo parámetros auxiliares, como el parámetro $anc(s,t)$ que indica el "ancestro" de un escenario final s en la etapa t, y restricciones de no anticipatividad (3.5)

$$\begin{aligned}
\min \quad & c^1x^1 + \sum_{t=2}^{H} \sum_{s \in S} \pi(s) c^t(s) x^t(s) \\
\text{s. a} \quad & W^1x^1 = h^1, \\
& T^2(s)x^1 + W^2(s)x^2(s) = h^2(s), \qquad s \in S, \\
& T^t(s)x^{t-1}(s) + W^t(s)x^t(s) = h^t(s), \quad t = 3, \dots, H, s \in S, \\
& x^1 \geq 0, x^t(s) \geq 0, \ \ t = 2, \dots, H, s \in S,
\end{aligned} \tag{3.4 revisitada}$$

$$x^t(s) = x^t(s'), \quad t = 2, \dots, H, \ \ s, s' \in S, \ \ anc(s,t) = anc(s',t). \tag{3.5 revisitada}$$

La formulación con escenarios separados es más simple y más sencilla de manipular ya que se puede trabajar con las etapas y los escenarios en forma independiente. Además, si bien controlar la no anticipatividad en forma implícita reduce la cantidad de variables necesarias, su representación explícita mediante restricciones puede ser de utilidad para resolver el problema mediante métodos de descomposición como los propuestos por Benders (1962) y Laporte y F. V. Louveaux (1993).

Cada una de las formulaciones anteriores puede construirse a partir de la otra. Dado un escenario final, su representante en los escenarios por etapa puede obtenerse como:

$$s' = anc(s,t), \quad s' \in S^t, \ \ t \in T, \ \ s \in S,$$

y por lo tanto, para cualquier parámetro p o variable x en la formulación con escenarios separados, puede obtenerse su equivalente en la formulación sin escenarios separados, p' y x', sustituyendo los índices en T y S por un índice en S^t, y manteniendo las igualdades siguientes:

$$p^t(s) = p'(anc(s,t)), \quad t \in T, \ \ s \in S, \ \ anc(s,t) \in S^t, \tag{3.9}$$

$$x^t(s) = x'(anc(s,t)), \quad t \in T, \ \ s \in S, \ \ anc(s,t) \in S^t. \tag{3.10}$$

Por otra parte, el parámetro $anc(s,t)$ puede definirse en términos de $pred(s)$ como sigue:

$$anc(s,t) = \begin{cases} s & \text{si } s \in S^t, \\ anc(pred(s),t) & \text{en otro caso,} \end{cases} \tag{3.11}$$

y los escenarios finales definirse como:

$$S = S^H. \tag{3.12}$$

Con las definiciones anteriores, si se sustituyen todos los parámetros y variables estocásticos siguiendo las igualdades (3.9) y (3.10), es posible pasar de una formulación con escenarios separados y representación explícita de la no anticipatividad, a una formulación sin escenarios separados y no anticipatividad implícita.

Un paso intermedio respecto al anterior consiste en tener dos juegos de parámetros y variables estocásticos, con y sin escenarios separados, e incorporar las igualdades (3.9) y (3.10) al modelo. La igualdad (3.9) puede incorporarse al modelo en la definición de los parámetros utilizando los atributos `:=` o `default`, y la igualdad (3.10) como restricciones (que cumplen la función de restricciones de no anticipatividad), manteniendo el resto del modelo sin cambios.

En `amphip` se trabaja con dos juegos de parámetros estocásticos, con y sin escenarios separados, manteniendo para las variables solamente la versión con escenarios separados, y generando restricciones de no anticipatividad explícitas de la forma (3.5). Manejar los parámetros estocásticos con y sin escenarios separados permite que la generación de datos para el modelo MathProg sea más eficiente ya que puede generarse sin redundancia y ser usada en la versión con escenarios separados mediante la igualdad (3.9). El manejo de dos juegos de variables o la sustitución completa de parámetros y variables estocásticas para controlar la no anticipatividad en forma implícita, queda como posible trabajo futuro (Capítulo 5).

Generación de restricciones de no anticipatividad

Para generar las restricciones de no anticipatividad se incorporan al modelo las declaraciones de S^t, *pred*, y *anc*. En el Listado 3.3 se muestran como quedan las declaraciones anteriores en `amphip`, siendo `ST` el conjunto S^t y `ancf` una versión de *anc* que trabaja únicamente con escenarios finales. Para evitar colisiones de nombre con entidades declaradas en los modelos, se agrega el prefijo `"ST_"` a los nombres de las entidades que se agregan. Se observa que el parámetro del modelo que representa el horizonte de planificación, H, se define a partir de T, cuando normalmente ocurre lo opuesto. Lo anterior se debe a que el conjunto T es manejado por `amphip` y no puede ser definido por otra vía, sin embargo, para poder obtener el conjunto S a partir de S^t es necesario poder hacer referencia a S^H, por lo que se introduce H como el valor máximo del conjunto T.

Una vez incorporadas las definiciones anteriores, es posible generar las restricciones de no anticipatividad. Por ejemplo, las siguientes son posibles restricciones de no anticipatividad para una extensión del caso de estudio que incorpore incertidumbre en la demanda (Sección 3.3):

```
val ST = set("ST_ST", T)

val pred = {
  val t = dummy
  param("ST_pred", t in T &~ List(1), ST(t)) in ST(t-1)
}

lazy val anc: ParamStat = {
  val t  = dummy
  val tp = dummy
  val s  = dummy
  param("ST_anc",
    ind(t in T, s in ST(t), tp in T) | tp <= t) in ST(tp) :=
    xif (tp === t) { s } { anc(t-1, pred(t,s), tp) }
}

val H = {
  val t = dummy
  param("ST_H") := max(t in T)(t)
}

val ancf = {
  val t = dummy
  val s = dummy
  param("ST_ancf", s in ST(H), t in T) in ST(t) := anc(H, s, t)
}
```

Listado 3.3: Parámetros y conjuntos auxiliares que se incorporan al modelo para generar las restricciones de no anticipatividad

```
val NA_v_ctr = st(
    ind(t in T, s1 in S, s2 in S, c in P) |
      ancf(s1,t) === ancf(s2,t) && t <= H-gamma(c)
  ) { v(c,t,s1) === v(c,t,s2) }

val NA_x_ctr = st(
    ind(t in T, s1 in S, s2 in S, c in A) |
      ancf(s1,t) === ancf(s2,t) && t <= tau(c)-1
  ) { x(c,t,s1) === x(c,t,s2) }

val NA_y_ctr = st(
    ind(t in T, s1 in S, s2 in S) | ancf(s1,t) === ancf(s2,t)
  ) { y(t,s1) === y(t,s2) }
```

La lógica para la generación de las restricciones de no anticipatividad está disponible en el objeto `nonanticipativity`, el cual define un método `apply` para poder utilizarlo como una función. La firma del método `apply` del objeto `nonanticipativity` es:

```
def apply(xvar: VarStat, T: SetStat, S: SetStat, na: STAdapter):
  Option[ConstraintStat]
```

donde `xvar` es la variable para la que se desea generar la restricción, `T` es el conjunto que representa las etapas, `S` es el conjunto que representa los escenarios finales, y `na` contiene las declaraciones del Listado 3.3. La función devuelve la restricción dentro de un `Option` para manejar el caso en que la variable no es estocástica.

Para extraer las entradas y el predicado de la expresión de indexación del dominio de la variable, se utiliza una *for comprehension* que, además, filtra que la variable sea estocástica y devuelve el resultado manteniendo el mismo "contenedor" que tiene el dominio de la variable (`Option`):

```
for {
  IndExpr(entries0, predicate0) <- xvar.domain
    if isStochastic(entries0, T, S)
} yield {
  // lógica para generar la restricción
}
```

El primer paso de la generación de la restricción consiste en introducir indices para cada entrada de la expresión de indexación de la variable, los que conforman el subíndice base a utilizar:

```
val tIdeal = dummy("t")
val sIdeal = dummy("s")

val (entries, t, s) = assignIndices(entries0, T, S, tIdeal, sIdeal)
val subscript       = entries.flatMap(_.indices)
```

La función `assignIndices` recibe los parámetros `T`, `S`, `sIdeal`, `tIdeal` para intentar usar subíndices más nemotécnicos para los conjuntos `T` y `S` en caso de ser posible, y devuelve la lista de entradas actualizada y los índices usados efectivamente para `T` y `S`.

A partir del subíndice generado se construyen los dos índices que se usan para seleccionar los distintos escenarios de `S`, con nombres nemotécnicos si no hay colisiones con los otros índices en el subíndice, y el predicado $ancf(s_1, t) = ancf(s_2, t)$, y se crea la expresión de indexación para la restricción (`indexing`):

```
val s1 = uniqueDummy(subscript, s, 1)
val s2 = uniqueDummy(subscript, s, 2)
val ancfEq = na.ancf(s1, t) === na.ancf(s2, t)

val detEntries = entries.filterNot(e => List(T(), S()).contains(e.set))

val indexing = IndExpr(
    (t in T) :: (s1 in S) :: (s2 in S) :: detEntries,
    Some(predicate0.fold(ancfEq)(ancfEq && _)))
```

Las entradas en `indexing` se construyen concatenando la indexación $t \in T, s_1 \in S, s_2 \in S$ a la parte "determinista" de las entradas de la variable, esto es, la lista de entradas quitando la indexación sobre T y S (`detEntries`). El predicado de la expresión de indexación de la restricción debe contemplar que el dominio de la variable ya tenga un predicado, lo que se consigue mediante la función `fold` de `Option`. En el caso que no hay predicado, se utiliza directamente `ancfEq`. Si el dominio ya tiene un predicado, se construye una conjunción con ambos predicados. El resultado final se envuelve en un nuevo `Option`.

Finalmente se construye la restricción de no anticipatividad correspondiente a la variable `xvar`, construyendo dos subíndices que utilizan `s1` y `s2` en lugar de `s`, e indexando la variable en cada uno:

```
val subscript1 = subscript.map(x => if (x == s) s1 else x)
val subscript2 = subscript.map(x => if (x == s) s2 else x)

st(s"NA_${xvar.name}_ctr", indexing) {
  xvar(subscript1) === xvar(subscript2)
}
```

El código completo de la función se muestra en el Listado 3.4.

```
object nonanticipativity {

  def apply(xvar: VarStat, T: SetStat, S: SetStat, na: STAdapter):
    Option[ConstraintStat] = {
    for {
      IndExpr(entries0, predicate0) <- xvar.domain
        if isStochastic(entries0, T, S)
    } yield {

      val tIdeal = dummy("t")
      val sIdeal = dummy("s")

      val (entries, t, s) =
        assignIndices(entries0, T, S, tIdeal, sIdeal)
      val subscript = entries.flatMap(_.indices)

      val s1 = uniqueDummy(subscript, s, 1)
      val s2 = uniqueDummy(subscript, s, 2)
      val ancfEq = na.ancf(s1, t) === na.ancf(s2, t)

      val detEntries =
        entries.filterNot(e => List(T(), S()).contains(e.set))

      val indexing = IndExpr(
          (t in T) :: (s1 in S) :: (s2 in S) :: detEntries,
          Some(predicate0.fold(ancfEq)(ancfEq && _)))

      val subscript1 = subscript.map(x => if (x == s) s1 else x)
      val subscript2 = subscript.map(x => if (x == s) s2 else x)

      st(s"NA_${xvar.name}_ctr", indexing) {
        xvar(subscript1) === xvar(subscript2)
      }
    }
  }

  // funciones auxiliares
}
```

Listado 3.4: Generación de restricción de no antivipatividad para una variable estocástica

Generación de parámetros estocásticos sin escenarios separados

La generación de las versiones sin escenarios separados de los parámetros estocásticos sigue un procedimiento que tiene puntos de contacto con el de generación de restricciones de no anticipatividad. Para la generación de los parámetros estocásticos sin escenarios separados se utiliza el método `adaptParam` de la clase `STAdapter` que, como se comentó anteriormente, incluye las declaraciones

del Listado 3.3. Dicha clase recibe en su constructor los conjuntos de etapas, `T`, y de escenarios finales, `S`. La firma del método `adaptParam` es la siguiente:

```
def adaptParam(param: ParamStat): Option[(ParamStat, ParamStat)]
```

El método `adaptParam` recibe como entrada la declaración de un parámetro del modelo y retorna una tupla con las dos versiones del mismo, con y sin escenarios separados. El método devuelve la versión con escenarios separados porque su declaración debe ser adaptada para que tome sus valores a partir de la versión sin escenarios separados, siguiendo la igualdad (3.9). El par resultado se devuelve dentro de un `Option` para manejar el caso en que el parámetro del modelo no sea estocástico.

En forma análoga al método para generar la restricciones de no anticipatividad, la deconstrucción de la expresión de indexación del parámetro del modelo, el chequeo de que sea estocástico, y el retorno del resultado como `Option`, es manejado con una *for comprehension* de Scala:

```
for {
  IndExpr(entries0, predicate0) <- param.domain
    if isStochastic(entries0, T, S)
} yield {
  // lógica para adaptar el parámetro
}
```

Nuevamente el primer paso consiste en introducir índices para todas las entradas de la expresión de indexación, utilizando el método `assignIndices`. La lista de entradas con los índices introducidos se utiliza para crear una copia del parámetro del modelo que es idéntica a la original salvo porque todas las entradas de su expresión de indexación declaran índices:

```
val tIdeal = dummy("t")
val sIdeal = dummy("s")

val (entries, t, s) = assignIndices(entries0, T, S, tIdeal, sIdeal)
val param0 =
  param.copy(domain = Some(IndExpr(entries, predicate0)))
```

La versión del parámetro sin escenarios separados se construye reemplazando cualquier referencia al conjunto `S` por la versión indexada por etapa, `ST(t)`, siendo `t` el índice usado para las etapas devuelto por `assignIndices`, y cambiando el nombre del parámetro para evitar colisiones:

```
val ST_param =
  replace(param0, S(), ST(t)).copy(name = s"ST_${param0.name}")
```

Para el reemplazo de las referencias al conjunto `S` por referencias al conjunto `ST(t)` en el parámetro del modelo `param0`, se utiliza una función provista por la capa de modelo (Sección 2.2), llamada `replace`. Dicha función está implementada siguiendo un esquema similar a la función `collect` (Sección 2.6), utilizándose en este caso un objeto `replace`, que define un método `apply`, y una *type class* `Replace`. Se observa que las coincidencias del valor a reemplazar se buscan dentro del componente del modelo objetivo utilizando la *igualdad universal* de objetos, que comparten tanto Scala como Java (Sección 2.2.3).

Finalmente se establece que el valor por defecto del parámetro con escenarios separados se obtiene del parámetro sin escenarios separados utilizando el atributo

`default`, y se devuelve el par con los dos parámetros:

```
val subscript  = entries.flatMap(_.indices)
val subscript1 = subscript.map { x =>
  if (x == s) ancf(s,t) else x: SimpleExpr
}

val paramA = param0 default ST_param(subscript1)

paramA -> ST_param
```

El subíndice utilizado para la referencia al parámetro sin escenarios separados (`subscript1`) es el subíndice construido a partir de la lista de entradas resultado de `assignIndices`, reemplazando las ocurrencias de `s` por `ancf(s,t)` de modo que para los escenarios finales en el parámetro original, indexados con `s`, en el parámetro sin escenarios separados se indexe con `ancf(s,t)` sobre los escenarios de la etapa, según (3.9).

En el Listado 3.5 se muestra el código completo de la función `adaptParam`. Como ejemplos de aplicación de la función `adaptParam`, se muestran la aplicación al parámetro $d^t(s)$ del Caso de estudio extendido (Sección 3.3) y al parámetro $\xi_i^t(s)$ del problema de planificación financiera (Sección 1.1):

```
// parámetro "demanda" en caso de estudio extendido
// original: val d = param(T, S)
val ST_d = param(t in T, s in ST(t))
val d    = param(t in T, s in S) default ST_d(t, ancf(s,t));

// parámetro "retorno" en problema de planificación financiera
// original: val xi = param(ind(t in T, S, I) | t > 1)
val ST_xi = param(ind(t in T, s in ST(t), i_ in I) | t > 1)
val xi    = param(ind(t in T, s in S,     i_ in I) | t > 1)
  default ST_xi(t, ancf(s,t), i_)
```

Generación de forma extensiva

Con las transformaciones descritas anteriormente, es posible generar el problema de programación entera mixta equivalente al estocástico y resolverlo. Se provee de un *extension method* para la clase `StochModel`, `mip`, que construye una instancia de `ModelWithData` (Sección 2.3.1) a partir del modelo estocástico. El método `mip` genera las restricciones de no anticipatividad para cada una de las variables estocásticas del modelo usando la función `nonanticipativity` (Listado 3.4):

```
val model1 = (model0 :++ nonanticipativityConstraints)
```

La función `:++` agrega las sentencias de la lista de la derecha al final de las sentencias del modelo a la izquierda, recolectando todas las declaraciones necesarias e incorporándolas también al modelo usando la función `collect` (Sección 2.6). Debido a lo anterior, al agregar las restricciones de no anticipatividad también se agregan los entidades `ancf`, `anc`, `pred`, `H`, y `ST` (Listado 3.3).

Luego se generan los pares de parámetros del modelo para especificar los valores sin escenarios separados utilizando la función `adaptParam` (Listado 3.5). Los parámetros sin escenarios separados se agregan al inicio del modelo, junto con el parámetro `ancf`, para que su declaración ocurra antes de que se haga referencia a él:

```
case class STAdapter(T: SetStat, S: SetStat) extends NAMode {

  // declaraciones de ST, pred, anc, H, y ancf...

  def adaptParam(param: ParamStat): Option[(ParamStat, ParamStat)] = {
    for {
      IndExpr(entries0, predicate0) <- param.domain
        if isStochastic(entries0, T, S)
    } yield {
      val tIdeal = dummy("t")
      val sIdeal = dummy("s")

      val (entries, t, s) =
        assignIndices(entries0, T, S, tIdeal, sIdeal)
      val param0 =
        param.copy(domain = IndExpr(entries, predicate0).some)

      val ST_param =
        replace(param0, S(), ST(t)).copy(name = s"ST_${param0.name}")

      val subscript  = entries.flatMap(_.indices)
      val subscript1 = subscript.map { x =>
        if (x == s) ancf(s,t) else x: SimpleExpr
      }

      val paramA = param0 default ST_param(subscript1)

      paramA -> ST_param
    }
  }
}
```

Listado 3.5: Generación de parámetro estocástico sin escenarios separados a partir de uno con escenarios separados, y modificación de este último para tomar los valores desde el primero

```
val adaptedParams = model1.stochasticParameters.flatMap(na.adaptParam)
val st_params     = adaptedParams.unzip._2

val model2 = (ancf :: st_params) ++: model1
```

El conjunto de escenarios finales ahora se define en función del conjunto de escenarios por etapa, por lo que se reemplaza su declaración en base a la igualdad (3.12), utilizando la función `replace` comentada anteriormente:

```
val SA     = S default ST(H)
val model3 = model2.replace(S, SA)
val piA    = model3.pi

val model4 = model3
  .setData(T , stochData.TData)
  .setData(ST, stochData.STData)
  .paramData(pred, stochData.predecessorsData)
  .paramData(piA , stochData.probabilityData)
```

El reemplazo de `S` por `SA` tiene efecto sobre las probabilidades, ya que están indexadas en `S`, por lo que el parámetro `pi` debe tomarse del modelo posterior al reemplazo. Los datos del modelo se especifican utilizando la funcionalidad de la capa de datos (Sección 2.3) a partir del procesamiento realizado en la clase `StochData` (Sección 3.2.1). Se especifican los datos de las etapas, los escenarios por etapa, los predecesores de cada escenario por etapa, y las probabilidades (Sección 3.2.3).

El paso final consiste en reemplazar las declaraciones de los parámetros con escenarios separados por su versión "adaptada" que toma los datos del parámetro sin escenarios separados, y especificar los valores de los parámetros. Para esto se acumula los cambios sobre un modelo utilizando `foldLeft` sobre la lista de pares producto de juntar, con `zip`, los parámetros estocásticos y los pares de parámetros adaptados calculados previamente. Como en las dos listas anteriores los parámetros están en el mismo orden, en cada tupla $(p, (pA, st\_p))$ se tiene el parámetro estocástico original, p, la versión sin escenarios separados st_p, y la versión con escenarios separados que toma el valor de la versión sin escenarios separados, pA:

```
val model5 = model4.stochasticParameters.zip(adaptedParams)
  .foldLeft(model4) { case (model, (param, (paramA0, st_param))) =>
    val paramA1 = replacef(paramA0, S, SA)
    model
      .replace(param, paramA1)
      .paramData(st_param, stochData.paramDataST(param))
  }

model5.model
```

En cada paso se reemplaza el parámetro del modelo por su versión adaptada con `replace`, y se especifican los valores de la versión sin escenarios separados utilizando `paramDataST` (Listado 3.2). Es necesario reemplazar el conjunto `S` por `SA` en los parámetros adaptados debido a que el conjunto original ya no existe en el modelo. El modelo final a devolver es el `ModelWithData` contenido dentro del modelo estocástico con todas las modificaciones aplicadas (`model5.model`). En el Listado 3.6 se muestra el código completo de la función `mip`.

```
def mip: ModelWithData = (m: StochModel) match {
  case model0 @ MultiStageStochModel(
    _, stochData, T @ _, S @ _, _, na: STAdapter) =>
    import amphip.model.{replace => replacef}
    import na.{ST, pred, H, ancf}

    val model1 = (model0 :++ nonanticipativityConstraints)

    val adaptedParams =
      model1.stochasticParameters
      .flatMap(na.adaptParam)
    val st_params = adaptedParams.unzip._2

    val model2 = (ancf :: st_params) ++: model1

    val SA     = S default ST(H)
    val model3 = model2.replace(S, SA)
    val piA    = model3.pi

    val model4 = model3
      .setData(T , stochData.TData)
      .setData(ST, stochData.STData)
      .paramData(pred, stochData.predecessorsData)
      .paramData(piA , stochData.probabilityData)

    val model5 = model4.stochasticParameters.zip(adaptedParams)
      .foldLeft(model4) { case (model, (param, (paramA0, st_param))) =>
        val paramA1 = replacef(paramA0, S, SA)
        model
          .replace(param, paramA1)
          .paramData(st_param, stochData.paramDataST(param))
      }

    model5.model
}
```

Listado 3.6: Generación de modelo equivalente MathProg a partir de modelo estocástico

3.3. Caso de estudio extendido

A continuación se presenta una extensión al problema determinista presentado como caso de estudio en la Sección 2.5, en el que la demanda de combustible se considera un parámetro aleatorio y se la incorpora reformulando el problema como uno de programación estocástica multietapa basada en escenarios. En las Tablas 3.1, 3.2 y 3.3 se muestra las entidades del problema adaptadas para el caso estocástico, y en el Listado 3.7 su representación en el EDSL. Se observa que en el EDSL, el parámetro del modelo que representa el horizonte de períodos, H, debe definirse en función del conjunto de etapas, T, en lugar de ser al revés como ocurre en el modelo matemático.

```
// Conjuntos (y parámetros de los conjuntos)
val t  = dummy
val tp = dummy
val T  = set
val H  = param := max(t in T)(t)

val c = dummy
val A = set
val P = set
val C = set := A | P

val s = dummy
val S = set

// Parámetros
val d    = param(T, S)
val y0   = param
val ymin = param
val ymax = param

val tau   = param(A) in T
val gamma = param(P) in (0 to H - 1)
val q     = param(C)

val ca = param(C)
val cc = param(A)
val h  = param(T)

val a = param(t in T) :=
  sum((c in A) | tau(c) === t) { q(c) }

val pi = param(S)

// Variables
val y = xvar(T, S) >= 0
val v = xvar(ind(c in P, t in T, S) | t <= H - gamma(c)).binary
val x = xvar(ind(c in A, t in T, S) | t <= tau(c) - 1).binary

val u = xvar(T, S) >= 0
val w = xvar(T, S) >= 0
```

Listado 3.7: Entidades del Caso de estudio extendido representadas en el EDSL propuesto

En la versión determinista, las posibles demandas de combustible se resumen

H	horizonte de planificación
T	períodos, $T := 1, \ldots, H$
A	cargamentos ya adquiridos
P	posibles cargamentos a adquirir
C	cargamentos totales, $C := A \cup P$
S	escenarios finales

Tabla 3.1: Conjuntos, y parámetros de los conjuntos, del Caso de estudio extendido

$d^t(s)$	volumen demandado en el período $t \in T$ y escenario $s \in S$
$y0$	volumen almacenado inicial
$\underline{y}, \bar{y}$	Capacidades mínima y máxima de volumen almacenado
τ_c	período en que se recibe el cargamento ya adquirido $c \in A$
γ_c	tiempo de entrega del cargamento $c \in P$, tal que $0 \leq \gamma_c \leq H - 1$
q_c	volumen del cargamento $c \in C$
ca_c	costo unitario de adquisición del cargamento $c \in C$
cc_c	costo unitario de cancelación del cargamento $c \in A$
h^t	costo unitario de almacenamiento en el período $t \in T$
a^t	volumen ya adquirido que se recibe en el período $t \in T$, $a^t := \sum_{c \in A \mid \tau_c = t} q_c$ (resumen)
$\pi(s)$	probabilidad del escenario $s \in S$
$anc(s,t)$	ancestro del escenario $s \in S$ en el período $t \in T$ en el árbol de escenarios

Tabla 3.2: Parámetros del Caso de estudio extendido

$y^t(s)$	volumen almacenado al final del período $t \in T$ en el escenario $s \in S$
$v_c^t(s)$	decisión de adquirir el cargamento $c \in P$ en el período $t \in T$ y escenario $s \in S$, tal que $1 \leq t \leq H - \gamma_c$ (binaria)
$x_c^t(s)$	decisión de cancelar el cargamento $c \in A$ en el período $t \in T$ y escenario $s \in S$, tal que $1 \leq t \leq \tau_c - 1$ (binaria)
$u^t(s)$	volumen adquirido en el período $t \in T$ y escenario $s \in S$ (resumen)
$w^t(s)$	volumen cancelado en el período $t \in T$ y escenario $s \in S$ (resumen)

Tabla 3.3: Variables del Caso de estudio extendido

en un único valor por período, d^t. Dicho valor puede ser, por ejemplo, la demanda promedio esperada para cada período. Sin embargo, debido al tiempo que transcurre entre que se toman las decisiones de compra y los cargamentos de combustibles efectivamente llegan, puede suceder que la demanda observada se distancie mucho de la demanda prevista en cada período, y provocar que las decisiones recomendadas por el modelo se tornen subóptimas o incluso infactibles. Por otra parte, el modelo determinista tampoco puede dar ninguna indicación sobre posibles decisiones correctivas a tomar para atender la diferencia entre el valor esperado al tomar las decisiones y el observado en la práctica.

La incertidumbre en la demanda de combustible puede modelarse como un proceso estocástico de tiempo discreto y espacio de probabilidades finito, donde cada etapa del proceso se corresponde con un período de planificación en T. En cada período distinto del primero, la demanda se considera aleatoria con una distribución de probabilidad conocida. La distribución de probabilidad de la demanda en dichos períodos está condicionada por la distribución de probabilidad de las demandas en los períodos anteriores. En el período inicial la demanda se considera determinista.

El proceso estocástico descrito en el párrafo anterior puede representarse mediante un árbol de escenarios donde cada nivel del árbol se corresponde con un período. Como en otras oportunidades, el conjunto S denota los escenarios finales. Utilizando la formulación con escenarios separados (3.4), dados un período $t \in T$ y un escenario $s \in S$, queda identificado un único nodo del árbol de escenarios, siendo $d^t(s)$ la demanda en dicho nodo (Figura 3.10).

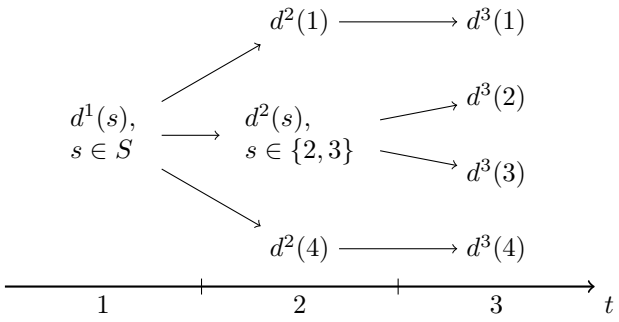


Figura 3.10: Ejemplo de árbol de escenarios para el Caso de estudio extendido con tres etapas ($H = 3$) y cuatro escenarios finales ($S = \{1, 2, 3, 4\}$), con las demandas asociadas a cada nodo del árbol

Incorporar los escenarios a la demanda impacta en las decisiones de compra y cancelación de cargamentos, y en las decisiones de almacenamiento de combustible, ya que pueden tomarse distintas decisiones según el escenario que se trate. Como se utiliza la formulación con escenarios separados, el cambio consiste simplemente en incorporar el escenario $s \in S$ como parámetro de las entidades (Tabla 3.3).

La restricción de que un cargamento no pueda ser comprado más de una vez (Ecuación (2.7)), ahora debe tomar en cuenta que en un escenario se puede decidir una compra pero en otro no, por lo que hay una restricción separada por cada escenario

$$\sum_{t=1}^{H-\gamma_c} v_c^t(s) \leq 1, \quad c \in P, \ s \in S$$

```
val singleAcquisition =
  st(c in P, s in S) {
    sum(t in (1 to H - gamma(c))) {
      v(c,t,s)
    } <= 1
  }
```

En forma similar, la restricción de que no pueda cancelarse un cargamento más de una vez (Ecuación (2.8)) también debe tomar en cuenta que las decisiones pueden ser diferentes según el escenario de demanda que se considere

$$\sum_{t=1}^{\tau_c-1} x_c^t(s) \leq 1, \quad c \in A, \ s \in S$$

```
val singleCancellation =
  st(c in A, s in S) {
    sum(t in (1 to tau(c) - 1)) {
      x(c,t,s)
    } <= 1
  }
```

Las variables u^t y w^t, que calculan los volúmenes a adquirir y cancelar por período (Ecuaciones (2.9) y (2.10)), también se ven afectadas por la incorporación de la incertidumbre, ya que dependen de las compras y las cancelaciones, y por lo tanto también incorporan al escenario como parámetro

$$u^t(s) = \sum_{c \in P | \gamma_c \leq t-1} q_c v_c^{t-\gamma_c}(s), \quad t \in T, \ s \in S$$

```
val acquiredFuel = st(t in T, s in S) {
  u(t,s) === sum((c in P) | gamma(c) <= t-1) {
              q(c) * v(c, t-gamma(c), s)
            }
}
```

$$w^t(s) = \sum_{c \in A | \tau_c = t} \left(q_c \sum_{t'=1}^{\tau_c - 1} x_c^{t'}(s) \right), \quad t \in T, \ s \in S$$

```
val cancelledFuel = st(t in T, s in S) {
  w(t,s) === sum((c in A) | tau(c) === t) {
              q(c) *
              sum(tp in (1 to tau(c)-1)) { x(c,tp,s) }
            }
}
```

Como se comentó anteriormente, las decisiones de almacenamiento de combustible en cada período, y^t, también dependen del escenario de demanda. La restricción que establece las cotas (Ecuación (2.6)) debe aplicarse en todos los escenarios

$$\underline{y} \leq y^t(s) \leq \bar{y}, \quad t \in T, \ s \in S$$

```
val inventory =
  st(t in T, s in S) {
    dlte(ymin, y(t,s), ymax)
  }
```

El balance entre las entradas y salidas de combustible (Ecuación (2.5)) puede ser diferentes en cada escenario, por lo que se necesita una restricción separada por cada uno. Se observa que el volumen ya adquirido que se recibe en cada

período, a^t, no depende del escenario ya que es producto de decisiones tomadas previamente. Lo que puede suceder con el volumen ya adquirido, es que parte de ese volumen sea cancelado en un escenario y en otro no, lo que se refleja en la variable $w^t(s)$

$$y^{t-1}(s) + a^t + u^t(s) = d^t(s) + w^t(s) + y^t(s), \quad t \in T, \;\; s \in S, \;\; y^0(s) = y0$$

```
val balance0 = st(            s in S         ) {
  y0        + a(1) + u(1,s) === d(1,s) + w(1,s) + y(1,s)
}
val balance  = st(ind(t in T, s in S) | t > 1) {
  y(t-1, s) + a(t) + u(t,s) === d(t,s) + w(t,s) + y(t,s)
}
```

La función objetivo (Ecuación (2.4)) también debe incorporar la incertidumbre, ya que los costos dependen de las decisiones tomadas en cada escenario. Se optimiza el valor esperado de los costos considerando todos los posibles escenarios al mismo tiempo. Dado un escenario $s \in S$, $\pi(s)$ denota la probabilidad de ocurrencia de dicho escenario

$$\min \sum_{t \in T, s \in S} \pi(s) \Bigg[\sum_{c \in P | t \leq H - \gamma_c} ca_c q_c v_c^t(s) + \sum_{c \in A | t \leq \tau_c - 1} (cc_c - ca_c) q_c x_c^t(s) + h^t y^t(s) \Bigg]$$

```
val cost = minimize { sum(t in T, s in S) {
  pi(s) * (
    sum((c in P) | t <= H-gamma(c)) {
                ca(c)  * q(c) * v(c,t,s)
    } +
    sum((c in A) | t <= tau(c)-1) {
      (cc(c) - ca(c)) * q(c) * x(c,t,s)
    } +
    h(t) * y(t,s)
  )
}}
```

Como se está utilizando la formulación con escenarios separados, deben incorporarse restricciones de no anticipatividad para las variables de decisión, por ejemplo en base a la Ecuación (3.5) utilizando el parámetro $anc(s,t)$

$$v_c^t(s) = v_c^t(s'), \quad c \in P, \;\; t \in T, \;\; s, s' \in S, \;\; anc(s,t) = anc(s',t),$$

$$x_c^t(s) = x_c^t(s'), \quad c \in A, \;\; t \in T, \;\; s, s' \in S, \;\; anc(s,t) = anc(s',t),$$

$$y^t(s) = y^t(s'), \quad t \in T, \;\; s, s' \in S, \;\; anc(s,t) = anc(s',t).$$

La versión en `amphip` no requiere la definición de restricciones de no anticipatividad, ya que las mismas son generadas automáticamente en base a la información provista al especificar la estructura del árbol. El modelo estocástico se especifica de la siguiente forma:

```
val stochModel =
  model(cost,
    balance0, balance, inventory,
    singleAcquisition, singleCancellation,
    acquiredFuel, cancelledFuel
  ).stochastic(T, S, pi)
```

Para completar el ejemplo estocástico, se extiende la instancia de datos utilizada en el Caso de estudio, incorporando el árbol de escenarios. Se considera un árbol de escenarios perfecto con aridad 2 y cuatro escenarios finales. La probabilidad de los escenarios finales se toma siguiendo una distribución de probabilidad $Beta(\alpha = 2, \beta = 2)$ (Figura 3.11, caso con cuatro escenarios finales). Al igual que en el caso determinista, la demanda se muestrea con distribución uniforme $d^t(s) \sim U[10, 50]$. La única diferencia es que en este caso se debe muestrear un número mayor de valores (uno por cada nodo del árbol). El árbol intenta simular en forma sencilla un caso donde los escenarios sobre los bordes representan demandas más extremas con menor probabilidad, y el centro del árbol demandas más cercanos al promedio con probabilidad mayor, aunque en este caso la demanda simplemente se asigna en forma aleatoria. La Figura 3.12 muestra el árbol de escenarios utilizado.

La instancia de datos puede especificarse en el EDSL de la siguiente forma:

```
val (t1, t2, t3) = (Stage("1"), Stage("2"), Stage("3"))
val (init, bsa, bsb) = (BasicScenario("init"), BasicScenario("a"),
    BasicScenario("b"))

val stochModelBS = stochModel
  .stochStages(t1, t2, t3)
  .stochBasicScenarios(t1, init -> r"1")
  .stochBasicScenarios(t2, bsa -> r"1/2", bsb -> r"1/2")
  .stochBasicScenarios(t3, bsa -> r"1/2", bsb -> r"1/2")

val stages         = stochModelBS.stages
val scenarios      = stochModelBS.scenarios
val finalScenarios = stochModelBS.finalScenarios

val AData = List("A1", "A2")
val PData = List("P1", "P2", "P3", "P4")

val stochModelWData = stochModelBS
  .setData(A, AData)
  .setData(P, PData)
  .paramData(y0  , 20)
  .paramData(ymin,  0)
  .paramData(ymax, 80)
  .paramData(tau, "A1" -> 1, "A2" -> 2)
  .paramData(gamma, PData.map(_ -> 1))
  .paramData(q , uniform(2)( 10,  50)(AData ::: PData))
  .paramData(ca, uniform(3)(150, 250)(AData ::: PData))
  .paramData(cc, uniform(4)( 30,  50)(AData))
  .paramData(h, stages.indices.map(_ + 1 -> 1))
  .stochScenarioData(d, uniformL(1)(10, 50)(scenarios))
  .stochProbabilities(betaR(2, 2)(finalScenarios))
```

El árbol perfecto de aridad 2 se construye con la funcionalidad básica para la especificación del generador (Sección 3.2.2), especificando probabilidades arbitrarias que son sobrescritas luego. La especificación de las probabilidades de

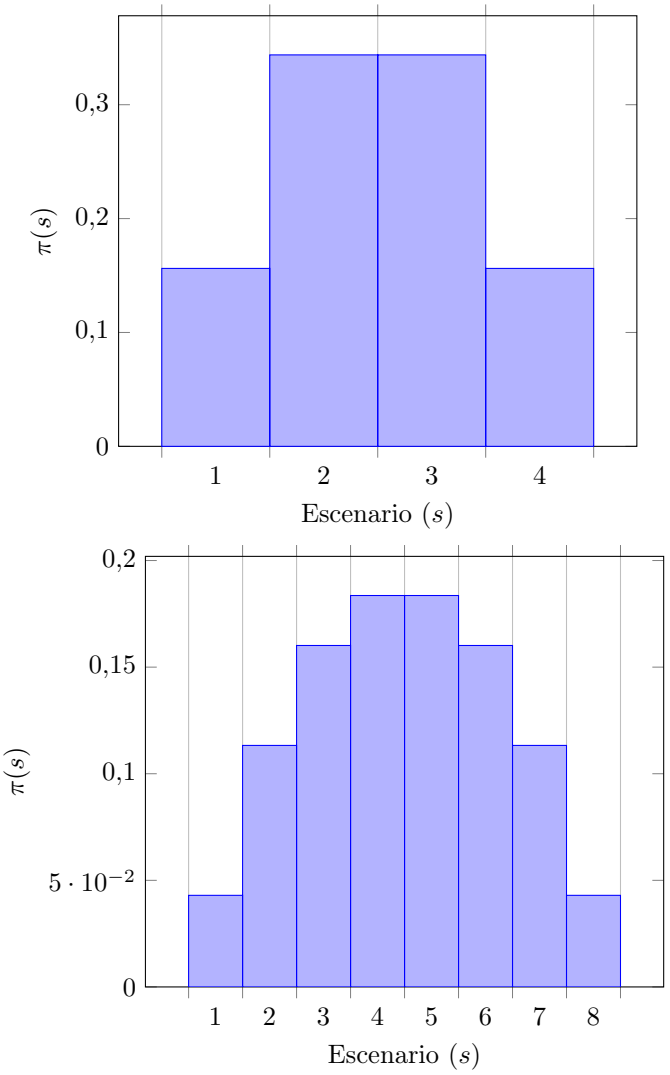


Figura 3.11: Probabilidad de los escenarios finales siguiendo la distribución $Beta(\alpha = 2, \beta = 2)$. Ejemplos para cuatro y ocho escenarios finales

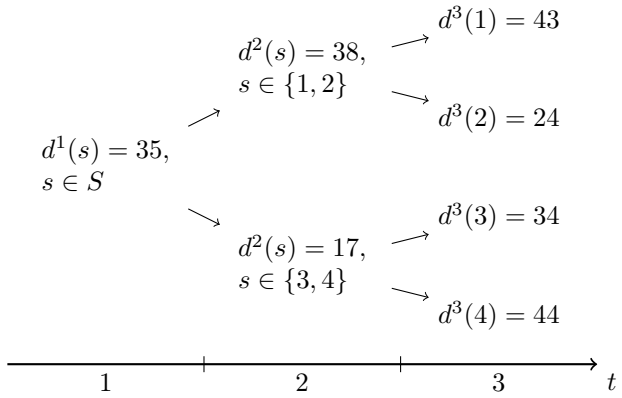


Figura 3.12: Árbol de escenarios utilizado en el Caso de estudio extendido con los valores de demanda de combustible por escenario

los escenarios finales se realiza con la ayuda de una función auxiliar, `betaR(2,2)`, que recibe una lista y devuelve una lista de pares asignando a cada valor de la lista la probabilidad correspondiente según la cantidad de elementos que contenga. La lista que se pasa a la función anterior es la lista de los escenarios finales, y el resultado de la misma es usado para especificar las probabilidades de los escenarios finales utilizando el método `stochProbabilities` (Listado 3.1), que propaga las probabilidades hacia los escenarios de las etapas anteriores. Los valores de los parámetros deterministas son especificados utilizando la funcionalidad de la capa de datos (Sección 2.3), de la misma forma que en el Caso de estudio determinista. Para la especificación del parámetro estocástico d, se utiliza el método `stochScenarioData` (Sección 3.2.4) que permite asignar valores para cada escenario individual. La función anterior es llamada con valores generados para el conjunto total de escenarios (`stochmodelBS.scenarios`), cubriendo el árbol de escenarios completo.

Se observa que los datos de los parámetros estocásticos se especifican para los escenarios definidos, por lo que la consistencia entre ambos se mantiene por construcción, y no tiene redundancia. Las probabilidades de los escenarios se especifican de la forma deseada combinando las primitivas provistas por `amphip` y las capacidades del lenguaje anfitrión.

Con la información disponible, es posible generar el modelo MathProg equivalente utilizando la función `mip` (Listado 3.6). Dicho modelo es más eficiente que el especificado manualmente ya que los datos de los parámetros estocásticos se generan sin redundancia. En el Apéndice E.1 se muestra una implementación del Caso de estudio extendido en MathProg sin utilizar `amphip`.

El modelo completo actualizado se muestra a continuación. En el Listado 3.8

se muestra su representación en el EDSL.

$$\begin{aligned}
\min \quad & \sum_{t\in T, s\in S} \pi(s)\Bigg[\sum_{c\in P|t\le H-\gamma_c} ca_c q_c v_c^t(s) \\
& \qquad + \sum_{c\in A|t\le \tau_c-1} (cc_c - ca_c) q_c x_c^t(s) \qquad (3.13)\\
& \qquad + h^t y^t(s)\Bigg] \\
\text{s. a} \quad & y^{t-1}(s) + a^t + u^t(s) = d^t(s) + w^t(s) + y^t(s), \quad t\in T,\ s\in S,\ y^0(s) = y0, \\
& \qquad (3.14)\\
& \underline{y} \le y^t(s) \le \bar{y}, \qquad t\in T,\ s\in S, \quad (3.15)\\
& \sum_{t=1}^{H-\gamma_c} v_c^t(s) \le 1, \qquad c\in P,\ s\in S, \quad (3.16)\\
& \sum_{t=1}^{\tau_c-1} x_c^t(s) \le 1, \qquad c\in A,\ s\in S, \quad (3.17)\\
& u^t(s) = \sum_{c\in P|\gamma_c\le t-1} q_c v_c^{t-\gamma_c}(s), \qquad t\in T,\ s\in S, \quad (3.18)\\
& w^t(s) = \sum_{c\in A|\tau_c=t} \left(q_c \sum_{t'=1}^{\tau_c-1} x_c^{t'}(s) \right), \qquad t\in T,\ s\in S, \quad (3.19)\\
& y^t(s), u^t(s), w^t(s) \ge 0, \quad t\in T,\ s\in S,\\
& v_c^t(s) \in \{0,1\}, \qquad c\in P,\ t = 1,\dots,H-\gamma_c,\ s\in S,\\
& x_c^t(s) \in \{0,1\}, \qquad c\in A,\ t = 1,\dots,\tau_c-1,\ s\in S,\\
& \textit{no anticipatividad de las variables.}
\end{aligned}$$

3.4. Cálculo de medidas de valor

El *embedding* desarrollado permite generar el problema MathProg equivalente y resolverlo de forma totalmente externa, sin posibilidad de posprocesar los resultados que devuelva el *solver*. Por lo anterior, el soporte para el cálculo de medidas de valor de los modelos como las descritas en la Sección 3.1.4, es limitado.

El problema RP es el problema estocástico, por lo que no hace falta realizar ninguna acción para obtenerlo.

```
// Función objetivo
val cost = minimize {
  sum(t in T, s in S) {
    pi(s) * (
      sum((c in P) | t <= H-gamma(c)) {
        ca(c) * q(c) * v(c,t,s)
      } +
      sum((c in A) | t <= tau(c)-1) {
        (cc(c) - ca(c)) * q(c) * x(c,t,s)
      } +
      h(t) * y(t,s)
    )
  }
}

// Restricciones
val balance0 = st(            s in S         ) {
  y0        + a(1) + u(1,s) === d(1,s) + w(1,s) + y(1,s)
}
val balance  = st(ind(t in T, s in S) | t > 1) {
  y(t-1, s) + a(t) + u(t,s) === d(t,s) + w(t,s) + y(t,s)
}

val inventory = st(t in T, s in S) { dlte(ymin, y(t,s), ymax) }

val singleAcquisition = st(c in P, s in S) {
  sum(t in (1 to H - gamma(c))) { v(c,t,s) } <= 1
}
val singleCancellation = st(c in A, s in S) {
  sum(t in (1 to tau(c) - 1)  ) { x(c,t,s) } <= 1
}

val acquiredFuel = st(t in T, s in S) {
  u(t,s) ===  sum((c in P) | gamma(c) <= t-1) {
                q(c) * v(c, t-gamma(c), s)
              }
}

val cancelledFuel = st(t in T, s in S) {
  w(t,s) ===  sum((c in A) | tau(c) === t) {
                q(c) * sum(tp in (1 to tau(c)-1)) { x(c,tp,s) }
              }
}

// Modelo
val stochModel =
  model(cost,
    balance0, balance, inventory,
    singleAcquisition, singleCancellation,
    acquiredFuel, cancelledFuel
  ).stochastic(T, S, pi)
```

Listado 3.8: Función objetivo, restricciones, y declaración de la instancia del modelo del Caso de estudio extendido representadas en el EDSL propuesto

```
object RP {
  val problem = stochModelWData // modelo estocástico con datos
  val value   = 5935 // valor del óptimo para el modelo generado
  /* Decisiones de compra y cancelación (sin ecenarios separados):
    v[P1,1,1] = 0
    v[P2,1,1] = 0
    v[P3,1,1] = 0
    v[P4,1,1] = 1
    v[P1,2,1] = 0
    v[P1,2,2] = 0
    v[P2,2,1] = 0
    v[P2,2,2] = 0
    v[P3,2,1] = 1
    v[P3,2,2] = 0
    v[P4,2,1] = 0
    v[P4,2,2] = 0

    x[A2,1,1] = 1
  */
}
```

En la solución óptima se decide importar el cargamento $P4$ en el primer período. Se observa que la decisión es la misma independientemente del escenario final, ya que en la primera etapa conceptualmente hay un único escenario. En el segundo período se decide importar el cargamento $P3$ en un escenario y no importarlo en el otro. Nuevamente, las decisiones se toman manteniendo la propiedad de no anticipatividad. En el tercer período no puede importarse ningún cargamento porque en todos los casos $\gamma_c = 1$. De los cargamentos ya adquiridos, se decide cancelar el $A2$ en el primer período, un período antes del período en que se preveía su llegada.

El problema WS puede ser calculado de diversas formas. Por ejemplo, se puede resolver el problema RP restringido a un único escenario para cada escenario de S, y luego calcular la esperanza de los resultados obtenidos en cada problema individual. La forma de cálculo anterior puede utilizarse en el caso en que las variables de primera etapa no estén "separadas" por escenario como en la formulación (3.4). Si las decisiones de primera etapa están separadas por escenario, como en la formulación (3.6), el valor WS coincide con resolver el problema RP sin considerar las restricciones de no anticipatividad. Dado que `amphip` no permite acceder a los resultados de la resolución del problema, la solución anterior es más atractiva que la que implica resolver $card(S)$ veces el problema RP restringido a un único escenario. La formulación (3.4) puede convertirse en la (3.6) mediante una función provista por `amphip`, llamada `separate`, que transforma una instancia de `StochModel` en un `ModelWithData` (Sección 2.3.1) con las variables de primera etapa separadas por escenario y todas restricciones y la función objetivo actualizadas para tomar en cuenta el cambio, además de instanciar la estructura del árbol de escenarios y los datos de los parámetros estocásticos en el modelo MathProg. Si el modelo ya se encuentra en la formulación (3.6), la función `separate` se reduce a instanciar los valores de los parámetros correspondientes de forma similar a la función `mip` (Listado 3.6), pero sin generar restricciones de no anticipatividad.

```
object WS {
  val problem = amphip.stoch.separate(RP.problem)
  val value   = 2738
  /* Decisiones de compra y cancelación (con escenario separados):
    v[P1,1,1] = 0    v[P1,2,1] = 0
    v[P1,1,2] = 0    v[P1,2,2] = 0
    v[P1,1,3] = 0    v[P1,2,3] = 1
    v[P1,1,4] = 0    v[P1,2,4] = 0
    v[P2,1,1] = 0    v[P2,2,1] = 0
    v[P2,1,2] = 0    v[P2,2,2] = 0
    v[P2,1,3] = 0    v[P2,2,3] = 0
    v[P2,1,4] = 0    v[P2,2,4] = 0
    v[P3,1,1] = 0    v[P3,2,1] = 0
    v[P3,1,2] = 0    v[P3,2,2] = 0
    v[P3,1,3] = 0    v[P3,2,3] = 0
    v[P3,1,4] = 0    v[P3,2,4] = 0
    v[P4,1,1] = 0    v[P4,2,1] = 1
    v[P4,1,2] = 1    v[P4,2,2] = 0
    v[P4,1,3] = 0    v[P4,2,3] = 0
    v[P4,1,4] = 0    v[P4,2,4] = 1

    x[A2,1,1] = 0
    x[A2,1,2] = 1
    x[A2,1,3] = 0
    x[A2,1,4] = 1
  */
}
```

En este caso, las decisiones sugeridas no son consistentes con el árbol de escenarios, porque el problema no incluye restricciones de no anticipatividad. Por ejemplo, se sugiere importar $P4$ en el primer período en el escenario final 2, pero no en los otros, lo cuál es inconsistente. En forma similar, el modelo sugiere cancelar $A2$ en el primer período para algunos escenarios finales y pero para otros no. El poder violar las restricciones de no anticipatividad es lo que permite que el problema WS alcance mejores valores para el óptimo.

El valor esperado de la información perfecta es:

```
val EVPI = RP.value - WS.value // 3197
```

El cálculo del problema de valor esperado para cada etapa (Ecuaciones (3.7) y (3.8)) implica adaptar la estructura del árbol y los valores de los parámetros estocásticos, y el fijado de las variables en etapas anteriores a la que se está calculando. Al no disponer de soporte para obtener el resultado de la resolución del modelo, el cálculo anterior no puede automatizarse completamente en `amphip`. El cálculo del problema EV^t puede automatizarse si no se considera el fijado de los valores de las variables. En `amphip` se provee una función, `EV`, que dados un `StochModel` y una etapa, devuelve el `StochModel` con la estructura del árbol ajustada y los valores de los parámetros estocásticos adaptados para tomar como valor su esperanza en las etapas posteriores a la especificada.

```
object EV1 {
  val problem = amphip.stoch.EV(RP.problem, t1)
  val value   = 2444
  /* Compra y cancelación en t1 (sin ecenarios separados):
    v[P1,1,1] = 0
    v[P2,1,1] = 0
    v[P3,1,1] = 0
    v[P4,1,1] = 1

    x[A2,1,1] = 1
  */
}
```

La solución de valor esperado en la etapa 1 para el problema EV^1, indica que se debe adquirir el cargamento $P4$ ($\bar{v}^1_{P4}(s) = 1, s \in S$) y cancelar el cargamento $A2$ ($\bar{x}^1_{A2}(s) = 1, s \in S$). El cálculo del EV^t para etapas posteriores puede intercalarse con la generación del problema EEV^t en el que se fijan los valores de las variables manualmente.

```
object EEV1 {
  val problem = {
    val vbar1 = param(ind(t in T, S, P) | t === 1)
    val xbar1 = param(ind(t in T, S, A) | t === 1)

    val fixV1 =
      st(ind(s in S, c in P) | gamma(c) <= H-1) {
        v(c,1,s) === vbar1(1,s,c)
      }
    val fixX1 =
      st(ind(s in S, c in A) | tau(c) >= 2) {
        x(c,1,s) === xbar1(1,s,c)
      }

    val List(s1) = RP.problem.stochData.scenariosByStage(t1)
    (RP.problem :++ List(fixV1, fixX1))
      .stochScenarioData(vbar1, s1,
        "P1" -> 0,
        "P2" -> 0,
        "P3" -> 0,
        "P4" -> 1)
      .stochScenarioData(xbar1, s1,
        "A2" -> 1)
  }
  val value = 5935
}
```

Como la función `EV` parte de un modelo estocástico arbitrario, puede utilizarse el modelo `EEV1.problem` como entrada para calcular EV^2, ya que dicho modelo tiene fijadas las variables de primera etapa en el valor correspondiente.

```
object EV2 {
  val problem = amphip.stoch.EV(EEV1.problem, t2)
  val value   = 2461
  /* Compra y cancelación en t2 (sin ecenarios separados):
    v[P1,2,1] = 0
    v[P1,2,2] = 0
    v[P2,2,1] = 0
    v[P2,2,2] = 0
    v[P3,2,1] = 0
    v[P3,2,2] = 0
    v[P4,2,1] = 0
    v[P4,2,2] = 0
  */
}
```

La solución obtenida para la etapa 2 en el problema EV^2 indica que no se deben adquirir más cargamentos ($\bar{v}_c^2(s) = 0, c \in P, s \in S$) en base a las demandas promedio previstas para la etapa 3. El problema EEV^2 queda:

```
object EEV2 {
  val problem = {
    val vbar2 = param(ind(t in T, S, P) | t === 2)
    val fixV2 =
      st(ind(s in S, c in P) | gamma(c) <= H-2) {
        v(c,2,s) === vbar2(2,s,c)
      }

    val List(s1, s2) = EEV1.problem.stochData.scenariosByStage(t2)
    (EEV1.problem :+ fixV2)
      .stochScenarioData(vbar2, s1, PData.map(_ -> 0))
      .stochScenarioData(vbar2, s2, PData.map(_ -> 0))
  }
  val value = Double.PositiveInfinity // infactible
}
```

Al utilizar la solución de valor esperado en el problema con el árbol de escenarios completo, ocurre que no es posible satisfacer la demanda en todos los escenarios de la etapa 3, por lo que el problema EEV^2 es infactible, y su valor se considera $+\infty$, al ser un problema de minimización. Se observa que no es posible decidir comprar cargamentos en la etapa 3 porque todos los cargamentos a adquirir especifican un tiempo de entrega $\gamma_c = 1$, es decir, deben adquirirse al menos con un período de antelación. En este caso el valor de la solución estocástica es $+\infty$ porque el problema RP permite obtener una solución mientras que si se utilizan soluciones de valor esperado, la demanda queda insatisfecha en algunos escenarios de la etapa 3.

```
val VSS = EEV2.value - RP.value // +inf
```

Capítulo 4

Trabajos relacionados

En este capítulo se presentan y analizan lenguajes de dominio específico para programación estocástica destacados en la bibliografía consultada. Para cada DSL interesa particularmente los mecanismos para a definición del árbol de escenarios, la especificación de las probabilidades de los escenarios, la definición y asignación de los valores de los parámetros estocásticos, el manejo de escenarios separados y restricciones de no anticipatividad, la posibilidad de definir mecanismos alternativos para especificar las probabilidades, y la posibilidad de especificar los datos de los parámetros estocásticos en forma agregada.

Los ejemplos de código mostrados para cada DSL están vinculados con el Caso de estudio extendido (Sección 3.3), tomando su implementación en cada DSL como medida de comparación. En particular, la capacidad para definir mecanismos alternativos para especificar las probabilidades, es de interés en el Caso de estudio extendido para poder especificar las probabilidades de los escenarios finales y propagarlas automáticamente a los escenarios de las etapas anteriores. La posibilidad es especificar los datos de los parámetros estocásticos en forma agregada puede facilitar cumplir con el requerimiento del caso de estudio extendido de que los valores de las demandas se asignen siguiendo una distribución de probabilidad.

Se listan primero los DSL externos y luego los embebidos. En el Apéndice E se listan las implementaciones completas del Caso de estudio extendido en cada uno de los DSL analizados.

4.1. MathProg

Los problemas estocásticos basados en escenarios pueden formularse como problemas de programación entera utilizando la forma extensiva. Se toma la implementación en MathProg de la forma extensiva del Caso de estudio extendido, como referencia para el análisis de los distintos lenguajes.

La implementación más directa del problema es utilizando escenarios separados. Se definen conjuntos de MathProg para representar las etapas y los escenarios finales, y parámetros del modelo para las probabilidades y la determinación del escenario ancestro en un período.

```
param H;
set T := 1..H;
set S;

param pi{S};
param ancf{S, T};
```

Los parámetros estocásticos se definen como parámetros de MathProg indexados en los conjuntos seleccionados para representar las etapas y los escenarios finales.

```
param d{T, S};
```

El modelo MathProg debe incluir obligatoriamente las restricciones de no anticipatividad explícitas para mantener la consistencia, ya que el *solver* ignora que se trata de un problema de programación estocástica formulado con escenarios separados.

```
s.t. na_v{
    c in P, t in T, s1 in S, s2 in S :
      t <= H - gamma[c] and ancf[s1,t] == ancf[s2,t]
  }: v[c,t,s1] == v[c,t,s2];

s.t. na_x{
    c in A, t in T, s1 in S, s2 in S :
      t <= tau[c] - 1 and ancf[s1,t] == ancf[s2,t]
  }: x[c,t,s1] == x[c,t,s2];

s.t. na_y{
    t in T, s1 in S, s2 in S : ancf[s1,t] == ancf[s2,t]
  }: y[t,s1] == y[t,s2];
```

El uso de la formulación con escenarios separados obliga a que los datos sean especificados con redundancia. Por otra parte, la consistencia entre los datos y la estructura del árbol de escenarios debe ser mantenida manualmente, o mediante un sistema externo, ya que MathProg no puede chequearla. Lo anterior implica que tampoco pueda utilizarse la función `Uniform()` de MathProg para generar los datos de la demanda, ya que no respetarían la estructura del árbol, violando la no anticipatividad.

```
param d :
  1  2  3  4 :=
1  35 35 35 35
2  38 38 17 17
3  43 24 34 44 ;

param ancf :
     1 2 3 :=
  1  1 1 1
  2  1 1 2
  3  1 2 3
  4  1 2 4 ;
```

MathProg no dispone de las primitivas necesarias para generar los valores de las probabilidades de los escenarios finales siguiendo la distribución requerida, por lo que deben ser generados mediante un sistema externo.

En el Apéndice E.1 se lista el modelo completo correspondiente al Caso de estudio extendido (Sección 3.3) implementado en MathProg.

4.2. SAMPL

SAMPL es un lenguaje de modelado algebraico que extiende AMPL para soportar programación estocástica y otras técnicas de optimización que involucran incertidumbre (Valente et al. 2009). Puede generar el equivalente determinista y trabajar con los *solvers* disponibles para AMPL, o generar la representación del problema en SMPS (Gassmann 2005), la cual puede ser leída por *solvers* especializados como FortSP (Ellison et al. 2009).

Se trabaja con la formulación con escenarios separados, generándose automáticamente las restricciones de no anticipatividad. Para la definición del árbol de escenarios, se manejan un conjunto que representa el avance del tiempo y otro que representa los escenarios finales, este último marcado con la palabra clave `scenarioset`.

```
param H;
set T  := 1..H;
scenarioset S;
```

Las etapas se definen como una partición del conjunto que representa el tiempo, con variables auxiliares utilizado sufijos de AMPL.

```
let{c in P, t in T, s in S : t <= H - gamma[c]} v[c,t,s].stage = t;
let{c in A, t in T, s in S : t <= tau[c] - 1}   x[c,t,s].stage = t;
let{t in T, s in S}                             y[  t,s].stage = t;
```

La estructura del árbol de escenarios se declara con una entidad nueva identificada con la palabra clave `tree`, a la que se le asigna la estructura requerida utilizando constructores predefinidos. Se proveen constructores para definir árboles con estructuras "estándar" y constructores para definir árboles con estructuras arbitrarias.

```
three theTree := nway{2};
```

Para el caso de árboles con estructura estándar pueden utilizarse, por ejemplo, los constructores `nway` para indicar que cada nodo del árbol tiene un número fijo de descendientes por etapa, generándose un árbol perfecto, y `twostage` para indicar que se trata de un problema de solo dos etapas. Para árboles con estructura arbitraria, pueden utilizarse los constructores `bundle_list` que permite indicar los escenarios finales que son distintos en cada etapa, lo que permite agruparlos para mantener la no anticipatividad, y `tlist` que permite definir el árbol en forma muy compacta, requiriendo solamente la lista de etapas en las que cada escenario se origina, haciéndose distinguible del escenario anterior, considerando que el primer escenario se origina en la primera etapa.

Las probabilidades de los escenarios se declaran utilizando un parámetro especial indexado sobre el conjunto de escenarios, marcado con la palabra clave `probability`.

```
probability param pi{S};
```

Los parámetros estocásticos se definen como parámetros de AMPL indexados en el conjunto especial de escenarios finales, y se los marca utilizando la palabra clave `random`.

```
random param d{T, S};
```

Los valores de los parámetros estocásticos es posible especificarlos en forma compacta utilizando una variante de la sintaxis para datos de AMPL, donde se especifican los valores del parámetro para cada etapa y escenario, pero solamente a partir de la etapa en que el escenario se origina, siguiendo el mismo esquema que el constructor `tlist`.

```
random param d :=
  1 1 35
  2 1 38
  2 3 17
  3 1 43
  3 2 24
  3 3 34
  3 4 44;
```

Los valores de la demanda podrían ser muestreados con la distribución requerida utilizando las capacidades de AMPL. En cuanto a la probabilidad de los escenarios finales, AMPL permite trabajar con la distribución *Beta*, y se podría construir el histograma correspondiente y trabajar con la distribución discreta de probabilidades para asignar a los escenarios finales utilizando la biblioteca de funciones extendida. Una vez definidas las probabilidades de los escenarios finales, es posible propagar los valores a los escenarios de etapas anteriores por medio de un *script* de AMPL. A partir de la literatura consultada no queda claro en que medida el sistema de modelado que se propone para la generación de escenarios e interacción con los *solvers*, *AMPLDevSP*[1], resuelve o complejiza los puntos anteriores (Ellison et al. 2009; Valente et al. 2009). En cualquier caso, siempre es posible generar los valores externamente y asignarlos uno a uno a cada escenario.

En el Apéndice E.2 se lista el modelo completo correspondiente al Caso de estudio extendido (Sección 3.3) implementado en SAMPL. Para la definición del árbol de escenario se utiliza el constructor `nway{2}` para indicar que se debe utilizar un árbol perfecto de aridad 2.

4.3. StAMPL

StAMPL propone una extensión para AMPL para soportar programación estocástica, pero con la característica de no requerir definiciones explícitas para los conjuntos de etapas y escenarios (Fourer y Lopes 2009). Se trabaja con *subproblemas por etapa* donde no se considera la incertidumbre, permitiéndose el vínculo entre etapas mediante construcciones especiales del lenguaje. El modelo en StAMPL puede representarse en el formato SMPS y ser resuelto utilizando *solvers* especializados (Gassmann 2005).

El lenguaje no se ocupa directamente de la definición del árbol de escenarios. Para la asignación de los subproblemas a las etapas se utiliza la sentencia `definestage` en la primera línea, que indica la etapa o rango de etapas a la que pertenece el modelo a continuación.

```
# Definición de subproblema correspondiente a las etapas intermedias.
definestage 2 .. stages() - 1;
```

[1] https://optirisk-systems.com

Dentro de cada subproblema es posible hacer referencia a variables de otras etapas utilizando la función `parent()`, que devuelve el subproblema de la etapa anterior posicionado en el escenario padre del actual. La función `parent()` admite un argumento *count* para especificar cuantas etapas hacia atrás se debe ir. Asimismo, se proveen las funciones `stage()` y `stages()` que proveen del valor de la "etapa actual" y del número total de etapas a cada subproblema.

```
# Para determinar si un cargamento comprado llegará en el período
# actual, se debe acceder al valor de la variable en un escenario
# "ancestro" del actual, 'gamma[c]' períodos atrás.
s.t. acquired_fuel:
  u = sum{c in P : gamma[c] <= stage() - 1}
        q[c] * parent(gamma[c]).v[c];
```

Normalmente se trabaja con tres subproblemas, uno para la primera etapa, otro para las etapas intermedias, y el tercero para la última etapa. No se trabaja en forma directa con los escenarios. La no anticipatividad se consigue en forma implícita.

En el archivo de datos se debe especificar el número de etapas utilizando la sentencia `setstages(n)`, y luego se especifican, por cada subproblema, los valores de los conjuntos y parámetros que no dependen de los escenarios. Para separar los datos pertenecientes a cada subproblema, nuevamente se utiliza la sentencia `definestage` replicando la estructura de subproblemas del archivo de modelo.

```
setstages(3);

definestage 1;
# ...Datos para la etapa 1...

definestage 2 .. stages() - 1;
# ...Datos para las etapas intermedias...

definestage stages();
# ...Datos para la última etapa...
```

Para la definición de los escenarios, se provee una biblioteca en el lenguaje C++ que permite la definición del árbol nodo por nodo. Se utilizan dos clases principales que son `RVNode`, para la construcción de los nodos del árbol, y `RandomElem`, para la especificación de los valores de los parámetros estocásticos por escenario.

```
RandomElem d_1_1, d_2_1, d_2_2, d_3_1, d_3_2, d_3_3, d_3_4;
d_1_1.param = "d";
d_2_1.param = d_2_2.param = "d";
d_3_1.param = d_3_2.param = d_3_3.param = d_3_4.param = "d";

d_1_1.val = 35;
d_2_1.val = 38; d_2_2.val = 17;
d_3_1.val = 43; d_3_2.val = 24; d_3_3.val = 34; d_3_4.val = 44;
```

En cada nodo se indica la etapa a la que pertenece, su probabilidad, la lista de valores de los parámetros estocásticos y la lista de nodos hijos.

```
RVNode s_1_1, s_2_1, s_2_2, s_3_1, s_3_2, s_3_3, s_3_4;
s_1_1.stage = 1;
s_2_1.stage = s_2_2.stage = 2;
s_3_1.stage = s_3_2.stage = s_3_3.stage = s_3_4.stage = 3;

s_1_1.pathprob = 1;
s_2_1.pathprob = 0.5;
s_2_2.pathprob = 0.5;
s_3_1.pathprob = 0.15625;
s_3_2.pathprob = 0.34375;
s_3_3.pathprob = 0.34375;
s_3_4.pathprob = 0.15625;

s_1_1.elems.push_back(d_1_1);
s_2_1.elems.push_back(d_2_1);
s_2_2.elems.push_back(d_2_2);
// ...

s_1_1.children.push_back(s_2_1);
s_1_1.children.push_back(s_2_2);
// ...
```

La biblioteca permite almacenar el árbol construido en un archivo que es leído por el compilador del lenguaje al momento de generar la representación del problema en formato SMPS. El argumento dado por los autores para dejar la generación del árbol de escenarios por fuera del lenguaje de modelado algebraico, es que el proceso de construcción del árbol normalmente es un proceso muy complejo y se precisan estructuras de datos y capacidades del lenguaje, como la recursión, que son provistas más naturalmente por lenguajes de propósito general.

Un problema general que presenta el lenguaje, es que las declaraciones de las entidades del modelo que se utilizan en cada subproblema deben repetirse, lo que genera un nivel de redundancia alto. Por otra parte, la imposibilidad de manipular explícitamente los escenarios hace que no sea posible definir restricciones globales que consideren al mismo tiempo escenarios de distintos caminos del árbol. Restricciones como la anterior son necesarias, por ejemplo, para establecer cotas al valor esperado de las variables de decisión. La asignación de variables a las etapas es estática y no puede depender en forma arbitraria de la instancia de datos utilizada. Lo anterior puede subsanarse utilizando expresiones `if/else` para desactivar las variables en determinadas etapas, pero no deja de ser un artificio.

El lenguaje no provee ningún mecanismo directo para muestrear los valores de la demanda a partir de una distribución de probabilidad ni para construir el histograma con la distribución requerida para las probabilidades de los escenarios finales. Tampoco provee mecanismos para inferir las probabilidades de los escenarios interiores a partir de la de los escenarios finales. Como el árbol se construye en un lenguaje de propósito general como C++, se puede utilizar cualquier biblioteca disponible para los dos primeros puntos. Para la propagación de los valores de probabilidad desde los escenarios finales hacia los de etapas anteriores se requiere de un desarrollo específico por parte del usuario.

En el Apéndice E.3 se lista el modelo completo correspondiente al Caso de estudio extendido (Sección 3.3) implementado en StAMPL, incluyendo el código C++ para la construcción del árbol de escenarios.

4.4. AIMMS

AIMMS es un sistema de modelado que incluye un entorno integrado de desarrollo, un lenguaje de propósito general, y un lenguaje de modelado con soporte para programación estocástica (Bisschop y Roelofs 2006).

En AIMMS, se parte de un modelo determinista al que se le modifican los parámetros y variables que son estocásticos agregándoles atributos que establecen esta propiedad y, en el caso de las variables, la etapa a la que pertenecen.

```
DeclarationSection deterministic_model_declarations {
  Parameter H;
  Set T {
    SubsetOf    : Integers;
    Index       : t, t1;
    Definition  : { 1 .. H };
  }

  ! ...

  Parameter d {
    IndexDomain : t;
    Property    : Stochastic;
  }

  ! ...

  Variable v {
    IndexDomain : (p,t) | t <= H - gamma(p);
    Range       : Binary;
    Property    : Stochastic;
    Stage       : t;
  }

  !...
}
```

El conjunto de escenarios y otras entidades que no existen en el modelo determinista se agregan en un modelo aparte.

```
DeclarationSection stochastic_model_declarations {
  ! ... Declaraciones de 'S', 'pi' y entidades auxiliares ...
}
```

Mediante el uso de funciones predefinidas se puede construir el árbol de escenarios y resolver el modelo calculando el equivalente determinista o utilizando el algoritmo de descomposición de Benders (Benders 1962).

Se trabaja con la formulación de escenarios separados. Para la definición del árbol de escenarios se maneja un conjunto que representa los escenarios finales, que debe ser subconjunto del conjunto predefinido `AllStochasticScenarios`, el cual representa a todos los escenarios del sistema.

```
Set S {
  SubsetOf  : AllStochasticScenarios;
  Index     : s;
}
```

Las etapas se declaran por cada variable utilizando el atributo `Stage` y pueden estar asociadas a un conjunto arbitrario, en particular el conjunto

que representa el avance del tiempo. Adicionalmente, se utiliza un parámetro auxiliar `ScenarioTreeMap` para asociar a cada escenario final un escenario "representativo" por etapa que es utilizado para establecer la no anticipatividad.

```
ElementParameter ScenarioTreeMap {
  IndexDomain : (s,t);
  Range       : S;
}
```

El árbol de escenarios puede construirse programáticamente utilizando el módulo para generación de escenarios incluido en la biblioteca estándar del lenguaje (por defecto llamado `ScenGen`). Dicho módulo se configura en base a *callbacks*, utilizando el lenguaje de programación de propósito general que acompaña al lenguaje de modelado. Con esto puede construirse el árbol de escenarios utilizado en el Caso de estudio, aunque es un proceso relativamente complejo que requiere conocimiento significativo de dicho lenguaje y su biblioteca estándar. Mediante uno de los *callbacks* es posible definir el conjunto de escenarios finales junto con sus probabilidades, y otro *callback* permite asignar valores a los parámetros estocásticos por etapa, agrupando los escenarios según la no anticipatividad.

```
ScenGen::InitializeStochasticScenarioDataCallbackFunction     :=
  'InitializeStochasticScenarioDataCallbackFunction';
ScenGen::DetermineScenarioGroupsCallbackFunction              :=
  'DetermineScenarioGroupsCallbackFunction';
ScenGen::AssignStochasticDataForScenarioGroupCallbackFunction :=
  'AssignStochasticDataForScenarioGroupCallbackFunction';
ScenGen::CompareScenariosCallbackFunction                     :=
  'CompareScenariosCallbackFunction';

ScenGen::CreateScenarioData(H, S, pi, ScenarioTreeMap);
```

La no anticipatividad puede controlarse en forma explícita, generando restricciones de no anticipatividad, o implícita, mediante la sustitución de variables que deben tener el mismo valor por una única copia, al momento de la resolución. Toda restricción en la que participa un parámetro o variable marcada con la propiedad `Stochastic`, es adaptada automáticamente para incorpora el escenario como índice.

```
StochModelGMP := GMP::Instance::GenerateStochasticProgram(
  basic_MIP,
  AllStochasticParameters,
  AllStochasticVariables,
  S,
  pi,
  ScenarioTreeMap,
  "DetOrExp",
  GenerationMode  : 'SubstituteStochasticVariables',
  Name            : "stochModel" );
```

Algunas de las características generales del lenguaje y su biblioteca estándar son especialmente útiles para programación estocástica, como la posibilidad trabajar con un calendario y poder realizar planificaciones con "horizonte rodante", y las funciones para análisis estadístico. El lenguaje permite especificar unidades medida para las entidades.

En lenguaje tiene soporte directo para la especificación de la demanda si-

guiendo una distribución determinada. También tiene una biblioteca extensa de funciones para el trabajo con distribuciones de probabilidad y el análisis estadístico, pudiéndose construir distribuciones de probabilidad discretas como la utilizada en el Caso de estudio, por ejemplo, en base a la función `DistributionCumulative`.

Un problema general del lenguaje es que es muy verboso. Se estima que dicha característica se debe a que AIMMS está orientado a ser utilizado mediante una interfaz gráfica, por lo que las mayoría del "código" no es escrito directamente por el usuario, y el lenguaje está subordinado a dicha interfaz.

En el Apéndice E.4 se lista el modelo completo correspondiente al Caso de estudio extendido (Sección 3.3) implementado en AIMMS. No hace uso de la posibilidad de especificar unidades de medida.

4.5. PySP

PySP es un EDSL para problemas de programación estocástica embebido en Python (Watson et al. 2012; van Rossum y Drake 2011). Funciona como una extensión del EDSL para optimización matemática, Pyomo (Hart et al. 2017), también embebido en Python.

El EDSL trabaja con el modelo determinista y con un modelo predefinido con entidades que representan los escenarios, las etapas y la asignación de variables a etapas. El modelo determinista se describe en Pyomo.

```
model = AbstractModel()

model.H = Param()
model.T = RangeSet(1, model.H)

model.A = Set()
model.P = Set()
model.C = model.A | model.P

# ...

model.d = Param(model.T)

# ...

def PT_init(model):
  return ((c,t) for c in model.P
                for t in model.T if t <= model.H - model.gamma[c])

model.PT = Set(dimen=2, initialize=PT_init)
model.v  = Var(model.PT, within=Binary)
```

En forma separada, en un lenguaje externo basado en AMPL, se especifican los datos para la construcción del árbol de escenarios y juegos de valores para los parámetros a utilizar en cada escenario final o en cada nodo del árbol. Combinando la información de los distintos modelos y juegos de datos construye el modelo de programación estocástica, que puede resolverse con un *solver* específico para este tipo de problemas provisto por el propio paquete, con *solvers* de MIP mediante la construcción de la forma extensiva, o generar la representación en formato SMPS aunque con limitaciones.

El modelo determinista debe ser especificado en un archivo independiente,

por defecto llamado `ReferenceModel.py`, con la restricción de que los costos asociados a cada etapa estén declarados en variables independientes, para usarlas al momento de construir la función objetivo, tomando en cuenta los escenarios y su probabilidad.

```
def cost_rule(model, t):
  return \
    sum(               model.ca[c]  * model.q[c] * model.v[c,t] \
                          for c in model.P if (c,t) in model.PT) + \
    sum((model.cc[c] - model.ca[c]) * model.q[c] * model.x[c,t] \
                          for c in model.A if (c,t) in model.AT) + \
    model.h[t] * model.y[t]
```

Los nombres de las entidades para la construcción del árbol de escenarios están fijos en el modelo predefinido incluido en PySP. Utiliza un conjunto ordenado, `Stages`, para la etapas y otro conjunto, `Nodes`, para los nodos en el árbol de escenarios.

```
set Stages := T1 T2 T3;

set Nodes := N11
             N21 N22
             N31 N32 N33 N34;
```

El parámetro `ConditionalProbability` permite especificar la probabilidad de cada nodo.

```
param ConditionalProbability := N11 1.0
                                N21 0.5
                                N22 0.5
                                N31 0.3125
                                N32 0.6875
                                N33 0.6875
                                N34 0.3125;
```

Los nodos se asocian a las etapas mediante un parámetro indexado `NodeStage`.

```
param NodeStage := N11 T1
                   N21 T2
                   N22 T2
                   N31 T3
                   N32 T3
                   N33 T3
                   N34 T3;
```

La dependencia entre los nodos se declaran con un parámetro `Children`, que permite asociar cada nodo con el conjunto de nodos que son sus descendientes directos y así determinar la estructura del árbol.

```
set Children[N11] := N21 N22;
set Children[N21] := N31 N32;
set Children[N22] := N33 N34;
```

Adicionalmente se debe proveer un conjunto `Scenarios` con los escenarios finales, que debe ser consistente con la estructura del árbol definida, y un parámetro `ScenarioLeafNode` que asocia los escenarios finales con los nodos hoja del árbol con el que se corresponden.

```
set Scenarios := S1 S2 S3 S4;

param ScenarioLeafNode := S1 N31
                          S2 N32
                          S3 N33
                          S4 N34;
```

La asignación de las variables a las etapas se especifica con el parámetro `StageVariables` que asocia a cada etapa el conjunto de variables correspondiente. En dicha asignación se establece el vínculo entre el conjunto que representa el avance del tiempo (si existe) y las etapas.

```
set StageVariables[T1] := y[1] v[*,1] x[*,1] u[1] w[1];
set StageVariables[T2] := y[2] v[*,2]        u[2] w[2];
set StageVariables[T3] := y[3]               u[3] w[3];
```

Finalmente, los costos asociados a cada etapa se especifican con el parámetro `StageCost`.

```
param StageCost := T1 cost[1]
                   T2 cost[2]
                   T3 cost[3];
```

Los valores de los parámetros estocásticos pueden especificarse por escenario final o por nodo. En el primero caso, para cada escenarios final deben especificarse los valores de todos los parámetros del modelo, estocásticos o no, y debe controlarse la consistencia de los valores respecto al árbol de escenarios. En el segundo caso, por cada nodo, solo se especifican los valores de los parámetros que cambian respecto al nodo "padre", lo que es más eficiente y menos propenso a errores.

La forma en que se especifican los valores de los parámetros se indica mediante el parámetro `ScenarioBasedData` que vale `true` si los valores se especifican por escenario y `false` si es por nodo. Tanto los datos para definir el árbol de escenarios como los datos de los parámetros se especifican en archivos separados utilizando un DSL externo, basado en el lenguaje de especificación de datos de AMPL. Los datos anteriores pueden especificarse también programáticamente mediante *callbacks*, pero exige el uso de modelos "concretos" de Pyomo, que tienen todos los valores de los parámetros instanciados salvo los relativos a la estructura del árbol de escenarios y los parámetros estocásticos.

El modelo de programación estocástica que construye combinando los modelos y los juegos de datos trabaja con escenarios separados y siempre representa las restricciones de no anticipatividad en forma explícita.

Si se utilizan modelos "concretos" de Pyomo es posible especificar todos los valores de los parámetros programáticamente, lo que habilita el uso de las bibliotecas de Python, tanto para la generación de los datos de demanda con distribución uniforme como para la construcción del árbol con las probabilidades requeridas en los nodos hoja y su propagación hacia los "padres".

Un problema identificado es que el tipo que asigna a las expresiones depende de los datos, lo que puede producir sorpresas en tiempo de ejecución. Por ejemplo, con la instancia de datos utilizada, la restricción `single_cancellation` debe ser definida utilizando la función `inequality` en lugar del operador <= para forzar el tipo, debido a que, para algunos cargamentos, nunca se cumple el predicado que condiciona la sumatoria, por lo que trivialmente evalúa a cero y el

operador <= termina produciendo un booleano en lugar de un objeto de Pyomo.

```
def single_acquisition_rule(model, c):
  return sum(model.v[c,t] for t in model.T if (c,t) in model.PT) <= 1

def single_cancellation_rule(model, c):
  return inequality(0,
          sum(model.x[c,t] for t in model.T if (c,t) in model.AT), 1)
```

En el Apéndice E.5 se lista el modelo completo correspondiente al Caso de estudio extendido (Sección 3.3) implementado en PySP. Se utiliza un modelo "abstracto" de Pyomo y los datos se especifican por nodo usando el DSL externo.

4.6. Otros DSL para programación estocástica

Se comentan brevemente otros DSL analizados que incluyen soporte para algún tipo de programación estocástica pero que no contemplan el caso general estudiado en este trabajo, es decir, programación estocástica multietapa basa en escenarios donde las variables pueden depender de la historia de eventos pasados (el proceso estocástico no es markoviano) y pueden ser enteras. Se comentan tres DSL, todos embebidos en el lenguaje de programación Julia (Bezanson et al. 2017): StochasticPrograms, StructJuMP, y SDDP, todos basados en JuMP, que es un EDSL para problemas de optimización en Julia (Dunning et al. 2017).

De los tres EDSL anteriores, el que se considera más interesante y cercano en sus objetivos a este trabajo es StochasticPrograms (Biel y Johansson 2019), teniendo como uno de sus objetivos el modelado de problemas de programación estocástica en forma amigable y flexible, pero todavía no tiene soporte para especificar el árbol de escenarios y sus probabilidades.

El foco de StructJuMP está en explotar la estructura de problemas *separables en bloques* de modo de resolver cada bloque en forma distribuida en un *cluster* de cómputo, no incluyendo sintaxis específica para la definición del árbol de escenarios ni la especificación de los valores de los parámetros estocásticos (Huchette et al. 2014).

SDDP permite modelar problemas multietapa pero requiere que el proceso estocástico sea markoviano, y el problema convexo, lo que deja afuera el caso general estudiado en este trabajo (Dowson y Kapelevich 2021).

Para finalizar se observa que el uso de macros por parte de JuMP para construir sus objetos hace que el DSL no esté tan integrado al resto del lenguaje, debido a la sintaxis particular que provee Julia para esto, lo que se entiende introduce un poco de "ruido" al leer los programas.

Capítulo 5

Conclusiones

MathProg es un lenguaje declarativo y por lo tanto resulta natural representarlo usando programación funcional. Las jerarquías de clases selladas y las *case classes* permiten implementar AST inmutables y que el compilador realice chequeos de exhaustividad en los *pattern matchings*, lo que fue de mucha utilidad para el desarrollo del *deep embedding* del lenguaje.

La sintaxis flexible de Scala, junto con la posibilidad de sobrecargar operadores y las funciones de alto orden, permitió representar la sintaxis de MathProg en Scala con gran fidelidad. El soporte de Scala para evaluar las expresiones en forma no estricta permite soportar la declaración de entidades del modelo con definiciones recursivas en forma adecuada. El soporte para macros permite ir un paso más allá y utilizar los nombres de las definiciones en Scala como nombres de las entidades de MathProg, siendo, de todas formas, el único uso de metaprogramación en la construcción del *embedding*.

Utilizando *type classes* y el esquema con tres componentes (*ops*, *syntax* e *instances*) se tiene un método sistemático, si bien algo trabajoso, para codificar cualquier operador con sus respectivas sobrecargas que, a su vez, es extensible por terceros. El soporte para priorización de valores implícitos de Scala es suficiente para resolver las ambigüedades surgidas entre las instancias definidas para soportar la sintaxis completa de la sección de modelo de MathProg.

Existen algunas interferencias entre los operadores predefinidos y las palabras reservadas en Scala, y los métodos y operadores de MathProg, pero que pudieron resolverse en el EDSL con modificaciones menores, como el uso de === para la igualdad y nombres alternativos en caso de colisiones. Una forma de evitar este problema podría ser utilizar un *quoted* DSL (Najd et al. 2016).

Con el EDSL es posible codificar modelos con mayores garantías estáticas que utilizando MathProg en forma independiente, aunque sería deseable poder realizar controles adicionales como el alcance de los índices y la aridad de los conjuntos.

El *embedding* permite acceder a las capacidades del lenguaje anfitrión al momento de especificar el modelo, lo que resulta de gran utilidad por ejemplo en la especificación de los datos de parámetros y conjuntos. Lo anterior puede verse en el Caso de estudio, donde los valores de los parámetros del modelo que responden a determinadas distribuciones de probabilidad son generados usando bibliotecas de Scala y asignados directamente a los parámetros usando el EDSL.

Disponer de MathProg embebido en Scala permitió extenderlo para incor-

porarle soporte para programación estocástica multietapa basada en escenarios. El soporte se desarrolló como una capa de funcionalidad por encima de las desarrolladas para la representación de MathProg en Scala.

El soporte para la especificación del árbol de escenarios permite construirlo en etapas que sucesivamente refinan su estructura. Se introdujo el concepto de generador, que permite definir una estructura básica con poco esfuerzo y que se utiliza siempre que es necesario generar subárboles, en particular al agregar escenarios en etapas que no son la última. Las primitivas incorporadas al EDSL permiten construir árboles con estructura arbitraria en forma concisa. Aprovechando las capacidades de abstracción del lenguaje anfitrión, es posible definir funciones reutilizables para construir árboles con estructuras particulares, como por ejemplo árboles perfectos n-arios.

Las probabilidades de los escenarios se establecen junto con la estructura del árbol. Para la representación de la probabilidad se utilizan números racionales, los que permiten operar con matemática precisa hasta el momento en que se genera el modelo MathProg. Nuevamente, las primitivas agregadas al EDSL y las capacidades del lenguaje anfitrión, permiten definir mecanismos alternativos para definir las probabilidades, como por ejemplo, las requeridas por el Caso de estudio extendido, en el que las probabilidades se especifican únicamente para los escenarios finales siguiendo una distribución de probabilidad determinada.

La definición de los valores de los parámetros estocásticos sigue una lógica similar a la de la definición del árbol de escenarios, permitiendo definir los valores con distinto nivel de detalle, apoyándose en el generador. En este caso también resulta de utilidad disponer de un *embedding*, ya que permite definir funciones auxiliares para especificar los datos siguiendo patrones particulares y así evitar código repetitivo, como sucede con el problema de planificación financiera mostrado en el Capitulo 1.

Las extensiones incorporadas al EDSL permiten definir el modelo en forma sencilla con escenarios separados y, al mismo tiempo, especificar los datos de los parámetros estocásticos sin escenarios separados, evitando la redundancia. Codificar el lenguaje como un *deep embedding*, permitió realizar transformaciones al modelo operando sobre el AST para generar una forma extensiva del problema estocástico que es más eficiente que la que se construiría manualmente, generando automáticamente versiones alternativas de los parámetros estocásticos sin escenarios separados, además de las restricciones de no anticipatividad.

Se provee soporte básico para el cálculo de medidas de valor de los modelos estocásticos. Si bien el EDSL no incluye funcionalidad para trabajar con los resultados de la resolución del modelo, por lo que el cálculo de los EEV no puede automatizarse por completo, se incluye funcionalidad para el cálculo de las transformaciones necesarias al árbol de escenarios y a los valores de los parámetros estocásticos, para obtener el problema EV^t para una etapa t. También se incluye funcionalidad para "separar" la primera etapa de un modelo y así poder calcular el valor WS en un paso.

Una de las ventajas de soportar directamente a los problemas de programación estocástica, es que puede aprovecharse su estructura y utilizar algoritmos de resolución específicos. En este trabajo la resolución de los modelos siempre se realiza por medio de la generación del problema equivalente en MathProg, por lo que lo anterior no es posible. Sería interesante desarrollar una "capa de resolución" por encima del EDSL desarrollado que implemente métodos de

resolución aprovechando la estructura del problema, ya sean los propuestos Benders (1962) y Laporte y F. V. Louveaux (1993), o heurísticas. Un paso intermedio sería generar la representación del problema en el formato SMPS y así poder utilizar *solvers* específicos de programación estocástica como FortSP.

La resolución de los modelos se realiza generando el modelo MathProg. Sería deseable interactuar directamente con la API de los *solvers*, lo que sería más eficiente y facilitaría la inspección de los valores asignados a las variables luego de la resolución. Existe un soporte preliminar para interacción directa con `cplex` pero quedó fuera del alcance del trabajo por razones de tiempo.

Los modelos de programación estocástica se especifican utilizando la formulación con escenarios separados, controlándose la no anticipatividad en forma explícita mediante restricciones. Podrían incorporarse transformaciones adicionales al modelo que permitan manejar dos juegos de variables estocásticas en forma análoga a lo que sucede con los parámetros estocásticos, o que permitan la sustitución completa de los parámetros y variables estocásticas por sus versiones sin escenarios separados, pasando a controlar la no anticipatividad en forma implícita.

Se estima que existen muchas oportunidades de mejora de performance, en particular optimizando las estructuras de datos utilizadas para el manejo de los valores de los parámetros y conjuntos en memoria, optimizando las operaciones más frecuentes.

El lenguaje MathProg puede ser extendido en otras direcciones. Podrían soportarse algunas expresiones no lineales, pero "linealizables", como el producto, la implicancia lógica y tomar el máximo o mínimo de variables binarias. En todos los casos podrían introducirse las transformaciones necesarias al modelo en forma automática.

Podría explorarse incorporar mayores garantías estáticas a las provistas por el EDSL. En particular, sería deseable controlar estáticamente el alcance de los índices y la aridad de los conjuntos. El lenguaje podría extenderse para incorporar la especificación de unidades de medida a los parámetros y variables, y así poder realizar chequeos de consistencia, ya sea estáticamente o previo a la generación del modelo para la interacción con el *solver*.

Con los cambios introducidos en Scala 3, en particular en torno a la definición de *extension methods*, es posible codificar *type classes* en forma más directa reduciendo sensiblemente la cantidad de código necesario (Scala 3 Team 2021). Otras características introducidas en Scala 3, como *export clauses*, pueden ser exploradas en particular para dar mayor flexibilidad al esquema de priorización de instancias.

Sería interesante explorar *encodings* alternativos, en particular para ver si es posible lograr una mayor integración con el lenguaje anfitrión, aunque implique desviarse un poco de la sintaxis de MathProg. *Encodings* más abstractos podrían permitir una mayor integración con el lenguaje anfitrión y mayor potencialidad de reutilización de componentes por fuera del *embedding*.

Bibliografía

Beale, E. M. L. (1955). «On Minimizing a Convex Function Subject to Linear Inequalities». En: *Journal of the Royal Statistical Society: Series B (Methodological)* 17.2, págs. 173-184. DOI: 10.1111/j.2517-6161.1955.tb00191.x.

Benders, J. F. (1962). «Partitioning procedures for solving mixed-variables programming problems». En: *Numerische Mathematik* 4.1, págs. 238-252. ISSN: 0945-3245. DOI: 10.1007/BF01386316.

Bezanson, Jeff et al. (2017). «Julia: A fresh approach to numerical computing». En: *SIAM Review* 59.1, págs. 65-98. DOI: 10.1137/141000671.

Biel, Martin y Mikael Johansson (2019). «Efficient Stochastic Programming in Julia». En: *CoRR* abs/1909.10451. arXiv: 1909.10451.

Birge, John R. y François Louveaux (2011). *Introduction to Stochastic Programming*. 2nd. Springer. DOI: 10.1007/978-1-4614-0237-4.

Bisschop, Johannes (2006). *AIMMS - Optimization Modeling*. Lulu.com. ISBN: 1411698991.

Bisschop, Johannes y Marcel Roelofs (2006). *AIMMS - Language Reference*. Lulu.com. ISBN: 1411698975.

Cats (2018). *Cats: Lightweight, modular, and extensible library for functional programming*. Ver. 1.1.0. URL: https://github.com/typelevel/cats.

Dantzig, George B. (1955). «Linear Programming under Uncertainty». En: *Management Science* 1.3-4, págs. 197-206. DOI: 10.1287/mnsc.1.3-4.197.

Doeraene, Sébastien Jean R (2018). «Cross-Platform Language Design». Tesis doct. École polytechnique fédérale de Lausanne. DOI: 10.5075/epfl-thesis-8733.

Dowson, Oscar y Lea Kapelevich (2021). «SDDP.jl: A Julia Package for Stochastic Dual Dynamic Programming». En: *INFORMS Journal on Computing* 33.1, págs. 27-33. DOI: 10.1287/ijoc.2020.0987.

Dunning, Iain, Joey Huchette y Miles Lubin (2017). «JuMP: A Modeling Language for Mathematical Optimization». En: *SIAM Review* 59.2, págs. 295-320. DOI: 10.1137/15M1020575.

Ellison, Francis et al. (2009). *FortSP: A Stochastic Programming Solver*. OptiRisk Systems.

Escudero, Laureano F. et al. (2007). «The value of the stochastic solution in multistage problems». En: *TOP* 15.1, págs. 48-64. ISSN: 1863-8279. DOI: 10.1007/s11750-007-0005-4.

Ferrari, Germán (2021). *amphip: GNU MathProg EDSL for Scala with support for scenario-based multistage stochastic programming*. URL: https://github.com/gerferra/amphip.

Fourer, Robert, David M. Gay y Brian W. Kernighan (2002). *AMPL: A Modeling Language for Mathematical Programming*. Cengage Learning. ISBN: 0534388094.

Fourer, Robert y Leo Lopes (2009). «StAMPL: A Filtration-Oriented Modeling Tool for Multistage Stochastic Recourse Problems». En: *INFORMS Journal on Computing* 21.2, págs. 242-256. DOI: 10.1287/ijoc.1080.0289.

Fowler, Martin (2010). *Domain Specific Languages*. 1st. Addison-Wesley Professional. ISBN: 0321712943.

Gassmann, Horand I. (2005). «The SMPS Format for Stochastic Linear Programs». En: *Applications of Stochastic Programming*. SIAM, págs. 9-19. DOI: 10.1137/1.9780898718799.ch2.

Gibbons, Jeremy (2015). «Functional Programming for Domain-Specific Languages». En: *Central European Functional Programming School*. Springer, págs. 1-28. DOI: 10.1007/978-3-319-15940-9_1.

Gosling, James et al. (2018). *The Java® Language Specification*. Java SE 11 Edition. Oracle America, Inc. URL: https://docs.oracle.com/javase/specs/jls/se11/jls11.pdf.

Haoyi, Li (2017). *sourcecode: Scala library providing "source" metadata to your program*. Ver. 0.1.4. URL: https://github.com/lihaoyi/sourcecode.

Hart, William E. et al. (2017). *Pyomo — Optimization Modeling in Python*. Springer. DOI: 10.1007/978-3-319-58821-6.

Huchette, Joey, Miles Lubin y Cosmin Petra (2014). «Parallel algebraic modeling for stochastic optimization». En: *2014 First Workshop for High Performance Technical Computing in Dynamic Languages*, págs. 29-35. DOI: 10.1109/HPTCDL.2014.6.

Laporte, Gilbert y François V. Louveaux (1993). «The integer L-shaped method for stochastic integer programs with complete recourse». En: *Operations Research Letters* 13.3, págs. 133-142. ISSN: 0167-6377. DOI: 10.1016/0167-6377(93)90002-X.

Lindholm, Tim et al. (2018). *The Java® Virtual Machine Specification*. Java SE 11 Edition. Oracle America, Inc. URL: https://docs.oracle.com/javase/specs/jvms/se11/jvms11.pdf.

Makhorin, Andrew (2016a). *GNU Linear Programming Kit, Version 4.58*. URL: http://www.gnu.org/software/glpk/glpk.html.

— (2016b). *Modeling Language GNU MathProg. Language Reference for GLPK Version 4.58*. (DRAFT, February 2016). Free Software Foundation.

Marlow, Simon et al. (2010). *Haskell 2010 language report*. URL: https://www.haskell.org/onlinereport/haskell2010/.

Najd, Shayan et al. (2016). «Everything Old is New Again: Quoted Domain-Specific Languages». En: *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. PEPM '16. Association for Computing Machinery, págs. 25-36. ISBN: 9781450340977. DOI: 10.1145/2847538.2847541.

Nemhauser, George L. y Laurence A. Wolsey (1988). *Integer and Combinatorial Optimization*. Wiley interscience series in discrete mathematics and optimization. John Wiley & Sons, Inc. ISBN: 9780471828198. DOI: 10.1002/9781118627372.

Odersky, Martin, Lex Spoon y Bill Venners (2019). *Programming in Scala: A Comprehensive Step-by-Step Guide*. 4th. Artima Incorporation. ISBN: 9780981531618.

Odersky, Martin et al. (2016). *The Scala Language Specification Version 2.12*. URL: https://scala-lang.org/files/archive/spec/2.12/.

Oliveira, Bruno C.d.S., Adriaan Moors y Martin Odersky (2010). «Type Classes as Objects and Implicits». En: *ACM SIGPLAN Notices* 45.10, págs. 341-360. ISSN: 0362-1340. DOI: 10.1145/1932682.1869489.

Osheim, Erik (2012). «Generic Numeric Programming Through Specialized Type Classes». En: ScalaDays 2012. URL: http://plastic-idolatry.com/osheim-scaladays-2012.pdf.

Osheim, Erik y Tom Switzer (2018). *Spire: Powerful new number types and numeric abstractions for Scala*. Ver. 0.15.0. URL: https://github.com/typelevel/spire.

Paulson, Lawrence C. (1996). *ML For the Working Programmer*. 2nd. Cambridge University Press. ISBN: 0521570506.

Pilquist, Michael (2018). *simulacrum: First class syntax support for type classes in Scala*. Ver. 0.12.0. URL: https://github.com/typelevel/simulacrum.

Scala 3 Team, The (2021). *Scala 3 Reference*. URL: https://docs.scala-lang.org/scala3/reference/overview.html.

Scalaz (2018). *Scalaz: Principled Functional Programming in Scala*. Ver. 7.2.20. URL: https://github.com/scalaz/scalaz.

Shabalin, Denys (2020). «Just-in-time performance without warm-up». Tesis doct. École polytechnique fédérale de Lausanne. DOI: 10.5075/epfl-thesis-9768.

Shapiro, Alexander y Andy Philpott (2007). *A Tutorial on Stochastic Programming*. URL: https://sites.gatech.edu/alexander-shapiro/files/2021/03/TutorialSP.pdf.

Testuri, Carlos E., Héctor Cancela y Víctor M. Albornoz (2019). «Stochastic discrete lot-sizing with lead times for fuel supply optimization». En: *Pesquisa Operacional* 39.1, págs. 37-55. DOI: 10.1590/0101-7438.2019.039.01.0037.

Testuri, Carlos E., Bernardo Zimberg y Germán Ferrari (2012). *Modelado estocástico múltiple etapa de adquisición de combustible para la generación de electricidad bajo demanda incierta*. Inf. téc. 12-07. UR. FI – INCO. URL: https://hdl.handle.net/20.500.12008/3470.

Valente, Christian et al. (2009). «Extending Algebraic Modelling Languages for Stochastic Programming». En: *INFORMS Journal on Computing* 21.1, págs. 107-122. DOI: 10.1287/ijoc.1080.0282.

van Rossum, Guido y Fred L. Drake (2011). *The Python Language Reference Manual*. Network Theory Ltd. ISBN: 1906966141.

Wadler, Philip (1995). «Monads for functional programming». En: *Advanced Functional Programming*. Springer, págs. 24-52. ISBN: 9783540492702. DOI: 10.1007/3-540-59451-5_2.

Wadler, Philip y Stephen Blott (1989). «How to make ad-hoc polymorphism less ad hoc». En: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '89. Association for Computing Machinery, págs. 60-76. ISBN: 0897912942. DOI: 10.1145/75277.75283.

Wadler, Philip, Walid Taha y David MacQueen (1998). «How to add laziness to a strict language without even being odd». En: Workshop on Standard ML. URL: https://homepages.inf.ed.ac.uk/wadler/papers/lazyinstrict/lazyinstrict.ps.gz.

Watson, Jean-Paul, David L. Woodruff y William E. Hart (2012). «PySP: modeling and solving stochastic programs in Python». En: *Mathematical Programming Computation* 4.2, págs. 109-149. ISSN: 1867-2957. DOI: 10.1007/s12532-012-0036-1.

Zimberg, Bernardo, Carlos E. Testuri y Germán Ferrari (2019). «Stochastic modeling of fuel procurement for electricity generation with contractual terms and logistics constraints». En: *Computers & Chemical Engineering* 123, págs. 49-63. ISSN: 0098-1354. DOI: 10.1016/j.compchemeng.2018.12.021.

Apéndices

Apéndice A

Sintaxis concreta de MathProg

Se presenta la sintaxis concreta de MathProg utilizando EBNF.

⟨*model*⟩ ::= {⟨*stmt*⟩} [**end;**]

⟨*stmt*⟩ ::= ⟨*set-stmt*⟩ | ⟨*param-stmt*⟩ | ⟨*var-stmt*⟩ | ⟨*ctr-stmt*⟩ | ⟨*obj-stmt*⟩

⟨*set-stmt*⟩ ::= **set** ⟨*ident*⟩ [⟨*string-lit*⟩] [⟨*ind-expr*⟩] {[**,**] ⟨*set-att*⟩} **;**

⟨*param-stmt*⟩ ::= **param** ⟨*ident*⟩ [⟨*string-lit*⟩] [⟨*ind-expr*⟩] {[**,**] ⟨*param-att*⟩} **;**

⟨*var-stmt*⟩ ::= **var** ⟨*ident*⟩ [⟨*string-lit*⟩] [⟨*ind-expr*⟩] {[**,**] ⟨*var-att*⟩} **;**

⟨*ctr-stmt*⟩ ::= [**subject to** | **subj to** | **s.t.**] ⟨*ident*⟩ [⟨*string-lit*⟩] [⟨*ind-expr*⟩] **:**
⟨*ctr-def*⟩ **;**

⟨*ctr-def*⟩ ::= ⟨*num-expr*⟩ [**,**] **>=** ⟨*lin-expr*⟩ [**,**] **>=** ⟨*num-expr*⟩
| ⟨*num-expr*⟩ [**,**] **<=** ⟨*lin-expr*⟩ [**,**] **<=** ⟨*num-expr*⟩
| ⟨*lin-expr*⟩ [**,**] (**>=** | **<=** | **=**) ⟨*lin-expr*⟩

⟨*obj-stmt*⟩ ::= (**maximize** | **minimize**) ⟨*ident*⟩ [⟨*string-lit*⟩] [⟨*ind-expr*⟩] **:** ⟨*lin-expr*⟩ **;**

⟨*set-att*⟩ ::= **dimen** ⟨*dimen*⟩ | (**within** | **:=** | **default**) ⟨*set-expr*⟩

⟨*dimen*⟩ ::= ⟨*digit*⟩ | **1** ⟨*digit*⟩ | **20**;

⟨*param-att*⟩ ::= **integer** | **binary** | **symbolic**
| (**<** | **<=** | **>** | **>=** | **==** | **=** | **<>** | **!=** | **:=** | **default**) ⟨*simple-expr*⟩
| **in** ⟨*set-expr*⟩

⟨*var-att*⟩ ::= **integer** | **binary** | (**<=** | **>=** | **==** | **=**) ⟨*num-expr*⟩

⟨*expr*⟩ ::= ⟨*num-expr*⟩ | ⟨*sym-expr*⟩ | ⟨*set-expr*⟩ | ⟨*logic-expr*⟩ | ⟨*lin-expr*⟩

⟨*unsubscr-ref*⟩ ::= ⟨*ident*⟩

⟨*subscr-ref*⟩ ::= ⟨*ident*⟩ ⟨*subscript*⟩

⟨*simple-expr*⟩ ::= ⟨*num-expr*⟩ | ⟨*sym-expr*⟩

⟨*subscript*⟩ ::= **[** ⟨*simple-expr*⟩ {**,** ⟨*simple-expr*⟩} **]**

⟨*tuple*⟩ ::= **(** ⟨*simple-expr*⟩ {**,** ⟨*simple-expr*⟩} **)**

⟨*num-expr*⟩ ::= ⟨*cond-num-expr*⟩ | ⟨*num-expr-6*⟩

⟨*cond-num-expr*⟩ ::= **if** ⟨*logic-expr*⟩ **then** ⟨*num-expr-6*⟩ [**else** ⟨*num-expr-6*⟩]

⟨*num-expr-6*⟩ ::= ⟨*num-expr-5*⟩ {(**+** | **-** | **less**) ⟨*num-expr-5*⟩}

⟨*num-expr-5*⟩ ::= ⟨*iter-num-expr*⟩ | ⟨*num-expr-4*⟩

⟨*iter-num-expr*⟩ ::= (**sum** | **prod** | **min** | **max**) ⟨*ind-expr*⟩ ⟨*num-expr-4*⟩

⟨*num-expr-4*⟩ ::= ⟨*num-expr-3*⟩ {(***** | **/** | **div** | **mod**) ⟨*num-expr-3*⟩}

⟨*num-expr-3*⟩ ::= ⟨*num-expr-2*⟩ | (**+** | **-**) ⟨*num-expr-2*⟩

⟨*num-expr-2*⟩ ::= ⟨*num-expr-1*⟩ {(****** | **^**) ⟨*num-expr-1*⟩}

⟨*num-expr-1*⟩ ::= ⟨*num-ref*⟩ | ⟨*num-func-ref*⟩ | ⟨*num-lit*⟩ | **(** ⟨*num-expr*⟩ **)** | ⟨*num-expr*⟩

⟨*num-ref*⟩ ::= ⟨*subscr-ref*⟩ | ⟨*unsubscr-ref*⟩

⟨*num-func-ref*⟩ ::= **abs(** ⟨*num-expr*⟩ **)**
| **atan(** ⟨*num-expr*⟩ **)**
| **atan(** ⟨*num-expr*⟩ **,** ⟨*num-expr*⟩ **)**
| **card(** ⟨*set-expr*⟩ **)**
| **ceil(** ⟨*num-expr*⟩ **)**
| **cos(** ⟨*num-expr*⟩ **)**
| **exp(** ⟨*num-expr*⟩ **)**
| **floor(** ⟨*num-expr*⟩ **)**
| **gmtime()**
| **length(** ⟨*sym-expr*⟩ **)**
| **log(** ⟨*num-expr*⟩ **)**
| **log10(** ⟨*num-expr*⟩ **)**
| **max(** ⟨*num-expr*⟩ {**,** ⟨*num-expr*⟩} **)**
| **min(** ⟨*num-expr*⟩ {**,** ⟨*num-expr*⟩} **)**
| **round(** ⟨*num-expr*⟩ [**,** ⟨*num-expr*⟩] **)**
| **sin(** ⟨*num-expr*⟩ **)**
| **sqrt(** ⟨*num-expr*⟩ **)**
| **str2time(** ⟨*sym-expr*⟩ **,** ⟨*sym-expr*⟩ **)**
| **trunc(** ⟨*num-expr*⟩ [**,** ⟨*num-expr*⟩] **)**
| **Irand224()**
| **Uniform01()**

⟨*num-lit*⟩ ::= [**+** | **-**] ⟨*digits*⟩ [**.** ⟨*digits*⟩] [(**d** | **D** | **e** | **E**) [**+** | **-**] ⟨*digits*⟩]

⟨*sym-expr*⟩ ::= ⟨*cond-sym-expr*⟩ | ⟨*sym-expr-2*⟩

⟨*cond-sym-expr*⟩ ::= **if** ⟨*logic-expr*⟩ **then** ⟨*sym-expr*⟩ [**else** ⟨*sym-expr*⟩]

⟨*sym-expr-2*⟩ ::= ⟨*sym-expr-1*⟩ {**&** ⟨*sym-expr-1*⟩}

⟨*sym-expr-1*⟩ ::= ⟨*sym-ref*⟩ | ⟨*sym-func-ref*⟩ | ⟨*string-lit*⟩ | ⟨*num-expr*⟩
| **(** ⟨*sym-expr*⟩ **)** | ⟨*sym-expr*⟩

⟨*sym-ref*⟩ ::= ⟨*subscr-ref*⟩ | ⟨*unsubscr-ref*⟩

⟨*sym-func-ref*⟩ ::= **substr(** ⟨*sym-expr*⟩ **,** ⟨*num-expr*⟩ [**,** ⟨*num-expr*⟩] **)**
| **time2str(** ⟨*num-expr*⟩ **,** ⟨*sym-expr*⟩ **)**

⟨*string-lit*⟩ ::= **"** ⟨*char-seq*⟩ **"** | **'** ⟨*char-seq*⟩ **'**

⟨*set-expr*⟩ ::= ⟨*cond-set-expr*⟩ | ⟨*set-expr-5*⟩

⟨*cond-set-expr*⟩ ::= **if** ⟨*logic-expr*⟩ **then** ⟨*set-expr-5*⟩ **else** ⟨*set-expr-5*⟩

⟨*set-expr-5*⟩ ::= ⟨*set-expr-4*⟩ {(**union** | **diff** | **symdiff**) ⟨*set-expr-4*⟩}

⟨*set-expr-4*⟩ ::= ⟨*set-expr-3*⟩ {**inter** ⟨*set-expr-3*⟩}

⟨*set-expr-3*⟩ ::= ⟨*set-expr-2*⟩ {**cross** ⟨*set-expr-2*⟩}

⟨*set-expr-2*⟩ ::= ⟨*iter-set-expr*⟩ | ⟨*arith-set*⟩ | ⟨*set-expr-1*⟩

⟨*iter-set-expr*⟩ ::= **setof** ⟨*ind-expr*⟩ (⟨*tuple*⟩ | ⟨*simple-expr*⟩) | ⟨*ind-expr*⟩

⟨*arith-set*⟩ ::= ⟨*num-expr*⟩ **..** ⟨*num-expr*⟩ [**by** ⟨*num-expr*⟩]

⟨*set-expr-1*⟩ ::= ⟨*set-ref*⟩ | ⟨*set-lit*⟩ | **(** ⟨*set-expr*⟩ **)** | ⟨*set-expr*⟩

⟨*set-ref*⟩ ::= ⟨*subscr-ref*⟩ | ⟨*unsubscr-ref*⟩

⟨*set-lit*⟩ ::= **{ }**
| **{** ⟨*tuple*⟩ {**,** ⟨*tuple*⟩} **}**
| **{** ⟨*simple-expr*⟩ {**,** ⟨*simple-expr*⟩} **}**

⟨*ind-expr*⟩ ::= **{** ⟨*ind-entry*⟩ {**,** ⟨*ind-entry*⟩} [**:** ⟨*logic-expr*⟩] **}**

⟨*ind-entry*⟩ ::= ⟨*ind-entry-tuple*⟩ **in** ⟨*set-expr*⟩
| ⟨*ident*⟩ **in** ⟨*set-expr*⟩
| ⟨*set-expr*⟩

⟨*ind-entry-tuple*⟩ ::= **(** (⟨*ident*⟩ | ⟨*simple-expr*⟩) {**,** (⟨*ident*⟩ | ⟨*simple-expr*⟩) } **)**

⟨*logic-expr*⟩ ::= ⟨*logic-expr-5*⟩ {(**or** | ||) ⟨*logic-expr-5*⟩}

⟨*logic-expr-5*⟩ ::= ⟨*iter-logic-expr*⟩ | ⟨*logic-expr-4*⟩

⟨*iter-logic-expr*⟩ ::= (**forall** | **exists**) ⟨*ind-expr*⟩ ⟨*logic-expr-4*⟩

⟨*logic-expr-4*⟩ ::= ⟨*logic-expr-3*⟩ {(**and** | **&&**) ⟨*logic-expr-3*⟩}

⟨*logic-expr-3*⟩ ::= (**not** | **!**) ⟨*logic-expr-2*⟩

⟨*logic-expr-2*⟩ ::= ⟨*rel-expr*⟩ | ⟨*logic-expr-1*⟩

⟨*rel-expr*⟩ ::= ⟨*simple-expr*⟩ (**<** | **<=** | **>** | **>=**) ⟨*simple-expr*⟩
| ⟨*simple-expr*⟩ (**==** | **=** | **<>** | **!=**) ⟨*simple-expr*⟩
| (⟨*tuple*⟩ | ⟨*simple-expr*⟩) [**not** | **!**] **in** ⟨*set-expr*⟩
| ⟨*set-expr*⟩ [**not** | **!**] **within** ⟨*set-expr*⟩

⟨*logic-expr-1*⟩ ::= ⟨*num-expr*⟩ | **(** ⟨*logic-expr*⟩ **)** | ⟨*logic-expr*⟩

⟨*lin-expr*⟩ ::= ⟨*cond-lin-expr*⟩ | ⟨*lin-expr-5*⟩

⟨*cond-lin-expr*⟩ ::= **if** ⟨*logic-expr*⟩ **then** ⟨*lin-expr-5*⟩ [**else** ⟨*lin-expr-5*⟩]

⟨*lin-expr-5*⟩ ::= ⟨*lin-expr-4*⟩ {(**+** | **-**) ⟨*lin-expr-4*⟩}

⟨*lin-expr-4*⟩ ::= ⟨*iter-lin-expr*⟩ | ⟨*lin-expr-3*⟩

⟨*iter-lin-expr*⟩ ::= **sum** ⟨*ind-expr*⟩ ⟨*lin-expr-3*⟩

⟨*lin-expr-3*⟩ ::= ⟨*lin-expr-2*⟩ {(***** | **/**) ⟨*num-expr-3*⟩}
| ⟨*num-expr-3*⟩ {***** ⟨*lin-expr-2*⟩}

⟨*lin-expr-2*⟩ ::= ⟨*lin-expr-1*⟩ | (**+** | **-**) ⟨*lin-expr-1*⟩

⟨*lin-expr-1*⟩ ::= ⟨*lin-ref*⟩ | ⟨*num-func-ref*⟩ | ⟨*num-lit*⟩ | **(** ⟨*lin-expr*⟩ **)** | ⟨*lin-expr*⟩

⟨*lin-ref*⟩ ::= ⟨*subscr-ref*⟩ | ⟨*unsubscr-ref*⟩

⟨*ident*⟩ ::= ((⟨*alpha*⟩ | _) {⟨*alphanum*⟩ | _}) - ⟨*keyword*⟩

⟨*keyword*⟩ ::= **if** | **then** | **else** | **mod** | **div** | **less** | **within**
| **union** | **cross** | **diff** | **inter** | **symdiff** | **in** | **by**
| **and** | **or** | **not**

⟨*alphanum*⟩ ::= (⟨*letter*⟩ | ⟨*digit*⟩) {(⟨*letter*⟩ | ⟨*digit*⟩)}

⟨*alpha*⟩ ::= ⟨*letter*⟩ {⟨*letter*⟩}

⟨*letter*⟩ ::= **A** | **B** | **C** | **D** | **E** | **F** | **G** | **H** | **I** | **J** | **K** | **L** | **M**
| **N** | **O** | **P** | **Q** | **R** | **S** | **T** | **U** | **V** | **W** | **X** | **Y** | **Z**
| **a** | **b** | **c** | **d** | **e** | **f** | **g** | **h** | **i** | **j** | **k** | **l** | **m**
| **n** | **o** | **p** | **q** | **r** | **s** | **t** | **u** | **v** | **w** | **x** | **y** | **z**

⟨*digits*⟩ ::= ⟨*digit*⟩ {⟨*digit*⟩}

⟨*digit*⟩ ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

⟨*char-seq*⟩ ::= (* Secuencia de caracteres arbitraria. Los caracteres utilizadas como delimitadores (comillas simples o dobles) solo pueden formar parte de la secuencia si aparecen codificados dos veces *)

Apéndice B

Referencia rápida de Scala

Se comentan algunas de las características generales del lenguaje de programación Scala, como una referencia rápida del mismo. Para una descripción en detalle del lenguaje se sugiere referirse a Odersky et al. (2019) y Odersky et al. (2016).

Scala corre en forma nativa sobre la máquina virtual de Java, de la que hereda su jerarquía de clases. La jerarquía de clases de Java es extendida por Scala para incorporar los tipos primitivos y el tipo `Nothing`, que actúa como *bottom type*. La Figura B.1 muestra la jerarquía de clases de Scala.

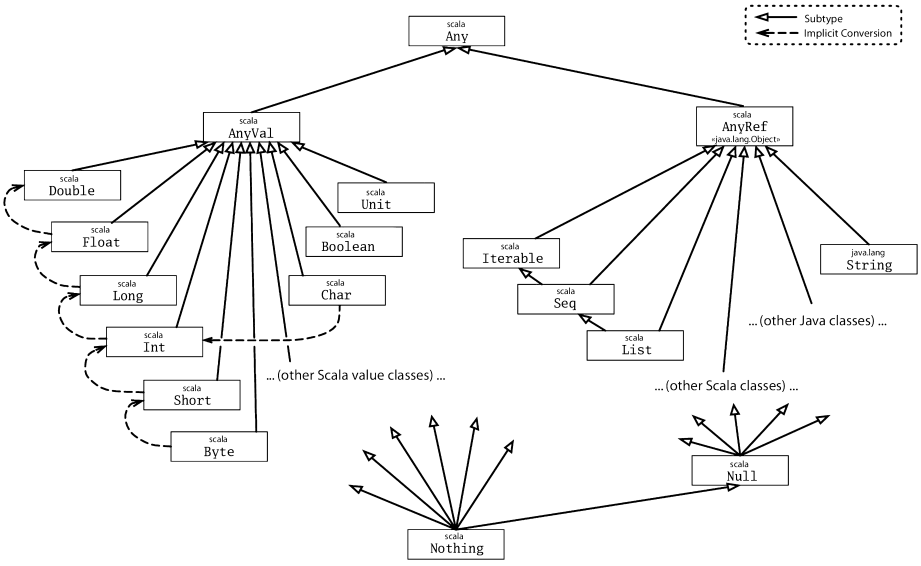


Figura B.1: Jerarquía de clases de Scala (Odersky et al. 2019)

Las clases en Scala se declaran con la palabra reservada `class`, seguida del nombre de la clase y la lista de parámetros del constructor entre paréntesis:

```
class A(x: Int, b: String) {
  // cuerpo de la clase
}
```

El cuerpo de la clase está constituido por una lista de declaraciones de campos y métodos. Los campos pueden declararse con `val`, para campos no modificables, o `var`, para campos modificables. Si el tipo asociado a un campo declarado con `val` es inmutable, el campo es efectivamente una constante. Los métodos se declaran con `def`, seguidos del nombre, la lista de parámetros entre paréntesis, y su cuerpo luego del signo de igual:

```
class A(x: Int, a: String) {
  val y = x + 1
  def m1(b: String) = a + b
  def m2(z: Int) = {
    val n = y * 2
    z + n
  }
}
```

Si el cuerpo de los métodos tiene una sola línea puede especificarse directamente, sin llaves. En otro caso es obligatorio utilizarlas, ocurriendo lo mismo con los bloques de código. Los métodos o funciones pueden tener declaraciones locales, usando también `val`, `var` o `def`, y el valor de retorno es el resultado de evaluar la última expresión.

Las clases y los métodos soportan polimorfismo paramétrico. La lista de parámetros de tipo se declara utilizando paréntesis rectos luego del nombre de la clase o método:

```
class X[A,B](n: Int, a: A, b: B) {
  def m[C](a1: A, c: C) = if (n > 0) a1 else c
  // ...
}
```

Scala no diferencia entre operadores y métodos, los operadores son métodos. En general, cualquier método que tenga un único argumento puede invocarse sin necesidad de utilizar el carácter punto para seleccionarlo y paréntesis para especificar el argumento, es decir, se puede escribir `a >= b` en lugar de `a.>=(b)`, siendo ambas expresiones equivalentes. Scala permite también "operadores posfijos" aunque su uso está desalentado.

Scala soporta inferencia de tipos a nivel local, siendo necesario especificar los tipos para los parámetros de constructores y métodos. Los tipos pueden ser declarados explícitamente con fines de documentación o para forzar un tipo específico:

```
val x = 1       // tipo 'Int' inferido
val y: Int = 1  // tipo explícito
```

En el caso de declaraciones recursivas, siempre es necesario especificar el tipo de retorno.

Las funciones en Scala son valores de tipo `Function[A,B]`, que también puede escribirse `A => B`. Scala soporta literales de función, cuya sintaxis consiste en la lista de parámetros entre paréntesis seguida de una flecha => y el cuerpo de la función:

```
val f = (x: Int, y: Int) => x + y // tipo de f inferido
val g: (Int, Int) => Int = (x, y) => x + y // tipo del literal inferido
```

Los tipos de los parámetros en los literales de función pueden ser inferidos a partir del tipo destino, como sucede en la declaración de `g` anterior. Si el literal de función tiene un solo parámetro, los paréntesis pueden ser omitidos.

Existen formas abreviadas de escribir funciones utilizando el subguión (_) como marcador de posición para uno o más parámetros, en la medida que el compilador pueda determinar el tipo apropiado:

```
val f : Int => Boolean    = _ > 0               // x => x > 0
val g1: (Int, Int) => Int = _ + _               // (x,y) => x + y
val g2                    = (_: Int) + (_: Int) // (x,y) => x + y
```

Los métodos y funciones pueden aplicarse parcialmente utilizando el subguión (_) para indicar los parámetros omitidos:

```
def m(x: Int, y: String): String = y * x
val f = (x: Int, y: String) => y * x

val mp = m(_, "a")  // : Int => String
val fp = f(2, _)    // : String => String

val g1 = m _ // : (Int, String) => String
val g2: (Int, String) => String = m
```

Al aplicarlos parcialmente, los métodos son convertidos a funciones. Es posible indicar que toda la lista de parámetros es omitida utilizando un subguión, sin paréntesis, como en el caso de `g1`. Si el tipo destino es una función con el tipo apropiado, los métodos pueden convertirse a funciones directamente, como en el caso de `g2`. Las conversiones anteriores hacen que pueda considerarse a métodos y funciones como intercambiables, si bien son entidades diferentes en Scala.

Scala soporta *pattern matching* permitiendo diversos tipos de patrones, por ejemplo:

`x: T`	Tipo. Se empareja con cualquier valor de tipo `T`, ligando la variable `x` al valor.
`"a"`	Patrón constante (válido para cualquier literal). Se empareja con exactamente ese valor.
`X`	Variable en mayúsculas. Se considera un patrón constante.
`A(x,y)`	Constructor. Se empareja con el constructor `A` ligando las variables `x` e `y` a los parámetros del constructor.
`x`	Variable. Se empareja con cualquier valor ligándolo a la variable `x`.
`_`	Idem anterior, pero no liga ninguna variable.
`x @ pat`	Patrón ligado. Se empareja según el patrón `pat` ligando el valor emparejado a la variable `x`.
`` `x` ``	Variable en minúsculas entre *backticks*. Se considera un patrón constante.
`(x,y)`	Tupla. Patrón constructor utilizando el constructor de tuplas.

Se observa que los patrones anteriores pueden combinarse y formar patrones anidados más complejos.

Las clases de Scala pueden tener el modificador *case*. Las *case classes* son clases a las que se le agrega funcionalidad adicional que las hace más convenientes de usar como constructores de datos. En particular, las *case classes* pueden ser utilizadas para deconstruir valores del tipo correspondiente cuando se utiliza *pattern matching*. Otra característica destacada de las *case classes* es que todos los parámetros declarados en el constructor, automáticamente quedan disponibles como variables finales (declaradas con `val` en Scala), por lo que son adecuadas para la definición de estructuras inmutables. Las *case classes* tienen implementaciones predeterminadas y consistentes para los métodos de Java `equals`, `hashCode` y `toString`, los que de todos modos se pueden sobreescribir para definir comportamientos específicos.

Para crear copias de objetos inmutables con algunas modificaciones, las *case classes* de Scala incluyen un método `copy` que permite crear una copia de la instancia utilizando los valores actuales como valores por defecto para cada parámetro de su constructor. Como Scala soporta valores por defecto para parámetros y parámetros nombrados, es posible llamar al método `copy` especificando únicamente los parámetros del constructor que debieran cambiar, manteniendo para los otros los valores por defecto. Gracias a la inmutabilidad, las estructuras que no cambian pueden compartirse libremente entre la instancia original y la "actualizada", sin ningún tipo de riesgo. Como ejemplo, se muestra el método `copy` generado para una *case class* junto con un posible uso:

```
case class Person(name   : String,
                  surname: String
                  age    : Int,
                  address: Address) {
  // el método `copy` es generado automáticamente para las case classes
  def copy(name   : String  = name,
           surname: String  = surname,
           age    : Int     = age,
           address: Address = address) =
    Person(name, surname, age, address)

  // posible uso de `copy` para ``actualizar'' edad
  def withAge(newAge: Int) = copy(age = newAge)
}
```

Los *traits* de Scala son similares a las clases pero se diferencian en que no tienen constructor y habilitan una forma de herencia múltiple. En Scala, las clases solo pueden extender de una única clase, pero pueden "mezclar" (*mix in*) cualquier número de *traits*.

En Scala es posible especificar que una clase o *trait* está sellada (*sealed*). Las clases o *traits* selladas no pueden ser extendidas, en el sentido de orientación a objetos, por fuera del archivo en que son definidas. Esto permite que la cantidad de subclases esté determinada en tiempo de compilación y puedan realizarse chequeos de exhaustividad en los *pattern matchings*, con lo que pueden modelarse tipos de datos algebraicos. Por ejemplo, utilizando una *trait* sellada y dos *case classes*, podemos representar un árbol binario con información en las hojas:

```
sealed trait BTree[A]
case class Leaf[A](value: A)                          extends BTree[A]
case class Fork[A](left: BTree[A], right: BTree[A]) extends BTree[A]
```

y definir funciones o métodos mediante recursión estructural sobre el tipo utilizando *pattern matching*, con los chequeos de exhaustividad de casos correspondientes. Por ejemplo:

```
def toList[A](tree: BTree[A]): List[A] = tree match {
  case Leaf(v)    => List(v)
  case Fork(l, r) => toList(l) ::: toList(r)
}
```

Scala dispone de mecanismos elaborados para la declaración y uso de valores y conversiones implícitas. El detalle de los mismos excede el alcance de esta referencia rápida, puede consultarse en Odersky et al. (2019) y Odersky et al. (2016). Las conversiones implícitas se aplican sobre una expresión cuando su tipo no coincide con el tipo esperado, cuando se selecciona un miembro no definido, o cuando se intenta aplicar un método que es definido por el tipo de la expresión pero que no es aplicable a los argumentos especificados. Los últimos dos casos permiten que las conversiones implícitas puedan utilizarse para codificar *extension methods* en Scala. Para esto se requiere definir una nueva clase que incluya los métodos a agregar al tipo objetivo, y una conversión implícita entre el tipo objetivo y la nueva clase. Por ejemplo:

```
// tipo objetivo. clase a extender
class C {
  def methodX(a: Int) = // ...
}

// nueva clase con métodos a agregarle a 'C'
class CExtension(c: C) {
  def methodY(b: String) = // ...
}

// conversión implícita
implicit def CToCExtension(c: C) = new CExtension(c)

val c = new C
/*
  aplicación de la conversión implícita para poder usar 'methodY' en
  el tipo 'C'
*/
c.methodY("hello")
```

La clase y conversión implícita anteriores pueden definirse en forma más concisa utilizando clases implícitas, las que, además de definir una clase, introducen una conversión implícita entre el tipo del argumento en su constructor y la clase que se está definiendo:

```
/*
  define nueva clase con métodos a agregarle a 'C' y la conversión
  implícita necesaria, en un paso
*/
implicit class CExtension(c: C) {
  def methodY(b: String) = // ...
}
```

El tipo de las funciones en Scala, `A => B`, no da información sobre si estas son totales o parciales. Scala provee soporte básico para trabajar con funciones parciales a través de un tipo especial, denominado `PartialFunction[A,B]`. Este tipo especial de funciones permite verificar en tiempo de ejecución si están definidas para un valor particular del dominio, mediante la función `isDefinedAt`. Los análisis de casos pueden utilizarse para definir funciones parciales en Scala. En ese caso, la función está definida solamente para los valores que pueden emparejarse con alguno de los patrones definidos en cada caso.

```
/*
  los análisis de casos pueden utilizarse como "literal de función
  parcial" si el tipo destino es 'PartialFunction'
*/
val pf: PartialFunction[BTree[Int], Int] = {
  case Leaf(v) => v
}

pf.isDefinedAt(Leaf(1))                // true
pf.isDefinedAt(Fork(Leaf(1), Leaf(3))) // false
```

Una forma alternativa de trabajar con las funciones parciales es convertirlas en funciones normales de Scala pero de tipo `A => Option[B]`, mediante el método `lift`. El método `lift` envuelve el resultado de la función en un `Option`, que es el tipo predefinido en Scala para representar la posibilidad de tener un valor o no. En el caso que la función no esté definida para el argumento con el que se está invocando, devolverá `None` (sin valor), mientras que si está definida devolverá `Some(b)` con el resultado de la función original, `b`.

```
val lifted = pf.lift // tipo: BTree[Int] => Option[Int]

lifted(Leaf(1))                // Some(1)
lifted(Fork(Leaf(1), Leaf(3))) // None
```

La biblioteca estándar de Scala incluye, entre otros, los tipos `List`, `Option` y `Map`, que representan listas, valores opcionales, y mapas genéricos. Las listas se definen mediante los constructores `::` (*cons*) y `Nil`. Algunos métodos comunes de listas son:

```
/* Se muestra nombre y tipo simplificado.
 * El primer argumento corresponde al objeto sobre el que se aplica el
 * método, salvo en `:::` que es el segundo por terminar en `:`.
 */

/* concatenación de listas */
::: : List[A] => List[A] => List[A]

/* aplica la función a cada elemento de la lista */
map : List[A] => (A => B) => List[B]

/* aplica la función a cada elemento de la lista, concatenando los
 * resultados
 */
flatMap : List[A] => (A => List[B]) => List[B]

/* combina cada elemento de la lista con su índice */
zipWithIndex: List[A] => List[(A,Int)]

/* aplica un operador binario a un valor inicial y a todos los
 * elementos de la lista, recorriendo de izquierda a derecha
 */
foldLeft: List[A] => B => ((B,A) => B) => B

/* aplica un operador binario a todos los elementos de la lista y a un
 * valor inicial, recorriendo de derecha a izquierda
 */
foldRight: List[A] => B => ((A,B) => B) => B
```

Como se comentó anteriormente, los valores opcionales se definen mediante los constructores `Some` y `None`, que representan la existencia y ausencia de valores respectivamente. Algunos métodos comunes de `Option` son:

```
/* Se muestra nombre y tipo simplificado.
 * El primer argumento corresponde al objeto sobre el que se aplica el
 * método.
 */

/* devuelve el valor contenido en el Option o un valor por defecto si
 * es vacío
 */
getOrElse : Option[A] => A => A

/* devuelve el primer Option si no es vacío, o el segundo en otro caso
 */
orElse: Option[A] => Option[A] => Option[A]

/* aplica la función al elemento, si existe */
map : Option[A] => (A => B) => Option[B]

/* aplica la función al elemento, si existe, "aplanando" el resultado
 */
flatMap : Option[A] => (A => Option[B]) => Option[B]

/* devuelve el resultado de aplicar la función si no es vacío, o un
 * valor por defecto en otro caso
 */
fold: Option[A] => B => (A => B) => B
```

Las tuplas en Scala pueden construirse directamente con `(,)`. En el caso específico de pares $(clave, valor)$ suele utilizarse el método `->` que hace visualmente más

clara la dependencia entre los valores. Los mapas se definen a partir de un constructor que recibe un número variable de pares $(clave, valor)$, por ejemplo:

```
val map = Map(1 -> "a", 2 -> "b")
```

Para buscar un elemento en un mapa se aplica el mapa a la clave, como si fuera una función, por ejemplo `map(1)`. Una versión alternativa, más segura, del método anterior es utilizar el método `get`, es decir `map.get(1)`, que devuelve un `Option` con el valor, si existe, o vacío en otro caso.

Para terminar con la referencia rápida, Scala dispone de *for comprehensions* que permiten expresar secuencias de computaciones en un contexto, con una notación liviana. Algunos ejemplos:

```
/* for-loop */
for {
  i <- List(1,2,3)
} {
  println(i)
}

/* genera lista con todas las combinaciones */
for {
  i <- List(1,2,3)
  j <- List(4,5)
} yield {
  (i,j)
}

/* genera lista con las combinaciones que cumplen el predicado */
for {
  i <- List(1,2,3)
  j <- List(4,5)
    if j % 2 == 0
} yield {
  (i,j)
}

/*
 * Devuelve un 'Option' con el valor de 'map.get(x)', si existe,
 * o el valor de 'map.get(y)', si existe. En otro caso devuelve un
 * 'Option' vacío ('None')
 */
for {
  optX = map.get(x)
  optY = map.get(y)
  z <- optX.orElse(optY)
} yield {
  z
}
```

Apéndice C

AST completo del *deep embedding*

```
package amphip.model

object ast {

  final case class Model(statements: List[Stat])

  /*
   * DECLARATIONS
   */
  sealed trait Decl extends Product with Serializable {
    def name: SymName
  }

  final case class DummyIndDecl(
      name: SymName,
      synthetic: Boolean = false
  ) extends Decl

  /*
   * STATEMENTS
   */
  sealed trait Stat extends Decl

  final case class SetStat(
      name: SymName,
      alias: Option[StringLit] = None,
      domain: Option[IndExpr] = None,
      atts: List[SetAtt] = Nil
  ) extends Stat

  final case class ParamStat(
      name: SymName,
      alias: Option[StringLit] = None,
      domain: Option[IndExpr] = None,
      atts: List[ParamAtt] = Nil
  ) extends Stat

  final case class VarStat(
      name: SymName,
      alias: Option[StringLit] = None,
```

```
    domain: Option[IndExpr] = None,
    atts: List[VarAtt] = Nil
) extends Stat

sealed trait ConstraintStat extends Stat

final case class EqConstraintStat(
    name: SymName,
    alias: Option[StringLit] = None,
    domain: Option[IndExpr] = None,
    left: LinExpr,
    right: LinExpr
) extends ConstraintStat

final case class LTEConstraintStat(
    name: SymName,
    alias: Option[StringLit] = None,
    domain: Option[IndExpr] = None,
    left: LinExpr,
    right: LinExpr
) extends ConstraintStat

final case class GTEConstraintStat(
    name: SymName,
    alias: Option[StringLit] = None,
    domain: Option[IndExpr] = None,
    left: LinExpr,
    right: LinExpr
) extends ConstraintStat

final case class DLTEConstraintStat(
    name: SymName,
    alias: Option[StringLit] = None,
    domain: Option[IndExpr] = None,
    lower: NumExpr,
    expr: LinExpr,
    upper: NumExpr
) extends ConstraintStat

final case class DGTEConstraintStat(
    name: SymName,
    alias: Option[StringLit] = None,
    domain: Option[IndExpr] = None,
    upper: NumExpr,
    expr: LinExpr,
    lower: NumExpr
) extends ConstraintStat

sealed trait ObjectiveStat extends Stat

final case class Minimize(
    name: SymName,
    alias: Option[StringLit] = None,
    domain: Option[IndExpr] = None,
    expr: LinExpr
) extends ObjectiveStat

final case class Maximize(
    name: SymName,
    alias: Option[StringLit] = None,
    domain: Option[IndExpr] = None,
    expr: LinExpr
```

```
) extends ObjectiveStat

sealed trait SetAtt

final case class SetDimen(expr: Int) extends SetAtt

final case class SetWithin(expr: SetExpr) extends SetAtt

final case class SetAssign(expr: SetExpr) extends SetAtt

final case class SetDefault(expr: SetExpr) extends SetAtt

sealed trait ParamAtt

final case class ParamLT(expr: SimpleExpr) extends ParamAtt

final case class ParamLTE(expr: SimpleExpr) extends ParamAtt

final case class ParamEq(expr: SimpleExpr) extends ParamAtt

final case class ParamNEq(expr: SimpleExpr) extends ParamAtt

final case class ParamGT(expr: SimpleExpr) extends ParamAtt

final case class ParamGTE(expr: SimpleExpr) extends ParamAtt

final case class ParamIn(expr: SetExpr) extends ParamAtt

final case class ParamAssign(expr: SimpleExpr) extends ParamAtt

final case class ParamDefault(expr: SimpleExpr) extends ParamAtt

sealed trait VarAtt

final case class VarGTE(expr: NumExpr) extends VarAtt

final case class VarLTE(expr: NumExpr) extends VarAtt

final case class VarEq(expr: NumExpr) extends VarAtt

case object Integer extends ParamAtt with VarAtt

case object Binary extends ParamAtt with VarAtt

case object Symbolic extends ParamAtt

/*
 * EXPRESSIONS
 */
sealed trait Expr

sealed trait SimpleExpr extends Expr // groups NumExpr and SymExpr

sealed trait SetExpr extends Expr

sealed trait LogicExpr extends Expr

sealed trait LinExpr extends Expr

/*
 * Groups SetRef, ParamRef, VarRef, ConstraintRef, ObjectiveRef
 */
```

```
sealed trait StatRef

/*
 * SIMPLE
 */
sealed trait NumExpr extends SimpleExpr with LogicExpr with LinExpr

sealed trait SymExpr extends SimpleExpr

/*
 * ParamStat reference with lazy semantic to allow recursive
 * definitions.
 * Overrides `hashCode`, `equals` and `Product` methods to only
 * consider the name of the ParamStat and avoid infinite loops.
 */
final class ParamRef(
    paramThunk: () => ParamStat,
    val subscript: List[SimpleExpr] = Nil
) extends NumExpr
    with SymExpr
    with StatRef
    with Product
    with Serializable {
  lazy val param: ParamStat = paramThunk()

  def copy(
      paramThunk: () => ParamStat = paramThunk,
      subscript: List[SimpleExpr] = subscript
  ) =
    new ParamRef(paramThunk, subscript)

  lazy private val product = (param.name, subscript)

  override def equals(obj: Any): Boolean = obj match {
    case that: ParamRef =>
      (that canEqual this) && this.product.equals(that.product)
    case _ => false
  }
  override def hashCode(): Int = this.product.hashCode()

  override def toString(): String = productPrefix + product

  // from Product
  override def canEqual(that: Any): Boolean =
    that.isInstanceOf[ParamRef]
  override def productArity: Int = product.productArity
  override def productElement(n: Int): Any =
    product.productElement(n)
  override def productPrefix: String = "ParamRef"
}
object ParamRef {
  def apply(
      param: => ParamStat,
      subscript: List[SimpleExpr] = Nil
  ) = new ParamRef(() => param, subscript)

  def unapply(x: ParamRef): Option[(ParamStat, List[SimpleExpr])] =
    Some((x.param, x.subscript))
}

final case class DummyIndRef(dummyInd: DummyIndDecl)
    extends NumExpr
```

```
    with SymExpr

/*
 * NUMERIC
 */
final case class CondNumExpr(
    test: LogicExpr,
    ifTrue: NumExpr,
    otherwise: Option[NumExpr] = None
) extends NumExpr

final case class NumAdd(left: NumExpr, right: NumExpr)
    extends NumExpr

final case class NumSub(left: NumExpr, right: NumExpr)
    extends NumExpr

final case class NumLess(left: NumExpr, right: NumExpr)
    extends NumExpr

final case class NumSum(indexing: IndExpr, integrand: NumExpr)
    extends NumExpr

final case class NumProd(indexing: IndExpr, integrand: NumExpr)
    extends NumExpr

final case class NumMax(indexing: IndExpr, integrand: NumExpr)
    extends NumExpr

final case class NumMin(indexing: IndExpr, integrand: NumExpr)
    extends NumExpr

final case class NumMult(left: NumExpr, right: NumExpr)
    extends NumExpr

final case class NumDiv(left: NumExpr, right: NumExpr)
    extends NumExpr

final case class NumDivExact(left: NumExpr, right: NumExpr)
    extends NumExpr

final case class NumMod(left: NumExpr, right: NumExpr)
    extends NumExpr

final case class NumUnaryPlus(x: NumExpr) extends NumExpr

final case class NumUnaryMinus(x: NumExpr) extends NumExpr

final case class NumRaise(left: NumExpr, right: NumExpr)
    extends NumExpr

final case class NumLit(num: BigDecimal) extends NumExpr

sealed trait NumFuncRef extends NumExpr

final case class Abs(x: NumExpr) extends NumFuncRef

final case class Atan(x: NumExpr) extends NumFuncRef

final case class Atan2(y: NumExpr, x: NumExpr) extends NumFuncRef

final case class Card(x: SetExpr) extends NumFuncRef
```

```
final case class Ceil(x: NumExpr) extends NumFuncRef

final case class Cos(x: NumExpr) extends NumFuncRef

final case class Exp(x: NumExpr) extends NumFuncRef

final case class Floor(x: NumExpr) extends NumFuncRef

final case class Gmtime() extends NumFuncRef

final case class Length(x: SymExpr) extends NumFuncRef

final case class Log(x: NumExpr) extends NumFuncRef

final case class Log10(x: NumExpr) extends NumFuncRef

final case class Max(x: NumExpr*) extends NumFuncRef

final case class Min(x: NumExpr*) extends NumFuncRef

final case class Round(x: NumExpr, n: Option[NumExpr] = None)
    extends NumFuncRef

final case class Sin(x: NumExpr) extends NumFuncRef

final case class Sqrt(x: NumExpr) extends NumFuncRef

final case class Str2time(s: SymExpr, f: SymExpr) extends NumFuncRef

final case class Trunc(x: NumExpr, n: Option[NumExpr] = None)
    extends NumFuncRef

final case class Irand224() extends NumFuncRef

final case class Uniform01() extends NumFuncRef

/*
 * SYMBOLIC
 */
final case class CondSymExpr(
    test: LogicExpr,
    ifTrue: SymExpr,
    otherwise: Option[SymExpr] = None
) extends SymExpr

final case class Concat(left: SymExpr, right: SymExpr)
    extends SymExpr

final case class StringLit(str: String) extends SymExpr

final case class SymNumExpr(numExpr: NumExpr) extends SymExpr

sealed trait SymFuncRef extends SymExpr

final case class Substr(
    x: SymExpr,
    from: NumExpr,
    length: Option[NumExpr] = None
) extends SymFuncRef

final case class Time2str(t: NumExpr, f: SymExpr) extends SymFuncRef
```

```
/*
 * SET
 */
final case class CondSetExpr(
    test: LogicExpr,
    ifTrue: SetExpr,
    otherwise: SetExpr
) extends SetExpr

final case class Union(left: SetExpr, right: SetExpr)
    extends SetExpr

final case class Diff(left: SetExpr, right: SetExpr) extends SetExpr

final case class SymDiff(left: SetExpr, right: SetExpr)
    extends SetExpr

final case class Inter(left: SetExpr, right: SetExpr)
    extends SetExpr

final case class Cross(left: SetExpr, right: SetExpr)
    extends SetExpr

final case class SetOf(
    indexing: IndExpr,
    integrand: List[SimpleExpr]
) extends SetExpr

final case class ArithSet(
    t0: NumExpr,
    tf: NumExpr,
    deltaT: Option[NumExpr] = None
) extends SetExpr

/*
 * SetStat reference with lazy semantic to allow recursive
 * definitions.
 * Overrides `hashCode`, `equals` and `Product` methods to only
 * consider the name of the SetStat and avoid infinite loops.
 */
final class SetRef(
    setThunk: () => SetStat,
    val subscript: List[SimpleExpr] = Nil
) extends SetExpr
    with StatRef
    with Product
    with Serializable {
  lazy val set: SetStat = setThunk()

  def copy(
      setThunk: () => SetStat = setThunk,
      subscript: List[SimpleExpr] = subscript
  ) =
    new SetRef(setThunk, subscript)

  lazy private val product = (set.name, subscript)

  override def equals(obj: Any): Boolean = obj match {
    case that: SetRef =>
      (that canEqual this) && this.product.equals(that.product)
    case _ => false
```

```
  }
  override def hashCode(): Int = this.product.hashCode()

  override def toString(): String = productPrefix + product

  // from Product
  override def canEqual(that: Any): Boolean =
    that.isInstanceOf[SetRef]
  override def productArity: Int = product.productArity
  override def productElement(n: Int): Any =
    product.productElement(n)
  override def productPrefix: String = "SetRef"
}
object SetRef {
  def apply(set: => SetStat, subscript: List[SimpleExpr] = Nil) =
    new SetRef(() => set, subscript)

  def unapply(x: SetRef): Option[(SetStat, List[SimpleExpr])] =
    Some((x.set, x.subscript))
}

final case class SetLit(values: Tuple*) extends SetExpr

/*
 * INDEXING
 */
final case class IndExpr(
    entries: List[IndEntry],
    predicate: Option[LogicExpr] = None
)

final case class IndExprSet(indexing: IndExpr) extends SetExpr

final case class IndEntry(
    indices: List[DummyIndDecl],
    set: SetExpr,
    predicate: Option[LogicExpr] = None
)

object IndEntry {
  def extract(predicate: LogicExpr): Map[DummyIndDecl, SimpleExpr] =
    predicate match {
      case Conj(expr1: LogicExpr, expr2: LogicExpr) =>
        extract(expr1) ++ extract(expr2)
      case Eq(DummyIndRef(ind), expr: SimpleExpr) =>
        Map(ind -> expr)
      case _ => Map()
    }

  def simplify(pred: LogicExpr): Option[LogicExpr] = pred match {
    case Conj(expr1: LogicExpr, expr2: LogicExpr) =>
      (simplify(expr1), simplify(expr2)) match {
        case (Some(se1), Some(se2)) => Some(Conj(se1, se2))
        case (x, None)              => x
        case (_, y)                 => y
      }
    case Eq(DummyIndRef(_), _: SimpleExpr) => Option.empty
    case x                                 => Some(x)
  }
}

/*
```

```
 * LOGIC
 */
final case class Disj(left: LogicExpr, right: LogicExpr)
    extends LogicExpr

final case class Forall(indexing: IndExpr, integrand: LogicExpr)
    extends LogicExpr

final case class Exists(indexing: IndExpr, integrand: LogicExpr)
    extends LogicExpr

final case class Conj(left: LogicExpr, right: LogicExpr)
    extends LogicExpr

final case class Neg(x: LogicExpr) extends LogicExpr

final case class LT(left: SimpleExpr, right: SimpleExpr)
    extends LogicExpr

final case class LTE(left: SimpleExpr, right: SimpleExpr)
    extends LogicExpr

final case class GT(left: SimpleExpr, right: SimpleExpr)
    extends LogicExpr

final case class GTE(left: SimpleExpr, right: SimpleExpr)
    extends LogicExpr

final case class Eq(left: SimpleExpr, right: SimpleExpr)
    extends LogicExpr

final case class NEq(left: SimpleExpr, right: SimpleExpr)
    extends LogicExpr

final case class In(values: List[SimpleExpr], set: SetExpr)
    extends LogicExpr

final case class NotIn(values: List[SimpleExpr], set: SetExpr)
    extends LogicExpr

final case class Within(left: SetExpr, right: SetExpr)
    extends LogicExpr

final case class NotWithin(left: SetExpr, right: SetExpr)
    extends LogicExpr

/*
 * LINEAR
 */
final case class CondLinExpr(
    test: LogicExpr,
    ifTrue: LinExpr,
    otherwise: Option[LinExpr] = None
) extends LinExpr

final case class LinAdd(left: LinExpr, right: LinExpr)
    extends LinExpr

final case class LinSub(left: LinExpr, right: LinExpr)
    extends LinExpr

final case class LinSum(indexing: IndExpr, integrand: LinExpr)
```

```
    extends LinExpr

final case class LinSumExp(summands: List[LinExpr]) extends LinExpr

final case class LinMult(left: NumExpr, right: LinExpr)
    extends LinExpr

final case class LinDiv(left: LinExpr, right: NumExpr)
    extends LinExpr

final case class LinUnaryPlus(x: LinExpr) extends LinExpr

final case class LinUnaryMinus(x: LinExpr) extends LinExpr

/*
 * VarStat reference with lazy semantic to allow recursive
 * definitions.
 * Overrides `hashCode`, `equals` and `Product` methods to only
 * consider the name of the VarStat and avoid infinite loops.
 */
final class VarRef(
    varThunk: () => VarStat,
    val subscript: List[SimpleExpr] = Nil
) extends LinExpr
    with StatRef
    with Product
    with Serializable {
  lazy val xvar: VarStat = varThunk()

  def copy(
      varThunk: () => VarStat = varThunk,
      subscript: List[SimpleExpr] = subscript
  ) =
    new VarRef(varThunk, subscript)

  lazy private val product = (xvar.name, subscript)

  override def equals(obj: Any): Boolean = obj match {
    case that: VarRef =>
      (that canEqual this) && this.product.equals(that.product)
    case _ => false
  }
  override def hashCode(): Int = this.product.hashCode()

  override def toString(): String = productPrefix + product

  // from Product
  override def canEqual(that: Any): Boolean =
    that.isInstanceOf[VarRef]
  override def productArity: Int = product.productArity
  override def productElement(n: Int): Any =
    product.productElement(n)
  override def productPrefix: String = "VarRef"
}
object VarRef {
  def apply(xvar: => VarStat, subscript: List[SimpleExpr] = Nil) =
    new VarRef(() => xvar, subscript)

  def unapply(x: VarRef): Option[(VarStat, List[SimpleExpr])] =
    Some((x.xvar, x.subscript))
}
```

```
  /*
   * STAT
   */
  sealed trait RowRef extends StatRef // to group ConstraintRef and
                                      // ObjectiveRef
  final case class ConstraintRef(
      crt: ConstraintStat,
      subscript: List[SimpleExpr] = Nil
  ) extends RowRef
  final case class ObjectiveRef(
      obj: ObjectiveStat,
      subscript: List[SimpleExpr] = Nil
  ) extends RowRef

  /*
   * BASIC
   */
  type SymName = String
  type Tuple = List[SimpleExpr]

}
```

Listado C.1: AST completo del *deep embedding*

Apéndice D

Caso de estudio en MathProg

```
# Conjuntos (y parámetros de los conjuntos)
param H;
set T := 1..H;
set A;
set P;
set C := A union P;

# Parámetros
param d{T};
param y0;
param ymin;
param ymax;

param tau{A}   in T;
param gamma{P} in 0..H-1;
param q{C};

param ca{C};
param cc{A};
param h{T};

param a{t in T} := sum{c in A : tau[c] == t} q[c];

# Variables
var y{T} >= 0;
var v{c in P, t in 1 .. H - gamma[c]} binary;
var x{c in A, t in 1 .. tau[c] - 1}   binary;

var u{T} >= 0;
var w{T} >= 0;
```

Listado D.1: Entidades del Caso de estudio representadas en MathProg

```
# Función objetivo
minimize cost:
  sum{t in T} (
    sum{c in P : t <= H - gamma[c]}          ca[c]  * q[c] * v[c,t] +
    sum{c in A : t <= tau[c] - 1  } (cc[c] - ca[c]) * q[c] * x[c,t] +
    h[t] * y[t]
  );

# Restricciones
s.t. balance0:
  y0     + a[1] + u[1] = d[1] + w[1] + y[1];
s.t. balance{t in T : t > 1}:
  y[t-1] + a[t] + u[t] = d[t] + w[t] + y[t];

s.t. inventory{t in T}: ymin <= y[t] <= ymax;

s.t. single_acquisition {c in P}:
  sum{t in 1 .. H - gamma[c]} v[c,t] <= 1;

s.t. single_cancellation{c in A}:
  sum{t in 1 .. tau[c] - 1}   x[c,t] <= 1;

s.t. acquired_fuel{t in T}:
  u[t] = sum{c in P : gamma[c] <= t-1} q[c] * v[c, t-gamma[c]];

s.t. cancelled_fuel{t in T}:
  w[t] = sum{c in A : tau[c] == t} (
          q[c] * sum{t1 in T : t1 <= tau[c] - 1} x[c,t1]
         );
```

Listado D.2: Función objetivo y restricciones del Caso de estudio representadas en MathProg

```
param H := 3;
set A   := A1, A2;
set P   := P1, P2, P3, P4;

param d    := 1 35, 2 27.5, 3 36.25;
param y0   := 20;
param ymin := 0;
param ymax := 80;

param tau   := A1 1, A2 2;
param gamma := P1 1, P2 1, P3 1, P4 1;
param q     := A1 38, A2 22, P1 10, P2 37, P3 39, P4 40;

param ca := A1 184, A2 210, P1 160, P2 231, P3 178, P4 152;
param cc := A1  32, A2  42;
param h  := 1 1, 2 1, 3 1;
```

Listado D.3: Instancia de datos del Caso de estudio representada en MathProg

Apéndice E

Caso de estudio extendido en los DSL analizados

Implementación del Caso de estudio extendido (Sección 3.3), en los diferentes DSL analizados.

E.1. MathProg

```
# Conjuntos (y parámetros de los conjuntos)
param H;
set T := 1..H;
set A;
set P;
set C := A union P;
set S;

# Parámetros
param d{T, S};
param y0;
param ymin;
param ymax;

param tau{A}   in T;
param gamma{P} in 0..H-1;
param q{C};

param ca{C};
param cc{A};
param h{T};

param a{t in T} := sum{c in A : tau[c] == t} q[c];

param pi{S};
param ancf{S, T};

# Variables
var y{T, S} >= 0;
var v{c in P, t in T, S : t <= H - gamma[c]} binary;
var x{c in A, t in T, S : t <= tau[c] - 1  } binary;

var u{T, S} >= 0;
```

```
var w{T, S} >= 0;
```

Listado E.1: Entidades del Caso de estudio extendido representadas en MathProg

```
# Función objetivo
minimize cost:
  sum{t in T, s in S} pi[s] * (
    sum{c in P : t <= H - gamma[c]}          ca[c]  * q[c] * v[c,t,s] +
    sum{c in A : t <= tau[c] - 1  } (cc[c] - ca[c]) * q[c] * x[c,t,s] +
    h[t] * y[t,s]
  );

# Restricciones
s.t. balance0{        s in S        }:
  y0        + a[1] + u[1,s] = d[1,s] + w[1,s] + y[1,s];
s.t. balance {t in T, s in S : t > 1}:
  y[t-1, s] + a[t] + u[t,s] = d[t,s] + w[t,s] + y[t,s];

s.t. inventory{t in T, s in S}: ymin <= y[t,s] <= ymax;

s.t. single_acquisition {c in P, s in S}:
  sum{t in 1 .. H - gamma[c]} v[c,t,s] <= 1;

s.t. single_cancellation{c in A, s in S}:
  sum{t in 1 .. tau[c] - 1  } x[c,t,s] <= 1;

s.t. acquired_fuel{t in T, s in S}:
  u[t,s] = sum{c in P : gamma[c] <= t-1} q[c] * v[c, t-gamma[c], s];

s.t. cancelled_fuel{t in T, s in S}:
  w[t,s] = sum{c in A : tau[c] == t} (
            q[c] * sum{tp in T : tp <= tau[c] - 1} x[c,tp,s]
           );

# No anticipatividad
s.t. na_v{
      c in P, t in T, s1 in S, s2 in S :
        t <= H - gamma[c] and ancf[s1,t] == ancf[s2,t]
    }: v[c,t,s1] == v[c,t,s2];

s.t. na_x{
      c in A, t in T, s1 in S, s2 in S :
        t <= tau[c] - 1 and ancf[s1,t] == ancf[s2,t]
    }: x[c,t,s1] == x[c,t,s2];

s.t. na_y{
      t in T, s1 in S, s2 in S : ancf[s1,t] == ancf[s2,t]
    }: y[t,s1] == y[t,s2];
```

Listado E.2: Función objetivo y restricciones del Caso de estudio extendido representadas en MathProg

```
param H := 3;
set A   := A1, A2;
set P   := P1, P2, P3, P4;

param y0   := 20;
param ymin := 0;
param ymax := 80;
```

```
param tau   := A1 1, A2 2;
param gamma := P1 1, P2 1, P3 1, P4 1;
param q     := A1 38, A2 22, P1 10, P2 37, P3 39, P4 40;

param ca := A1 184, A2 210, P1 160, P2 231, P3 178, P4 152;
param cc := A1  32, A2  42;
param h  := 1 1, 2 1, 3 1;

set S    := 1 2 3 4;
param pi := 1 0.15625, 2 0.34375, 3 0.34375, 4 0.15625;

param ancf :
     1 2 3 :=
  1  1 1 1
  2  1 1 2
  3  1 2 3
  4  1 2 4 ;

param d :
      1  2  3  4 :=
  1  35 35 35 35
  2  38 38 17 17
  3  43 24 34 44 ;
```

Listado E.3: Instancia de datos del Caso de estudio extendido representada en MathProg

E.2. SAMPL

```
# Conjuntos (y parámetros de los conjuntos)
param H;
set T := 1..H;
set A;
set P;
set C := A union P;

scenarioset S;

# Parámetros
random param d{t in T, s in S};
param y0;
param ymin;
param ymax;

param tau{A}   in T;
param gamma{P} in 0..H-1;
param q{C};

param ca{C};
param cc{A};
param h{T};

param a{t in T} := sum{c in A : tau[c] == t} q[c];

probability param pi{S};
three theTree := nway{2};
suffix stage IN;

# Variables
```

```
var y{T, S} >= 0;
var v{c in P, t in T, S : t <= H - gamma[c]} binary;
var x{c in A, t in T, S : t <= tau[c] - 1  } binary;

var u{T, S} >= 0;
var w{T, S} >= 0;

# Etapa asociada a las variables
let{t in T, s in S}                              y[  t,s].stage = t;
let{c in P, t in T, s in S : t <= H - gamma[c]} v[c,t,s].stage = t;
let{c in A, t in T, s in S : t <= tau[c] - 1}   x[c,t,s].stage = t;
```

Listado E.4: Entidades del Caso de estudio extendido representadas en SAMPL

```
# Función objetivo
minimize cost:
  sum{t in T, s in S} pi[s] * (
    sum{c in P : t <= H - gamma[c]}          ca[c]  * q[c] * v[c,t,s] +
    sum{c in A : t <= tau[c] - 1  } (cc[c] - ca[c]) * q[c] * x[c,t,s] +
    h[t] * y[t,s]
  );

# Restricciones
s.t. balance0{        s in S        }:
  y0        + a[1] + u[1,s] = d[1,s] + w[1,s] + y[1,s];
s.t. balance {t in T, s in S : t > 1}:
  y[t-1, s] + a[t] + u[t,s] = d[t,s] + w[t,s] + y[t,s];

s.t. inventory{t in T, s in S}: ymin <= y[t,s] <= ymax;

s.t. single_acquisition {c in P, s in S}:
  sum{t in 1 .. H - gamma[c]} v[c,t,s] <= 1;

s.t. single_cancellation{c in A, s in S}:
  sum{t in 1 .. tau[c] - 1  } x[c,t,s] <= 1;

s.t. acquired_fuel{t in T, s in S}:
  u[t,s] = sum{c in P : gamma[c] <= t-1} q[c] * v[c, t-gamma[c], s];

s.t. cancelled_fuel{t in T, s in S}:
  w[t,s] = sum{c in A : tau[c] == t} (
            q[c] * sum{tp in T : tp <= tau[c] - 1} x[c,tp,s]
           );
```

Listado E.5: Función objetivo y restricciones del Caso de estudio extendido representadas en SAMPL

```
param H := 3;
set A   := A1, A2;
set P   := P1, P2, P3, P4;

param y0   := 20;
param ymin := 0;
param ymax := 80;

param tau   := A1 1, A2 2;
param gamma := P1 1, P2 1, P3 1, P4 1;
param q     := A1  38, A2  22, P1  10, P2  37, P3  39, P4  40;

param ca := A1 184, A2 210, P1 160, P2 231, P3 178, P4 152;
```

```
param cc := A1  32, A2  42;
param h  := 1 1, 2 1, 3 1;

scenarioset S        := 1 2 3 4;
probability param pi := 1 0.15625, 2 0.34375, 3 0.34375, 4 0.15625;

random param d :=
1 1 35
2 1 38
2 3 17
3 1 43
3 2 24
3 3 34
3 4 44;
```

Listado E.6: Instancia de datos del Caso de estudio extendido representada en SAMPL

E.3. StAMPL

```
definestage 1;

# Conjuntos
set A;
set P;
set C := A union P;

# Parámetros
param d;
param y0;
param ymin;
param ymax;

param tau{A}   in 1 .. stages();
param gamma{P} in 0 .. stages() - 1;
param q{C};

param ca{C};
param cc{A};
param h;

param a := sum{c in A : tau[c] == stage()} q[c];

# Variables
var y >= 0;
var v{c in P} binary <= if (stage() <= stages() - gamma[c]) 1 else 0;
var x{c in A} binary <= if (stage() <= tau[c] - 1)          1 else 0;

var u >= 0;
var w >= 0;

# Función objetivo
minimize cost:
    sum{c in P : stage() <= stages() - gamma[c]} ca[c]  * q[c] * v[c] +
    sum{c in A : stage() <= tau[c] - 1} (cc[c] - ca[c]) * q[c] * x[c] +
    h * y;

# Restricciones
s.t. balance0: y0 + a + u = d + w + y;
```

```
s.t. inventory: ymin <= y <= ymax;

s.t. acquired_fuel:
  u = sum{c in P : gamma[c] <= stage() - 1}
        q[c] * parent(gamma[c]).v[c];

s.t. cancelled_fuel:
  w = sum{c in A : tau[c] == stage()} (
        q[c] * sum{t in 1 .. tau[c] - 1} parent(stage() - t).x[c]
      );
```

Listado E.7: Modelo para la primera etapa del Caso de estudio extendido representado en StAMPL

```
definestage 2 .. stages() - 1;

# Conjuntos
set A;
set P;
set C := A union P;

# Parámetros
param d;
param ymin;
param ymax;

param tau{A}   in 1 .. stages();
param gamma{P} in 0 .. stages() - 1;
param q{C};

param ca{C};
param cc{A};
param h;

param a := sum{c in A : tau[c] == stage()} q[c];

# Variables
var y >= 0;
var v{c in P} binary <= if (stage() <= stages() - gamma[c]) 1 else 0;
var x{c in A} binary <= if (stage() <= tau[c] - 1)          1 else 0;

var u >= 0;
var w >= 0;

# Función objetivo
minimize cost:
    sum{c in P : stage() <= stages() - gamma[c]} ca[c]  * q[c] * v[c] +
    sum{c in A : stage() <= tau[c] - 1} (cc[c] - ca[c]) * q[c] * x[c] +
    h * y;

# Restricciones
s.t. balance: parent().y  + a + u = d + w + y;

s.t. inventory: ymin <= y <= ymax;

s.t. acquired_fuel:
  u = sum{c in P : gamma[c] <= stage() - 1}
        q[c] * parent(gamma[c]).v[c];

s.t. cancelled_fuel:
```

```
  w = sum{c in A : tau[c] == stage()} (
        q[c] * sum{t in 1 .. tau[c] - 1} parent(stage() - t).x[c]
      );
```

Listado E.8: Modelo para las etapas intermedias del Caso de estudio extendido representado en StAMPL

```
definestage stages();

# Conjuntos
set A;
set P;
set C := A union P;

# Parámetros
param d;
param ymin;
param ymax;

param tau{A}   in 1 .. stages();
param gamma{P} in 0 .. stages() - 1;
param q{C};

param ca{C};
param cc{A};
param h;

param a := sum{c in A : tau[c] == stage()} q[c];

# Variables
var y >= 0;
var v{c in P} binary <= if (stage() <= stages() - gamma[c]) 1 else 0;
var x{c in A} binary <= if (stage() <= tau[c] - 1)          1 else 0;

var u >= 0;
var w >= 0;

# Función objetivo
minimize cost:
    sum{c in P : stage() <= stages() - gamma[c]} ca[c]  * q[c] * v[c] +
    sum{c in A : stage() <= tau[c] - 1} (cc[c] - ca[c]) * q[c] * x[c] +
    h * y;

# Restricciones
s.t. balance: parent().y  + a + u = d + w + y;

s.t. inventory: ymin <= y <= ymax;

s.t. single_acquisition {c in P}:
  sum{t in 1 .. stages() - gamma[c]} parent(stages() - t).v[c] <= 1;

s.t. single_cancellation{c in A}:
  sum{t in 1 .. tau[c] - 1}          parent(stages() - t).x[c] <= 1;

s.t. acquired_fuel:
  u = sum{c in P : gamma[c] <= stage() - 1}
        q[c] * parent(gamma[c]).v[c];

s.t. cancelled_fuel:
  w = sum{c in A : tau[c] == stage()} (
        q[c] * sum{t in 1 .. tau[c] - 1} parent(stage() - t).x[c]
```

```
    );
```

Listado E.9: Modelo para la última etapa del Caso de estudio extendido representado en StAMPL

```
# Definición de total de etapas
setstages(3);

definestage 1;

set A := A1, A2;
set P := P1, P2, P3, P4;

param y0   := 20;
param ymin := 0;
param ymax := 80;

param tau   := A1 1, A2 2;
param gamma := P1 1, P2 1, P3 1, P4 1;
param q     := A1 38, A2 22, P1 10, P2 37, P3 39, P4 40;

param ca := A1 184, A2 210, P1 160, P2 231, P3 178, P4 152;
param cc := A1  32, A2  42;
param h  := 1;
```

Listado E.10: Instancia de datos de la primera etapa del Caso de estudio extendido representado en StAMPL (incluye definición de total de etapas)

```
definestage 2 .. stages() - 1;

set A := A1, A2;
set P := P1, P2, P3, P4;

param ymin := 0;
param ymax := 80;

param tau   := A1 1, A2 2;
param gamma := P1 1, P2 1, P3 1, P4 1;
param q     := A1 38, A2 22, P1 10, P2 37, P3 39, P4 40;

param ca := A1 184, A2 210, P1 160, P2 231, P3 178, P4 152;
param cc := A1  32, A2  42;
param h  := 1;
```

Listado E.11: Instancia de datos de las etapas intermedias del Caso de estudio extendido representado en StAMPL

```
definestage stages();

set A := A1, A2;
set P := P1, P2, P3, P4;

param ymin := 0;
param ymax := 80;

param tau   := A1 1, A2 2;
param gamma := P1 1, P2 1, P3 1, P4 1;
param q     := A1 38, A2 22, P1 10, P2 37, P3 39, P4 40;
```

```
param ca := A1 184, A2 210, P1 160, P2 231, P3 178, P4 152;
param cc := A1  32, A2  42;
param h  := 1;
```

Listado E.12: Instancia de datos de la última etapa del Caso de estudio extendido representado en StAMPL

```
#include <fstream>
#include <iostream>
#include <rvnode/rvnode.h>
#include <utils/leostl.h>

int main(void) {
  const int nStages = 3;
  const int nScens = 4;

  RandomElem d_1_1, d_2_1, d_2_2, d_3_1, d_3_2, d_3_3, d_3_4;
  d_1_1.param = "d";
  d_2_1.param = d_2_2.param = "d";
  d_3_1.param = d_3_2.param = d_3_3.param = d_3_4.param = "d";

  d_1_1.val = 35;
  d_2_1.val = 38; d_2_2.val = 17;
  d_3_1.val = 43; d_3_2.val = 24; d_3_3.val = 34; d_3_4.val = 44;

  RVNode s_1_1, s_2_1, s_2_2, s_3_1, s_3_2, s_3_3, s_3_4;
  s_1_1.stage = 1;
  s_2_1.stage = s_2_2.stage = 2;
  s_3_1.stage = s_3_2.stage = s_3_3.stage = s_3_4.stage = 3;

  s_1_1.pathprob = 1;
  s_2_1.pathprob = 0.5;
  s_2_2.pathprob = 0.5;
  s_3_1.pathprob = 0.15625;
  s_3_2.pathprob = 0.34375;
  s_3_3.pathprob = 0.34375;
  s_3_4.pathprob = 0.15625;

  s_1_1.elems.push_back(d_1_1);
  s_2_1.elems.push_back(d_2_1);
  s_2_2.elems.push_back(d_2_2);
  s_3_1.elems.push_back(d_3_1);
  s_3_2.elems.push_back(d_3_2);
  s_3_3.elems.push_back(d_3_3);
  s_3_4.elems.push_back(d_3_4);

  s_1_1.children.push_back(s_2_1);
  s_1_1.children.push_back(s_2_2);
  s_2_1.children.push_back(s_3_1);
  s_2_1.children.push_back(s_3_2);
  s_2_2.children.push_back(s_3_3);
  s_2_2.children.push_back(s_3_4);

  s_1_1.NumberScens();
  s_1_1.ReconcileProbabilities();

  cout << s_1_1 << endl;
}
```

Listado E.13: Código C++ para generación de árbol de escenarios correspondiente

al Caso de estudio extendido representado en StAMPL

E.4. AIMMS

```
Model fuel_supply {
  DeclarationSection deterministic_model_declarations {
    ! Conjuntos (y parámetros de los conjuntos)
    Parameter H;
    Set T {
      SubsetOf   : Integers;
      Index      : t, t1;
      Definition : { 1 .. H };
    }
    Set A {
      Index : a;
    }
    Set P {
      Index : p;
    }
    Set C {
      Index      : c;
      Definition : A + P;
    }

    ! Parámetros
    Parameter d {
      IndexDomain : t;
      Property    : Stochastic;
    }
    Parameter y0 {
    }
    Parameter ymin {
    }
    Parameter ymax {
    }

    Parameter tau {
      IndexDomain : a;
      Range       : T
    }
    Parameter gamma {
      IndexDomain : p;
      Range       : { 0 .. H-1 };
    }
    Parameter q {
      IndexDomain : c;
    }

    Parameter ca {
      IndexDomain : c;
    }
    Parameter cc {
      IndexDomain : c;
    }
    Parameter h {
      IndexDomain : t;
    }

    Parameter a {
```

```
    IndexDomain : t;
    Definition  : Sum(a | tau(a) = t, q(a));
  }

  ! Variables
  Variable y {
    IndexDomain : t;
    Range       : Nonnegative;
    Property    : Stochastic;
    Stage       : t;
  }
  Variable v {
    IndexDomain : (p,t) | t <= H - gamma(p);
    Range       : Binary;
    Property    : Stochastic;
    Stage       : t;
  }
  Variable x {
    IndexDomain : (a,t) | t <= tau(a) - 1;
    Range       : Binary;
    Property    : Stochastic;
    Stage       : t;
  }

  Variable u {
    IndexDomain : t;
    Range       : Nonnegative;
    Property    : Stochastic;
    Stage       : t;
  }
  Variable w {
    IndexDomain : t;
    Range       : Nonnegative;
    Property    : Stochastic;
    Stage       : t;
  }

  ! Función objetivo
  Variable cost {
    Definition : {
      Sum[t,
        Sum(p | t <= H - gamma(p),        ca(p)  * q(p) * v(p,t)) +
        Sum(a | t <= tau(a) - 1, (cc(a) - ca(a)) * q(c) * x(c,t)) +
        h(t) * y(t)
      ]
    }
  }

  ! Restricciones
  Constraint balance0 {
    Definition  : y0 + a(1) + u(1) = d(1) + w(1) + y(1);
  }
  Constraint balance {
    IndexDomain : t | t > 1;
    Definition  : y(t-1) + a(t) + u(t) = d(t) + w(t) + y(t);
  }

  Constraint inventory {
    IndexDomain : t;
    Definition  : ymin <= y(t) <= ymax;
  }
```

```
  Constraint single_acquisition {
    IndexDomain : p;
    Definition  : Sum(t | t <= H - gamma(p), v(p,t)) <= 1;
  }

  Constraint single_cancellation {
    IndexDomain : a;
    Definition  : Sum(t | t <= tau(a) - 1, x(a,t)) <= 1;
  }

  Constraint acquired_fuel {
    IndexDomain : t;
    Definition  : {
      u(t) = Sum(p | gamma(p) <= t-1, q(p) * v(p, t-gamma(p)));
    }
  }

  Constraint cancelled_fuel {
    IndexDomain : t;
    Definition  : {
      w(t) = Sum(a | tau(a) == t,
               q(a) * Sum(t1 | t1 <= tau(a) - 1, x(a,t1)));
    }
  }

  ! Modelo determinista
  MathematicalProgram basic_MIP {
    Objective: cost;
    Direction: minimize;
  }
}

DeclarationSection stochastic_model_declarations {
  Set S {
    SubsetOf : AllStochasticScenarios;
    Index    : s;
  }
  ElementParameter ScenarioTreeMap {
    IndexDomain : (s,t);
    Range       : S;
  }
  Parameter pi {
    IndexDomain : s;
  }
  ElementParameter StochModelGMP {
    Range: AllGeneratedMathematicalPrograms;
  }
}

! Datos y scripts
Procedure MainInitialization {
  Body: {
    H := 3;

    A := DATA { A1, A2 };
    P := DATA { P1, P2, P3, P4 };

    y0   := 20;
    ymin :=  0;
    ymax := 80;

    tau   := DATA { A1 : 1, A2 : 2 };
```

```
    gamma := DATA { P1 : 1, P2 : 1, P3 : 1, P4 : 1 };
    q     := DATA { A1 : 38, A2 : 22,
                    P1 : 10, P2 : 37, P3 : 39, P4 : 40 };

    ca := DATA { A1 : 184, A2 : 210,
                 P1 : 160, P2 : 231, P3 : 178, P4 : 152 };
    cc := DATA { A1 : 32, A2 : 42 };
    h  := DATA { 1 : 1, 2 : 1, 3 : 1 };

    ! Valores dummy para problema determinista
    d := DATA { 1 : 35, 2 : 27.5, 3 : 36.25 };
  }
}
Procedure MainExecution {
  Body: {

    empty AllStochasticScenarios;
    cleandependents AllStochasticScenarios;

    ScenGen::InitializeStochasticScenarioDataCallbackFunction     :=
      'InitializeStochasticScenarioDataCallbackFunction';
    ScenGen::DetermineScenarioGroupsCallbackFunction              :=
      'DetermineScenarioGroupsCallbackFunction';
    ScenGen::AssignStochasticDataForScenarioGroupCallbackFunction :=
      'AssignStochasticDataForScenarioGroupCallbackFunction';
    ScenGen::CompareScenariosCallbackFunction                     :=
      'CompareScenariosCallbackFunction';

    option seed := 1;

    ScenGen::CreateScenarioData(H, S, pi, ScenarioTreeMap);

    StochModelGMP := GMP::Instance::GenerateStochasticProgram(
      basic_MIP,
      AllStochasticParameters,
      AllStochasticVariables,
      S,
      pi,
      ScenarioTreeMap,
      "DetOrExp",
      GenerationMode  : 'SubstituteStochasticVariables',
      Name            : "stochModel" );

    GMP::Instance::Solve( StochModelGMP );
  }
}
Procedure MainTermination {
  Body: {
    return 1;
  }
}

Section ScenGen_Callback_Funtions {
  Procedure InitializeStochasticScenarioDataCallbackFunction {
    Arguments: (Scenario, Scenarios);
    Body: {

      switch card(Scenarios) do
        0       : return
                    DistributionCumulative(Beta(2,2), 1/4) -
                    DistributionCumulative(Beta(2,2), 0);
        1       : return
```

```
                    DistributionCumulative(Beta(2,2), 2/4) -
                    DistributionCumulative(Beta(2,2), 1/4);
      2       : return
                    DistributionCumulative(Beta(2,2), 3/4) -
                    DistributionCumulative(Beta(2,2), 2/4);
      3       : return
                    DistributionCumulative(Beta(2,2), 1) -
                    DistributionCumulative(Beta(2,2), 3/4);
      default : return 0;
    endswitch;

  }
  Comment: {
    "Define escenarios finales junto con sus probabilidades."
  }
  ElementParameter Scenario {
      Range     : AllStochasticScenarios;
      Property  : Input;
  }
  Set Scenarios {
      SubsetOf  : AllStochasticScenarios;
      Index     : cs;
      Property  : Input;
  }
}

Procedure DetermineScenarioGroupsCallbackFunction {
  Arguments: (CurrentStage, ScenarioGroup, ScenarioGroupOrder);
  Body: {

    ScenarioGroupSize := card(ScenarioGroup);
    if CurrentStage > 1 then
      for (sgr) do
        if sgr <= ScenarioGroupSize / 2 then
          ScenarioGroupOrder(sgr) := 1;
        else
          ScenarioGroupOrder(sgr) := 2;
        endif;
      endfor;
    else
      ScenarioGroupOrder(sgr) := 1;
    endif;

  }
  Comment: {
      "Divide el grupo actual de escenarios en dos subgrupos,
      acorde con las dos ramificaciones posibles por escenario."
  }
  ElementParameter CurrentStage {
      Range: Integers;
      Property: Input;
  }
  Set ScenarioGroup {
      SubsetOf: AllStochasticScenarios;
      Index: sgr;
      Property: Input;
  }
  Parameter ScenarioGroupOrder {
      IndexDomain: (sgr);
      Range: {
          { 1 .. ScenarioGroupSize }
      }
```

```
        Property: Output;
    }
    Parameter ScenarioGroupSize;
  }

  Procedure AssignStochasticDataForScenarioGroupCallbackFunction {
    Arguments: (CurrentStage, ScenarioGroup);
    Body: {

      ScenarioDemand := Uniform(10,50);
      for (sgr) do
        d.Stochastic(sgr, CurrentStage) := ScenarioDemand;
      endfor;

    }
    Comment: {
        "Asigna los valores de los parámetros estocásticos para cada
        grupo de escenarios considerando la no anticipatividad."
    }
    ElementParameter CurrentStage {
        Range     : Integers;
        Property  : Input;
    }
    Set ScenarioGroup {
        SubsetOf  : AllStochasticScenarios;
        Index     : sgr;
        Property  : Input;
    }
    Parameter ScenarioDemand;
  }

  Procedure CompareScenariosCallbackFunction {
    Arguments: (Scenario1, Scenario2, Stages, FirstDifferentStage);
    Body: {

      for (cs) do
        if (d.Stochastic(Scenario1, cs) <
              d.Stochastic(Scenario2, cs)) then
          FirstDifferentStage := cs;
          return -1;
        endif;
        if (d.Stochastic(Scenario1, cs) >
              d.Stochastic(Scenario2, cs)) then
          FirstDifferentStage := cs;
          return  1;
        endif;
      endfor;

      return 0;

    }
    Comment: {
      "Elimia escenarios que, luego de asignar los valores de
      parámetros estocásticos, quedaron identicos."
    }
    ElementParameter Scenario1 {
        Range: AllStochasticScenarios;
        Property: Input;
    }
    ElementParameter Scenario2 {
        Range: AllStochasticScenarios;
        Property: Input;
```

```
      }
      Set Stages {
          SubsetOf: Integers;
          Index: cs;
          Property: Input;
      }
      ElementParameter FirstDifferentStage {
          Range: Integers;
          Property: Output;
      }
    }
  }
}
```

Listado E.14: Modelo, datos y lógica para generación del árbol de escenarios y resolución del problema correspondientes al Caso de estudio extendido representado en AIMMS

E.5. PySP

```
from pyomo.core import *

model = AbstractModel()

# Conjuntos (y parámetros de los conjuntos)
model.H = Param()
model.T = RangeSet(1, model.H)
model.A = Set()
model.P = Set()
model.C = model.A | model.P

# Parámetros
model.d    = Param(model.T)
model.y0   = Param()
model.ymin = Param()
model.ymax = Param()

model.tau   = Param(model.A, within=model.T)
model.gamma = Param(model.P, within=RangeSet(0, model.H - 1))
model.q     = Param(model.C)

model.ca = Param(model.C)
model.cc = Param(model.A)
model.h  = Param(model.T)

def a_init(model, t):
  return sum(model.q[c] for c in model.A if model.tau[c] == t)

model.a = Param(model.T, initialize=a_init)

# Variables
model.y = Var(model.T, within=NonNegativeReals)

def PT_init(model):
  return ((c,t) for c in model.P
                for t in model.T if t <= model.H - model.gamma[c])

def AT_init(model):
```

```
  return ((c,t) for c in model.A
                for t in model.T if t <= model.tau[c] - 1)

model.PT = Set(dimen=2, initialize=PT_init)
model.AT = Set(dimen=2, initialize=AT_init)

model.v = Var(model.PT, within=Binary)
model.x = Var(model.AT, within=Binary)

model.u = Var(model.T, within=NonNegativeReals)
model.w = Var(model.T, within=NonNegativeReals)
```

Listado E.15: Entidades del modelo base determinista correspondiente al Caso de estudio extendido representadas en PySP

```
# Costos por etapa
# La función objetivo la construye automáticamente a
# partir de estas expresiones
def cost_rule(model, t):
  return \
    sum(               model.ca[c]  * model.q[c] * model.v[c,t] \
                          for c in model.P if (c,t) in model.PT) + \
    sum((model.cc[c] - model.ca[c]) * model.q[c] * model.x[c,t] \
                          for c in model.A if (c,t) in model.AT) + \
    model.h[t] * model.y[t]

model.cost = Expression(model.T, rule=cost_rule)

# Restricciones
def T2_init(model):
  return ((t) for t in model.T if t > 1)

model.T2 = Set(initialize=T2_init)

def balance0_rule(model):
  return \
    model.y0     + model.a[1] + model.u[1] == \
      model.d[1] + model.w[1] + model.y[1]

def balance_rule(model, t):
  return \
    model.y[t-1] + model.a[t] + model.u[t] == \
      model.d[t] + model.w[t] + model.y[t]

model.balance0 = Constraint(rule=balance0_rule)
model.balance  = Constraint(model.T2, rule=balance_rule)

def inventory_rule(model, t):
  return inequality(model.ymin, model.y[t], model.ymax)

model.inventory = Constraint(model.T, rule=inventory_rule)

def single_acquisition_rule(model, c):
  return sum(model.v[c,t] for t in model.T if (c,t) in model.PT) <= 1

def single_cancellation_rule(model, c):
  return inequality(0, sum(model.x[c,t] \
            for t in model.T if (c,t) in model.AT), 1)

model.single_acquisition  = \
  Constraint(model.P, rule=single_acquisition_rule)
```

```
model.single_cancellation = \
  Constraint(model.A, rule=single_cancellation_rule)

def acquired_fuel_rule(model, t):
  return \
    model.u[t] == \
      sum(model.q[c] * model.v[c, t - model.gamma[c]] \
        for c in model.P if model.gamma[c] <= t - 1)

def cancelled_fuel_rule(model, t):
  return \
    model.w[t] == \
      sum(model.q[c] * sum(model.x[c, t1] \
        for t1 in model.T if t1 <= model.tau[c] - 1) \
        for  c in model.A if model.tau[c] == t)

model.acquired_fuel  = Constraint(model.T, rule=acquired_fuel_rule)
model.cancelled_fuel = Constraint(model.T, rule=cancelled_fuel_rule)

# Función objetivo usada en versión determinista.
# La función objetivo del modelo estocástico es generada por PySP
def cost_objective_rule(model):
    return summation(model.cost)

model.cost_objective = \
  Objective(rule=cost_objective_rule, sense=minimize)
```

Listado E.16: Función objetivo y restricciones del modelo base determinista correspondiente al Caso de estudio extendido representadas en PySP

```
set Stages := T1 T2 T3;

set Nodes := N11
             N21 N22
             N31 N32 N33 N34;

param NodeStage := N11 T1
                   N21 T2
                   N22 T2
                   N31 T3
                   N32 T3
                   N33 T3
                   N34 T3;

set Children[N11] := N21 N22;
set Children[N21] := N31 N32;
set Children[N22] := N33 N34;

param ConditionalProbability := N11 1.0
                                N21 0.5
                                N22 0.5
                                N31 0.3125
                                N32 0.6875
                                N33 0.6875
                                N34 0.3125;

set Scenarios := S1 S2 S3 S4;

param ScenarioLeafNode := S1 N31
                          S2 N32
                          S3 N33
```

```
                          S4 N34;

set StageVariables[T1] := y[1] v[*,1] x[*,1] u[1] w[1];
set StageVariables[T2] := y[2] v[*,2]        u[2] w[2];
set StageVariables[T3] := y[3]               u[3] w[3];

param StageCost := T1 cost[1]
                   T2 cost[2]
                   T3 cost[3];

param ScenarioBasedData := False;
```

Listado E.17: Estructura de escenarios correspondiente al Caso de estudio extendido representada en PySP

```
# N11.dat
param H := 3;
set A   := A1 A2;
set P   := P1 P2 P3 P4;

param y0   := 20;
param ymin := 0;
param ymax := 80;

param tau   := A1 1  A2 2;
param gamma := P1 1  P2 1  P3 1  P4 1;
param q     := A1  38  A2  22  P1  10  P2  37  P3  39  P4  40;

param ca := A1 184  A2 210  P1 160  P2 231  P3 178  P4 152;
param cc := A1  32  A2  42;
param h  := 1 1  2 1  3 1;

param d := 1 35;

# N21.dat
param d := 2 38;

# N22.dat
param d := 2 17;

# N31.dat
param d := 3 43;

# N32.dat
param d := 3 24;

# N33.dat
param d := 3 34;

# N34.dat
param d := 3 44;
```

Listado E.18: Instancia de datos por nodo del árbol de escenarios del Caso de estudio extendido representado en PySP