# Formalización y resolución de problemas

Laboratorio 4
Lógica para Computación
Otoño 2025

El objetivo de este laboratorio es poner en práctica una de las posibles aplicaciones de la lógica formal, específicamente en la resolución de problemas. A grandes rasgos, la idea es formalizar problemas enunciados en el lenguaje natural para hallar soluciones de forma automática mediante herramientas de decisión de satisfacibilidad (SMT solvers). Esta manera de resolver problemas es la base de un paradigma de programación lógica.

## 1 Introducción

El objetivo de esta sección es servir como una breve introducción de conceptos básicos así como de algunas herramientas basadas en dichos conceptos.

### 1.1 Clasificación fundamental de problemas

En general nos interesa resolver problemas. Pero no cualquier tipo de problemas, sino aquellos que se puedan formular de manera matemáticamente precisa, diremos *problemas formales*. El interés en estos problemas no es solo práctico, además constituyen la base de la Teoría de la Computación.
Los problemas formales se pueden clasificar en tres categorías principales:

1. **Problemas de decisión.** Son aquellos cuya respuesta es "sí" o "no" (o sea, verdadero o falso). Ejemplos:
   - Determinar si una fórmula de LP es satisfacible (SAT).
   - Determinar si es posible colorear un grafo usando ciertos colores.
   - Determinar si existe un camino entre dos vértices de un grafo.
2. **Problemas de búsqueda.** Son aquellos que consisten en encontrar un objeto que cumpla ciertas propiedades especificadas. Dicho objeto puede no ser único. Ejemplos:
   - Encontrar una interpretación que satisface (un modelo) una fórmula de LP.
   - Encontrar un coloreo de un grafo usando ciertos colores.
   - Encontrar un camino entre dos vértices de un grafo.
3. **Problemas de optimización.** Son aquellos que consisten en encontrar el "mejor" objeto según algún criterio cuantificable. De nuevo, dicho objeto puede no ser único. Ejemplos:
   - Encontrar una interpretación que satisface la mayor cantidad de cláusulas de una fórmula en FNC.
   - Encontrar un coloreo de un grafo usando la mínima cantidad de colores (número cromático).

- Encontrar el camino más corto entre dos vértices de un grafo.

Además, estas categorías se pueden relacionar de la siguiente manera dejando clara la existencia de una jerarquía entre ellas.

- Los problemas de optimización pueden reducirse a problemas de búsqueda. Por ejemplo, consideremos el problema de encontrar un coloreo de un grafo usando la mínima cantidad de colores. Supongamos un grafo (no vacío) con $n$ vértices. Transformamos el problema de optimización a una serie de problemas de búsqueda sobre un intervalo de $m$ colores comenzando con $m = 1$ y mientras $m$ no sea suficiente incrementamos $m$ sucesivamente hasta a lo sumo $n$: [1]

  - *Encontrar un coloreo del grafo usando* $m = 1$ *color.*
  - *Encontrar un coloreo del grafo usando* $m = 2$ *colores.*

    ⋮
  - *Encontrar un coloreo del grafo usando* $m = n$ *colores.*

  Es claro que en general, para un problema de optimización dado, la cota inferior y superior para el intervalo de búsqueda depende del problema.

- Los problemas de búsqueda pueden reducirse a problemas de decisión. Por ejemplo, consideremos el problema de encontrar la posición de un elemento $x$ en un array ordenado $A$. Podemos transformarlo en una serie de problemas de decisión de la forma:

  *¿Está el elemento* $x$ *en la primera mitad del arreglo* $A$*?*

  Si la respuesta es positiva, repetimos la pregunta para la primera mitad, de lo contrario repetimos la pregunta para la segunda mitad. Esto permite progresar dividiendo el espacio de búsqueda hasta encontrar el elemento o determinar que no existe. En esencia, este razonamiento explica como opera el conocido algoritmo de búsqueda binaria.

En vista de las reducciones anteriores, los problemas de decisión se pueden considerar fundamentales en la jerarquía. Sin embargo, por razones obvias, tendemos a enfocarnos en la búsqueda porque en la práctica queremos encontrar las soluciones de un problema y no solamente saber si ellas existen. En lo siguiente, cuando hablemos de *resolver* un problema vamos a referirnos a encontrar soluciones.

## 1.2 Lógica(s) y el problema de decisión de SAT

Además de las transformaciones entre categorías mencionadas antes, resulta que muchos problemas aparentemente distintos pueden transformarse entre sí. En este laboratorio vamos a resolver una variedad de problemas de búsqueda y optimización reduciéndolos al problema de decisión y búsqueda de SAT en una lógica apropiada. Podemos pensar en el problema de decisión de SAT como un problema de decisión universal.
Sea $\mathcal{L}$ un lenguaje formal de lógica y $\alpha \in \mathcal{L}$ una fórmula, el problema de decisión de SAT se puede formular como la pregunta

*¿Existe una interpretación* $\mathcal{I}$ *de* $\mathcal{L}$ *que satisface* $\alpha$*?*

[1] Asumiendo grafos planos, el Teorema de los 4 colores nos asegura que 4 colores son suficientes. Por lo tanto, bajo dicha hipótesis, podríamos dejar de buscar luego de 4.

o alternativamente en símbolos

$$\exists \mathcal{I} \ : \ \mathcal{I} \models \alpha$$

Algunas lógicas son más expresivas que otras, y es importante tener en cuenta que la computabilidad del problema depende de la misma, por ejemplo:

- Si $\mathcal{L}$ es el lenguaje de la Lógica Proposicional (LP), sabemos que existe un algoritmo[2] que permite resolver la cuestión para cualquier $\alpha \in \mathcal{L}$ y por lo tanto decimos que la decisión de SAT en LP es decidible.
- Si $\mathcal{L}$ es un lenguaje de Lógica de Predicados (LPO), es demostrable que no existe un algoritmo que permite resolver la cuestión para cualquier $\alpha \in \mathcal{L}$ y por lo tanto decimos que la decisión de SAT en LPO es indecidible. [3]

En este laboratorio, para el primer problema será suficiente trabajar con LP. En el segundo problema parecerá más conveniente trabajar con LPO pero de todas maneras vamos a ver que podemos también resolverlo en LP. Para el último problema será mejor trabajar con una "lógica especial": una variante de LPO que al nivel de las fórmulas es más simple que la versión estándar de LPO vista en clase (porque no tiene cuantificadores, entre otras cosas) y que al nivel de los términos incluye símbolos para representar algunas operaciones aritméticas. El lenguaje para esta lógica especial, que llamaremos aquí $\mathcal{L}_{\mathbb{Z}}$, queda definido por las siguientes reglas:

$$\begin{array}{rcl} t_1, t_2 & ::= & 0 \\ & | & 1 \\ & | & x \\ & | & c \\ & | & (t_1 + t_2) \\ & | & (t_1 - t_2) \\ & | & (c * t_1) \end{array} \qquad \begin{array}{rcl} \alpha, \beta & ::= & \top \\ & | & (t_1 = t_2) \\ & | & (t_1 < t_2) \\ & | & (\neg \alpha) \\ & | & (\alpha \wedge \beta) \\ & | & (\alpha \vee \beta) \end{array}$$

Este lenguaje es representado en Haskell mediante los tipos `Term` (para los términos) y `LPO` (para las fórmulas). Será interpretado de manera fija bajo el dominio $\mathbb{Z}$ y con el significado usual para los símbolos 0 (cero), 1 (uno), $+$ (suma), $-$ (resta), $*$ (producto), $=$ (igualdad) y $<$ (menor). Observar que los productos siempre están restringidos a aplicar sobre una constante del lado izquierdo. El motivo de esta limitación quedará más claro luego de la siguiente sección.

## 1.3 De SAT-*solvers* (lógica) a SMT-*solvers* (lógica + teorías)

Distinguimos aquí entre SAT-*solvers* y SMT-*solvers*. Los primeros sirven solo para lógica pura (específicamente LP), y los segundos extienden las capacidades de los primeros e involucran la noción de *teoría* que será brevemente introducida.

**SAT-*solver*.** Por SAT-*solver* nos referimos a una herramienta computacional que resuelve problemas de SAT exclusivamente en LP. Este tipo de herramienta es al fin y al cabo una

[2] Por ejemplo el Método por Tabla de verdad o el Método de Tableaux como fue visto en la primera parte del curso. Además, notar que la semántica de LP es un álgebra de Boole, por lo cual resolver SAT es en definitiva hallar una asignación de valores sobre variables booleanas para que una expresión sea igual a 1 (p.ej. resolver SAT de $\neg p \vee q$ es lo mismo que en álgebra resolver $\overline{p} + q = 1$).

[3] Este resultado negativo no significa que LPO es inutilizable. Podemos en LPO usar el Método de Tableaux sobre una gran cantidad de fórmulas de interés sin problema.

implementación muy optimizada de algún algoritmo de resolución de SAT (como lo podría ser por ejemplo el Método de Tableaux) siendo capaz de manejar cientos o incluso miles de variables[4]. Para este tipo de herramientas, la respuesta solo puede ser `sat` o `unsat`. En el caso `sat` es posible además pedirle que exhiba un modelo.

**Teoría QF_LIA.** También es de interés tener herramientas para resolver SAT en LPO. Más específicamente, muchas veces tenemos claro el universo del discurso con el cual queremos trabajar y sobre el cual pretendemos interpretar nuestras fórmulas de LPO. Por ejemplo, puede ser que estamos interesados en razonar sobre ciertas estructuras de datos (como pueden ser listas, colas o árboles) o que estamos interesados en razonar sobre valores de $\mathbb{Z}$ (como es el caso del ejercicio 3 de este laboratorio). Esto tiene la consecuencia de que nuestro razonamiento ya no involucra solo "lógica pura", sino también otros objetos que deberán ser formalizados en términos de la lógica subyacente.
Aquí surge por primera vez la noción de *teoría* con respecto a una lógica: *una colección de axiomas formulados en la lógica que caracterizan aquellos objetos sobre los cuales queremos razonar lógicamente*. Podemos pensar en una teoría como una lógica especial donde la interpretación (elección de universo y significado del vocabulario) ha quedado fija y por tanto solo nos interesa la verdad dentro de esa interpretación.
Hay muchas teorías, algunas son decidibles y otras no. Para los propósitos de este laboratorio, será suficiente trabajar con la teoría decidible QF_LIA (*Quantifier Free Linear Integer Arithmetic*) que formaliza un fragmento de la aritmética de $\mathbb{Z}$ donde no hay cuantificadores y solo aparecen expresiones aritméticas lineales (o sea de la forma $a_1\,x_1 + a_2\,x_2 + \cdots + a_n\,x_n + c$). Por ejemplo:

- $3\,x + 2\,y - 7 \leq 0$ *está en* QF_LIA.
- $(\forall y)(2\,x + y = 0)$ *no está en* QF_LIA porque aparece un cuantificador.
- $x^2 + y < 4$ *no está en* QF_LIA porque el primer término no es lineal.

El lenguaje $\mathcal{L}_{\mathbb{Z}}$ presentado en la sección anterior tiene como propósito la construcción formal de este tipo de expresiones, por eso su restricción en el producto. No vamos a estudiar en detalle la axiomatización de QF_LIA y sus procedimiento de decisión asociados porque excede el alcance de nuestro curso, pero vamos a introducir una herramienta que se encargará de mecanizar el razonamiento en esta lógica.
Adicionalmente, como QF_LIA incluye la "lógica pura" (sin cuantificadores) como un fragmento más pequeño, también la vamos a usar para resolver problemas de lógica proposicional.

**SMT-*solver*.** Un SMT-*solver* (*Satisfiability modulo theories*) es una herramienta computacional que generaliza el problema de resolver SAT de LP a teorías de LPO que describen una variedad de objetos como números naturales, números enteros, conjuntos, arrays, strings, etc. En particular, un SMT-*solver* también es un SAT-*solver*, por lo que internamente depende de algún procedimiento de decisión de SAT. Estas herramientas pueden ser usadas para verificar mecánicamente proposiciones matemáticas. Por ejemplo, la conjetura de Goldbach afirma que todo número entero par y mayor a 2 puede escribirse como la suma de dos números primos. Siendo un poco más formales, esta afirmación puede escribirse como

$$(\forall x \in \mathbb{Z})(x > 2 \wedge par(x) \supset (\exists p_1, p_2 \in \mathbb{Z})(primo(p_1) \wedge primo(p_2) \wedge x = p_1 + p_2))$$

[4]Considerar que para este problema no se conoce hasta la fecha una manera mejor de resolverlo que no tenga complejidad computacional exponencial en el peor caso (cuestión estrechamente vinculada al problema abierto P vs NP). Las mejores implementaciones que tenemos disponibles son el resultado de décadas de esfuerzo.

donde asumimos que los predicados *par* y *primo* están definidos apropiadamente. Podemos usar un SMT-*solver* para verificar el enunciado con un $x$ particular, es decir la parte

$$(\exists p_1, p_2 \in \mathbb{Z})(primo(p_1) \wedge primo(p_2) \wedge x = p_1 + p_2)$$

En particular, para $x = 28$ la respuesta será `sat`, y si le pedimos además un modelo este podría ser $p_1 = 11$ y $p_2 = 17$ (hay otros). Sin embargo, no es razonable esperar que sea capaz de verificar el enunciado general para todo $x \in \mathbb{Z}$[5]. Siempre es importante conocer las limitaciones de las herramientas que utilizamos. En general, el razonamiento sobre fórmulas con cuantificadores es indecidible[6]. Por este motivo, la respuesta de un SMT-*solver* puede ser `sat`, `unsat` o `unknown`. Esta última ocurre habiendo pasado cierto límite de tiempo configurable (*timeout*). No obstante, cuando sea posible ordenar al SMT-*solver* limitar su operación a una teoría decidible (como lo es QF_LIA) podemos estar seguros de que la respuesta será solamente `sat` o `unsat`.
Algunos SMT-*solvers* muy conocidos son Z3, CVC5 y Yices. En este laboratorio proponemos el uso de CVC5 simplemente por ser *open-source*. Tiene disponible una interface online que puede ser ocasionalmente útil pero vamos a usarlo de forma local (ver guía de instalación).

**SMT-LIB.** Antiguamente, cada SMT-*solver* tenía su propio formato de entrada y salida, dificultando la colaboración entre usuarios de distintas herramientas. Esto motivó la creación de SMT-LIB, un estándar para el formato de entrada y salida de SMT-*solvers*, entre otras cosas. Las tres herramientas mencionadas previamente (Z3, CVC5, Yices) soportan este estándar, por lo que son en gran parte compatibles entre si.
SMT-LIB define un lenguaje de lógica usando una sintaxis estilo LISP con *S-expressions* por lo cual todos los operadores aparecen prefijos y las expresiones con espacios quedan siempre parentizadas. Por ejemplo, nuestra fórmula $p \supset q \wedge r$ de $\mathcal{L}_{\text{LP}}$ se escribe `(=> p (and q r))` en SMT-LIB. En general, no se trata de una sintaxis muy conveniente para ser usada directamente a mano porque fue pensada para ser generada de forma programática a partir de interfaces disponibles para cada lenguaje de programación. Presentaremos en la siguiente sección un módulo de Haskell que servirá para tal fin.
Si bien en la práctica el usuario no escribe en el formato de SMT-LIB directamente, es instructivo saber (aunque solo sea por razones de *debugging*) que el formato de entrada para el SAT-*solver* consiste en un script que en general puede tener la forma de la figura 1.[7]
Este script ordena al SAT-*solver* usar una lógica determinada, declara o define los símbolos del vocabulario a usar y luego afirma una o más fórmulas $\alpha_i$ para resolver la satisfacibilidad de

$$\alpha_1 \wedge \alpha_2 \wedge \cdots \wedge \alpha_n$$

Un ejemplo concreto de script formalizando el ejemplo previo de la conjetura de Goldbach para $x = 28$ se puede ver aquí. Nótese que en este caso se requiere el uso de la lógica NIA (*Non-linear Integer Arithmetic*): una extensión (no decidible pero semi-decidible) de QF_LIA que admite cuantificadores y expresiones no lineales. Además se puede poner a prueba en el editor online de CVC5, su ejecución debería responder `sat` con un modelo para las variables $p_1$ y $p_2$. Recomendamos experimentarlo para familiarizarse con el funcionamiento de la herramienta.

[5] Se trata de una conjetura porque la evidencia existente indica que se trata de una proposición verdadera (un teorema) pero que todavía no se ha podido demostrar matemáticamente. En otras palabras, su estatus es *unknown* incluso para las capacidades de razonamiento humano.

[6] Como el razonamiento matemático es en general indecidible, existen otras herramientas llamadas *proof assistants* que combinan la aplicación automática de tácticas de prueba asistiendo al usuario durante la prueba (ver p.ej. Isabelle). Estas herramientas se complementan con los SMT-*solvers*.

[7] Un tutorial de SMT-LIBv2 se puede ver aquí, sin embargo no lo consideramos una lectura necesaria.

```
(set-logic LOGIC)                    ; elegir una lógica, p.ej. QF_LIA
(set-option :produce-models true)    ; opción para generar modelo

; Definición de simbolos con interpretación fija
; Se usa el comando define-fun

; Declaración de simbolos no interpretados
; Se usa el comando declare-fun

(assert α1)
(assert α2)
...
(assert αn)

(check-sat)                  ; decidir SAT
(get-model)                  ; obtener modelo en caso positivo
```

Figura 1: Ejemplo de script genérico de SMT-LIB.

## 1.4 Presentación del módulo `SMTlib.hs`

El módulo `SMTlib.hs` incluido en este laboratorio tiene como objetivo facilitar la comunicación con cualquier SMT-*solver* compatible con SMT-LIB. El uso de este módulo (junto a los tipos `L`, `Term`, `LPO` y algunas funciones auxiliares) servirán como interfase entre el programador y el SMT-*solver* facilitando la creación de scripts y la interacción con dicha herramienta.

**Flujo de ejecución.** El flujo de ejecución al trabajar con este módulo es ilustrado en la figura 2. El módulo provee dos funciones principales que sirven para invocar al SMT-*solver*:

```
decide :: SATproblem → IO Bool
solve :: SATproblem → IO (Maybe Model)
```

La única diferencia es que mientras la primera solo decide SAT (responde `Bool`), la segunda además posiblemente devuelve un modelo (responde `Maybe Model`). Si el resultado es SAT, obtenemos `Just m` (donde `m` es un modelo) y en caso contrario obtenemos `Nothing`.
El tipo `Model` se define como [(`String`, `String`)] representando parejas (símbolo, valor). Al trabajar en LP el valor será un booleano `true` o `false`, mientras que al trabajar en la lógica especial para la aritmética el valor será un número entero.
Nos concentraremos aquí en la función `solve` ya que será nuestra manera de resolver el problema de búsqueda de SAT.
*¿Qué hace* `solve`*?* Será suficiente saber que la misma debe "salir" del ambiente de ejecución de Haskell para solicitarle al sistema subyacente:

1. La creación de un archivo `script.smt2` en el directorio de ejecución actual (nombre y ubicación por defecto) donde se guarda el script SMT-LIB (ver figura 1) con el problema a resolver.

2. La creación de un proceso para ejecutar el SMT solver pasándole la ubicación del archivo `script.smt`. Del resultado de esta ejecución obtenemos la respuesta `Maybe Model`.

*¿Por qué el retorno es de tipo* `IO`*?* Como Haskell es un lenguaje de programación funcional *puro* (no hay estado ni efectos), la manera de interactuar con el mundo exterior es a través de mónadas. La consecuencia es que el resultado de la invocación viene dentro de la mónada

`IO` (*Input Output*), y por eso el tipo de salida es `IO (Maybe Model)` y no simplemente `Maybe Model`. En otras palabras, una función como `solve` que pretende salir al mundo exterior (donde hay estado y efectos) se considera impura y por esto su tipo de salida quedará forzosamente encapsulado por `IO`. No vamos a entrar en más en detalle sobre las mónadas en este laboratorio ya que no será necesario y además es una característica más bien específica de lenguajes de programación como Haskell.

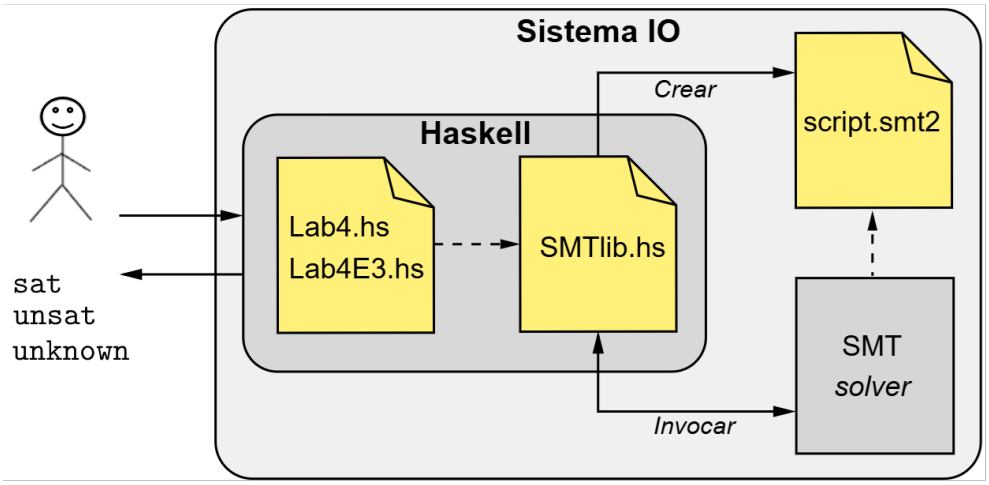



Figura 2: Flujo de ejecución. Las flechas punteadas representan dependencias. Las Flechas solidas representan acciones. La posición en la que se encuentra el módulo `SMTlib.hs` en el diagrama sugiere que se trata de una interfase entre el trabajo de nuestro laboratorio y la herramienta a utilizar. Por "Sistema IO" entendemos aquí cualquier ambiente de ejecución (interpretado o nativo) dentro del que se está ejecutando nuestro código Haskell.

**Tipo `SATproblem`.** El único parámetro de la función `solve` es de tipo `SATproblem`, esto es una especificación completa del problema de SAT a resolver. Se define

```
SATproblem = (Logic, [SymDecl], [Formula])
```

donde

- **`Logic`**. Es la lógica a usar, según estándar SMT-LIB. En este laboratorio vamos a usar siempre QF_LIA.
- **`[SymDecl]`**. Una lista de declaraciones de símbolos del vocabulario. Se distinguen dos tipos de símbolos

  1. *Símbolos interpretados (o definiciones)*: estos son símbolos a los que se les asigna un significado fijo. Decimos que "juegan el rol de constantes" ya que no varían de una interpretación a otra.
     Este es el tipo de símbolos que usamos para representar la información ya conocida del problema.
  2. *Símbolos no interpretados*: estos son símbolos que a priori no tienen significado. Decimos que "juegan el rol de variables" ya que pueden variar de una interpretación a otra.
     Este es el tipo de símbolos que usamos para representar incógnitas que queremos resolver.

- **`[Formula]`**. Una lista de fórmulas cuya conjunción se pretende analizar.

Todos estos componentes mencionados son por simplicidad codificados como *strings* y deben respetar el estándar SMT-LIB. Tenemos disponibles funciones auxiliares que nos facilitan la especificación de estos componentes. Algunas son:

- **`lp2SMT :: L → Formula`**
  Traduce una fórmula de $\mathcal{L}_{\text{LP}}$ (tipo `L`) a la sintaxis de SMT-LIB.

- **`term2SMT :: Term → Formula`**
  Traduce un término de $\mathcal{L}_{\mathbb{Z}}$ (tipo `Term`) a la sintaxis de SMT-LIB.

- **`lpo2SMT :: LPO → Formula`**
  Traduce una fórmula de $\mathcal{L}_{\mathbb{Z}}$ (tipo `LPO`) a la sintaxis de SMT-LIB.

- **`varDecl :: Type → String → SymDecl`**
  Genera la declaración de una variable con un tipo y un nombre.
  Esto en SMT-LIB tiene la forma `(declare-fun NAME () TYPE)`.
  Además, la función `mapVarDecl` generaliza esta operación para una lista de variables asumiendo que tienen el mismo tipo.

- **`constDef :: Type → String → String → SymDecl`**
  Genera la definición de una constante con un tipo, nombre y valor asignado.
  Esto en SMT-LIB tiene la forma `(define-fun NAME () TYPE VALUE)`.
  Además, la función `mapConstDef` generaliza esta operación para una lista de parejas constante-valor asumiendo que tienen el mismo tipo.

Hacemos algunas aclaraciones técnicas sobre SMT-LIB y como se compara con algunos de los conceptos teóricos estudiados en clase:

- En el vocabulario de SMT-LIB todos los símbolos son funciones, por lo tanto lo que nosotros comúnmente denominamos "constantes" son aquí funciones de aridad 0. Cuando estas constantes son simplemente *declaradas*, es decir no se les asigna valor, les denominamos variables ya que se comportan como tales al variar entre interpretaciones (por eso hemos preferido el nombre `varDecl` y no `constDecl` en las funciones auxiliares). En cambio, cuando son *definidas*, es decir se les asigna valor, les denominamos constantes como corresponde ya que no varían entre interpretaciones.

- El lenguaje lógico de SMT-LIB es *multi-sorted*, esto significa que es una lógica con tipos. El lenguaje de LPO visto en clase no tiene tipos, pero los simulamos a través de predicados apropiados (y la hipótesis implícita de que estos son exhaustivos y mutuamente excluyentes).

- En particular, `Bool` siempre es un tipo disponible sin importar cual es el dominio sobre el cual pretendemos trabajar. Por lo tanto lo que nosotros comúnmente denominamos "predicados" son aquí funciones con tipo de salida booleano. Esto permite que los predicados puedan aparecer adentro de funciones (u otros predicados) a diferencia del lenguaje de LPO visto en clase que mantiene una separación estricta entre términos y predicados. Sin embargo, no le vamos a dar ningún uso a esta facilidad. Desde nuestro punto de vista, los términos solo aparecen dentro de predicados y así es como tenemos definido $\mathcal{L}_{\mathbb{Z}}$.

**Ejemplo.** Finalizamos esta introducción mostrando un ejemplo simple de como usar el módulo para resolver una cuestión aritmética. Queremos resolver la siguiente inecuación lineal en $\mathbb{Z}$[8]

$$3\,x + 2\,y - 7 \;\leq\; 0 \tag{1}$$

Aquí identificamos tres constantes principales ($a_1 = 3$, $a_2 = 2$ y $c = 7$) y dos variables ($x$ e $y$) [9]. La figura 3 muestra la formalización en Haskell del vocabulario y la inecuación como una fórmula. Nuestra intensión es interpretar todos los símbolos en $\mathbb{Z}$.

```
voc :: [SymDecl]                    -- Vocabulario:
voc =    mapConstDef "Int" consts   -- ponemos constantes
      ++ mapVarDecl  "Int" vars     -- ponemos variables
  where
    consts = [("a1","3"), ("a2","2"), ("c","7")]
    vars   = ["x", "y"]

form :: LPO              -- Formula:
form = lhs `leq` Zero    -- 3x + 2y - 7 <= 0
  where
    lhs = ("a1" `Mul` (V "x")) `Add` ("a2" `Mul` (V "y")) `Sub` (C "c")
```

Figura 3: Código Haskell formalizando el problema de resolver la inecuación 1.

La inecuación 1 está en QF_LIA. Para resolverla, invocamos al SMT-*solver* indicando la lógica, el vocabulario y la fórmula correspondiente (traducida con `lpo2SMT`):

```
ghci> solve ("QF_LIA", voc, [lpo2SMT form])
Just [("x","0"),("y","0")]
```

Así obtuvimos una de las soluciones posibles: $x = y = 0$. Esta invocación por detrás ha generado el script de la figura 4.

```
(set-logic QF_LIA)
(set-option :produce-models true)

(define-fun a1 () Int 3)   ; Definición de constantes
(define-fun a2 () Int 2)
(define-fun c  () Int 7)

(declare-fun x () Int)     ; Declaración de variables
(declare-fun y () Int)

(assert (<= (- (+ (* a1 x) (* a2 y)) c) 0))

(check-sat)
(get-model)
```

Figura 4: Script de SMT-LIB generado por el módulo `SMTlib.hs` a partir del código de la figura 3.

[8] Por convención, al resolver una (in)ecuación se entiende que las variables de la fórmula están implícitamente cuantificadas por existenciales y no universales, o sea que estamos afirmando $(\exists x)(\exists y)(3\,x + 2\,y - 7 \leq 0)$. Pero no solo queremos saber su valor de verdad sino que también queremos conocer valores para $x$ e $y$.

[9] Alternativamente, el 0 se puede definir explícitamente como otra constante. No hay una sola manera de armar esta formalización.

# 2 Ejercicios

1. (**Veraces y mentirosos**). En una extraña isla, viven tan sólo dos tipos de habitantes: los *caballeros*, que siempre dicen la verdad, y los *escuderos*, que siempre mienten. Además, los forasteros no pueden distinguir el tipo de los habitantes a simple vista.
   En una ocasión, usted visita la isla y se encuentra con tres habitantes $A$, $B$ y $C$, entablando un diálogo. Hay distintas versiones de como fue exactamente dicho diálogo, las cuales se presentan a continuación. En cualquier caso, el objetivo es determinar el tipo de los tres habitantes.

   1.
   $A$: Todos somos escuderos.
   $B$: Solo uno de nosotros es caballero.

   2.
   $A$: $B$ y $C$ son de distinto tipo.
   $B$: $A$ no es caballero y yo soy escudero.
   $B$: $C$ es escudero.
   $C$: $A$ es caballero.

   Se pide lo siguiente:

   - Declarar el vocabulario de símbolos que se usará en la formalización, incluyendo un comentario de cómo se interpretan. Esto quedará expresado mediante la función:

     ```
     vocEj1 :: [SymDecl]
     ```

   - Completar la formalización de cada caso mediante las funciones:

     ```
     caso1 :: [L]
     caso2 :: [L]
     ```

   - Para cada caso completar el comentario con la respuesta al problema, según el resultado que se obtiene de invocar al SMT-*solver*.

   *Sugerencia.* En general, pensar:

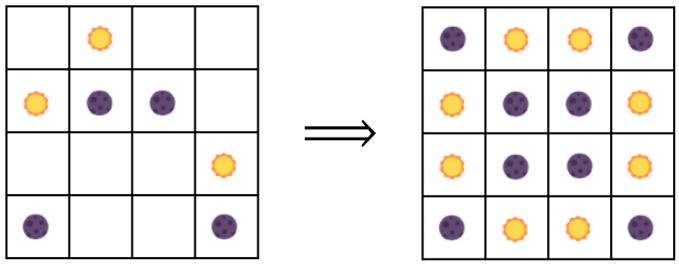
   - ¿Bajo qué condiciones la declaración de un habitante es verdadera?.
   - ¿Bajo qué condiciones la declaración de un habitante es falsa?.
   - ¿Cómo podemos expresar dichas condiciones formalmente?

2. (**Binairo y Tango**). Binairo (o Takuzu) es un juego de puzzle que se desarrolla sobre una cuadricula de tamaño $n \times n$ donde $n \geq 4$ y $n$ es par. Las celdas se pueden marcar con dos tipos de símbolos, estos pueden ser círculos negros y blancos, 1s y 0s, o como vamos a hacer aquí con soles y lunas. Al iniciar el puzzle, algunas celdas empiezan marcadas y el resto vacías. En la versión más básica de este juego, el objetivo es marcar todas las celdas según las siguientes reglas:

   (a) *Cada fila y cada columna debe contener el mismo número de ambos tipos de símbolos.*

   (b) *Cada fila y cada columna no puede contener más de dos símbolos adyacentes del mismo tipo.*

   La siguiente figura muestra un ejemplo de Binairo básico $4 \times 4$ en su forma inicial y su forma completa:



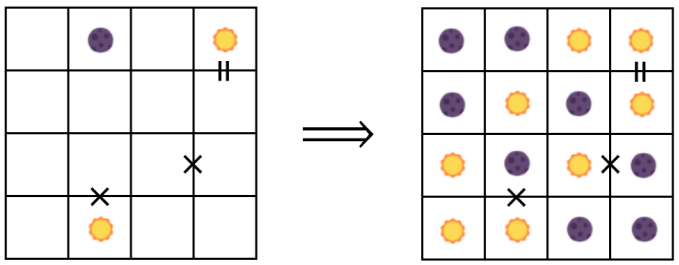
Puzzle inicial Puzzle completo

   La versión estándar usual de Binairo impone además la siguiente regla:

   (c) *No hay dos filas o columnas idénticas.*

   Observar que para el ejemplo previo el puzzle completo mostrado ya no es una solución admisible. Este tipo de puzzle se puede jugar online aquí.

   Por otro lado, la plataforma Linkedin ha popularizado otra versión de Binairo bajo el nombre de Tango (también conocido como Binairo+). Tango le añade a la versión básica condiciones de adyacencia entre celdas contiguas que imponen igualdad (indicadas con $=$) o desigualdad (indicadas con $\times$). La siguiente figura muestra un ejemplo de Tango $4 \times 4$ en su forma inicial y su forma completa:



Puzzle inicial Puzzle completo

   Este tipo de puzzle se puede jugar online aquí (como Tango) y aquí (como Binairo+).

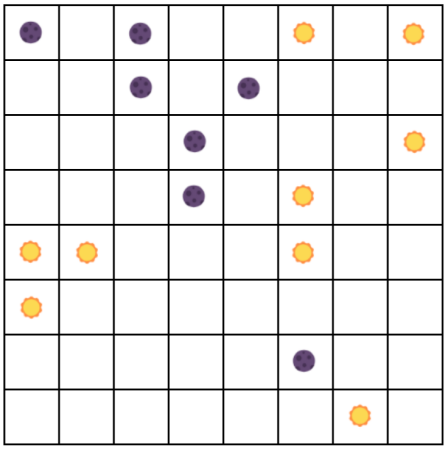
   Para la identificación de celdas, vamos a convenir en su numeración desde la esquina superior izquierda. En este ejercicio, se pide lo siguiente:

1. Formalizar el problema de resolver Binairo básico. La respuesta quedará expresada en Haskell mediante las funciones:

$$\texttt{binairo :: Binairo} \rightarrow \texttt{[L]}$$
$$\texttt{solveBinairo :: Binairo} \rightarrow \texttt{IO (Maybe Model)}$$

   donde el tipo **`Binairo`** representa una instancia de este problema.

2. Resolver el siguiente Binairo básico usando el punto anterior buscando dos soluciones. Deberá ilustrar las soluciones obtenidas sobre los archivos `binairo8x8s1.png` y `binairo8x8s2.png`.



3. Extender la formalización del punto 1 con la regla (c). La respuesta quedará expresada en Haskell mediante las funciones:

$$\texttt{binairoStd :: Binairo} \rightarrow \texttt{[L]}$$
$$\texttt{solveBinairoStd :: Binairo} \rightarrow \texttt{IO (Maybe Model)}$$

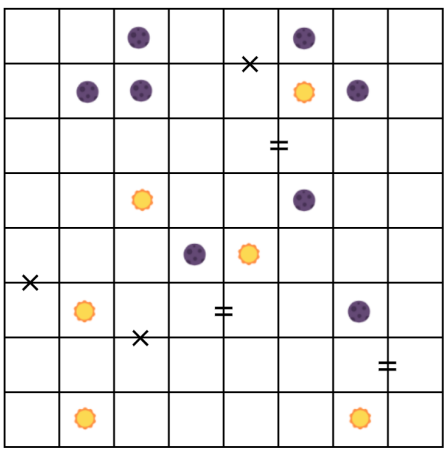
4. Resolver de nuevo el Binairo del punto 2 pero como Binairo estándar. Compare con las soluciones obtenidas anteriormente.

5. Formalizar el problema de resolver Tango. La respuesta quedará expresada en Haskell mediante las funciones:

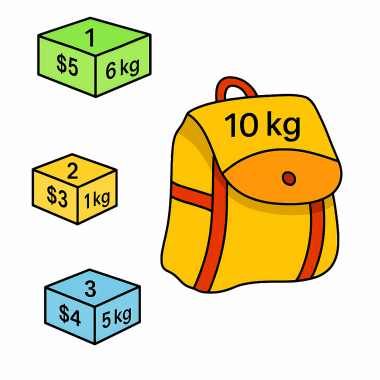
$$\texttt{tango :: Tango} \rightarrow \texttt{[L]}$$
$$\texttt{solveTango :: Tango} \rightarrow \texttt{IO (Maybe Model)}$$

   donde el tipo **`Tango`** representa una instancia de este problema.

6. Resolver el siguiente Tango usando el punto anterior. Deberá ilustrar la solución obtenida sobre el archivo `tango8x8.png`.



3. **(Problema de la mochila).** El problema de la mochila (en inglés *Knapsack problem*) es un problema de optimización combinatoria que permite modelar una situación análoga a tener una mochila con cierta capacidad máxima de peso y un conjunto de objetos con cierto valor y peso donde el objetivo es determinar qué objetos debemos elegir para colocar en la mochila de tal manera que el valor total sea maximizado y el peso máximo no sea excedido.
Por ejemplo, la siguiente figura y tabla presentan una situación concreta donde la mochila a considerar tiene 10kg de capacidad y tenemos disponible un conjunto de tres objetos, ilustrados como cajas con valor y peso, de donde podemos elegir.




| Objeto | Valor ($) | Peso (kg) |
|---|---|---|
| 1 | 5 | 6 |
| 2 | 3 | 1 |
| 3 | 4 | 5 |

Este problema es generalmente formulado en términos matemáticos de la siguiente manera

$$\begin{aligned} \text{Maximizar} \quad & \sum_{i=1}^{n} v_i \, x_i \\ \text{tal que} \quad & \sum_{i=1}^{n} w_i \, x_i \leq w_{max} \\ \text{y} \quad & x_i \in \{0, 1\} \end{aligned}$$

donde $w_{max}$ es la capacidad de peso máximo de la mochila y para cada objeto $i \in \{1, 2, \ldots, n\}$ tenemos su valor $v_i$, peso $w_i$ y además una variable $x_i$ (incógnita) indicando su selección.

Lo que acabamos de describir es más precisamente lo que se conoce como el "problema de la mochila 0-1", la variante más conocida del problema. Vamos a considerar aquí tres posibles variantes de este problema y mencionar su solución para el ejemplo de mochila y objetos ilustrado previamente:

(a) **Problema de la mochila 0-1.** Cada objeto se asume único, y para cada uno tenemos que elegir si lo llevamos o no.
*Solución del ejemplo*: elegimos los objetos 1 y 2. De esta manera tenemos un valor total de 8$ y un peso total de 7kg.

(b) **Problema de la mochila 0-$\infty$.** Cada objeto tiene ilimitadas copias disponibles, y para cada uno tenemos que elegir cuantas copias del mismo llevamos.
*Solución del ejemplo*: elegimos diez copias del objeto 2. De esta manera tenemos un valor total de 30$ y un peso total de 10kg.

(c) **Problema de la mochila 0-$c$.** Cada objeto tiene $c$ copias disponibles, y para cada uno tenemos que elegir cuantas copias del mismo llevamos. Notar que esta variante del problema depende de un nuevo parámetro $c$.

*Solución del ejemplo asumiendo $c = 2$*: elegimos una copia del objeto 1 y dos copias del objeto 2. De esta manera tenemos un valor total de 11$ y un peso total de 8kg.
*Solución del ejemplo asumiendo $c = 5$*: elegimos una copia del objeto 3 y cinco copias del objeto 2. De esta manera tenemos un valor total de 19$ y un peso total de 10kg.

En este ejercicio, se pide lo siguiente:

1. La formulación matemática dada anteriormente es para el problema de la mochila 0-1. Adaptarla para las otras dos variantes.
2. Explique si es posible reducir los problemas de la mochila 0-1 y 0-$\infty$ a la versión 0-$c$.
3. Formalizar el problema de la mochila 0-$c$. La respuesta quedará expresada en Haskell mediante las funciones:

```
mochila0c :: KSc → [LPO]
solveMochila0c :: KSc → IO (Maybe Model)
maxMochila0c :: KSc → IO (Maybe (Int,Model))
```

   donde el tipo `KSc` representa una instancia de este problema.
4. Formalizar el problema de la mochila 0-1. La respuesta quedará expresada en Haskell mediante las funciones:

```
mochila01 :: KS1 → [LPO]
solveMochila01 :: KS1 → IO (Maybe Model)
maxMochila01 :: KS1 → IO (Maybe (Int,Model))
```

   donde el tipo `KS1` representa una instancia de este problema.
5. Formalizar el problema de la mochila 0-$\infty$. La respuesta quedará expresada en Haskell mediante las funciones:

```
mochila0inf :: KSinf → [LPO]
solveMochila0inf :: KSinf → IO (Maybe Model)
maxMochila0inf :: KSinf → IO (Maybe (Int,Model))
```

   donde el tipo `KSinf` representa una instancia de este problema.
6. Para una mochila de capacidad 15kg y la siguiente lista de objetos:

| **Objeto** | **Valor ($)** | **Peso (kg)** |
|---|---|---|
| 1 | 12 | 3 |
| 2 | 8 | 2 |
| 3 | 2 | 5 |
| 4 | 15 | 3 |
| 5 | 5 | 1 |
| 6 | 22 | 6 |
| 7 | 9 | 2 |
| 8 | 13 | 3 |
| 9 | 7 | 2 |
| 10 | 24 | 4 |

Se pide resolver:

(a) Problema de la mochila 0-1.

(b) Problema de la mochila 0-$\infty$.

(c) Problema de la mochila 0-$c$ con $c = 2$.

7. Introducimos ahora el problema del Subconjunto suma. El problema es, dado un conjunto de enteros $S$ y un número $k$, *¿existe algún subconjunto de $S$ cuya suma sea exactamente $k$?*

   Por ejemplo, para $S = \{6, 3, 2, 5, 8\}$ y $k = 10$, la respuesta es "sí" porque el subconjunto $\{3, 2, 5\}$ suma 10.

   En esta parte se pide formalizar este nuevo problema. La respuesta quedará expresada en Haskell mediante las funciones:

```
subsetSum :: SS → [LPO]
solveSubsetSum :: SS → IO (Maybe Model)
```

   donde el tipo `SS` representa una instancia de este problema.

8. Resolver el problema del Subconjunto suma para:

$$S = \{42,\ 18,\ 24,\ 78,\ 12,\ 66,\ 2,\ 14,\ 50,\ 34,\ 30,\ 28,\ 56,\ 6,\ 10\}$$
$$k_1 = 84$$
$$k_2 = 95$$

9*. Construyendo sobre el trabajo previo, escribir una función que cuenta todas las soluciones para un problema `SS` y usarla para determinar el valor en el caso de $S$ y $k_1$ del item precedente.

*Sugerencia.* Se recomienda implementar las siguientes funciones de soporte que le permitirán construir en Haskell operadores sobre intervalos.

- **`bigAnd :: [Nat] → (Nat → L) → L`**
  Dada una lista de índices $I$ y una fórmula indexada $\alpha$, devuelve la conjuntaria de $\alpha$ sobre $I$, o sea $\bigwedge_{i \in I} \alpha_i$.
  Ejemplo: Para representar $\bigwedge_{i \in [1,2,3]} p_i$ en Haskell escribimos
  `bigAnd` $[1,2,3]$ $(\lambda i \to$ `V` $(\text{“}$`p`$\text{”}$`++ show i`$))$ = (`V` “$p1$”) `And` (`V` “$p2$”) `And` (`V` “$p3$”)

- **`bigOr :: [Nat] → (Nat → L) → L`**
  Dada una lista de índices $I$ y una fórmula indexada $\alpha$, devuelve la disyuntoria de $\alpha$ sobre $I$, o sea $\bigvee_{i \in I} \alpha_i$.
  Ejemplo: Para representar $\bigvee_{i \in [1,2,3]} p_i$ en Haskell escribimos
  `bigOr` $[1,2,3]$ $(\lambda i \to$ `V` $(\text{“}$`p`$\text{”}$`++ show i`$))$ = (`V` “$p1$”) `Or` (`V` “$p2$”) `Or` (`V` “$p3$”)

- **`bigAdd :: [Int] → (Int → Term) → Term`**
  Dada una lista de índices $I$ y un término indexado $t$, devuelve la sumatoria de $t$ sobre $I$, o sea $\sum_{i \in I} t_i$.
  Ejemplo: Para representar $\sum_{i \in [1,2,3]} x_i$ en Haskell escribimos
  `bigAdd` $[1,2,3]$ $(\lambda i \to$ `V` $(\text{“}$`x`$\text{”}$`++ show i`$))$ = (`V` “$x1$”) `Add` (`V` “$x2$”) `Add` (`V` “$x3$”)

Ilustramos con un ejemplo como se pueden combinar algunas de las funciones anteriores en el contexto de LP. La conjunción de disyunciones de literales indexados (una FNC) sobre $I = [1..n]$ y $J = [1..m]$:

$$\bigwedge_{i \in I} \bigvee_{j \in J} p_{i,j}$$

es representada en Haskell escribiendo

`bigAnd` $[1..n]$ $(\lambda i \to$ `bigOr` $[1..m]$ $(\lambda j \to$ `v2` $\text{“}p\text{”}\, i\, j))$

En particular, si $n = 2$ y $m = 2$, esto construye en Haskell la fórmula de LP

`Bin` (`Bin` (`V` “$p1\_1$”) `Or` (`V` “$p1\_2$”)) `And` (`Bin` (`V` “$p2\_1$”) `Or` (`V` “$p2\_2$”))

que en la sintaxis de SMT-LIB (requerido para trabajar con CVC5) se escribe

(`and` (`or` $p1\_1$ $p1\_2$) (`or` $p2\_1$ $p2\_2$))

y en nuestra sintaxis concreta (“notación de mano alzada”) se escribe

$$(p_{1,1} \vee p_{1,2}) \wedge (p_{2,1} \vee p_{2,2})$$