UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Trabajo fin de máster

Máster en Ingeniería del Software, Métodos Formales y Sistemas de Información

Extensiones a la comprobación de satisfacibilidad de restricciones

Autor:
Pablo Viciano Negre

Director de tesina:
Santiago Escobar

17 de septiembre de 2012

Resumen

En esta tesina de máster se estudia la satisfacibilidad de fórmulas en la aritmética Presburger para el lenguaje de programación de alto rendimiento Maude y cómo se pueden extender estos algoritmos a modelos que extiendan la aritmética Presburger con propiedades ecuacionales tales como asociatividad, conmutatividad e identidad, así como el caso más complejo (con un algoritmo de semi-decisión en vez de un algoritmo de decisión) para teorías ecuacionales con propiedades ecuacionales orientadas como reglas.

Palabras Clave

satisfacibilidad de fórmulas ; unificación ecuacional ; estrechamiento ecuacional

Índice general

1

Introducción

La satisfacibilidad de fórmulas (*Satisfiability* - *SAT*) ha atraído [1] a muchos investigadores de diferentes disciplinas tales como la inteligencia artificial o la verificación formal durante los últimos años debido a la increíble mejora en rendimiento de los *SAT solvers*. Se ha convertido en la pieza de tecnología más decisiva en muchas áreas de verificación hardware y software. Por ejemplo, la verificación de modelos con límites (*Bounded Model Checking* - *BMC*) es una de las áreas que más extensamente utiliza *SAT solvers*. Donde un modelo (generado hasta cierto límite) se traduce en cantidades enormes de fórmulas booleanas tal que la verificación de una propiedad concreta sobre ese modelo consiste en añadir un conjunto extra de fórmulas booleanas y utilizar un *SAT solver* para comprobar la satisfacibilidad del conjunto extendido de fórmulas booleanas.

Sin embargo, aunque la satisfacibilidad de fórmulas ha ayudado a muchas áreas, ocurre cada vez con más frecuencia que estas aplicaciones requieren la satisfacibilidad de fórmulas en lógicas más ricas o con propiedades semánticas extra (correspondientes a la denominada teoría de fondo). Para muchas teorías de fondo, los métodos especializados han permitido disponer de procedimientos de decisión para la satisfacibilidad de fórmulas libres de cuantificadores o para algunas subclases; ver [2]. De hecho, muchos procedimientos han sido descubierto o inventados para teorías de fondo tales como varias teorías aritméticas, ciertas teorías de vectores, teorías de listas, tuplas, registros y vectores de bits (muy comunes y necesarias en lenguajes de programación como C). La satisfacibilidad de fórmulas bajo una teoría de fondo se denomina satisfacibilidad modulo teorías (*Satisfiability Modulo Theories* -

SMT).

En esta memoria se estudia la adaptación de procedimientos de satisfacibilidad para la aritmética Presburger (que incluye números naturales, la suma y el símbolo de menor-que) en Maude. También estudiamos su extensión a cualquier otro símbolo con propiedades algebraicas como la asociatividad, conmutatividad y la identidad. Y estudiamos también el caso más complejo, pero más difícil, de la satisfacibilidad para teorías ecuacionales con más propiedades ecuaciones (orientables como reglas). En la tesina se crea un prototipo que implementa la teoría asociada a cada parte.

Esta tesina se ha desarrollado dentro de una iniciativa internacional para añadir SMT al lenguaje Maude, liderada por el profesor José Meseguer de la *University of Illinois at Urbana-Champaign*, Vijay Ganesh del *MIT* y Santiago Escobar de la *Universitat Politècnica de València*.

Este prototipo ha sido definido usando una versión modificada de **Maude** (Sección 2.2) que lleva integrado el SAT-Solver **CVC3** (Sección 2.3). Además, se utiliza internamente una interfaz definida por *Camilo Rocha* (Sección 2.4) de la *University of Illinois at Urbana-Champaign* para la interacción entre **Maude-CVC3**. El prototipo se divide en tres partes (se pueden observar en la Figura 1.1)

- Primera parte: Transformación de términos en valores válidos para la interfaz **Maude-CVC3**.
- Segunda parte: Abstracción y combinación de variables en los términos (utilizando *unificación*).
- Tercera parte: Expansión de funciones creando restricciones de las reglas definidas en módulos y obtención de sustituciones (utilizando *narrowing*).

La estructura de la memoria constará, en primer lugar, de la definición de algunos conceptos básicos como el lenguaje **Maude** (tipos de datos, definición de reglas, ecuaciones, étc.), cómo funciona **CVC3** y la definición y ejecución de la interfaz **Maude-CVC3**.

Posteriormente se presentará cada una de las partes del prototipo donde, al principio se detallará la teoría relativa a dicha parte y, seguidamente se

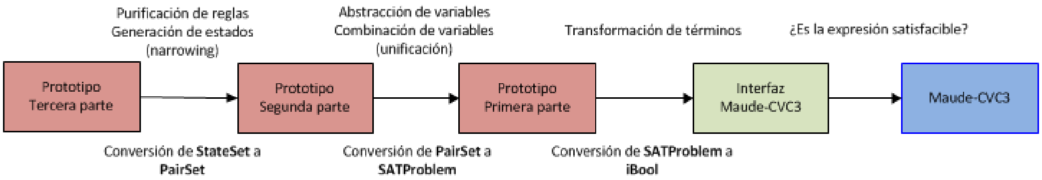

Figura 1.1: Esquema general del prototipo

describirá cómo se ha implementado, los tipos de datos, flujos de ejecución y algunos ejemplos prácticos.

Finalmente se presentan las conclusiones y las posibles extensiones que se pueden realizar en el futuro.

2

Conceptos básicos

2.1. Sistemas de reescritura de términos

En esta tesina se sigue la notación clásica de [3] para reescritura de términos y la notación clásica de [4] para lógica de reescritrua y nociones de tipos ordenados. Asumimos una signatura de tipos ordenados $\mathbf{\Sigma} = (S, \leq, \Sigma)$ con un conjunto de tipos parcialmente ordenado $(S, \leq)$. También asumimos una familia de variables ordenadas por los tipos S definida de la siguiente forma $\mathcal{X} = \{\mathcal{X}_{\mathsf{s}}\}_{\mathsf{s} \in \mathsf{S}}$ basada en conjuntos disjuntos de variables con cada conjunto $\mathcal{X}_{\mathsf{s}}$ siendo infinitamente enumerable. El conjunto $\mathcal{T}_{\Sigma}(\mathcal{X})_{\mathsf{s}}$ denota los términos de tipo s, y $\mathcal{T}_{\Sigma,\mathsf{s}}$ denota los términos sin variables del tipo s. Escribiremos $\mathcal{T}_{\Sigma}(\mathcal{X})$ y $\mathcal{T}_{\Sigma}$ para las correspondientes algebras de términos de tipos ordenados. Dado un término t, $Var(t)$ representa el conjunto de variables de t.

Las posiciones de un término se representan con secuencias de números naturales que denotan un camino de acceso en el término cuando el término se ve como un árbol. El tope o posición raíz se denota con la secuencia vacía Λ. La relación $p \leq q$ entre posiciones se define como $p \leq p$ para toda posición p; y $p \leq p.q$ para toads las posiciones p y q. Dado un conjunto $U \subseteq \Sigma \cup \mathcal{X}$, $Pos_U(t)$ denote el conjunto de posiciones del término t que están encabezadas por símbolos o variables de U. El conjunto de posiciones de un término t se escribe $Pos(t)$, y el conjunto de posiciones no variables $Pos_{\Sigma}(t)$. El subtérmino de t en la posición p se escribe $t|_p$ y $t[u]_p$ representa el término t donde el subtérmino $t|_p$ se ha reemplazado por u.

Una *sustitución* $\sigma \in \mathcal{S}ubst(\Sigma, \mathcal{X})$ es un mapeo ordenado de variables en $\mathcal{X}$ hacia términos en $\mathcal{T}_\Sigma(\mathcal{X})$ que es casi siempre la identidad salvo un conjunto finito $X_1, \ldots, X_n$ de variables de $\mathcal{X}$, denominado el dominio de σ. Las sustituciones se escriben $\sigma = \{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$ donde el domino de σ is $Dom(\sigma) = \{X_1, \ldots, X_n\}$ y el conjunto de variables introducidas por $t_1, \ldots, t_n$ se escribe $Ran(\sigma)$. La sustitución identidad es *id*. Las sustituciones se extienden homomorficamente al conjunto de términos $\mathcal{T}_\Sigma(\mathcal{X})$. La aplicación de una sustitución σ a u término t se representa como $t\sigma$ ó $\sigma(t)$. Por simplicidad, se asume que todas las sustituciones son idempotentes, es decir, cada sustitución σ satisface $Dom(\sigma) \cap Ran(\sigma) = \emptyset$. La idempotencia de las sustituciones asegura la siguiente propiedad $t\sigma = (t\sigma)\sigma$. La restricción de σ a un conjunto de variables V es $\sigma|_V$. La composición de dos sustituciones σ y σ' se denota como $\sigma\sigma'$. La combinación de dos sustituciones σ y σ' tal que $Dom(\sigma) \cap Dom(\sigma') = \emptyset$ se denota como $\sigma \cup \sigma'$. Una sustitución idempotente σ es un *renombramiento* si existe otra sustitución idempotente σ^{-1} tal que $(\sigma\sigma^{-1})|_{Dom(\sigma)} = id$.

Una Σ-*ecuación* es un par no ordenado $t = t'$, donde $t, t' \in \mathcal{T}_\Sigma(\mathcal{X})_{\mathsf{s}}$ para un tipo $\mathsf{s} \in \mathsf{S}$. Dada la signatura Σ y un conjunto E de Σ-equations, la lógica ecuacional de tipos ordenados induce una relación de congruencia $=_E$ sobre los términos $t, t' \in \mathcal{T}_\Sigma(\mathcal{X})$ (ver [5]). En esta tesina, se asume que $\mathcal{T}_{\Sigma,\mathsf{s}} \neq \emptyset$ para cada tipo s, ya que proporciona un sistema de deducción más sencillo. Una *teoría ecuacional* (Σ, E) es un par consistente en la signatura de tipos ordenados Σ y un conjunto E de Σ-ecuaciones.

El preorden de E-*subsunción* $\sqsupseteq_E$ (ó simplemente $\sqsupseteq$ si E se sobreentiende) se satisface entre dos términos $t, t' \in \mathcal{T}_\Sigma(\mathcal{X})$, denotado $t \sqsupseteq_E t'$ (entendiéndose que t es *más general* que t' modulo E), si existe una sustitución σ tal que $t\sigma =_E t'$; dicta sustitución σ se denomina un E-*match* entre t' y t. La relación de E-renombrado $t \approx_E t'$ se satisface si existe una renombramiento de variables θ tal que $t\theta =_E t'$. Dadas dos sustituciones σ, ρ y un conjunto de variables V, se satisface $\sigma|_V =_E \rho|_V$ si $x\sigma =_E x\rho$ para todas las variables $x \in V$; $\sigma|_V \sqsupseteq_E \rho|_V$ si existe una sustitución η tal que $(\sigma\eta)|_V =_E \rho|_V$; y $\sigma|_V \approx_E \rho|_V$ si existe un renombramiento η tal que $(\sigma\eta)|_V =_E \rho|_V$.

Un E-*unificador* para una Σ-ecuación $t = t'$ es una sustitución σ tal que $t\sigma =_E t'\sigma$. Dado el conjunto de variables W tal que $Var(t) \cup Var(t') \subseteq W$, un conjunto de sustituciones $CSU_E^W(t = t')$ se dice que es un conjunto *completo* de unificadores para la igualdad $t = t'$ modulo E fuera del conjunto W de variables si y sólo si: (i) cada $\sigma \in CSU_E^W(t = t')$ es un E-unificador de $t = t'$; (ii) para cada E-unificador ρ de $t = t'$ existe un $\sigma \in CSU_E^W(t = t')$ tal que

$\sigma|_W \sqsupseteq_E \rho|_W$; (iii) para cada $\sigma \in CSU_E^W(t = t')$, $Dom(\sigma) \subseteq (Var(t) \cup Var(t'))$ y $Ran(\sigma) \cap W = \emptyset$. Si el conjunto de variables W es irrelevante o se sobreentiende del contexto, escribiremos $CSU_E(t = t')$ en vez de $CSU_E^W(t = t')$. Un algoritmo de E-unificación es *completo* si para cada ecuación $t = t'$ genera un conjunto completo de E-unificadores. Un algoritmo de unificación se dice que es *finitario* y completo si siempre termina generando un conjunto finito y completo de soluciones.

Una *regla de reescritura* es un par orientado $l \to r$, donde $l \notin \mathcal{X}$, $Var(r) \subseteq Var(l)$, y $l, r \in \mathcal{T}_\Sigma(\mathcal{X})_{\mathsf{s}}$ para un tipo $\mathsf{s} \in \mathsf{S}$. Una *teoría incondicional de reescritura para tipos ordenados* es una tripleta (Σ, E, R) con Σ una signatura de tipos ordenados, E un conjunto de Σ-ecuaciones, y R un conjunto de reglas de reescritura.

La relación de reescritura entre términos $\mathcal{T}_\Sigma(\mathcal{X})$, escrita $t \to_R t'$ ó $t \to_{p,R} t'$ se satisface entre dos términos t y t' si y sólo si existe una posición $p \in Pos_\Sigma(t)$, una regla $l \to r \in R$ y una sustitución σ, tal que $t|_p = l\sigma$ y $t' = t[r\sigma]_p$. El subtérmino $t|_p$ se denomina un *redex*. La relación $\to_{R/E}$ sobre $\mathcal{T}_\Sigma(\mathcal{X})$ is $=_E; \to_R; =_E$. Nótese que la relación $\to_{R/E}$ sobre $\mathcal{T}_\Sigma(\mathcal{X})$ induce una relación $\to_{R/E}$ sobre el (Σ, E)-algebra libre $\mathcal{T}_{\Sigma/E}(\mathcal{X})$ de la forma $[t]_E \to_{R/E} [t']_E$ si y sólo si $t \to_{R/E} t'$. El cierre transitivo (resp. transitivo y reflexivo) de $\to_{R/E}$ se denota $\to_{R/E}^+$ (resp. $\to_{R/E}^*$). Un término t se denomina $\to_{R/E}$-irreducible (o simplemente R/E-irreducible) si no existe ningún término t' tal que $t \to_{R/E} t'$.

Dada una regla de reescritura $l \to r$, se dice que es *decreciente en tipo* (*sort-decreasing*) si para cada sustitución σ, se tiene que $r\sigma \in \mathcal{T}_\Sigma(\mathcal{X})_{\mathsf{s}}$ implica $l\sigma \in \mathcal{T}_\Sigma(\mathcal{X})_{\mathsf{s}}$. Una teoría de reescritura (Σ, E, R) es decreciente en tipo si todas las reglas R lo son. Dada una Σ-ecuación $t = t'$, se dice que es *regular* si $Var(t) = Var(t')$, y se dice que es *decreciente en tipo* si para cada sustitución σ, se tiene que $t\sigma \in \mathcal{T}_\Sigma(\mathcal{X})_{\mathsf{s}}$ implica $t'\sigma \in \mathcal{T}_\Sigma(\mathcal{X})_{\mathsf{s}}$ y viceversa.

Dadas dos sustituciones σ, ρ y un conjunto de variables V la siguiente relación de reescritura para sustituciones se satisface $\sigma|_V \to_{R/E} \rho|_V$ si existe $x \in V$ tal que $x\sigma \to_{R/E} x\rho$ y para todas las demás variables $y \in V$ se tiene $y\sigma =_E y\rho$. Una sustitución σ se denomina *R/E-irreducible* (ó normalizada) si $x\sigma$ es R/E-irreducible para toda variable $x \in V$.

La relación de reescritura $\to_{R/E}$ se denomina *terminante* si no existe una secuencia infinita $t_1 \to_{R/E} t_2 \to_{R/E} \cdots t_n \to_{R/E} t_{n+1} \cdots$. Además, la relación

$\rightarrow_{R/E}$ es *confluente* si cuando $t \rightarrow^*_{R/E} t'$ y $t \rightarrow^*_{R/E} t''$, existe un término t''' tal que $t' \rightarrow^*_{R/E} t'''$ y $t'' \rightarrow^*_{R/E} t'''$. Una teoría de reescritura de tipos ordenados (Σ, E, R) es confluente (resp. terminante) si la relación $\rightarrow_{R/E}$ es confluente (resp. terminante). En una teoría de reescritura de tipos ordenados que sea confluente, terminante y decreciente en tipo, para cada término $t \in \mathcal{T}_\Sigma(\mathcal{X})$, existe una única forma R/E-irreducible t' (modulo E-equivalencia) obtenida de t por reescritura hasta la forma canonica, la cual se denota como $t \rightarrow^!_{R/E} t'$, o $t{\downarrow}_{R/E}$ cuando t' es irrelevante.

La relación $\rightarrow_{R,E}$ sobre $\mathcal{T}_\Sigma(\mathcal{X})$ se define de la siguiente forma: $t \rightarrow_{p,R,E} t'$ (o simplemente $t \rightarrow_{R,E} t'$) si y sólo si existe una posición $p \in Pos_\Sigma(t)$, una regla $l \rightarrow r$ en R y una sustitución σ tal que $t|_p =_E l\sigma$ y $t' = t[r\sigma]_p$. Nótese que si la relación de E-emparejamiento es decidible, la relación $\rightarrow_{R,E}$ es decidible. Las nociones de confluencia, terminación y términos y sustituciones irreducibles se adaptan trivialmente para la relación $\rightarrow_{R,E}$. Si el conjunto de reglas R es confluente, terminante y decreciente en tipo, la relación $\rightarrow^!_{R,E}$ es decidible, ya que $\rightarrow_{R,E} \subseteq \rightarrow_{R/E}$.

La relación $\rightarrow_{R/E}$ es indecidible en general ya que las clases de E-congruencia pueden ser arbitrariamente extensas. Por lo tanto, la relación de reescritura $\rightarrow_{R/E}$ se implementa normalmente a través de la relación $\rightarrow_{R,E}$ (ver [6]). Se asumen las siguientes propiedades sobre R y E:

1. E es regular y decreciente en tipo; además, para cada ecuación $t = t'$ en E, todas las variables de $Var(t)$ tienen un tipo máximo.

2. E tiene un algoritmo finitario y completo de unificación.

3. Las reglas R son confluentes, terminantes y decrecientes en tipo modulo E.

4. $\rightarrow_{R,E}$ es *localmente E-coherente* (ver [6]), es decir, para todos los términos t_1, t_2, t_3 tenemos que $t_1 \rightarrow_{R,E} t_2$ y $t_1 =_E t_3$ implica existe t_4, t_5 tal que $t_2 \rightarrow^*_{R,E} t_4$, $t_3 \rightarrow^+_{R,E} t_5$, y $t_4 =_E t_5$.

Dada una teoría ecuacional para tipos ordenados (Σ, G), decimos que (Σ, E, R) es una *descomposición* de (Σ, G) si $G = R \cup E$ y (Σ, E, R) es una teoría de reescritura para tipos ordenados que satisfaga las propiedades (1)–(4) indicadas más arriba.

Dada una teoría de reescritura para tipos ordenados (Σ, E, R), un término t y un conjunto W de variables tales que $Var(t) \subseteq W$, la relación de R, E-*estrechamiento* (R, E-*narrowing*) sobre $\mathcal{T}_\Sigma(\mathcal{X})$ se define como $t \leadsto_{p,\sigma,R,E} t'$ (ó $\leadsto_{\sigma,R,E}$ si p se sobreentiende, $\leadsto_\sigma$ si R, E se sobreentiende, y $\leadsto$ si σ se sobreentiende) si existe una posición no variable $p \in Pos_\Sigma(t)$, una regla $l \to r \in R$ apropiadamente renombrada tal que $(Var(l) \cup Var(r)) \cap W = \emptyset$, y un unificador $\sigma \in CSU_E^{W'}(t|_p = l)$ para el conjunto de variables $W' = W \cup Var(l)$, tal que $t' = (t[r]_p)\sigma$. Por conveniencia, en cada paso de estrechamiento $t \leadsto_\sigma t'$ sólo se espeficica la parte de la sustitución σ que liga variables del término t. El cierre transitivo (resp. transitive y reflexivo) de la relación $\leadsto$ se denota como $\leadsto^+$ (resp. $\leadsto^*$). Escribiremos $t \leadsto_\sigma^k t'$ si existen términos $u_1, \ldots, u_{k-1}$ y sustituciones $\rho_1, \ldots, \rho_k$ tales que $t \leadsto_{\rho_1} u_1 \cdots u_{k-1} \leadsto_{\rho_k} t'$, $k \geq 0$ y $\sigma = \rho_1 \cdots \rho_k$.

2.2. Maude

El lenguaje de programación **Maude** utiliza reglas de reescritura, como los lenguajes denominados funcionales tales como **Haskell**, **ML**, **Scheme**, o **Lisp**. En concreto, el lenguaje **Maude** está basado en la lógica de reescritura que permite definir multitud de modelos computacionales complejos tales como programación concurrente o programación orientada a objetos. Por ejemplo, **Maude** permite especificar objetos directamente en el lenguaje, siguiendo una aproximación declarativa a la programación orientada a objetos que no está disponible ni en lenguajes imperativos como **C++**o **Java** ni en lenguajes declarativos como **Haskell**.

El desarrollo del lenguaje **Maude** parte de una iniciativa internacional cuyo objetivo consiste en diseñar una plataforma común para la investigación, docencia y aplicación de los lenguajes declarativos. Se puede encontrar más información en:

`http://maude.cs.uiuc.edu`

A continuación se resumen las principales carácterísticas del lenguaje **Maude**. Sin embargo, hay un extenso manual y un "primer"(libro de introducción muy sencillo y basado en ejemplos) en la dirección web indicada antes. Existe también un libro sobre **Maude**, con ejemplares adquiridos por la Biblioteca General y la ETSInf, y accesible online en :

http://www.springerlink.com/content/p6h32301712p

2.2.1. Un programa en Maude

Un programa **Maude** esta compuesto por diferentes módulos. Cada módulo se define entre las palabras reservadas `mod` y `endm`, si es un *módulo de sistema*, o entre `fmod` y `endfm`, si es un *módulo funcional*. Cada módulo incluye declaraciones de tipos y símbolos, junto con las reglas, encabezadas por `rl`, que describen la lógica de algunos de los símbolos, llamadas *funciones*. Básicamente, los símbolos y reglas definidos en un módulo de sistema tienen un comportamiento indeterminista y ejecuciones posiblemente infinitas en el tiempo (es decir, que no terminen nunca); mientras que los símbolos y reglas definidos en un módulo funcional, encabezadas por `eq` ya que se denominan ecuaciones en este caso, tienen un comportamiento determinista y siempre terminan su ejecución. Es decir, un módulo de sistema permite reglas indeterministas y no terminantes, ya que modela un sistema de estados (o autómata) y, claramente, pueden haber ciclos y varias posibles acciones a tomar para cada estado del sistema. Sin embargo, un módulo funcional sólo permite ecuaciones (es decir, reglas deterministas y terminantes), ya que representa un programa funcional y todo programa termina y debe devolver siempre el mismo valor. Por ejemplo, el siguiente módulo de sistema simula una máquina de café y galletas:

```
mod VENDING-MACHINE is
  sorts Coin Coffee Cookie Item State .
  subsorts Coffee Cookie < Item .
  subsorts Coin Item < State .
  op null : -> State .
  op __ : State State -> State [assoc comm id: null] .
  op $ : -> Coin .
  op q : -> Coin .
  op a : -> Cookie .
  op c : -> Coffee .
  var St : State .

  rl St => St q .     --- Modela que se ha anyadido un cuarto de dolar
  rl St => St $ .     --- Modela que se ha anyadido un dolar
  rl $  => c .        --- Modela que se ha trabado el dolar
                                      --- y ha devuelto un cafe
  rl $  => aq .       --- Devuelve una galleta y un cuarto de dolar
  eq q q q q = $.     --- Cambia cuatro cuartos de dolar por un dolar
endm
```

Este sistema es indeterminista (p.ej. para un dólar "$"hay dos posibles acciones) y no terminante (siempre se puede añadir más dinero a la máquina). Además el módulo incluye una ecuación para el cambio de cuatro cuartos de dólar por un dólar de manera transparente, es decir sin que haya una transición entre dos estados.

Sin embargo, podemos especificar el siguiente módulo que simula la función factorial:

```
fmod FACT is
  protecting INT .
  op _! : Int -> Int .
  var N : Int .
  eq 0 ! = 1 .                       --- factorial de N=0 es 1
  eq N ! = (N - 1)! * N [owise] . --- factorial de N>0 es N*factorial de N-1
endfm
```

Este sistema es determinista y termina para cada posible ejecución. Nótese que cada línea de texto se termina con un espacio y un punto. En un módulo de sistema podremos incluir reglas y ecuaciones, pero en un módulo funcional sólo pueden aparecer ecuaciones.

2.2.2. Tipos de datos predefinidos

Maude dispone de varios tipos de datos predefinidos incluidos en el fichero `prelude.maude` de la instalación. En concreto, se dispone del tipo `Bool` definido en el módulo `BOOL`, el tipo `Nat` definido en el módulo `NAT`, el tipo `Int` en el módulo `INT`, el tipo `Float` en el módulo `FLOAT`, y los tipo `Char` y `Stirng` en el módulo `STRING`. Para hacer uso de alguno de esos tipos, sus operadores, deberemos importar el módulo donde se encuentran con una de las palabras reservadas `including`, `protecting` o `extending`. Por ejemplo, el módulo `FACT` para factorial mostrado anteriormente importa el módulo `INT` de los números enteros.

2.2.3. Declaración obligatoria de variables

Es obligatorio declarar el tipo de las variables antes de ser usadas en el programa, p.ej. la siguiente declaración de variables

```
        var N : Nat .
        var NL : NatList .
        var NS : NatSet .
```

o añadirles el tipo directamente a las variables cuando vamos a usarlas, p.ej. "`X:Nat + Y:Nat`".

2.2.4. Declaraciones de Tipos (Sorts), Símbolos Constructores y Variables

Una declaración de tipo tiene la forma

`sort T .`

e introduce un nuevo tipo de datos `T`. Si se desea introducir varios tipos a la vez, se escribe

`sorts` T_1 `...` T_n `->T .`

Después se define los constructores que formarán los datos asociados a ese tipo de la forma

`op C :` T_1 T_2 `...` T_n `->T .`

donde T_1, T_2, ..., T_n son los tipos de los parámetros de ese símbolo. También se puede escribir

`ops` C_1 `...` C_n `:` T_1 T_2 `...` T_n `->T .`

y denota que todos los símbolos C_1, ..., C_n tienen el mismo tipo. Por ejemplo, las declaraciones de tipo:

```
  sort Bool .
  ops true false : -> Bool .
  sort NatList .
  op nil : -> NatList .
  op _:_ : Nat NatList -> NatList ..
```

introducen el tipo `Bool` con dos constantes `true` y `false`, y el tipo `Natlist` (listas cuyos elementos son naturales, es decir, de tipo `Nat`). Hay que tener en cuenta que **Maude** no soporta tipos de datos paramétricos, como **Haskell**, por lo tanto no se puden definir listas paramétricas sino específicas para cada tipo, como en el caso de `NatList`. Sin embargo, es interesante fijarse en la forma de definir el operador binario infijo de construcción de una lista, "`_:_`", donde se indica que el primer argumento debe aparecer antes de los dos puntos mientras que el segundo detrás de los dos puntos. Una lista de enteros se podrá definir por lo tanto en notación infija como `0 : (1 : (2 : nil))` en vez de la notación prefija `:(0,:(1,:(2,nil)))` simplemente indicando que el símbolo a utilizar es "`_:_`". Esto es muy práctico y versátil ya que simplemente se debe indicar con un "`_`"dónde va a aparecer el argumento, p.ej., se pueden definir símbolos tan versátiles como

```
op if_then_else_fi : Bool Exp Exp -> Exp
op for(_;_;_) {_} : Nat Bool Bool Exp -> Exp
```

En concreto en el ejemplo `VENDING_MACHINE` tenemos un símbolo

`op __ : State State ->State`

que denota que el carácter "vacío" es un símbolo válido para concatenar estados. Y en el ejemplo `FACT` tenemos

`op _! : Int ->Int .`

que denota el símbolo factorial en notación postfija.

2.2.5. Tipos de datos ordenados y sobrecarga de operadores

En **Maude** se pueden crear tipos de datos ordenados o divididos en jerarquías. Por ejemplo, podemos indicar que los números naturales se dividen en números naturales positivos y el cero usando la palabra reservada `subsort` de la siguiente forma

```
sorts Nat Zero NzNat .
subsort Zero < Nat .
subsort NzNat < Nat .
op 0 : -> Zero .
op s : Nat -> NzNat .
```

De esta forma, la expresión `s(0)` es de tipo `NzNat` y a la vez es de tipo `Nat`, mientras que no es de tipo `Zero`. E igualmente, la expresión `0` es de tipo `Zero` y `Nat`, pero no es de tipo `NzNat`.

Otra característica interesante del sistema de tipos de **Maude** es la sobrecarga de operadores. Por ejemplo, se puede reutilizar el símbolo `0` en el tipo de datos `Binary` sin ningún problema

```
sorts Nat Zero NzNat .
subsort Zero < Nat .
subsort NzNat < Nat .
op 0 : -> Zero .
op s : Nat -> NzNat .
sort Binary .
op 0 : -> Binary .
op 1 : -> Binary .
```

En este caso, pueden surgir ambigüedades sobre algún término que se resuelven especificando el tipo detrás del término, por ejemplo `(0).Zero` ó `(0).Binary`. El sistema informará sólo de ambigüedades que no pueda resolver por su cuenta.

La unión de la sobrecarga de símbolos y los tipos ordenados le confiere una gran flexibilidad al lenguaje. Por ejemplo, se puede redefinir el anterior tipo de datos de lista de números naturales de la siguiente forma, donde `ENatList` denota lista vacía (es decir `nil`) y `NeNatList` denota lista no vacía de elementos

```
sorts NatList ENatList NeNatList .
subsort ENatList < NatList .
op nil : -> ENatList .
subsort NeNatList < NatList .
op _:_ : Nat NeNatList -> NeNatList .
op _:_ : Nat ENatList -> NeNatList .
```

2.2.6. Propiedades avanzadas (o algebráicas) de los símbolos

El lenguaje **Maude** incorpora la posibilidad de especificar símbolos con propiedades algebraicas extra como asociatividad, conmutatividad, elemento neutro, etc. que facilitan mucho la creación de programas. Por ejemplo, se puede redefinir el tipo de datos lista de números naturales de la siguiente forma

```
sorts NatList ENatList NeNatList .
subsort ENatList < NatList .
op nil : -> ENatList .
subsort Nat < NeNatList < NatList .
op _:_ : NatList NatList -> NeNatList [assoc] .
```

donde _:_ es un símbolo asociativo, es decir, no son necesarios los paréntesis para separar los términos. Nótese que los dos argumentos del símbolo _:_ tienen que ser del mismo tipo para poder indicar que el símbolo es asociativo. Ahora **Maude** entiende que las siguiente expresiones significan exactamente lo mismo

```
s(0) : s(s(0)) : nil
s(0) : (s(s(0)) : nil)
(s(0) : s(s(0))) : nil
```

Otra posibilidad es añadir un elemento neutro al operador asociativo:

```
sorts NatList .
subsort Nat < NatList .
op nil : -> NatList .
op _:_ : NatList NatList -> NatList [assoc id: nil] .
```

donde en este momento _:_ es un símbolo asociativo y el término `nil` es el elemento neutro del tipo de datos, que por lo tanto se puede eliminar salvo cuando aparece sólo. Ahora **Maude** entiende que las siguiente expresiones significan exactamente lo mismo

```
s(0) : s(s(0)) : nil
s(0) : s(s(0))
nil : s(0) : nil : s(s(0)) : nil
```

También se puede añadir la propiedad de conmutatividad a la lista, creando el tipo de datos multiconjunto

```
sorts NatMultiSet .
subsort Nat < NatMultiSet .
op nil : -> NatMultiSet .
op _:_ : NatMultiSet NatMultiSet -> NatMultiSet [assoc comm id: nil] .
```

donde la propiedad de conmutatividad indica que se puede intercambiar el orden de los elementos. Ahora **Maude** entiende que las siguientes expresiones significan exactamente lo mismo

```
0 : s(0) : s(s(0)) : s(0)
0 : s(0) : s(0) : s(s(0)) : nil
s(0) : 0 : s(s(0)) : s(0) : nil
nil : s(0) : nil : s(s(0)) : nil : s(0) : nil : 0 : nil
```

Finalmente, se puede añadir la propiedad que no pueden haber elementos repetidos, convirtiendo el multiconjunto en un conjunto

```
sorts NatSet .
subsort Nat < NatSet .
op nil : -> NatSet .
op _:_ : NatSet NatSet -> NatSet [assoc comm id: nil] .
eq X:Nat : X:Nat = X:Nat .
```

donde la ecuación, encabezada por la palabra `eq`, elimina aquellas ocurrencias repetidas de un término. Ahora **Maude** entiende que las siguiente expresiones significan exactamente lo mismo

```
0 : s(0) : s(s(0))
0 : s(0) : s(0) : s(0) : s(0) : s(s(0)) : nil
s(0) : 0 : s(s(0)) : s(0) : nil
nil : s(0) : nil : s(s(0)) : nil : s(0) : nil : 0 : nil
```

2.2.7. Declaración de Funciones

Aquellos operadores o símbolos que dispongan de reglas o ecuaciones que los definan son denominados *funciones* mientras que los que no dispongan de reglas o ecuaciones son denominados *constructores*. Las reglas de una función se definen con el operador reservado "`rl_=>_.`" y las ecuaciones con el operador reservado "`eq_=_.`". Nótese que es obligatorio en **Maude** declarar el tipo de todas las funciones, tipo de todas las variables, etc. En otros lenguajes funcionales, como **Haskell**, esto no es necesario aunque se recomienda. En particular, esto puede ayudar a detectar fácilmente errores en el programa, cuando se definen funciones que no se ajustan al tipo declarado.

Respecto a las reglas/ecuaciones que definen las funciones, éstas pueden ser de la forma

`rl f(`t_1`, ..., `t_n` =>e .`
`eq f(`t_1`, ..., `t_n` = e .`

donde t_1`, ..., `t_n y `e` son términos. Las ecuaciones pueden etiquetarse con la palabra reservada `owise` (otherwise) y en ese caso se indica que sólo se aplicará si ninguna otra ecuación para símbolo es aplicable. La palabra `owise` sólo se puede aplicar a una ecuación, nunca a una regla ya que tienen un significado indeterminista. Por ejemplo, se puede dar el siguiente módulo funcional

```
fmod FACT is
protecting INT .
op _! : Int -> Int .
var N : Int .
eq 0 ! = 1 .
eq N ! = (N - 1)! * N [owise] .
endfm
```

Las funciones se pueden definir también mediante reglas/ecuaciones condicionales

`crl f(`t_1`, ..., `t_n` =>e if c .`
`ceq f(`t_1`, ..., `t_n` = e if c .`

donde la condición `c` es un conjunto de emparejamientos de la forma `t := t'` separados por el operador `/\`. Un emparejamiento `t := t'` indica que el término `t'` debe tener la forma del término `t`, instanciando las variables de `t` si es necesario, ya que las variables de `t` pueden ser usadas en la expresión `e` de la regla para extraer información de `t'`. Las ecuaciones condicionales sólo pueden aplicarse, si la condición tiene éxito. También es posible definir una ecuación condicional en la que las guardas sean expresiones de tipo `Bool` en vez de `t := t'`; en ese caso se interpretan como `true := t`. Por ejemplo, podemos escribir la anterior función factorial de la siguiente forma

```
  ceq N ! = 1 if N == 0 .
  ceq N ! = (N - 1)! * N if N =/= 0 .
```

donde la igualdad '`==`' se evalúa a `true` si ambas expresiones son iguales y '`=/=`' se evalúa a `true` si ambas expresiones son distintas. Nótese que en este caso, se puede usar también el operador condicional `if_then_else_fi`.

```
  eq N ! = if N == 0 then 1 else (N - 1)! * N fi .
```

Además, debido a los tipos de datos ordenados, se permiten expresiones lógicas de la forma `t :: T` que se evalúan a `true` si la expresión `t` es del tipo `T`. Por ejemplo, esto puede ser útil en ecuaciones condicionales como

```
op emptyList : NatList -> Bool .
eq emptyList(NL) = NL :: ENatList .
```

donde se dice que una lista `NL` está vacía, es decir `emptyList` retorna `true`, si `NL` es del tipo `ENatlist`.

2.2.8. Búsqueda eficiente de elementos en listas y conjuntos

Una ventaja de disponer de listas, multiconjuntos y conjuntos es que determinadas operaciones de búsqueda y emparejamiento de patrones resultan mucho más rápidas. De hecho, más rápidas que en otros lenguajes declarativos como **Prolog** o **Haskell**. Por ejemplo, la pertenencia de un elemento a una lista se realiza de forma secuencial en muchos lenguajes declarativos

```
sort NatList .
op nil : -> NatList .
op _:_ : Nat NatList -> NatList .
op _in_ : Nat NatList -> Bool .
eq N:Nat in nil
 = false .
eq N:Nat in (X:Nat : XS:NatList)
 = (N:Nat == X:Nat) or-else (N:Nat in XS:NatList) .
```

Sin embargo, cuando disponemos de un operador asociativo con un elemento neutro, esta operación se hace de forma mucho más elegante y eficiente como sigue a continuación:

```
sort NatList .
subsort Nat < NatList .
op nil : -> NatList .
op _:_ : NatList NatList -> NatList [assoc id: nil] .
op _in_ : Nat NatList -> Bool .
eq N:Nat in (L1:NatList : N:Nat : L2:NatList)
 = true .
eq N:Nat in L:NatList
 = false [owise] .
```

La expresión ’`1 in 1 : 2 : 3`’ se puede ver como ’`1 in nil : 1 : 2 : 3`’ gracias a la propiedad del elemento neutro, donde ’`L1:NatList`’ se emparejaría con ’`nil`’, ’`N:Nat`’ con ’`1`’ y ’`L2:NatList`’ con ’`"2 : 3`’; ocurre algo parecido con ’`2 in 1 : 2 : 3`’ y ’`3 in 1 : 2 : 3`’. Esto tiene la ventaja de que es el propio sistema el que decide la mejor técnica de búsqueda y así no está restringido al mecanismo usado por el programador. Además, en el caso de un conjunto (o multiconjunto) es aún más simple gracias a la conmutatividad.

```
sort NatSet .
subsort Nat < NatSet .
op empty : -> NatSet .
op _:_ : NatSet NatSet -> NatSet [assoc comm id: nil] .
op _in_ : Nat NatSet -> Bool .
eq N:Nat in (N:Nat : L:NatSet)
 = true .
eq N:Nat in L:NatSet
 = false [owise] .
```

2.2.9. Ejecución de programas en `Maude`

El sistema `Maude`, dispone, entre otros, de los siguientes comandos:

- `load` $<$ `name` $>$ `.` Lee y carga los distintos módulos almacenados en el archivo $<$ `name` $>$. Los archivos `Maude` suelen tener la extensión *.maude* aunque son ficheros de texto plano.

- `show modules .` Muestra los módulos cargados actualmente en el sistema.

- `show module` $<$ `module` $>$ `.` Muestra el módulo $<$ `module` $>$ en pantalla.

- `select` $<$ `module` $>$ `.` Selecciona un nuevo módulo para ser el módulo actual de ejecución de expresiones.

- `rewrite in` $<$ `module` $>$ `:` $<$ `expresion` $>$ `.` Evalúa la expresión $<$ `expresion` $>$ con respecto al módulo $<$ module $>$.

- `cd` $<$ `dir` $>$ Permite cambiar de directorio (no se usa con el punto final).

- `ls` Ejecuta el comando **UNIX** *ls* y muestra todos los ficheros en el directorio actual (no lleva punto al final).

- `quit` Salir del sistema (no lleva punto final).

La semántica operacional de **Maude** se basa en la lógica de reescritura y básicamente consiste en reescribir la expresión de entrada usando las reglas y ecuaciones del programa hasta que no haya más posibilidad. **Maude** utiliza una estrategia de ejecución impaciente, como ocurre en los lenguajes de programación imperativos como **C** o **Pascal** y en algunos lenguajes funcionales como **ML**, en vez de una estrategia de ejecución perezosa, como en el lenguaje funcional **Haskell**. Por ejemplo, dada la función _! mostrada anteriormente (listado 2.2.7), y asumiendo que está almacenada en el fichero *fact.maude* se puede escribir

```
$ maude
           \||||||||||||||||||/
         --- Welcome to Maude ---
           /||||||||||||||||||\
    Maude 2.1.1 built: Jun 15 2004 15:23:40
     Copyright 1997-2004 SRI International
         Wed Oct 14 19:53:21 2005
Maude> load fact.maude
Maude> rewrite 4 ! .
       rewrite in FACT : 4 ! .
       rewrites: 23 in 0ms cpu (0ms real) (~ rewrites/second)
       result NzNat: 24
Maude> quit
```

Figura 2.1: Posible ejecución en `Maude`

2.2.10. Metanivel en Maude

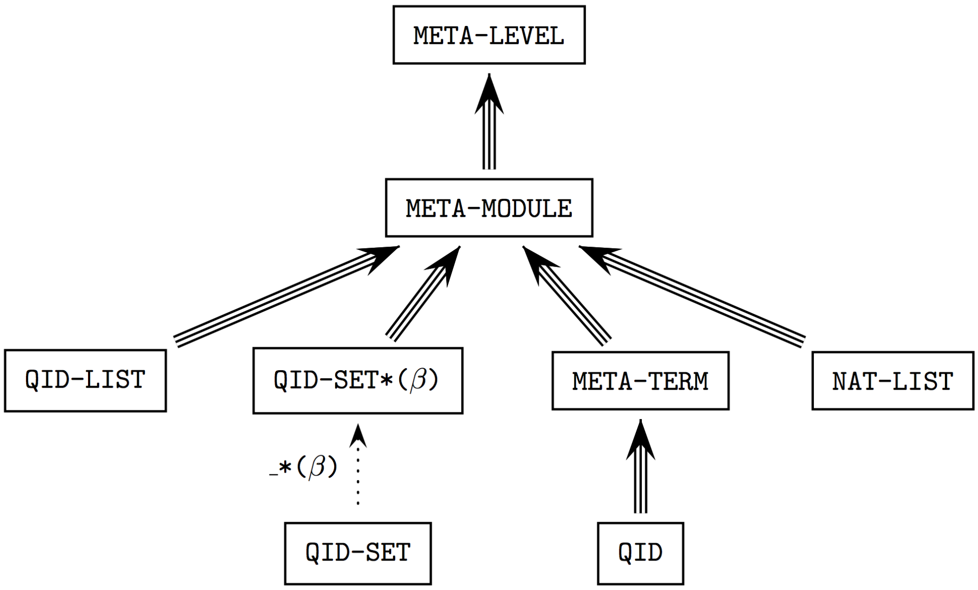

Figura 2.2: Esquema de los módulos parte del metanivel en Maude

En **Maude**, la clave de la funcionalidad de una teoría universal U ha sido implementada de forma eficiente el módulo META-LEVEL el cual está dividido de forma elegante en varios módulos (Figura 2.2) donde cada uno se encarga de un aspecto diferente. Si se describe el metanivel de **Maude** se puede decir que:

- Los términos **Maude** están reificados como elementos del tipo de datos Term en el módulo META-TERM.
- Los módulos **Maude** estan reificados como términos en un tipo de datos Module en el módulo META-MODULE.
- Existen operaciones upModule, upTerm, downTerm, entre otras, que permiten navegar entre los distintos niveles de reflexión.
- El proceso de reducir un término a una forma canónica usando el comando reduce de **Maude** esta meta-representado por la función incluida metaReduce.

- Los procesos de reescritura de un término en un módulo de sistema usando los comandos de **Maude** `rewrite` y `frewrite` están meta-representadas por las funciones incluidas `metaRewrite` y `metaFrewrite` respectivamente.
- El proceso de aplicar una regla de un módulo de sistema **en la cima** de un término está meta-representado por la función incluida `metaApply`.
- El proceso de aplicar una regla de un módulo de sistema en cualquier posición de un término está meta-representado por la función incluida `metaXapply`.
- El proceso de emparejamiento (*matching*) de dos términos está reificado por las funciones incluidas `metaMatch` y `metaXmatch`.
- El proceso de buscar un término que satisface algunas condiciones empezando en un término inicial está reificado por las funciones incluidas `metaSearch` y `metaSearchPath`.
- Las funciones de analizar (*parsing*) y presentación limpia (*pretty-printing*) de un término en un módulo, así como las operaciones de comparar tipos (*sorts*) sobre el subtipo ordenado de una signatura, también están meta-representadas por las correspondientes funciones incluidas.

Representación de términos

La definición de los términos en el módulo `META-LEVEL` de **Maude** se especifica de la siguiente forma

```
sorts Constant Variable Term .
subsorts Constant Variable < Qid Term .

op <Qids> : -> Constant [special (...)] .
op <Qids> : -> Variable [special (...)] .

sort TermList .
subsort Term < TermList .
op _,_ : TermList TermList -> TermList
    [ctor assoc gather (e E) prec 120] .

op _[_] : Qid TermList -> Term [ctor] .
```

A continuación se muestran un ejemplo sobre la diferencia de representación de un mismo término según su nivel; estos términos se basan en la definición del módulo `VENDING-MACHINE` (listado 2.2.1).

- Término común: `c (q M:State)`
- Meta-representación de un término:
  `'__['c.Item, '__['q.Coin, 'M:State]]`
- Meta-meta-representación de un término:

```
'_`[_`]['`__.Qid,
        '_`,_[''c.Item.Constant,
              '_`[_`]['`__.Qid,
                     '_`,_[''q.Coin.Constant,
                           ''M:State.Variable]]]]
```

Representación de módulos

Los módulos están definidos en **Maude** en el módulo `META-LEVEL` con la siguiente especificación

```
sorts FModule SModule FTheory STheory Module .
subsorts FModule < SModule < Module .
subsorts FTheory < STheory < Module .
sort Header .
subsort Qid < Header .
op _{_}  : Qid ParameterDeclList -> Header [ctor] .
op fmod_is_sorts_._____endfm : Header ImportList SortSet
 SubsortDeclSet OpDeclSet MembAxSet EquationSet -> FModule
 [ctor gather (& & & & & & &)] .
op mod_is_sorts_.______endm : Header ImportList SortSet
 SubsortDeclSet OpDeclSet MembAxSet EquationSet RuleSet
 -> SModule [ctor gather (& & & & & & & &)] .
op fth_is_sorts_._____endfth : Qid ImportList SortSet SubsortDeclSet
 OpDeclSet MembAxSet EquationSet -> FTheory
 [ctor gather (& & & & & & &)] .
op th_is_sorts_.______endth : Qid ImportList SortSet SubsortDeclSet
 OpDeclSet MembAxSet EquationSet RuleSet -> STheory
 [ctor gather (& & & & & & & &)] .
```

A continuación se muestra cómo se representarían diferentes módulos según su nivel (se toma como ejemplo la signatura del módulo `VENDING-MACHINE`).

En el primer ejemplo la representación común en **Maude** sería

```
fmod VENDING-MACHINE-SIGNATURE is
  sorts Coin Item State .
  subsorts Coin Item < State .
  op __ : State State -> State [assoc comm] .
  op $ : -> Coin [format (r! o)] .
  op q : -> Coin [format (r! o)] .
  op a : -> Item [format (b! o)] .
  op c : -> Item [format (b! o)] .
endfm
```

Por contra la meta-representación del mismo módulo sería

```
fmod 'VENDING-MACHINE-SIGNATURE is
  nil
  sorts 'Coin ; 'Item ; 'State .
  subsort 'Coin < 'State .
  subsort 'Item < 'State .
  op '__ : 'State 'State -> 'State [assoc comm] .
  op 'a : nil -> 'Item [format('b! 'o)] .
  op 'c : nil -> 'Item [format('b! 'o)] .
  op '$ : nil -> 'Coin [format('r! 'o)] .
  op 'q : nil -> 'Coin [format('r! 'o)] .
  none
none endfm
```

El siguiente ejemplo define las reglas del módulo anterior y, además asociada etiquetas a las reglas. La representación común sería

```
mod VENDING-MACHINE is
  including VENDING-MACHINE-SIGNATURE .
  var M : State .
  rl [add-q] : M => M q .
  rl [add-$] : M => M $ .
  rl [buy-c] : $ => c .
  rl [buy-a] : $ => a q .
  rl [change] : q q q q => $ .
endm
```

La meta-representación sería

```
mod 'VENDING-MACHINE is
 including 'VENDING-MACHINE-SIGNATURE .
 sorts none .
 none
 none
 none
 none
 rl 'M:State => '__['M:State, 'q.Coin] [label('add-q)] .
 rl 'M:State => '__['M:State, '$.Coin] [label('add-$)] .
 rl '$.Coin => 'c.Item [label('buy-c)] .
 rl '$.Coin => '__['a.Item, 'q.Coin] [label('buy-a)] .
 rl '__['q.Coin,'q.Coin,'q.Coin,'q.Coin]
```

```
 => '$.Coin [label('change)] .
endm
```

Ejemplos de cambios de nivel representación

Como se ha comentado con anterioridad, existen distintas funciones auxiliares que permiten mover términos, módulos, tipos, étc. entre los distintos niveles de representación. Su especificación es

```
op upModule : Qid Bool ~> Module [special (...)] .
op upSorts : Qid Bool ~> SortSet [special (...)] .
op upSubsortDecls : Qid Bool ~> SubsortDeclSet [special (...)] .
op upOpDecls : Qid Bool ~> OpDeclSet [special (...)] .
op upMbs : Qid Bool ~> MembAxSet [special (...)] .
op upEqs : Qid Bool ~> EquationSet [special (...)] .
op upRls : Qid Bool ~> RuleSet [special (...)] .
```

Como se habrá podido comprobar, estas funciones son parciales (pueden dar un error) donde:

- El primer argumento se espera que sea un nombre de un módulo.
- El segundo argumento es `Bool`, indicando si se está interesado en importar además el módulo o no.

En el siguiente ejemplo se obtiene la meta-representación de las ecuaciones del módulo `VENDING-MACHINE` y se indica en el segundo argumento `true` para importar el módulo.

```
Maude> reduce in META-LEVEL : upEqs('VENDING-MACHINE, true) .
 result EquationSet:
 eq '_and_['true.Bool, 'A:Bool] = 'A:Bool [none] .
 eq '_and_['A:Bool, 'A:Bool] = 'A:Bool [none] .
 eq '_and_['A:Bool, '_xor_['B:Bool, 'C:Bool]]
  = '_xor_['_and_['A:Bool, 'B:Bool], '_and_['A:Bool, 'C:Bool]]
   [none] .
 eq '_and_['false.Bool, 'A:Bool] = 'false.Bool [none] .
 eq '_or_['A:Bool,'B:Bool]
  = '_xor_['_and_['A:Bool, 'B:Bool],'_xor_['A:Bool, 'B:Bool]]
   [none] .
 eq '_xor_['A:Bool, 'A:Bool] = 'false.Bool [none] .
 eq '_xor_['false.Bool, 'A:Bool] = 'A:Bool [none] .
 eq 'not_['A:Bool] = '_xor_['true.Bool, 'A:Bool] [none] .
 eq '_implies_['A:Bool, 'B:Bool]
  = 'not_['_xor_['A:Bool, '_and_['A:Bool, 'B:Bool]]] [none] .
```

A continuación se realiza la misma llamada pero sin importar el módulo

```
Maude> reduce in META-LEVEL : upEqs('VENDING-MACHINE, false) .
 result EquationSet: (none).EquationSet
```

En el siguiente ejemplo se muestra cómo meta-representación de las reglas del mismo módulo

```
Maude> reduce in META-LEVEL : upRls('VENDING-MACHINE, true) .
result RuleSet:
  rl '$.Coin => 'c.Item [label('buy-c)] .
  rl '$.Coin => '__['q.Coin,'a.Item] [label('buy-a)] .
  rl 'M:State => '__['$.Coin,'M:State] [label('add-$)] .
  rl 'M:State => '__['q.Coin,'M:State] [label('add-q)] .
  rl '__['q.Coin,'q.Coin,'q.Coin,'q.Coin] => '$.Coin
   [label('change)] .
```

Finalmente se muestra un ejemplo de navegación de niveles en términos. Si se dispone de la definición del módulo

```
fmod UP-DOWN-TEST is protecting META-LEVEL .
 sort Foo .
 ops a b c d : -> Foo .
 op f : Foo Foo -> Foo .
 op error : -> [Foo] .
 eq c = d .
endfm
```

Si se llama a la función `upTerm` para mostrar la meta-representación de un término `f(a, f(b,c))`.

```
Maude> reduce in UP-DOWN-TEST : upTerm(f(a, f(b, c))) .
 result GroundTerm: 'f['a.Foo,'f['b.Foo,'d.Foo]]
```

Si se ejecuta la función `downTerm` permite navegar de meta-representación a representación

```
Maude> reduce downTerm('f['a.Foo,'f['b.Foo,'c.Foo]], error) .
 result Foo: f(a, f(b, c))
```

Si se intenta mostrar un metá-término que no está definido en el módulo se genera un error

```
Maude> reduce downTerm('f['a.Foo,'f['b.Foo,'e.Foo]], error) .
```

```
Advisory: could not find a constant e of
  sort Foo in meta-module UP-DOWN-TEST.
  result [Foo]: error
```

2.2.11. Unificación y Estrechamiento en Maude

El prototipo que se detallará en posteriores secciones utiliza íntegramente meta-representaciones de términos y módulos y para realizar ciertas acciones necesita, además de las comentadas anteriormente dos muy importantes son `metaNarrowSearch` y `metaUnify`.

La función `metaNarrowSearch`[1] es la meta-representación que se usa para realizar análisis de alcanzabilidad basados en *narrowing*. Esta función está definido (junto con su infraestructura necesaria) en el módulo `META-NARROWING-SEARCH` y se define de la siguiente forma.

```
op metaNarrowSearch :
   Module Term Term Substitution Qid Bound Bound -> ResultTripleSet .
```

donde

- `Module` es la meta-representación del módulo donde está definida la teoría.
- `Term` es la meta-representación del término inicial.
- `Term` es la meta-representación del término final.
- `Substitution` (si está dado, normalmente es `none`) cualquier sustitución computada debe ser una instancia de la pasada por argumentos.
- `Qid` meta-representa la búsqueda adecuada, en número de pasos (normalmente `*`, es decir, indeterminado).
- `Bound` indica el número máximo de soluciones que se desean (profundidad del árbol de *narrowing*).
- `Bound` indica el número de soluciones computadas (normalmente `unbounded`).

[1] `http://maude.cs.uiuc.edu/maude2-manual/html/maude-manualch16.html`

El tipo de datos `ResultTripleSet` representa un conjunto formado por

- El término resultante calculado.
- El tipo (*sort*) del término.
- Una lista de sustituciones para cada variable.

Por ejemplo

```
result ResultTripleSet :
 {'s_^1['0.Zero], 'Nat,
  '#1:Nat <- '0.Zero;
  '#2:Nat <- '0.Zero}
  |
 {'s_^2['0.Zero], 'Nat,
  '#3:Nat <- '0.Zero}
```

Para la ejecución de los ejemplos se usará el módulo del listado 2.2.11. (Nótese que la definición del módulo está envuelva por paréntesis, esto es necesario)

```
(mod NARROWING-VENDING-MACHINE is
 sorts Coin Item Marking Money State .
 subsort Coin < Money .
 op __ : Money Money -> Money [assoc comm] .
 subsort Money Item < Marking .
 op __ : Marking Marking -> Marking [assoc comm] .
 op <_> : Marking -> State .
 op $ : -> Coin [format (r! o)] .
 op q : -> Coin [format (r! o)] .
 op a : -> Item [format (b! o)] .
 op c : -> Item [format (b! o)] .

 var M : Marking .
 rl [buy-c] : < $ > => < c > .
 rl [buy-c] : < M $ > => < M c > .
 rl [buy-a] : < $ > => < a q > .
 rl [buy-a] : < M $ > => < M a q > .
 rl [change]: < q q q q > => < $ > .
 rl [change]: < M q q q q > => < M $ > .
endm)
```

Si se quisiera ejecutar la función `metaNarrowSearch`, un posible comando sobre el anterior módulo sería

```
Maude> (red in META-NARROWING-SEARCH :
     metaNarrowSearch(upModule(NARROWING-VENDING-MACHINE),
       '<_>['M:Money],
```

```
        '<_>['__['a.Item, 'c.Item]],
        none, '*, 4, unbounded) .)
  result ResultTripleSet :
    {'<_>['__['a.Item,'c.Item]], 'State,
      '#1:Marking <- '__['q.Coin, 'q.Coin, 'q.Coin];
      '#3:Money <- '__['q.Coin, 'q.Coin, 'q.Coin];
      '#4:Marking <- 'a.Item ;
      '#6:Marking <- 'a.Item ;
      'M:Money <- '__['$.Coin, '__['q.Coin, 'q.Coin, 'q.Coin]]}
  | {'<_>['__['a.Item, 'c.Item]], 'State,
      '#1:Marking <- '__['q.Coin, 'q.Coin, 'q.Coin];
      '#3:Money <- '__['q.Coin, 'q.Coin, 'q.Coin];
      '#4:Marking <- '__['q.Coin, 'q.Coin, 'q.Coin];
      '#6:Money <- '__['q.Coin, 'q.Coin, 'q.Coin];
      '#7:Marking <- 'a.Item ;
      '#9:Marking <- 'a.Item ;
      'M:Money <- '__['q.Coin, 'q.Coin, 'q.Coin, 'q.Coin,
                       '__['q.Coin, 'q.Coin, 'q.Coin]]}
```

La función `metaUnify`[2] es la meta-representación de la unificación. Esto es importante por dos razones:

- Muchas de las aplicaciones de razonamiento formal de unificación requieren acceso a funciones de unificación al metanivel. Por ejemplo, la computación de *pares críticos* para determinar si un módulo funcional es localmente confluente. Esto se realizará correctamente mediante una función que coja la meta-representación de dicho módulo funcional como datos, y entonces llame a las funciones de unificación como parte de sus computaciones de *pares críticos*.

- El algoritmo de unificación es dependiente de la teoría, así que a partir de la combinación de cada signatura con unos axiomas generan algoritmos de unificación *order-sorted* diferentes. Gracias a la función `metaUnify`, que recibe la meta-representación del módulo que se desee, se puede realizar la unificación de forma correcta.

La definición de la función `metaUnify` es

```
op metaUnify :
 Module UnificationProblem Nat Nat ~> UnificationPair?
 special (...)] .
```

donde

[2] `http://maude.cs.uiuc.edu/maude2-manual/html/maude-manualch12.html`

- `Module` es la meta-representación del módulo donde está definida la teoría.

- `UnificationProblem` Es una lista de pares de la forma `T:Term =? T:Term`.

- `Nat` Indica el identificador en el que se deben empezar a crear variables fescas (en caso que se necesiten).

- `Nat` Se usa para seleccionar el resultado que quiere (empezando desde el 0)

El tipo de datos `UnificationProblem` se define de la siguiente manera donde cada componente es un `UnificationPair` formado por `T:Term =? T:Term`. En cuanto al resultado de dicha función (`UnificationPair?`) está formado por una lista de `Sustitution, Nat`.

```
sorts UnificandPair UnificationProblem .
subsort UnificandPair < UnificationProblem .
op _=?_ : Term Term -> UnificandPair [ctor prec 71] .
op _/\_ : UnificationProblem UnificationProblem -> UnificationProblem
 [ctor assoc comm prec 73] .
subsort UnificationPair < UnificationPair? .
subsort UnificationTriple < UnificationTriple? .
op {_,_} : Substitution Nat -> UnificationPair [ctor] .
op {_,_,_} : Substitution Substitution Nat -> UnificationTriple
 [ctor] .
op noUnifier : -> UnificationPair? [ctor] .
```

Un ejemplo de uso de `metaUnify`

```
Maude> reduce in META-LEVEL :
   metaUnify(upModule('UNIFICATION-EX1, false),
   'f['X:Nat, 'Y:NzNat] =? 'f['Z:NzNat, 'U:Nat] /\
   'V:NzNat =? 'f['X:Nat, 'U:Nat], 0, 0) .
  result UnificationPair:
    {'U:Nat <- '#1:NzNat ;
    'V:NzNat <- 'f['#2:NzNat, '#1:NzNat] ;
    'X:Nat <- '#2:NzNat ;
    'Y:NzNat <- '#1:NzNat ;
    'Z:NzNat <- '#2:NzNat, 2}
```

2.3. CVC3

CVC3[3] es un solver (testeador/provador) automático de Teorías Módulo Satisfacibilidad (SMT Solver). Puede usarse para comprobar la validez (o, dualmente, la satisfacibilidad) de fórmulas de primer orden en un número grande de teorías lógicas y combinaciones de éstas.

CVC3 es el último descendiente de una serie de testers SMT originados en la Universidad de Stanford con el sistema SVC. En particular se ha generado a partir del código base de CVC Lite[4] (su más reciente predecesor, discontinuado en la actualidad).

CVC3 trata con una versión de lógica de primer orden con tipos polimórficos y tiene una gran variedad de características como:

- Algunas teorías base incluidas como aritmética lineal racional y entera, arrays, tuplas, tipos de datos inductivos, etc.
- Soporte para cuantificadores.
- Interfaz interactiva basada en texto.
- Una API creada en C y C++ para ser incluida en otros sistemas.
- Generación de pruebas y modelos.
- Subtipado de predicados.
- No tiene límites de uso ya sea para investigación o fines comerciales.

A continuación se detallan algunas características y tipos de **CVC3**, se puede encontrar más información en la documentación en la web

`http://www.cs.nyu.edu/acsys/cvc3/doc/user_doc.html`

[3] `http://www.cs.nyu.edu/acsys/cvc3/`
[4] `http://www.cs.nyu.edu/acsys/cvcl/`

2.3.1. Ejecutando CVC3 desde línea de comandos

Asumiendo que se ha instalado correctamente **CVC3** (apartado instalación[5] del manual) existe un ejecutable denominado `cvc3`. Este ejecutable lee la entrada (una secuencia de comandos) desde la entrada estándar y escribe los resultados en la salida estándar. Los errores y otros mensajes (salidas de depuración) se redirigen a la salida de error estándar.

Típicamente, la entrada de `cvc3` se guarda en una archivo y se redirige al ejecutable por ejemplo

```
# Reading from standard input:
  cvc3 < input-file.cvc
# Reading directly from file:
  cvc3 input-file.cvc
```

Nótese que por razones de eficiencia **CVC3** usa búffers de entrada, y la entrada no siempre se procesa inmediatamente después de recibir cada comando. De este modo, si se desea escribir los comandos de forma interactiva y recibir los resultados de forma rápida se debe usar la opción `+interactiva` o también acortado mediante `+int`

```
cvc3 +int
```

Si se desea obtener la ayuda de `cvc3` se puede usar el comando `-h`.

El *front-end* de línea de comandos de **CVC3** soporta dos lenguajes de entrada:

- El propio lenguaje de presentación **CVC3** cuya sintáxis estaba inicialmente inspirada por los sistemas **PVS**[6] (Prototype Verification System) y **SAL** y es casi idéntico al lenguaje de entrada de **CVC** y **CVC Lite**, los predecesores de **CVC3**.

- El lenguaje estándar promovido por la iniciativa **SMT-LIB**[7] para benchmarks **SMT-LIB**.

[5] http://www.cs.nyu.edu/acsys/cvc3/doc/INSTALL.html
[6] http://en.wikipedia.org/wiki/Prototype_Verification_System
[7] http://www.smt-lib.org/

A continuación se describen otras carácterísticas de **CVC3** enfocándose en el primero de los lenguajes.

2.3.2. Sistema de tipos de CVC3

El sistema de tipos de **CVC3** incluye una serie de tipos incluidos que pueden ser expandidos por otros definidos por el usuario. Este sistema de tipos consiste en tipos *valuados*, tipos *no valuados* y *subtipos*, todos ellos interpretados como conjuntos. Por conveniencia, algunas veces se identificará la interpretación de un tipo con el propio tipo.

Los tipos *valuados* pueden ser tipos *atómicos* y tipos *estructurados*. Los tipos *atómicos* son `REAL`, `BITVECTOR(n)` para todo `n >0`, así como los tipos definidos por el usuario (llamados también tipos *no interpretados*). Los tipos *estructurados*, son `array`, `tuple`, y `record`, así como los tipos estilo ML definidos por el usuario (tipos inductivos).

Los tipos *no valuados* consisten en el tipo `BOOLEAN` y los tipos `function`. Los subtipos incluyen el subtipo incluido `INT` o `REAL` y se detallan seguidamente.

Tipo `REAL`

El tipo `REAL` está interpretado como el conjunto de números racionales. El nombre `REAL` está justificado por el hecho que una fórmula **CVC3** es válida en la teoría de números racionales sí y solo sí es válida en la teoría de números reales.

Tipos `Bit Vector`

Para cada numeral positivo `n`, el tipo `BITVECTOR(n)` esta interpretado como el conjunto de todos los vectores de bits de tamaño `n`.

Tipo atómicos definidos por el usuario

Los tipos atómicos definidos por el usuario son cada interpretación como un conjunto de cardinalidad no especificada pero disjunto desde cualquier otro tipo. Ellos son creados por declaraciones como la siguiente

```
% User declarations of atomic types:
MyBrandNewType: TYPE;
Apples, Oranges: TYPE;
```

Tipo `BOOLEAN`

El tipo `BOOLEAN` es, quizá confusamente, el tipo de las fórmulas **CVC3**, no el conjunto de los dos valores Booleanos. El hecho que `BOOLEAN` no es un tipo valor en práctica significa que no es posible por los símbolos de función en **CVC3** tener argumentos de tipos `BOOLEAN`. La razón es que **CVC3** sigue la estructura de dos niveles de la lógica de primer orden clásica que distingue entre fórmulas y términos y permite que los términos aparezcan en fórmulas pero no al revés. (Una excepción es la construcción `IF-THEN-ELSE`). La única diferencia es que, sintácticamente, las fórmulas en **CVC3** son términos de tipo `BOOLEAN`. Una símbolo de función `f` entonces *puede* tener `BOOLEAN` como su tipo de retorno. Pero como ocurría en los predecesores de **CVC3** se diría que `f` es un símbolo de predicado.

CVC3 tiene un tipo que se comporta como tipo Booleano, esto es, un tipo valuado que sólo dos elementos con las operaciones booleananas comunes definidas en ellas.

Tipos `function`

Todos los tipos estructurados son actualmente tipos *familia*. Los tipos función `(->)` son creados por constructores mixfix

```
_ -> _
(_,_) -> _
(_,_,_) -> _
.....
```

cuyos argumentos pueden ser instanciarse por cualquier (sub)tipo valor, con la restricción que el último argumento puede además ser `BOOLEAN`.

```
% Function type declarations

UnaryFunType: TYPE = INT -> REAL;
BinaryFunType: TYPE = (REAL, REAL) -> ARRAY REAL OF REAL;
TernaryFunType: TYPE = (REAL, BITVECTOR(4), INT) -> BOOLEAN;
```

Un tipo `function` de la forma (T_1,..,T_n) `->T` con `n >0` es interpretado como el conjunto de todas las funciones del producto cartesiano donde `T` no es `BOOLEAN`. De otra manera, es interpretado como el conjunto de todas las relaciones sobre T_1 `x ... x` T_n.

El ejemplo anterior además muestra cómo introducir *nombres de tipo*. Un nombre como `UnaryFunType` anterior es una abreviación para el tipo `INT ->REAL` y pueden usarse de la misma forma (intercambiándolos).

En general, cualquier tipo definido por un tipo expresión `E` se puede dar con la declaración

```
name : TYPE = E;
```

Tipos `Array`

Los tipos `Array` están creados por los constructores tipo mixfix `ARRAY_OF_` cuyos argumentos pueden ser instanciados por cualquier tipo valuado.

```
T1 : TYPE;

% Array types:

ArrayType1: TYPE = ARRAY T1 OF REAL;
ArrayType2: TYPE = ARRAY INT OF (ARRAY INT OF REAL);
ArrayType3: TYPE = ARRAY [INT, INT] OF INT;
```

Un tipo array de la forma `ARRAY` T_1 `OF` T_2 se interpreta como el conjunto de todos los mapeos desde T_1 a T_2. La principal diferencia conceptual con el tipo $T_1 - > T_2$ es que los arrays, al contrario de las funciones, son objetos de *primera-clase* del lenguaje: pueden ser argumentos o resultados de funciones. Además, los tipos `array` pueden tener una operación de actualización.

Tipo Tuple

Los tipos Tuple están creados por los constructores tipo mixfix

```
[_]
[_,_]
[_,_,_]
....
```

cuyos argumentos pueden ser instanciados por cualquier tipo valuado.

```
% Tuple declaration

TupleType: TYPE = [ REAL, ArrayType1, [INT, INT] ];
```

Un tipo tuple de la forma [T_1,...,T_2] se interpreta como el producto cartesiano. Nótese que mientras los tipos (T_1,...,T_2) ->T y [T_1 x ... x T_n] ->T son semánticamente equivalentes, ellos son operacionalmente diferentes en **CVC3**. El primero es el tipo de las funciones que coge n argumentos mientras el segundo es el tipo de funciones con 1 argumento de tipo n-tuple.

Tipos Record

Los tipos Record son similares pero más generales que los tipos tuple. Están creados por el constructor tipo de la forma

[# l_1:_, ..., l_n :_ #]

donde n >0 l_1,...,l_n son campos etiqueta y los argumentos pueden ser instanciados por cualquier tipo valuado.

```
% Record declaration

RecordType: TYPE = [# number: INT, value: REAL, info: TupleType #];
```

El orden de los campos en un tipo record es significativo, en otras palabras, permutando el nombre de los campos da un diferente tipo. Nótese que los records son no-recursivos. Por ejemplo, no es posible declarar un tipo record llamado Person conteniendo un campo de tipo Person. Los tipos recursivos se proporcionan en **CVC3** como tipos de datos estilo **ML**.

Tipos de datos inductivos

Los tipos de datos inductivos están creados por declaraciones de la forma

DATATYPE
$type\_name_1$ = $C_{1,1}$ | $C_{1,2}$ | .... | C_{1,m_1}
.
.
.
$type\_name_n$ = $C_{n,1}$ | $C_{n,2}$ | .... | C_{n,m_1}

Cada $C_{i,j}$ es cualquier símbolo de constante o una expresión de la forma

cons(sel_1 : T_1, ..., sel_k : T_k)

donde T_1, ..., T_k son cualquier tipo valor o nombre de tipo para tipos valuados incluyendo cualquier $type\_name_i$. Estas declaraciones introducen para el tipo de datos

- símbolos *constructores* `cons` de tipo (T_1, ..., T_2) ->$type\_name_i$
- símbolos *selectores* sel_i de tipo $type\_name_i$ ->T
- símbolos *tester* `is_cons` de tipo $type\_name_i$ ->BOOLEAN

Otros ejemplos de declaraciones serían

```
% simple enumeration type
% implicitly defined are the testers: is_red, is_yellow and is_blue
% (similarly for the other datatypes)

DATATYPE
  PrimaryColor = red | yellow | blue
END;


% infinite set of pairwise distinct values ...v(-1), v(0), v(1), ...

DATATYPE
  Id = v (id: INT)
END;

% ML-style integer lists
```

```
DATATYPE
  IntList = nil | cons (head: INT, tail: IntList)
END;

% ASTs

DATATYPE
  Term = var (index: INT)
       | apply (arg_1: Term, arg_2: Term)
       | lambda (arg: INT, body: Term)
END;

% Trees

DATATYPE
  Tree = tree (value: REAL, children: TreeList),
  TreeList = nil_tl
           | cons_tl (first_t1: Tree, rest_t1: TreeList)
END;
```

Los símbolos constructor, selector y tester definidos para tipos de datos tienen un ámbito global. Así, por ejemplo, no es posible para dos diferentes usar el mismo nombre para un constructor.

Un tipo de datos es interpretado como una álgebra de términos construida por los símbolos constructores sobre algunos conjuntos generadores. Por ejemplo, el tipo de datos `IntList` es interpretado como el conjunto de todos los términos construidos con `nil` y `cons` sobre enteros.

Es gracias a la definición de esta semántica que **CVC3** permite sólo tipos de datos *inductivos*, esto es, tipos de datos cuyos valores son esencialmente (etiquetados, ordenados) árboles infinitos. Estructuras infinitas tales como flujos infinitos o estructuras cíclicas como listas circulares están excluidas.

Por ejemplo, ninguna de las siguientes declaraciones definen tipos de datos inductivos y son rechazados por **CVC3**

```
DATATYPE
 IntStream = s (first:INT, rest: IntStream)
END;

DATATYPE
 RationalTree = node1 (first_child1: RationalTree)
              | node2 (first_child2: RationalTree, second_child2:
                  RationalTree)
END;

DATATYPE
 T1 =  c1 (s1: T2),
 T2 =  c2 (s2: T1)
END;
```

2.3.3. Comprobación de tipos

En esencia, los términos **CVC3** son estáticamente tipados en el nivel de tipos (opuesto a subtipos) conforme a las reglas normales de la lógica de primer orden parcialmente ordenada (las reglas de tipo para las fórmulas son análogas):

- Cada variable tiene asociado un tipo (no `function`).
- Cada símbolo de constante tiene asociado un tipo (no `function`).
- Cada símbolo de `function` tiene uno o más tipo `function` asociado.
- El tipo de un término consistente en un símbolo de variable o constante es el tipo asociado a ese símbolo de variable o constante.
- El término obtenido al aplicar un símbolo de función `f` a los términos t_1`,...,`t_2) es `T` si `f` tiene tipo T_1`,...,`T_2) `->T` y cada t_i tiene tipo T_i.

Si se intenta introducir un término mal tipado **CVC3** mostrará un error.

La principal diferencia con la lógica parcialmente ordenada estándar es que algunos símbolos incluidos son polimórficos paramétricamente. Por ejemplo, el símbolo de función para extraer el elemento de cualquier array tiene tipo `ARRAY` T_1 `OF` T_2`,` $T_1 -> T_2$) para todos los tipos T_1`,` T_2 que no contienen tipos `function` o predicado.

2.3.4. Términos y fórmulas

Además de las expresiones tipadas, **CVC3** tiene expresiones para términos y fórmulas (por ejemplo el tipo `BOOLEAN`). Estos son términos de primer orden estándar construidos de variables (tipadas), operadores predefinidos especificos para una teoría, símbolos de función libres (definidos por el usuario), y cuantificadores. Las extensiones incluyen un operador `if-then-else`, abstracciones lambda, y declaraciones locales de símbolos. Nótese que estas extensiones se mantienen en el lenguaje de primer orden de CVC3. En particular, las abstracciones lambda están restringidas para coger y devolver sólo términos de tipos valuados. De la misma forma, los cuantificadores pueden sólo cuantificar variables de tipos valuados.

Los símbolos de funciones libres incluyen símbolos *constantes* y símbolos *predicado*, respectivamente los símbolos de función nularios y símbolos de función con un tipo de retorno BOOLEAN. Los símbolos libres están introducidos con declaraciones globales de la forma $f_1, \ldots, f_m$: T; donde m >0, f_i son los nombres de los símbolos y T es su tipo:

```
% integer constants

a, b, c: INT;

% real constants

x,y,z: REAL;

% unary function

f1: REAL -> REAL;

% binary function

f2: (REAL, INT) -> REAL;

% unary function with a tuple argument

f3: [INT, REAL] -> BOOLEAN;

% binary predicate

p: (INT, REAL) -> BOOLEAN;

% Propositional "variables"

P,Q; BOOLEAN;
```

Igual que la declaración de tipos, las declaraciones de símbolos libres tienen un ámbito global y deben ser únicos. En otras palabras, no es posible globalmente declarar un símbolo más de una vez. Esto implica otras cosas como que los símbolos no pueden ser sobrecargados con tipos diferentes.

Al igual que los tipos, un nuevo símbolo libre puede ser definido como el nombre de un término del correspondiente tipo. Con símbolos de constante esto es correcto con una declaración de la forma f : T = t;

```
c: INT;
i: INT = 5 + 3*c;
j: REAL = 3/4;
t: [REAL, INT] = (2/3, -4);
r: [# key: INT, value: REAL #] = (# key := 4, value := (c + 1)/2 #);
f: BOOLEAN = FORALL (x:INT): x <= 0 OR x > c ;
```

Una restricción sobre constantes del tipo `BOOLEAN` es que su valor sólo puede ser una fórmula *cerrada*, esto es, sin variables libres.

Un término y su nombre puede ser usados indistintamente en expresiones posteriores. Los términos con nombre son a menudo útiles para compartir subtérminos (términos usados varias veces en diferente lugares) desde su uso pueden hacer la entrada exponencialmente más concisa. Los términos con nombre son procesados muy eficientemente por **CVC3**. Es mucho más eficiente asociar un término complejo con un nombre directamente en lugar de declarar una constante y después comprobar si es igual al mismo término.

En **CVC3** uno puede asociar un término a un símbolo de función de cualquier aridad. Para símbolos de función no constantes se declara de la forma

`f : (`T_1`,...,`T_n`) ->T = LAMBDA (`x_1 `:` T_1`,...,`x_n `:` T_n`) : t ;`

donde `t` es cualquier término de tipo `T` con variables libres $x_1,\ldots,x_n$. El conector lambda tiene la semántica normal y se ajusta a las reglas léxicas de ámbito normales: con el término `t` la declaración de los símbolos $x_1,\ldots,x_n$ como variables locales de l tipo respectivo $T_1,\ldots,T_n$ ocultando cualquier declaración global previa sobre estos símbolos.

Cuando hay k tipos consecutivos $T_i,\ldots,T_{i+k-1}$ en la expresión lambda `LAMBDA(`x_1 `:` T_1`,...,x :` T_n`) : t` son idénticos, la sintaxis `LAMBDA(`x_1 `:` T_1`,...,`x_i`,...,`x_{i+k-1} `:` T_i`,...,x :` T_n`) : t` también se permite.

```
% Global declaration of x as a unary function symbol
x: REAL -> REAL;

% Local declarations of x as a constant symbol
```

```
f: REAL -> REAL = LAMBDA (x: REAL): 2*x + 3;

p: (INT, INT) -> BOOLEAN = LAMBDA (x,i: INT): i*x - 1 > 0;

g: (REAL, INT) -> [REAL, INT] = LAMBDA (x: REAL, i:INT): (x + 1, i - 3);
```

Los símbolos de constante y de función pueden también ser declarados localmente en cualquier lugar con un término por medio del enlazador `let`. Una posible definición usando `let` sería

```
t: REAL =
  LET g = LAMBDA(x:INT): x + 1,
      x1 = 42,
      x2 = 2*x1 + 7/2
  IN
     (LET x3 = g(x1) IN x3 + x2) / x1;
```

2.4. Interfaz CVC3 en Maude

Para la realización de esta tesina se ha utilizado una versión modificada de **Maude** (creada por un estudiante de *Grigore Rosu*) que lleva incluido el SAT-Solver **CVC3** en el propio ejecutable (disponible públicamente en [8]). Para poder interactuar entre **Maude** y **CVC3** se ha utilizado una interfaz desarrollada por *Camilo Rocha*[9] (estudiante de doctorado de la University of Illinois) en la cual se realiza '*deep-embedding*'[10] de la sintaxis de **PLEXIL**[11] en **Maude**. Esta interfaz define un pequeño lenguaje que posteriormente se transforma a lenguaje SMT-LIB y, éste último, se envía al solver.

A continuación se detallan los aspectos más importantes de esta interfaz.

2.4.1. Tipos de datos

Las expresiones de la interfaz pueden ser constantes, variables o términos recursivamente formados a partir de los dos anteriores. Cada constante o

[8] `http://code.google.com/p/sraplx/downloads/list`
[9] `http://www.camilorocha.info/`
[10] `http://en.wiktionary.org/wiki/deep_embedding`
[11] `http://en.wikipedia.org/wiki/PLEXIL`

variable puede ser de tipo booleano o entero. Algunas definiciones de datos se muestran en la siguiente lista:

```
sorts iBool iBoolAtom iBoolCns iBoolVar .
sorts iInt iIntAtom iIntCns iIntVar .
subsort iBoolCns iBoolVar < iBoolAtom < iBool .
subsort iIntCns  iIntVar  < iIntAtom  < iInt .
subsorts iBool iInt < iExpr .
subsorts iBoolAtom iIntAtom < iExprAtom .
subsorts iExprAtom < iExpr .
```

donde se puede observar que el tipo más general se denomina `iExpr`. Además se definen dos tipos generales, ’`iBool`’ para booleanos e ’`iInt`’ para enteros. A su vez, hay dos subtipos para cada uno, uno para las constantes booleanas ’`iBoolCns`’ y para las variables booleanas ’`iBoolVar`’. Del mismo modo se han definido para el tipo entero, para constantes ’`iIntCns`’ y para variables ’`iIntVar`’.

Las constantes se definen mediante el operador ’`c`’ y un `Bool` (en el caso de ser booleano) y un `Int` (en caso de ser entero):

```
--- Boolean and integer constants
op c : Bool -> iBoolCns [ctor] .
op c : Int -> iIntCns [ctor] .
```

Por su parte, para representar variables existen dos operadores distintos ’`b`’ para variables booleanas e ’`i`’ para enteras. Ambos necesitan un `Nat` para poder identificar la variable concreta.

```
--- Boolean and integer variables
op b : Nat -> iBoolVar [ctor] .
op i : Nat -> iIntVar [ctor] .
```

Al mismo tiempo se han definido las típicas operaciones entre tipos de datos booleanos (tanto variables como constantes) tales como la negación ( ), igualdad (`===`), desigualdad (= // =), disyunción (`^`) y conjunción (`v`) .

```
--- Boolean expressions
op ~_ : iBool -> iBool [prec 41] .
ops _^_ _v_ : iBool iBool -> iBool [assoc comm prec 45] .
op _->_ : iBool iBool -> iBool [prec 47] .
ops _===_ _=//=_ : iBool iBool -> iBool [comm prec 60] .
```

Del mismo modo, también se han definido las operaciones comunes para los datos de tipo entero

```
--- Integer expressions
op -_ : iInt -> iInt [prec 31] .
ops _+_ _*_ : iInt iInt -> iInt [assoc comm prec 35] .
op _-_ : iInt iInt -> iInt .

--- Relational expressions on integers
ops _<=_ _<_ _>=_ _>_ : iInt iInt -> iBool [prec 37] .
ops _===_ _=//=_ : iInt iInt -> iBool [comm prec 60] .
```

2.4.2. Modo de uso de la interfaz

La función para realizar comprobaciones de satisfacibilidad sobre una expresión definida en la interfaz (`iBool`) es `check-sat`, del mismo modo existe una función para comprobar si una expresión no es satisfacible llamada `check-unsat`. Ambas función están definidas en el módulo `SMT-INTERFACE`.

```
--- SMT interface for checking (un)satisfiability of PLEXIL's Boolean
    expressions
fmod SMT-INTERFACE is
pr SMT-HOOK .
pr SMT-TRANSLATE .

var  iB      : iBool .
--- checks if the given Boolean expression is satisfiable
op check-sat : iBool -> Bool [memo] .
eq check-sat(iB)
 = if iB == c(true)
   then true
   else
     if iB == c(false)
     then false
     else
       if check-sat(translate(iB)) == "sat"
       then true
       else false
       fi
     fi
   fi .
--- checks if the given Boolean expression is unsatisfiable
op check-unsat : iBool -> Bool [memo] .
eq check-unsat(iB)
 = if iB == c(false)
   then true
   else
     if iB == c(true)
     then false
     else
       if check-sat(translate(iB)) == "unsat"
       then true
```

```
        else false
        fi
      fi
    fi .
endfm
```

La función `translate` (definida en el módulo `SMT-TRANSLATE`) es la que se encarga de de transformar un término de tipo `iBool` en una cadena (`String`) que sigue la sintaxis **SMT-LIB** estándar (en la lista indican las primeras líneas del módulo)

```
fmod SMT-TRANSLATE is
pr 3TUPLE{String,NatSet,NatSet}
   * (sort Tuple{String,NatSet,NatSet} to Translation) .
pr EXPR .
pr SMT-CONSTANTS .
pr CONVERSION .

var  B          : Bool .
vars iB iB'     : iBool .
vars iE iE'     : iExpr .
vars iI iI'     : iInt .
vars I I'       : Int .
vars N N'       : Nat .
vars NS NS'     : NatSet .
vars NS2 NS3    : NatSet .
vars Str Str'   : String .
vars Str2 Str3  : String .

--- translates a given Boolean expression into the
--- SMTLIB syntax
op translate : iBool -> String [memo] .
--- translates a given expression into the SMTLIB syntax
--- accumulating the Boolean and integer symbolic variables in it
op $trans : iExpr -> Translation [memo] .
ceq translate(iE)
  = add-smt-metadata(Str' + Str2)
 if (Str,NS,NS') := $trans(iE)
 /\ Str' := declare-bool-vars(NS) + declare-int-vars(NS')
 /\ Str2 := "(assert " + Str + ")" .
```

Este `String` resultante se le envía por parámetros a la función `check-sat` definida el módulo `SMT-HOOK` que actúa a modo de enlace (*hook*) entre la interfaz y el solver. Es decir, es la función que le envía los datos al solver integrado en **Maude**. Esta función devuelve un `String` que únicamente puede tener dos valores:

- '`sat`' si la expresión es satisfacible.
- '`unsat`' si la expresión no es satisfacibe.

```
fmod SMT-HOOK is
  including STRING .
  op check-sat : String -> String
  [special (id-hook StringOpSymbol (callSolvers)
            op-hook stringSymbol (<Strings> : ~> String))] .
endfm
```

2.4.3. Ejemplos de expresiones

A continuación se definen algunos ejemplos de expresiones con la sintaxis definida en la interfaz.

En el primer ejemplo se comprueba si la expresion 1 + 2 es igual a la expresión 4 - 1

```
red check-sat( (c(1) + c(2))  === (c(4) - c(1)) ) .
```

El resultado obtenido es el esperado (`true`)

```
reduce in METASAT : check-sat(c(1) + c(2) === c(4) - c(1)) .
rewrites: 37 in 1ms cpu (1ms real) (28179 rewrites/second)
result Bool: true
```

También se pueden mezclar tipos de datos `iBool` e `iInt`, en este caso se verifica si `true` (del ejemplo anterior) `^ c(false)` es `false`.

```
red check-sat( (c(1) + c(2)  === c(4) - c(1)) ^ c(false) ) .
```

El resultado es

```
reduce in METASAT : check-sat(c(false) ^ (c(1) + c(2) === c(4) - c(1))) .
rewrites: 27 in 1ms cpu (1ms real) (23663 rewrites/second)
result Bool: false
```

En el siguiente ejemplo se comprueba si existe una variable x que haga que la siguiente ecuación sea cierta (x * 1 = x * 2)

```
red check-sat( (i(0) * c(1)) === (i(0) * c(2)) ) .
```

El resultado es `true` ya que existe un único caso cuando x = 0.

```
reduce in METASAT : check-sat(c(1) * i(0) === c(2) * i(0)) .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
```

Como último ejemplo se muestra una expresión muy parecida a las que se generarán usando el prototipo. En este caso se comprueba si existen algún valor para las variables X, Y, W, Z y K tal que $X = 1 \wedge W = 2 \wedge X = Y \wedge Y = W + Z \wedge Z = K$, donde las variables se codifican de la siguiente forma x == `i(1)`, y == `i(2)`, w == `i(3)`, z == `i(4)` y k == `i(5)`.

```
red check-sat(c(true) ^ (c(1) === i(1)) ^ (c(2) === i(3)) ^ (i(1) === i(2))
    ^ (i(2) === i(3) + i(4)) ^ (i(4) === i(5)) ) .
```

El resultado obtenido es

```
reduce in METASAT : check-sat(c(true) ^ ((((i(2) === i(3) + i(4)) ^ (i(4)
    ===
    i(5))) ^ (i(1) === i(2))) ^ (c(2) === i(3))) ^ (c(1) === i(1))) .
rewrites: 150 in 4ms cpu (4ms real) (32930 rewrites/second)
result Bool: true
```

ya que puede darse el caso si $X = 1 \wedge Y = 1 \wedge W = 2 \wedge Z = -1 \wedge k = -1$.

3

Prototipo - Primera parte

Tomando como base la implementación de la interfaz desarrollada por *Camilo Rocha* (Sección 2.4) se ha definido un prototipo en el que, dado un problema de satisfacibilidad como una secuencia de términos (en metanivel) se convierte dicho problema en una expresión (`iBool`, Sección 2.4.1) que pueda ser evaluada por el SAT **CVC3**. El presente prototipo se ha definido en el módulo `METASAT-INT`. Primero procedemos a detallar formalmente la resolución de problemas de satisfacibilidad en Maude.

3.1. Satisfacibilidad de igualdades sobre los números naturales en Maude

En esta tesina, partimos del model estándar de la teoría de primer orden para los números naturales, denominada *aritmética Presburger* en honor a Mojżesz Presburger, quien la introdujo. La aritmética Presburger incluye solamente los números naturales, la operación de suma (+) y la operación de igualdad entre términos (=). Dicha teoría de primer orden omite la operación de multiplicación, donde la *aritmética de Peano* corresponde a la aritmética Presburger junto con la multiplicación y no se disponen de procedimientos de decisión para la aritmética de Peano mientras que sí existen para la aritmética Presburger. Normalmente, los procedimientos de decisión para la aritmética Presburger asumen también una operación binaria de mayor-que entre dos números naturales (>), así como los operadores lógicos típicos (conjunción

$\wedge$, disyunción $\vee$, negación $\neg$). Sin embargo, nosotros vamos a asumir sólo la función de mayor-que y conjunciones de igualdades, manejando cada igualdad de forma separada.

Formalmente, definimos la teoría de primer orden para la aritmética Presburger como $\mathbb{N} = (\mathcal{N}, +_{\mathbb{N}}, 0_{\mathbb{N}}, 1_{\mathbb{N}}, >_{\mathbb{N}})$.

La satisfacibilidad de una conjunción de igualdades $C = \{u_1 = v_1 \wedge \cdots \wedge u_k = v_k\}$ para la aritmética Presburger $\mathbb{N}$ donde los términos $u_1, v_1, \ldots, u_k, v_k$ son términos con variables, se define como C es $\mathbb{N}$-*satisfacible* si existe una sustitución $\sigma : \mathcal{X}_C \to \mathcal{N}$, $\mathcal{X}_C = \mathit{Vars}(\bigwedge_i u_i = v_i)$, tal que $\mathbb{N} \models \bigwedge_i u_i\sigma = v_i\sigma$.

Sin embargo, asumimos una teoría ecuacional en Maude que caracterice la aritmética Presburger. Es decir, asumimos una teoría ecuacional $(\Sigma_{\mathbb{N}}, E_{\mathbb{N}}, R_{\mathbb{N}})$ para tipos ordenados basada en un tipo especial Nat que sea una descomposición de la aritmética Presburger $\mathbb{N}$. Es decir, una conjunción de igualdades $C = \{u_1 = v_1 \wedge \cdots \wedge u_k = v_k\}$ para la aritmética Presburger $\mathbb{N}$ donde los términos $u_1, v_1, \ldots, u_k, v_k$ son términos de $\mathcal{T}_{\Sigma}(\mathcal{X})_{\mathsf{Nat}}$, es satisfacible si y sólo si existe una sustitución $\sigma : \mathcal{X}_C \to \mathcal{T}_{\Sigma,\mathsf{Nat}}$, $\mathcal{X}_C = \mathit{Vars}(\bigwedge_i u_i = v_i)$, tal que para cada i, $(u_i\sigma){\downarrow}_{R_{\mathbb{N}},E_{\mathbb{N}}} =_{E_{\mathbb{N}}} (v_i\sigma){\downarrow}_{R_{\mathbb{N}},E_{\mathbb{N}}}$.

Maude dispone de un tratamiento interno para los números naturales y la aritmética Presburger, como se puede observar en la teoría ecuacional asociada a los números naturales en el fichero `prelude.maude` de la instalación de Maude.

```
fmod NAT is
  protecting BOOL .
  sorts Zero NzNat Nat .
  subsort Zero NzNat < Nat .
  op 0 : -> Zero [ctor] .

  op s_ : Nat -> NzNat
       [ctor iter
        special (id-hook SuccSymbol
                 term-hook zeroTerm (0))] .

  op _+_ : NzNat Nat -> NzNat
       [assoc comm prec 33
        special (id-hook ACU_NumberOpSymbol (+)
                 op-hook succSymbol (s_ : Nat ~> NzNat))] .
  op _+_ : Nat Nat -> Nat [ditto] .

  op sd : Nat Nat -> Nat
       [comm
        special (id-hook CUI_NumberOpSymbol (sd)
                 op-hook succSymbol (s_ : Nat ~> NzNat))] .
```

```
op _*_ : NzNat NzNat -> NzNat
      [assoc comm prec 31
       special (id-hook ACU_NumberOpSymbol (*)
                op-hook succSymbol (s_ : Nat ~> NzNat))] .
op _*_ : Nat Nat -> Nat [ditto] .

op _quo_ : Nat NzNat -> Nat
      [prec 31 gather (E e)
       special (id-hook NumberOpSymbol (quo)
                op-hook succSymbol (s_ : Nat ~> NzNat))] .

op _rem_ : Nat NzNat -> Nat
      [prec 31 gather (E e)
       special (id-hook NumberOpSymbol (rem)
                op-hook succSymbol (s_ : Nat ~> NzNat))] .

op _^_ : Nat Nat -> Nat
      [prec 29  gather (E e)
       special (id-hook NumberOpSymbol (^)
                op-hook succSymbol (s_ : Nat ~> NzNat))] .
op _^_ : NzNat Nat -> NzNat [ditto] .

op modExp : Nat Nat NzNat ~> Nat
      [special (id-hook NumberOpSymbol (modExp)
                op-hook succSymbol (s_ : Nat ~> NzNat))] .

op gcd : NzNat Nat -> NzNat
      [assoc comm
       special (id-hook ACU_NumberOpSymbol (gcd)
                op-hook succSymbol (s_ : Nat ~> NzNat))] .
op gcd : Nat Nat -> Nat [ditto] .

op lcm : NzNat NzNat -> NzNat
      [assoc comm
       special (id-hook ACU_NumberOpSymbol (lcm)
                op-hook succSymbol (s_ : Nat ~> NzNat))] .
op lcm : Nat Nat -> Nat [ditto] .

op min : NzNat NzNat -> NzNat
      [assoc comm
       special (id-hook ACU_NumberOpSymbol (min)
                op-hook succSymbol (s_ : Nat ~> NzNat))] .
op min : Nat Nat -> Nat [ditto] .

op max : NzNat Nat -> NzNat
      [assoc comm
       special (id-hook ACU_NumberOpSymbol (max)
                op-hook succSymbol (s_ : Nat ~> NzNat))] .
op max : Nat Nat -> Nat [ditto] .

op _xor_ : Nat Nat -> Nat
      [assoc comm prec 55
       special (id-hook ACU_NumberOpSymbol (xor)
                op-hook succSymbol (s_ : Nat ~> NzNat))] .

op _&_ : Nat Nat -> Nat
      [assoc comm prec 53
       special (id-hook ACU_NumberOpSymbol (&)
                op-hook succSymbol (s_ : Nat ~> NzNat))] .

op _|_ : NzNat Nat -> NzNat
      [assoc comm prec 57
```

```
         special (id-hook ACU_NumberOpSymbol (|)
                  op-hook succSymbol (s_ : Nat ~> NzNat))] .
  op _|_ : Nat Nat -> Nat [ditto] .

  op _>>_ : Nat Nat -> Nat
        [prec 35 gather (E e)
         special (id-hook NumberOpSymbol (>>)
                  op-hook succSymbol (s_ : Nat ~> NzNat))] .

  op _<<_ : Nat Nat -> Nat
        [prec 35 gather (E e)
         special (id-hook NumberOpSymbol (<<)
                  op-hook succSymbol (s_ : Nat ~> NzNat))] .

  op _<_ : Nat Nat -> Bool
        [prec 37
         special (id-hook NumberOpSymbol (<)
                  op-hook succSymbol (s_ : Nat ~> NzNat)
                  term-hook trueTerm (true)
                  term-hook falseTerm (false))] .

  op _<=_ : Nat Nat -> Bool
        [prec 37
         special (id-hook NumberOpSymbol (<=)
                  op-hook succSymbol (s_ : Nat ~> NzNat)
                  term-hook trueTerm (true)
                  term-hook falseTerm (false))] .

  op _>_ : Nat Nat -> Bool
        [prec 37
         special (id-hook NumberOpSymbol (>)
                  op-hook succSymbol (s_ : Nat ~> NzNat)
                  term-hook trueTerm (true)
                  term-hook falseTerm (false))] .

  op _>=_ : Nat Nat -> Bool
        [prec 37
         special (id-hook NumberOpSymbol (>=)
                  op-hook succSymbol (s_ : Nat ~> NzNat)
                  term-hook trueTerm (true)
                  term-hook falseTerm (false))] .

  op _divides_ : NzNat Nat -> Bool
        [prec 51
         special (id-hook NumberOpSymbol (divides)
                  op-hook succSymbol (s_ : Nat ~> NzNat)
                  term-hook trueTerm (true)
                  term-hook falseTerm (false))] .
endfm
```

Sin embargo, podemos asociar una teoría ecuacional inductivamente definida para la aritmética Presburger como sigue. Esta teoría ecuacional corresponde con el modelo asociado en Maude a la aritmética Presburger y es una descomposición de la aritmética Presburger.

```
fmod NAT is
  protecting BOOL .
```

```
  sorts Zero NzNat Nat .
  subsort Zero NzNat < Nat .

  op 0 : -> Zero [ctor] .

  op s_ : Nat -> NzNat [ctor] .

  op _+_ : NzNat Nat -> NzNat .
  op _+_ : Nat Nat -> Nat [ditto] .
  eq 0       + Y:Nat = Y:Nat .
  eq s X:Nat + Y:Nat = s (X:Nat + Y:Nat) .

  op _<_ : Nat Nat -> Bool .
  eq 0       < 0       = false .
  eq 0       < s X:Nat = true .
  eq s X:Nat < s Y:Nat = X:Nat < Y:Nat .
  eq s X:Nat < 0       = false .

endfm
```

Dado el módulo `NAT` de Maude, un ejemplo de decisión sobre la satisfacibilidad de una conjunción de igualdades sería como sigue, usando la función `metasat-interface` que transforma la conjunción de igualdades descritas en Maude en una llamada al SAT-Solver CVC3.

```
reduce in METASAT : metasat-interface(upModule('NAT, false),
    'X === 's_^1['0.Zero] ^ 'X:Nat === '0.Zero ^ 'X:Nat === 's_^2['0.Zero])
      .
result Bool: false
```

A continuación describimos en detalle cómo se realiza la transformación de una llamada de satisfacibilidad a `metasat-interface` en una llamada de satisfacibilidad apropiada en CVC3.

3.2. Tipos de datos

Los tipos de datos definidos en el prototipo son `SATPair`, `SATProblem` y `SubstiExprTetra`. El tipo `SATPair` define un par compuesto por términos (*metatérminos*) que tienen el mismo valor (es una suposición, ésta puede ser cierta o falsa). El operador constructor de `SATPair` se define mediante la igualdad (representada por `===`) entre dos términos (`Term`). El tipo `SATProblem` representa un conjunto de `SATPair` (que a su vez son subtipos de `SATProblem`) y, mediante el operador `^`, se pueden concatenar. Finalmente, se ha definido el tipo `SubstiExprTetra` que representa una tupla formada por una substitución, la expresión generada (para ser usada por la implementación de *Camilo Rocha*), un entero para identificar la próxima variable que se generará y un conjunto de restricciones sobre las variable enteras generadas (para obligar que sean positivas). También se han definido varias operaciones para extraer de esta tupla cada una de sus componente. Cada uno de estos tipos y operaciones se pueden observar en la lista 3.1.

Listing 3.1: Definiciones de tipos y operaciones básicas

```
sorts SATPair SATProblem .
subsort SATPair < SATProblem .


*** SAT problems

op empty : -> SATProblem .
op _===_ : Term Term -> SATPair [ctor prec 71] .
op _^_ : SATProblem SATProblem -> SATProblem
 [ctor assoc comm id: empty prec 73] .

*** success results

sort SubstiExprTetra .
op {_,_,_,_} : Substitution iExpr Nat iExpr -> SubstiExprTetra .

op getSubst : SubstiExprTetra -> Substitution .
eq getSubst ({S:Substitution, B:iExpr, N:Nat, i:iExpr}) = S:Substitution .

op getiExpr : SubstiExprTetra -> iExpr .
eq getiExpr ({S:Substitution, B:iExpr, N:Nat, i:iExpr}) = B:iExpr .

op getNat : SubstiExprTetra -> Nat .
eq getNat ({S:Substitution, B:iExpr, N:Nat, i:iExpr}) = N:Nat .

op getVars : SubstiExprTetra -> iExpr .
eq getVars({S:Substitution, B:iExpr, N:Nat, i:iExpr}) = i:iExpr .
```

3.3. Transformación de `SATProblem` a expresiones `iBool`

La función que inicia todo el proceso de generación se ha denominado `metasat-interface` que recibe un `SATProblem` y realiza una llamada a la función `check-sat` (descrita en la Sección 2.4.2) con la expresión transformada.

```
op metasat-interface : Module SATProblem -> Bool .
```

```
eq metasat-interface(M:Module, X:SATProblem) =
        check-sat (iExprUnion(trans(M:Module, X:SATProblem, none, 1, c(true)
            ))) .
```

Toda la transformación recae sobre la función `trans` que va dividiendo el `SATProblem` en `SATPair` y las expresiones resultantes se van concatenando en la tupla `SubstiExprTetra` final. Cada uno de los términos que forman el `SATPair` se envía a la función `transT` (*translateTerm*) que los convierte y devuelve el resultado.

```
op trans : Module SATProblem Substitution Nat iExpr -> SubstiExprTetra .
eq trans(M:Module, empty,S:Substitution, N:Nat, i:iExpr) = {S:Substitution,
    c(true), N:Nat, i:iExpr} .

ceq trans(M:Module, T1:Term === T2:Term ^ X:SATProblem, S:Substitution, N:
    Nat, i:iExpr)
        = {S3:Substitution, (B1:iExpr === B2:iExpr) ^ B3:iBool, N3:Nat, i3:
            iExpr}
        if  {S1:Substitution, B1:iExpr, N1:Nat, i1:iExpr}
                := transT(M:Module, T1:Term,       S:Substitution,  N:Nat, i
                    :iExpr)
        /\  {S2:Substitution, B2:iExpr, N2:Nat, i2:iExpr}
                := transT(M:Module, T2:Term,      S1:Substitution, N1:Nat,
                    i1:iExpr)
        /\  {S3:Substitution, B3:iBool, N3:Nat, i3:iExpr}
                := trans (M:Module, X:SATProblem, S2:Substitution, N2:Nat,
                    i2:iExpr) .
```

La función `transT`, como se ha comentado anteriormente, transforma un término (`Term`) en una `iExpr` aceptada por la implementación de *Camilo Rocha* (Sección 2.4). Esta función recibe un término, un conjunto de sustituciones (para las variables generadas), un `Nat` que indica el identificador de la siguiente variable a generar y el conjunto de variables (en formato `iExpr`). Según el tipo de término se realiza una transformación u otra. En el caso que el término sea una operación (un `QUID` con dos o más términos) se sustituye dicha operación por la equivalente definida en la interfaz de *Camilo Rocha* (Sección 2.4.1) y se envía recursivamente cada uno de los términos de la operación nuevamente a la misma función.

```
op transT : Module Term Substitution Nat iExpr -> SubstiExprTetra .

ceq transT(M:Module, '_+_[T1:Term, T2:TermList], S:Substitution, N:Nat, i:
    iExpr)
        = {S2:Substitution, (B1:iExpr + B2:iExpr), N2:Nat, i2:iExpr}
if {S1:Substitution, B1:iExpr, N1:Nat, i1:iExpr}
        := transT(M:Module, T1:Term, S:Substitution, N:Nat, i:iExpr)
/\ {S2:Substitution, B2:iExpr, N2:Nat, i2:iExpr}
        := transT(M:Module, '_+_[T2:TermList], S1:Substitution, N1:Nat, i1:
            iExpr) .
```

```
ceq transT(M:Module, 's_[T:Term], S:Substitution, N:Nat, i:iExpr)
        = {S1:Substitution, (B1:iExpr + c(1)), N1:Nat, i1:iExpr}
if {S1:Substitution, B1:iExpr, N1:Nat, i1:iExpr}
        := transT(M:Module, T:Term, S:Substitution, N:Nat, i:iExpr) .

eq transT(M:Module, F:Qid[empty], S:Substitution, N:Nat, i:iExpr)
        = {S:Substitution, c(0) , N:Nat, i:iExpr} .

ceq transT(M:Module, 'sd[T1:Term, T2:TermList], S:Substitution, N:Nat, i:iExpr)
        = {S2:Substitution, (B1:iExpr - B2:iExpr), N2:Nat, i2:iExpr}
if {S1:Substitution, B1:iExpr, N1:Nat, i1:iExpr}
        := transT(M:Module, T1:Term, S:Substitution, N:Nat, i:iExpr)
/\ {S2:Substitution, B2:iExpr, N2:Nat, i2:iExpr}
        := transT(M:Module, 'sd[T2:TermList], S1:Substitution, N1:Nat, i1:iExpr) .

ceq transT(M:Module, '_*_[T1:Term, T2:TermList], S:Substitution, N:Nat, i:iExpr)
        = {S2:Substitution, (B1:iExpr * B2:iExpr), N2:Nat, i2:iExpr}
if {S1:Substitution, B1:iExpr, N1:Nat, i1:iExpr}
        := transT(M:Module, T1:Term, S:Substitution, N:Nat, i:iExpr)
/\ {S2:Substitution, B2:iExpr, N2:Nat, i2:iExpr}
        := transT(M:Module, '_*_[T2:TermList], S1:Substitution, N1:Nat, i1:iExpr) .

ceq transT(M:Module, '_=//=_[T1:Term, T2:Term], S:Substitution, N:Nat, i:iExpr)
        = {S2:Substitution, B1:iExpr =//= B2:iExpr, N2:Nat, i2:iExpr}
if {S1:Substitution, B1:iExpr, N1:Nat, i1:iExpr}
        := transT(M:Module, T1:Term, S:Substitution, N:Nat, i:iExpr)
/\ {S2:Substitution, B2:iExpr, N2:Nat, i2:iExpr}
        := transT(M:Module, T2:Term, S1:Substitution, N1:Nat, i1:iExpr) .

ceq transT(M:Module, '_==_[T1:Term, T2:Term], S:Substitution, N:Nat, i:iExpr)
        = {S2:Substitution, B1:iExpr === B2:iExpr, N2:Nat, i2:iExpr}
if {S1:Substitution, B1:iExpr, N1:Nat, i1:iExpr}
        := transT(M:Module, T1:Term, S:Substitution, N:Nat, i:iExpr)
/\ {S2:Substitution, B2:iExpr, N2:Nat, i2:iExpr}
        := transT(M:Module, T2:Term, S1:Substitution, N1:Nat, i1:iExpr) .

ceq transT(M:Module, '_<_[T1:Term, T2:Term], S:Substitution, N:Nat, i:iExpr)
        = {S2:Substitution, B1:iExpr < B2:iExpr, N2:Nat, i2:iExpr}
if {S1:Substitution, B1:iExpr, N1:Nat, i1:iExpr}
        := transT(M:Module, T1:Term, S:Substitution, N:Nat, i:iExpr)
/\ {S2:Substitution, B2:iExpr, N2:Nat, i2:iExpr}
        := transT(M:Module, T2:Term, S1:Substitution, N1:Nat, i1:iExpr) .

ceq transT(M:Module, '_>_[T1:Term, T2:Term], S:Substitution, N:Nat, i:iExpr)
        = {S2:Substitution, B1:iExpr > B2:iExpr, N2:Nat, i2:iExpr}
if {S1:Substitution, B1:iExpr, N1:Nat, i1:iExpr}
        := transT(M:Module, T1:Term, S:Substitution, N:Nat, i:iExpr)
/\ {S2:Substitution, B2:iExpr, N2:Nat, i2:iExpr}
        := transT(M:Module, T2:Term, S1:Substitution, N1:Nat, i1:iExpr) .

ceq transT(M:Module, '_<=_[T1:Term, T2:Term], S:Substitution, N:Nat, i:iExpr)
        = {S2:Substitution, B1:iExpr <= B2:iExpr, N2:Nat, i2:iExpr}
if {S1:Substitution, B1:iExpr, N1:Nat, i1:iExpr}
        := transT(M:Module, T1:Term, S:Substitution, N:Nat, i:iExpr)
```

```
/\ {S2:Substitution, B2:iExpr, N2:Nat, i2:iExpr}
        := transT(M:Module, T2:Term, S1:Substitution, N1:Nat, i1:iExpr) .

ceq transT(M:Module, '_>=_[T1:Term, T2:Term], S:Substitution, N:Nat, i:iExpr
    )
        = {S2:Substitution, B1:iExpr >= B2:iExpr, N2:Nat, i2:iExpr}
if {S1:Substitution, B1:iExpr, N1:Nat, i1:iExpr}
        := transT(M:Module, T1:Term, S:Substitution, N:Nat, i:iExpr)
/\ {S2:Substitution, B2:iExpr, N2:Nat, i2:iExpr}
        := transT(M:Module, T2:Term, S1:Substitution, N1:Nat, i1:iExpr) .
```

Cuando el término que se recibe en la función `transT` es un booleano o un 0 se sustituye dicho término por el correspondiente: `'true.Bool` por `c(true)`, `'false.Bool` por `c(false)` y `'0.Zero` por `c(0)` (los otros datos pasados por parámetros no se alteran). Si el término no es ninguno de estos se envía a la función `transT2`.

```
eq transT(M:Module, 'true.Bool, S:Substitution, N:Nat, i:iExpr)
        = {S:Substitution, c(true), N:Nat, i:iExpr} .

eq transT(M:Module, 'false.Bool, S:Substitution, N:Nat, i:iExpr)
        = {S:Substitution, c(false), N:Nat, i:iExpr} .

eq transT(M:Module, '0.Zero, S:Substitution, N:Nat, i:iExpr)
        = {S:Substitution, c(0), N:Nat, i:iExpr} .

eq transT(M:Module, T:Term, S:Substitution, N:Nat, i:iExpr)
        = transT2(M:Module, T:Term, S:Substitution, N:Nat, i:iExpr) [owise]
          .
```

La función `transT2` se encarga de transformar los términos en caso que sean de tipo variable y exista una substitución (en el conjunto de substituciones) para ésta. En primer lugar se comprueba el tipo de variable:

- Si es de tipo `Bool` entonces se añade el término `b(downTerm(T:Term, 0))` donde `T:Term` es la substitución de la variable.
- Si es de tipo `Nat` entonces se añade el término `i(downTerm(T:Term, 0))` y, además se comprueba que exista dicha variable en el conjunto de restricciones de variables (para obligar que sean variables positivas, como se ha comentado con anterioridad).

```
op transT2 : Module Term Substitution Nat iExpr -> SubstiExprTetra .

ceq transT2(M:Module, V:Variable, V:Variable <- T:Term ; S:Substitution, N:
    Nat, i:iExpr)
        = {V:Variable <- T:Term ; S:Substitution, b(downTerm(T:Term, 0)), N:
            Nat, i:iExpr}
```

```
        if sortLeq(M:Module, 'Bool, getType(V:Variable)) == true .

ceq transT2(M:Module, V:Variable, V:Variable <- T:Term ; S:Substitution, N:
    Nat, i:iExpr)
        = {V:Variable <- T:Term ; S:Substitution, i(downTerm(T:Term, 0)), N:
            Nat,
        if existVar(i:iExpr, i(downTerm(T:Term, 0))) == true
                then i:iExpr
                else i:iExpr ^ (i(N:Nat) >= c(0)) fi }
        if sortLeq(M:Module, 'Nat, getType(V:Variable)) == true .

eq transT2(M:Module, T:Term, S:Substitution, N:Nat, i:iExpr)
        = transT3(M:Module, T:Term, S:Substitution, N:Nat, i:iExpr) [owise]
            .
```

En el caso que no exista una substitución sobre la variable entonces se llama a la función `transT3` para que siga con la traducción. Si no existe una substitución sobre una variable significa que dicha variable se está convirtiendo por primera vez y, por tanto, a parte de realizar la conversión que se produce en la función `transT2` se ha de añadir la substitución al conjunto de substituciones. En caso que el término no unifique con ninguna cabecera, se delega su transformación a la función `transT4`.

```
op transT3 : Module Term Substitution Nat iExpr -> SubstiExprTetra .

eq transT3(M:Module, V:Variable, S:Substitution, N:Nat, i:iExpr)
        = { V:Variable <- upTerm(N:Nat) ; S:Substitution,
    if sortLeq(M:Module, 'Bool, getType(V:Variable)) == true
                then b(N:Nat)
                else i(N:Nat)
        fi,
        N:Nat + 1,
        if existVar(i:iExpr, i(N:Nat)) == true
                then i:iExpr
                else i:iExpr ^ (i(N:Nat) >= c(0)) fi } .

eq transT3(M:Module, T:Term, S:Substitution, N:Nat, i:iExpr)
        = transT4(M:Module, T:Term, S:Substitution, N:Nat, i:iExpr) [owise]
            .
```

La función `transT4` se ejecuta en caso que no se pueda convertir por ninguna función anterior y únicamente comprueba que el término sea de tipo `Nat` y pone ese número dentro de un constructor de variables (`c(X:Nat)`).

```
eq transT3(M:Module, T:Term, S:Substitution, N:Nat, i:iExpr) = transT4(M:
    Module, T:Term, S:Substitution, N:Nat, i:iExpr) [owise] .

op transT4 : Module Term Substitution Nat iExpr -> SubstiExprTetra .

ceq transT4(M:Module, T:Term, S:Substitution, N:Nat, i:iExpr) = {S:
    Substitution, c(X:Nat), N:Nat, i:iExpr}
        if X:Nat := downTerm(T:Term, 0) .
```

El motivo de utilizar distintas funciones para transformar (`transT`, `transT2`, `transT3`, `transT4`) radica en que se desea llevar un orden de transformación ya que si todo el proceso se realizase dentro de una única función podrían darse casos en el que las transformaciones no se realizasen de forma correcta.

Finalmente, está la función `iExprUnion` que únicamente une la expresión generada con el conjunto de restricciones sobre las variables.

```
op iExprUnion : SubstiExprTetra  -> iExpr .
eq iExprUnion(S:SubstiExprTetra) =  getiExpr(S:SubstiExprTetra) ^ getVars(S:
    SubstiExprTetra) .
```

3.4. Ejemplos de transformación

Todos los ejemplos de esta sección utilizan la meta-representación del módulo `NAT` incluido en **Maude**.

A continuación se muestra un ejemplo de conversión, supongamos que tenemos el problema

```
'#6:Nat === '#0:Nat ^ '#6:Nat === '0.Zero ^ '#7:Nat === '#0:Nat ^ '#7:Nat
    === 's_^1['0.Zero]
```

Donde cada #Entero:Nat representa una variable entera y las constantes enteras se representan en notación de sucesor.

La transformación que realiza la función `trans` genera la tupla `SubstiExprTetra`:

```
{
'#0:Nat <- 's_^2['0.Zero] ;
'#6:Nat <- 's_['0.Zero] ;
'#7:Nat <- 's_^3['0.Zero],
c(true) ^ (c(0) === i(1)) ^ (c(1) === i(3)) ^ (i(1) === i(2)) ^ (i(2) === i
    (3)),
4,
c(true) ^ i(1) >= c(0) ^ i(2) >= c(0) ^ i(3) >= c(0)
}
```

De la cual el conjunto de substituciones es

```
'#0:Nat <- 's_^2['0.Zero] ; '#6:Nat <- 's_['0.Zero] ; '#7:Nat <- 's_^3['0.
    Zero]
```

La expresión generada es

```
c(true) ^ (c(0) === i(1)) ^ (c(1) === i(3)) ^ (i(1) === i(2)) ^ (i(2) === i
    (3))
```

El identificador de la próxima variable es

```
4
```

Y el conjunto de restricciones sobre variables es

```
c(true) ^ i(1) >= c(0) ^ i(2) >= c(0) ^ i(3) >= c(0)
```

Para poder ejecutar el anterior ejemplo en el prototipo se debería realizar de la siguiente forma.

```
red metasat-interface(upModule('MULT,false), '#6:Nat === '#0:Nat ^ '#6:Nat
    === '0.Zero ^ '#7:Nat === '#0:Nat ^ '#7:Nat === 's_^1['0.Zero]) .
```

Internamente se convertiría en la siguiente llamada sobre la interfaz de `Camilo Rocha`

```
check-sat(c(true) ^ (c(0) === i(1)) ^ (c(1) === i(3)) ^ (i(1) === i(2)) ^ (i
    (2) === i(3)) ^ c(true) ^ i(1) >= c(0) ^ i(2) >= c(0) ^ i(3) >= c(0) ) .
```

El resultado proporcionado por **Maude** sería:

```
reduce in METASAT :
       metasat-interface(upModule('MULT, false),
       '#6:Nat === '#0:Nat ^ '#6:Nat === '0.Zero ^
       '#7:Nat === '#0:Nat ^ '#7:Nat === 's_^1['0.Zero]) .
rewrites: 223 in 3ms cpu (3ms real) (60221 rewrites/second)
result Bool: false
```

Otro ejemplo de una posible conversión sería el siguiente

```
'#5:Nat === 'M:Nat ^ '#5:Nat === 's_^1['0.Zero] ^ '#6:Nat === '0.Zero ^ '#6:
    Nat === 's_^2['0.Zero]
```

Donde la función `trans` devolvería la tupla

```
{
'#5:Nat <- 's_['0.Zero] ;
'#6:Nat <- 's_^3['0.Zero] ;
'M:Nat <- 's_^2['0.Zero],
c(true) ^ (c(0) === i(3)) ^ (c(1) === i(1)) ^ (c(2) === i(3)) ^ (i(1) === i
    (2)),
4,
c(true) ^ i(1) >= c(0) ^ i(2) >= c(0) ^ i(3) >= c(0)
}
```

Y el resultado obtenido por **Maude** sería:

```
reduce in METASAT : metasat-interface(upModule('MULT, false), '#5:Nat ===
    'M:Nat ^ '#5:Nat === 's_^1['0.Zero] ^ '#6:Nat === '0.Zero ^ '#6:Nat ===
    's_^2['0.Zero]) .
rewrites: 219 in 3ms cpu (3ms real) (59125 rewrites/second)
result Bool: false
```

4

Prototipo - Segunda parte

En esta segunda parte se extiende el procedimiento de decisión de la satisfacibilidad de una conjunción de igualdades para la aritmética Presburger para que admita términos cualesquiera en una teoría ecuacional extendida con más propiedades algebraicas pero sin ninguna regla. Para ello se realizará un proceso de abstracción de variables (sustituyendo las variables contenidas en los pares por nuevas variables frescas) y combinación de variables (utilizando unificación ecuacional) para encontrar unificadores entre las variables. Este prototipo se ha definido en el módulo `METASAT-TRANS`.

4.1. Satisfacibilidad de igualdades para términos cualesquiera

En esta sección, partimos de la teoría de primer orden para la aritmética Presburger $\mathbb{N} = (\mathcal{N}, +_{\mathbb{N}}, 0_{\mathbb{N}}, 1_{\mathbb{N}}, >_{\mathbb{N}})$ definida en la Sección 3.1 y asumimos una teoría ecuacional $(\Sigma_{\mathbb{N}}, E_{\mathbb{N}}, R_{\mathbb{N}})$ para tipos ordenados basada en un tipo especial `Nat` que es una descomposición de la aritmética Presburger $\mathbb{N}$ como se indica también en la Sección 3.1.

Dada una teoría ecuacional para tipos ordenados (Σ, E, R), decimos que es una *extensión válida* de la aritmética Presburger si cumple las siguientes condiciones:

1. se añaden más símbolos y tipos de datos, es decir, $\Sigma = \Sigma_{\mathbb{N}} \uplus \Sigma'$ donde el conjunto de tipos S asociado a Σ contiene el único tipo Nat permitido en $\Sigma_{\mathbb{N}}$,

2. se añaden más propiedades ecuacionales, es decir, $E = E_{\mathbb{N}} \uplus E'$,

3. no existen más ecuaciones que las de la aritmética Presburger, es decir, $R = R_{\mathbb{N}}$,

4. la teoría ecuacional protege el algebra inicial de los naturales, es decir, para cada Σ-término t sin variables del tipo Nat, existe un Σ-término t_0 sin variables del tipo Nat tal que $t{\downarrow_{R,E}} =_{E_{\mathbb{N}}} t_0$.

Dada una teoría ecuacional (Σ, E, R) que sea una extensión válida de la aritmética Presburger $(\Sigma_{\mathbb{N}}, E_{\mathbb{N}}, R_{\mathbb{N}})$ y una conjunción de igualdades $C = \{u_1 = v_1 \wedge \cdots \wedge u_k = v_k\}$ donde los términos $u_1, v_1, \ldots, u_k, v_k$ son términos con variables de $\mathcal{T}_{\Sigma}(\mathcal{X})$, la conjunción C es satisfacible si y sólo si existe una sustitución $\sigma : \mathcal{X}_C \to \mathcal{T}_{\Sigma}$, $\mathcal{X}_C = \mathit{Vars}(\bigwedge_i u_i = v_i)$, tal que para cada i, $(u_i\sigma){\downarrow_{R,E}} =_E (v_i\sigma){\downarrow_{R,E}}$.

Dada una teoría ecuacional (Σ, E, R) que sea una extensión válida de la aritmética Presburger $(\Sigma_{\mathbb{N}}, E_{\mathbb{N}}, R_{\mathbb{N}})$ y un algoritmo de unificación finitario y completo para la teoría ecuacional E' de $E = E_{\mathbb{N}} \uplus E'$, es decidible si una conjunción de igualdades $C = \{u_1 = v_1 \wedge \cdots \wedge u_k = v_k\}$ donde los términos $u_1, v_1, \ldots, u_k, v_k$ son términos con variables de cualquier tipo de datos es satisfacible.

El algoritmo asociado al proceso de satisfacibilidad es muy sencillo gracias a que la teoría extendida protege el algebra inicial de los números naturales. Es decir, dada una conjunción de igualdades $C = \{u_1 = v_1 \wedge \cdots \wedge u_k = v_k\}$ donde los términos $u_1, v_1, \ldots, u_k, v_k$ son términos con variables de $\mathcal{T}_{\Sigma}(\mathcal{X})$, se realiza un proceso de abstracción con variables, generando dos conjuntos $\widehat{C} = \{\widehat{u_1} = \widehat{v_1} \wedge \cdots \wedge \widehat{u_k} = \widehat{v_k}\}$ y $C_{\mathbb{N}} = \{X_1 = t_1 \wedge \cdots \wedge X_n = t_n\}$ donde $X_1, \ldots, X_n \in \mathcal{X}_{\mathsf{Nat}}$ son variables frescas que no aparezcan en C y $t_1, \ldots, t_n \in \mathcal{T}_{\Sigma}(\mathcal{X})_{\mathsf{Nat}}$, tal que para todo $1 \leq i \leq n$, existe un índice $1 \leq j \leq k$ y una posición $p_{i,j}$ tal que $u_i|_{p_{i,j}} = t_i$ y $\widehat{u_i}|_{p_{i,j}} = X_i$ ó $v_i|_{p_{i,j}} = t_i$ y $\widehat{v_i}|_{p_{i,j}} = X_i$.

Una vez generados las conjunciones $\widehat{C}$ y $C_{\mathbb{N}}$, el proceso de decisión es simple gracias a que la teoría protege los naturales, ya que primero resolvemos el conjunto $\widehat{C}$ por unificacion ecuacional y, para cada unificador, invocamos el proceso de decisión para los naturales.

4.2. Tipos de datos

El principal tipo de datos definido en este prototipo es `Pair` (construido mediante el operador `_=?=_` entre dos términos) y representa una pregunta, ¿es el término 1 igual al término 2?. `Pair` es a su vez subtipo de `PairSet` que define un conjunto de `Pair` que permite concatenarlos utilizando el operador `_^_`.

```
sort Pair .
op _=?=_ : Term Term -> Pair [prec 71] .

sort PairSet .
subsort Pair < PairSet .
op emptyPairSet : -> PairSet .
op _^_ : PairSet PairSet -> PairSet [assoc comm id: emptyPairSet prec 73] .
```

Para mantener el estado entre llamadas sobre funciones se ha definido el tipo de datos `Term&PairSet&Counter`, formado por una lista de términos, un conjunto `PairSet` y el `Nat` que identifica la próxima variable que se puede generar. Además se han definido distintas operaciones para acceder a cada uno de sus miembros.

```
*** Variable de estado

sort Term&PairSet&Counter .
op (_,_,_) : TermList PairSet Nat -> Term&PairSet&Counter .
op getTermList : Term&PairSet&Counter -> TermList .
op getPairSet  : Term&PairSet&Counter -> PairSet .
op getCounter  : Term&PairSet&Counter -> Nat .

eq getTermList(T:TermList, P:PairSet, N:Nat) = T:TermList .
eq getPairSet (T:TermList, P:PairSet, N:Nat) = P:PairSet  .
eq getCounter (T:TermList, P:PairSet, N:Nat) = N:Nat      .
```

Se ha definido otro tipo denominado `PairSet&PairSet&Counter` usado para (al igual que el tipo de datos `Term&PairSet&Counter`) mantener el estado cuando se generan secuencias de `PairSet`. Los datos que forman cada término `PairSet&PairSet&Counter` son, un término `PairSet` con la transformación del término, otro término `PairSet` que contiene la lista de restricciones de las variables y, por último, un `Nat` que identifica la próxima variable a generar. Otro tipo generado es el denominado

`PairSet&PairSet&Counter&Substitution`

que únicamente difiere del anterior en que contiene una lista de `Substitution` para evitar creaciones incecesarias de variables.

```
*** Tipo auxiliar para metasat
sort PairSet&PairSet&Counter .
sort PairSet&PairSet&Counter&Substitution .

op `(_`,_`,_`) : PairSet PairSet Nat -> PairSet&PairSet&Counter .
op `(_`,_`,_`,_`) : PairSet PairSet Nat Substitution -> PairSet&PairSet&
    Counter&Substitution .


op getFirstPair  :  PairSet&PairSet&Counter -> PairSet .
op getSecondPair :  PairSet&PairSet&Counter -> PairSet .
op getCounter    :  PairSet&PairSet&Counter -> Nat     .

eq getFirstPair ((P1:PairSet, P2:PairSet, N:Nat)) = P1:PairSet .
eq getSecondPair((P1:PairSet, P2:PairSet, N:Nat)) = P2:PairSet .
eq getCounter   ((P1:PairSet, P2:PairSet, N:Nat)) = N:Nat      .


op getFirstPair  :  PairSet&PairSet&Counter&Substitution -> PairSet .
op getSecondPair :  PairSet&PairSet&Counter&Substitution -> PairSet .
op getCounter    :  PairSet&PairSet&Counter&Substitution -> Nat     .
op getSubst          :  PairSet&PairSet&Counter&Substitution -> Substitution
     .

eq getFirstPair ((P1:PairSet, P2:PairSet, N:Nat, S:Substitution)) = P1:
    PairSet .
eq getSecondPair((P1:PairSet, P2:PairSet, N:Nat, S:Substitution)) = P2:
    PairSet .
eq getCounter   ((P1:PairSet, P2:PairSet, N:Nat, S:Substitution)) = N:Nat
        .
eq getSubst             ((P1:PairSet, P2:PairSet, N:Nat, S:Substitution)) =
    S:Substitution .
```

El último tipo definido es el tipo `Boole` (supertipo de `Bool`) al que se define un nuevo valor `error`, de forma que se convierte en un tipo *tri-estado*. Este estado será muy importante ya que indicará que el prototipo no encontró resultado satisfactorio y debe abortar su ejecución.

```
*** Tipo Boole tri-estado
sort Boole .
subsort Bool < Boole .
op error : -> Boole .
```

4.3. Proceso de conversión

El proceso que se ejecuta en el prototipo para convertir el conjunto `PairSet` en un `SATProblem` se puede dividir en varios pasos:

- Una abstracción de variables sustituyendo las variables encontradas en cada `Pair` por nuevas variables frescas
- Una combinación de variables utilizando la unificación ecuacional para encontrar sustituciones que hagan los pares ciertos
- Una conversión de los `PairSet` resultantes a un `SATProblem`

A continuación se detalla cada punto.

4.3.1. Abstracción de variables

El proceso de abstracción de variables se encarga de sustituir cada término o lista de términos de un `Pair` por una nueva variable fresca y, posteriormente añade una nueva restricción al conjunto de restricciones indicando que esa nueva variable debe ser igual al termino (o lista de términos) sustituido.

Para realizar el proceso anteriormente descrito se utiliza la función `parseInput` (véase listado 4.1). Dicha función recibe el módulo donde se ha descrito la teoría, un conjunto de `PairSet`, un natural (`Nat`) y una lista de substituciones (`Substitution`) y, su función es modificar la entrada recibida (el conjunto `PairSet`) y devolver un `PairSet&PairSet&Counter&Substitution` donde el primer `PairSet` es el término renombrado con las nuevas variable, el segundo `PairSet` contiene el conjunto de restricciones generadas sobre las variables frescas, el tercer valor (`Counter`) es un `Nat` que indica cual es el próximo identificador en caso que se genere una variable nueva. El último valor contiene la lista de sustituciones creada durante la ejecución del algoritmo.

Listing 4.1: Definición función parseInput

```
op parseInput : Module Nat PairSet Substitution -> PairSet&PairSet&Counter&
    Substitution .
```

```
eq  parseInput (M:Module, N:Nat, emptyPairSet, S:Substitution)
        = (emptyPairSet, emptyPairSet, N:Nat, S:Substitution) .
ceq parseInput (M:Module, N:Nat, T1:Term =?= T2:Term ^ PS:PairSet, S:
    Substitution)
        = (T1':Term =?= T2':Term ^ PS3:PairSet, PS2:PairSet ^ PS4:PairSet,
            N3:Nat, S3:Substitution)
if (T1':Term, PS1:PairSet, N1:Nat, S1:Substitution)
        := transform(M:Module,(T1:Term, emptyPairSet, N:Nat, S:Substitution)
            )
/\ (T2':Term, PS2:PairSet, N2:Nat, S2:Substitution)
        := transform(M:Module,(T2:Term, PS1:PairSet, N1:Nat, S1:Substitution
            ))
/\ (PS3:PairSet, PS4:PairSet, N3:Nat, S3:Substitution)
        := parseInput (M:Module, N2:Nat, PS:PairSet, S2:Substitution) .
```

El proceso que se sigue en la función `parseInput` es muy similar al definido en la función `transT` de la Sección 3.3, donde en primer lugar se descompone cada `Pair` en los componentes que lo forman (en este caso `T1:Term =?= T2:Term`) y se delega en la función `transform` la conversión de cada uno de estos términos. Una vez se han convertido estos términos, recursivamente, se vuelve a ejecutar la función `parseInput` sobre el resto del `PairSet` hasta que únicamente quede el `PairSet` vacío (`emptyPairSet`).

La función `transform` (listado 4.2) puede contemplar varios casos. Podría ser que el término recibido sea un operador sobre una lista de términos, `F:QiD[TL:TermList]`, en este caso, se comprueba primero si esa operación es de tipo `Nat` (mediante la función `typeLeq` junto la función `leastSort`) y, si el resultado es afirmativo, se crea una nueva variable que sustituye al término actual (`V:Variable := newVar*(N2:Nat, 'Nat)`) y se añade una restricción al conjunto de restricciones (`V:Variable =?= F:Qid[TL':TermList]`), además se añade la sustitución (`V:Variable <- F:Qid[TL':TermList]`) a la lista de sustituciones. Si el resultado es negativo y no es de tipo `Nat`, se mantiene el símbolo (`QiD`) y se intenta transformar la lista de términos.

Listing 4.2: Función transform - definición

```
op transform : Module Term&PairSet&Counter&Substitution -> Term&PairSet&
    Counter&Substitution .

ceq transform(M:Module, (F:Qid[TL:TermList], PS:PairSet, N:Nat, S:
    Substitution))
        = (V:Variable, PS':PairSet ^ V:Variable =?= F:Qid[TL':TermList], N2:
            Nat + 1,
        (V:Variable <- F:Qid[TL':TermList] ; S1:Substitution))
if typeLeq(M:Module,leastSort(M:Module, F:Qid[TL:TermList]), 'Nat)
/\  (TL':TermList, PS':PairSet, N2:Nat, S1:Substitution)
        := transform(M:Module, (TL:TermList, PS:PairSet, N:Nat, S:
            Substitution))
/\   V:Variable := newVar*(N2:Nat, 'Nat) .

```

```
ceq transform(M:Module, (F:Qid[TL:TermList], PS:PairSet, N:Nat, S:Substitution))
        = (F:Qid[TL':TermList],PS':PairSet,N2:Nat, S1:Substitution)
if not typeLeq(M:Module,leastSort(M:Module, F:Qid[TL:TermList]), 'Nat)
/\ (TL':TermList, PS':PairSet, N2:Nat, S1:Substitution)
        := transform(M:Module, (TL:TermList, PS:PairSet, N:Nat, S:Substitution)) .
```

Los otros casos contemplados en la función `transform` son en el caso que las variables sean `Constant` o `Variable`. En estos casos únicamente se comprueba si la sustitución de dicha constantes o variable existe en la lista. Si existe se reemplaza por la sustitución y si no existe se crea una nueva variable fresca que la sustituye y se añade la sustitución a la lista de sustituciones. En cualquier caso se inserta una nueva restricción al conjunto de restricciones (de la forma `V =?= T`, donde `V` es la `Variable` de la `Susbstitution` generada y `T` es el `Term` reemplazado).

Listing 4.3: Función transform - conversiones de variables y constantes

```
eq transform(M:Module, (V:Variable, PS:PairSet, N:Nat, (V:Variable <- T:Term ; S:Substitution)))
        = (V:Variable, PS:PairSet, N:Nat, (V:Variable <- T:Term ; S:Substitution)) .

eq transform(M:Module, (C:Constant, PS:PairSet, N:Nat, (V:Variable <- C:Constant ; S:Substitution)))
        = (V:Variable, PS:PairSet, N:Nat, (V:Variable <- C:Constant ; S:Substitution)) .

ceq transform(M:Module, (C:Constant, PS:PairSet, N:Nat, S:Substitution))
        = (V2:Variable, PS:PairSet ^ V2:Variable =?= C:Constant,
        N:Nat + 1, (V2:Variable <- C:Constant  ; S:Substitution))
if V2:Variable := newVar*(N:Nat, 'Nat) .

eq transform(M:Module, (V:Variable, PS:PairSet, N:Nat, (V2:Variable <- V:Variable ; S:Substitution)))
        = (V2:Variable, PS:PairSet, N:Nat, (V2:Variable <- V:Variable ; S:Substitution)) .

ceq transform(M:Module, (V:Variable, PS:PairSet, N:Nat, S:Substitution))
        = (V2:Variable, PS:PairSet ^ V2:Variable =?= V:Variable,
        N:Nat + 1, (V2:Variable <- V:Variable  ; S:Substitution))
if V2:Variable := newVar*(N:Nat, 'Nat) .
```

Finalmente, existe un último caso (listado 4.4) en el que el término es un `TermList` (un término seguido de una lista), en este caso el procedimiento es sencillo, pues se envía cada uno de ellos nuevamente a la función `transform` por separado ya que será uno de los casos anteriormente detallados.

Listing 4.4: Función transform - conversión delistas

```
ceq transform(M:Module, ((T:Term , TL:TermList), PS:PairSet, N:Nat, S:
    Substitution))
        = ((TL1:TermList, TL2:TermList), PS2:PairSet, N2:Nat, S2:
            Substitution)
if (TL1:TermList, PS1:PairSet, N1:Nat, S1:Substitution)
        := transform(M:Module, (T:Term, PS:PairSet, N:Nat, S:Substitution))
/\ (TL2:TermList, PS2:PairSet, N2:Nat, S2:Substitution)
        := transform(M:Module, (TL:TermList, PS1:PairSet, N1:Nat, S1:
            Substitution)) .
```

Seguidamente se presentarán algunos ejemplos de conversión

Ejemplos

Dado el siguiente ejemplo

```
's_['0.Zero] =?= 's_^2['0.Zero]
```

Si lo ejecutamos en `Maude` mediante la sentencia

```
red parseInput(upModule('ACNAT,false), 0, 's_['0.Zero] =?= 's_^2['0.Zero]) .
```

El resultado es

```
reduce in METASAT : parseInput(upModule('ACNAT, false), 0, 's_['0.Zero] =?=
    's_^2['0.Zero]) .
rewrites: 28 in 0ms cpu (0ms real) (43818 rewrites/second)
result PairSet&PairSet&Counter: ('#0:Nat =?= '#1:Nat,'#0:Nat =?= 's_['0.Zero
    ] ^
    '#1:Nat =?= 's_^2['0.Zero],2)
```

Donde el `PairSet` reescrito es

```
'#0:Nat =?= '#1:Nat
```

Y el conjunto de restricciones es

```
'#0:Nat =?= 's_['0.Zero] ^ '#1:Nat =?= 's_^2['0.Zero]
```

Otro posible ejemplo sería incluir una variable y una operación en uno de los términos como muestra el ejemplo siguiente

```
'_+_['s_['0.Zero], 'X:Nat] =?= 'Y:Nat
```

El comando para ejecutarlo en `Maude` sería

```
red parseInput(upModule('ACNAT,false), 0, '_+_['s_['0.Zero], 'X:Nat] =?= 'Y:
    Nat) .
```

El resultado obtenido sería

```
reduce in METASAT : parseInput(upModule('ACNAT, false), 0, '_+_['s_['0.Zero
    ],
    'X:Nat] =?= 'Y:Nat) .
rewrites: 26 in 0ms cpu (0ms real) (37626 rewrites/second)
result PairSet&PairSet&Counter: ('#0:Nat =?= '#1:Nat,'#0:Nat =?= '_+_['s_[
    '0.Zero],'X:Nat] ^ '#1:Nat =?= 'Y:Nat,2)
```

Donde el termino resultante sería

```
'#0:Nat =?= '#1:Nat
```

Y el conjunto de restricciones

```
'#0:Nat =?= '_+_['s_['0.Zero],'X:Nat] ^ '#1:Nat =?= 'Y:Nat
```

También se pueden concatenar distintos `Pair` creando `PairSet` más complejos

```
'_+_['s_['0.Zero], 'X:Nat] =?= 'Y:Nat ^
        '_+_['Z:Nat, 'W:Nat] =?= '_+_['s_^3['0.Zero], 's_^2['0.Zero]]
```

La sentencia para ejecutarlo sería

```
red parseInput(upModule('ACNAT,false), 0,
'_+_['s_['0.Zero], 'X:Nat] =?= 'Y:Nat ^
'_+_['Z:Nat, 'W:Nat] =?= '_+_['s_^3['0.Zero], 's_^2['0.Zero]]) .
```

Y el correspondiente resultado

```
rewrites: 46 in 0ms cpu (0ms real) (235897 rewrites/second)
result PairSet&PairSet&Counter: ('#0:Nat =?= '#1:Nat ^ '#2:Nat =?= '#3:Nat,
    '#0:Nat =?= '_+_['Z:Nat,'W:Nat] ^ '#1:Nat =?= '_+_['s_^3['0.Zero],'s_^2[
    '0.Zero]] ^ '#2:Nat =?= '_+_['s_['0.Zero],'X:Nat] ^ '#3:Nat =?= 'Y:Nat
        ,4)
```

Donde el resultado reescrito sería

```
'#0:Nat =?= '#1:Nat ^ '#2:Nat =?= '#3:Nat
```

Y el conjunto de restricciones

```
'#0:Nat =?= '_+_['Z:Nat,'W:Nat] ^
'#1:Nat =?= '_+_['s_^3['0.Zero],'s_^2['0.Zero]] ^
'#2:Nat =?= '_+_['s_['0.Zero],'X:Nat] ^
'#3:Nat =?= 'Y:Nat
```

4.3.2. Combinación de variables

El proceso de combinación de variables intenta encontrar una lista sustituciones entre las variables tal que al aplicar dichas sustituciones el `PairSet` sea cierto y se cumplan todas las restricciones (al enviarlo como `SATProblem` a la primera parte del prototipo, Sección 3). Estas sustituciones se calculan utilizando la función `metaUnify` (Sección 2.2.11).

Este proceso se divide en dos partes:

- Transformar los `PairSet` en `UnificationProblem` para que `metaUnify` pueda tratarlos.
- Llamar a la función `metaUnify` y realizar las sustituciones resultantes sobre las restricciones.

Para realizar la transformación de `PairSet` a `UnificationProblem` se utiliza la función `transformU` (listado 4.5) en la que se sustituye en cada `Pair` el operador `=?=` por `=?`.

Listing 4.5: Función transformU

```
op transformU : Module PairSet -> UnificationProblem .
eq transformU(M:Module, T1:Term =?= T2:Term ^ PS:PairSet) =
        if PS:PairSet == emptyPairSet
        then  T1:Term =? T2:Term
        else T1:Term =? T2:Term /\ transformU(M:Module, PS:PairSet)
        fi .
```

Posteriormente se realiza la llamada a `metaUnify` mediante la función `callUnification` (listado 4.6)

Listing 4.6: Función callUnification

```
***(Llamada a meta unificacion
Nat -> Indice de variables
Nat -> Unificacion
)
op callUnification : Module UnificationProblem Nat Nat -> UnificationPair .
eq callUnification (M:Module, U:UnificationProblem, N:Nat, N2:Nat)
        = metaUnify(M:Module, U:UnificationProblem, N:Nat, N2:Nat) .
```

La llamada que se realiza internamente en el prototipo es la siguiente

```
callUnification(M:Module, transformU(M:Module, getFirstPair(P:PairSet&
    PairSet&Counter)), getCounter(P:PairSet&PairSet&Counter), N:Nat) .
```

Donde en primer lugar se obtiene el primer `PairSet` de un término de tipo `PairSet&PairSet&Counter` y, sobre éste se realiza la conversión a `UnificationProblem`. El contador `getCounter` sirve para indicarle a la función `metaUnify` a partir de qué identificador debe crear variables frescas. El último dato `N:Nat` sirve para indicar que sustitución se desea obtener (la 0, la 1, etc.). La función `callUnification` devuelve un conjunto de `UnificationPair` en caso que haya encontrado sustituciones o `noUnifier` en caso que no haya encontrado ninguna substitución.

Seguidamente se presentarán algunos ejemplos.

Ejemplos

En esta sección se partirá de los ejemplos vistos en la Sección 4.3.1 y se observará cómo se convierten y qué resultados se obtienen. Se ha definido un módulo para la realización de los ejemplos denominada `ACNAT` (listado 4.7).

Listing 4.7: Módulo ACNAT

```
mod ACNAT is
pr NAT .
sorts S Nat2 .
subsort Nat < Nat2 < S .
op _;_ : S S -> S [assoc comm prec 75] .
endm
```

El primer ejemplo es

```
's_['0.Zero] =?= 's_^2['0.Zero]
```

En el que el resultado al aplicar la abstracción de variables es

```
'#0:Nat =?= '#1:Nat
```

Al llamar a la función `callUnification` mediante el comando

```
red callUnification(upModule('ACNAT,false),
        transformU(upModule('ACNAT,false), '#0:Nat =?= '#1:Nat), 2, 0) .
```

Devuelve el `UnificationPair` resultante

```
reduce in METASAT : callUnification(upModule('ACNAT, false), transformU(
    upModule('ACNAT, false), '#0:Nat =?= '#1:Nat), 2, 0) .
rewrites: 6 in 1ms cpu (3ms real) (4373 rewrites/second)
result UnificationPair: {
        '#0:Nat <- '#3:Nat ;
        '#1:Nat <- '#3:Nat,3}
```

Si repetimos el comando con el mismo ejemplo para obtener la siguiente sustitución

```
red callUnification(upModule('ACNAT,false),
        transformU(upModule('ACNAT,false), '#0:Nat =?= '#1:Nat), 2, 1) .
```

Obtenemos que no existen más unificadores

```
reduce in METASAT : callUnification(upModule('ACNAT, false), transformU(
    upModule('ACNAT, false), '#0:Nat =?= '#1:Nat), 2, 1) .
rewrites: 6 in 0ms cpu (0ms real) (~ rewrites/second)
result UnificationPair?: (noUnifier).UnificationPair?
```

Si utilizamos como ejemplo

```
'_+_['s_['0.Zero], 'X:Nat] =?= 'Y:Nat ^
'_+_['Z:Nat, 'W:Nat] =?= '_+_['s_^3['0.Zero], 's_^2['0.Zero]]
```

En el que el resultado era

```
'#0:Nat =?= '#1:Nat ^ '#2:Nat =?= '#3:Nat
```

Si utilizamos el comando anterior

```
red callUnification(upModule('ACNAT,false),
        transformU(upModule('ACNAT,false), '#0:Nat =?= '#1:Nat ^
                '#2:Nat =?= '#3:Nat), 4, 0) .
```

Obtenemos la sustitución

```
reduce in METASAT : callUnification(upModule('ACNAT, false), transformU(
    upModule('ACNAT, false), '#0:Nat =?= '#1:Nat ^ '#2:Nat =?= '#3:Nat), 4,
        0)
    .
rewrites: 9 in 0ms cpu (0ms real) (47872 rewrites/second)
result UnificationPair: {
        '#0:Nat <- '#5:Nat ;
        '#1:Nat <- '#5:Nat ;
        '#2:Nat <- '#6:Nat ;
        '#3:Nat <- '#6:Nat,6}
```

4.3.3. Conversión de `PairSet` a `SATProblem`

La finalidad de este proceso es, a partir del resultado de la Sección 4.3.2, sustituir cada `UnificationPair` sobre el término que corresponda del conjunto de restricciones y crear una petición `SATProblem` para enviar a la primera parte del prototipo (Sección 3).

La sustitución de términos se realiza mediante las funciones `substitutePairs` (listado 4.8) y `substitutePair` (listado 4.9) cuya ejecución es muy simple pues únicamente sustituye cada término de cada `Pair` por la sustitución indicada.

Listing 4.8: Función substitutePairs

```
*** Substitution PairSet
op substitutePairs : PairSet Substitution -> PairSet .
eq substitutePairs(P:PairSet, none) = P:PairSet .
eq substitutePairs(P:PairSet, S:Substitution ; S2:Substitution)
        = substitutePairs(substitutePair(P:PairSet, S:Substitution) , S2:
            Substitution) .
```

Listing 4.9: Función substitutePair

```
op substitutePair : PairSet Substitution -> PairSet .
eq substitutePair(emptyPairSet, S:Substitution) = emptyPairSet .
ceq substitutePair(T1:Term =?= T2:Term ^ P:PairSet, S:Substitution) = T1':
    Term =?= T2':Term ^ P':PairSet
        if T1':Term := T1:Term << S:Substitution
        /\ T2':Term := T2:Term << S:Substitution
        /\ P':PairSet := substitutePair(P:PairSet, S:Substitution) .
```

La conversión de `PairSet` a `SATProblem` también es muy sencilla pues únicamente se sustituye el operador de construcción de `Pair` (`=?=`) por el operador de construcción de `SATPair` (`===`). Esta conversión la realiza la función `createSATRequest` que se puede observar en el listado 4.10.

Listing 4.10: Función createSATRequest

```
*** createSATRequest
*** PairSet -> Pares variable =?= valor
op createSATRequest : PairSet -> SATProblem .
eq createSATRequest(emptyPairSet) = empty .
eq createSATRequest(T1:Term =?= T2:Term ^ P:PairSet) = T1:Term === T2:Term ^
    createSATRequest(P:PairSet) .
```

Finalmente la llamada concreta para crear una petición `SATProblem` sería

```
red createSATRequest(substitutePairs(P:PairSet, getSubst(U:UnificationPair))
    .
```

Donde `P:PairSet` es el conjunto de restricción y `getSubst(U:UnificationPair)` es la lista de sustituciones obtenida mediante la función `callUnification`.

Al igual que las secciones anteriores, a continuación se presentan unos ejemplos.

Ejemplos

En esta sección se usan los resultados de los ejemplos de la Sección 4.3.2.

Para el ejemplo

```
's_['0.Zero] =?= 's_^2['0.Zero]
```

Se había obtenido el `UnificationPair`

```
reduce in METASAT : callUnification(upModule('ACNAT, false), transformU(
    upModule('ACNAT, false), '#0:Nat =?= '#1:Nat), 2, 0) .
rewrites: 6 in 1ms cpu (3ms real) (4373 rewrites/second)
result UnificationPair: {
        '#0:Nat <- '#3:Nat ;
        '#1:Nat <- '#3:Nat,3}
```

Si se llama a la función `substitutePairs` sobre las restricciones obtenidas en el proceso de abstracción (Sección 4.3.1) con estas sustituciones

```
red substitutePairs('#0:Nat =?= 's_['0.Zero] ^ '#1:Nat =?= 's_^2['0.Zero],
        '#0:Nat <- '#3:Nat ; '#1:Nat <- '#3:Nat) .
```

Se obtiene el `PairSet` resultante

```
reduce in METASAT : substitutePairs('#0:Nat =?= 's_['0.Zero] ^ '#1:Nat =?=
    's_^2['0.Zero],
        '#0:Nat <- '#3:Nat ;
        '#1:Nat <- '#3:Nat) .
rewrites: 21 in 0ms cpu (0ms real) (~ rewrites/second)
result PairSet: '#3:Nat =?= 's_['0.Zero] ^ '#3:Nat =?= 's_^2['0.Zero]
```

Si se le pasa este `PairSet` a la función `createSATRequest`

```
red createSATRequest('#3:Nat =?= 's_['0.Zero] ^ '#3:Nat =?= 's_^2['0.Zero])
    .
```

Se obtiene el siguiente `SATProblem` listo para ser enviado a la interfaz

```
reduce in METASAT : createSATRequest('#3:Nat =?= 's_['0.Zero] ^ '#3:Nat =?=
    's_^2['0.Zero]) .
rewrites: 3 in 0ms cpu (0ms real) (~ rewrites/second)
result SATProblem: '#3:Nat === 's_['0.Zero] ^ '#3:Nat === 's_^2['0.Zero]
```

Para el caso del ejemplo

```
'_+_['s_['0.Zero], 'X:Nat] =?= 'Y:Nat ^
'_+_['Z:Nat, 'W:Nat] =?= '_+_['s_^3['0.Zero], 's_^2['0.Zero]]
```

El `UnificationPair` obtenido era

```
reduce in METASAT : callUnification(upModule('ACNAT, false), transformU(
    upModule('ACNAT, false), '#0:Nat =?= '#1:Nat ^ '#2:Nat =?= '#3:Nat), 4,
        0)
    .
rewrites: 9 in 0ms cpu (0ms real) (47872 rewrites/second)
result UnificationPair: {
        '#0:Nat <- '#5:Nat ;
        '#1:Nat <- '#5:Nat ;
        '#2:Nat <- '#6:Nat ;
        '#3:Nat <- '#6:Nat,6}
```

Al llamar a la función `substitutePairs` sobre las restricciones obtenidas en el proceso de abstracción (Sección 4.3.1) con el anterior resultado se obtiene el `PairSet`

```
red substitutePairs('#0:Nat =?= '_+_['Z:Nat,'W:Nat] ^
        '#1:Nat =?= '_+_['s_^3['0.Zero],'s_^2['0.Zero]] ^
        '#2:Nat =?= '_+_['s_['0.Zero],'X:Nat] ^
        '#3:Nat =?= 'Y:Nat,
        '#0:Nat <- '#5:Nat ; '#1:Nat <- '#5:Nat ;
        '#2:Nat <- '#6:Nat ; '#3:Nat <- '#6:Nat ) .
```

Al llamar a la función `substitutePairs` sobre las restricciones obtenidas en el proceso de abstracción (Sección 4.3.1) con el anterior resultado

```
reduce in METASAT : substitutePairs('#0:Nat =?= '_+_['Z:Nat,'W:Nat] ^ '#1:
    Nat
    =?= '_+_['s_^3['0.Zero],'s_^2['0.Zero]] ^ '#2:Nat =?= '_+_['s_['0.Zero],
    'X:Nat] ^ '#3:Nat =?= 'Y:Nat,
        '#0:Nat <- '#5:Nat ;
        '#1:Nat <- '#5:Nat ;
        '#2:Nat <- '#6:Nat ;
        '#3:Nat <- '#6:Nat) .
rewrites: 105 in 0ms cpu (0ms real) (889830 rewrites/second)
result PairSet: '#5:Nat =?= '_+_['Z:Nat,'W:Nat] ^ '#5:Nat =?= '_+_['s_^3[
    '0.Zero],'s_^2['0.Zero]] ^ '#6:Nat =?= 'Y:Nat ^ '#6:Nat =?= '_+_['s_[
    '0.Zero],'X:Nat]
```

Finalmente, al llamar a la función `createSATRequest` mediante

```
red createSATRequest( '#5:Nat =?= '_+_['Z:Nat,'W:Nat] ^
```

```
        '#5:Nat =?= '_+_['s_^3['0.Zero],'s_^2['0.Zero]] ^
        '#6:Nat =?= 'Y:Nat ^
        '#6:Nat =?= '_+_['s_['0.Zero],'X:Nat]) .
```

Se obtiene el `SATProblem`

```
reduce in METASAT : createSATRequest('#5:Nat =?= '_+_['Z:Nat,'W:Nat] ^ '#5:
    Nat
    =?= '_+_['s_^3['0.Zero],'s_^2['0.Zero]] ^ '#6:Nat =?= 'Y:Nat ^ '#6:Nat
        =?=
    '_+_['s_['0.Zero],'X:Nat]) .
rewrites: 5 in 0ms cpu (0ms real) (~ rewrites/second)
result SATProblem: '#5:Nat === '_+_['Z:Nat,'W:Nat] ^ '#5:Nat === '_+_['s_^3[
    '0.Zero],'s_^2['0.Zero]] ^ '#6:Nat === 'Y:Nat ^ '#6:Nat === '_+_['s_[
    '0.Zero],'X:Nat]
```

4.3.4. Flujo de ejecución del prototipo

El punto de entrada del prototipo es la función `metasat-trans` (listado 4.11) y únicamente realiza una llamada a la función `parseInput` (explicada en la Sección 4.3.1) y envía el resultado a la función más importante del prototipo, `executesat`.

Listing 4.11: Función metasat-trans

```
op metasat-trans : Module PairSet Nat -> Bool  .
ceq metasat-trans(M:Module, P:PairSet, N:Nat)
        = executesat(M:Module,(PS:PairSet, PS1:PairSet, N1:Nat), 0)
        if (PS:PairSet, PS1:PairSet, N1:Nat, S1:Substitution)
                := parseInput(M:Module, N:Nat, P:PairSet, none) .
```

La función `executesat` (listado 4.12) actúa a modo de bucle (cada iteración se ha modelado mediante la función `SATIteration`, listado 4.13). Cada `SATIteration` ejecuta el proceso que se ha realizado en la Sección 4.3.3, es decir, realiza las sustituciones obtenidas de la función `callUnification` sobre el conjunto de restricciones, crea la petición `SATProblem` y realiza la llamada sobre la función del prototipo `metasat-interface` (Sección 3.3). Una vez se ha ejecutado la función `SATIteration`, se devuelve un `Bool` que se mapea a `Boole` (Sección 4.2). Los posibles casos son los siguientes:

- `SATIteration` devuelve `true`: Finaliza la ejecución del algoritmo y devuelve `true`.

- `SATIteration` devuelve `false`: Se realiza recursivamente una nueva llamada a la función `executesat` incremententando el `N:Nat` final en una unidad. Esto significa que se llamará a la función `callUnification` para que obtenga el siguiente `UnificationPair`, en caso que exista.

- `SATIteration` devuelve `error`: No ha podido ejecutarse la función `SATIteration` porque la función `callUnification` no ha encontrado más `UnificationPair` y el resultado de `metasat-interface` devuelve `false`, se aborta la ejecución del algoritmo y se devuelve `false`.

Listing 4.12: Función executesat

```
*** Execute SAT
*** Nat -> Numero de unificacion
op executesat : Module PairSet&PairSet&Counter Nat -> Bool .
ceq executesat(M:Module, P:PairSet&PairSet&Counter, N:Nat)
                = if B:Boole == true
                        then true
                else if B:Boole == error
                                then false
                        else executesat(M:Module, P:PairSet&PairSet&Counter,
                             N:Nat + 1)
                        fi
                fi
                if B:Boole := SATIteration(M:Module, getSecondPair(P:PairSet
                    &PairSet&Counter),
                                callUnification(M:Module, transformU(M:
                                    Module,
                                        getFirstPair(P:PairSet&PairSet&
                                            Counter)),
                                        getCounter(P:PairSet&PairSet&Counter
                                            ), N:Nat)) .  d
```

Listing 4.13: Función SATIteration

```
op SATIteration : Module PairSet UnificationPair -> Boole .
ceq SATIteration (M:Module, P:PairSet, noUnifier)
        = if B:Bool == true then true
                else error
                fi
        if B:Bool := metasat-interface(M:Module, createSATRequest(P:PairSet)
            ) .
eq SATIteration (M:Module, P:PairSet, U:UnificationPair)
                = metasat-interface(M:Module,
                        createSATRequest(substitutePairs(P:PairSet, getSubst
                            (U:UnificationPair)))) .
```

A continuación se detallan algunos ejemplos.

Ejemplos

Si se toma el ejemplo de la Sección 4.3.3

```
'_+_['s_['0.Zero], 'X:Nat] =?= 'Y:Nat ^
'_+_['Z:Nat, 'W:Nat] =?= '_+_['s_^3['0.Zero], 's_^2['0.Zero]]
```

Del que se había obtenido el siguiente `SATProblem`

```
'#5:Nat === '_+_['Z:Nat,'W:Nat]
        ^ '#5:Nat === '_+_['s_^3['0.Zero],'s_^2['0.Zero]]
    ^ '#6:Nat === 'Y:Nat
    ^ '#6:Nat === '_+_['s_['0.Zero],'X:Nat]
```

Y se ejecuta la función `metasat-trans`

```
red metasat-trans(upModule('ACNAT,false),
        '_+_['s_['0.Zero], 'X:Nat] =?= 'Y:Nat ^
        '_+_['Z:Nat, 'W:Nat] =?= '_+_['s_^3['0.Zero], 's_^2['0.Zero]], 0 ) .
```

Se obtiene el siguiente resultado

```
reduce in METASAT : metasat-trans(upModule('ACNAT, false), '_+_['Z:Nat,'W:
    Nat]
    =?= '_+_['s_^3['0.Zero],'s_^2['0.Zero]] ^ '_+_['s_['0.Zero],'X:Nat] =?=
    'Y:Nat, 0) .
rewrites: 681 in 13ms cpu (49ms real) (50662 rewrites/second)
result Bool: true
```

Otro ejemplo sería el siguiente

```
's_['0.Zero] =?= 's_^2['0.Zero]
```

Donde el `SATProblem` resultante era

```
reduce in METASAT : createSATRequest('#3:Nat =?= 's_['0.Zero] ^ '#3:Nat =?=
    's_^2['0.Zero]) .
rewrites: 3 in 0ms cpu (0ms real) (~ rewrites/second)
result SATProblem: '#3:Nat === 's_['0.Zero] ^ '#3:Nat === 's_^2['0.Zero]
```

Al ejecutar la función `metasat-trans`

```
red metasat-trans(upModule('ACNAT,false),
        's_['0.Zero] =?= 's_^2['0.Zero], 0 ) .
```

Se obtiene el resultado

```
reduce in METASAT : metasat-trans(upModule('ACNAT, false), 's_['0.Zero] =?=
    's_^2['0.Zero], 0) .
rewrites: 212 in 2ms cpu (3ms real) (73204 rewrites/second)
result Bool: false
```

5

Prototipo - Tercera parte

En esta tercera parte se extiende el procedimiento de decisión de la satisfacibilidad de una conjunción de igualdades para la aritmética Presburger para cualquier teoría ecuacional, no sólo con símbolos y propiedades ecuaciones extra sino también incluyendo ecuaciones (orientadas como reglas). El procedimiento de decisión con reglas extras se realiza a través del estrechamiento ecuacional. Para esta parte, se define el concepto de "*estado*", donde cada estado es un conjunto de pares formado por los `Pair` que forman la ecuación junto con los `Pair` que forman las restricciones de dicha ecuación. La idea fundamental de esta fase es la generación de estados en forma de árbol y averiguar si el conjunto de pares y restricciones se satisface con alguna sustitución obtenida mediante *narrowing* sobre operaciones definidas en un módulo definido por el usuario. Este prototipo se ha desarrollado como un *modulo de sistema* de **Maude** denominado `METASAT`.

5.1. Satisfacibilidad de igualdades con reglas extra

En esta sección, partimos de la teoría de primer orden para la aritmética Presburger $\mathbb{N} = (\mathcal{N}, +_{\mathbb{N}}, 0_{\mathbb{N}}, 1_{\mathbb{N}}, >_{\mathbb{N}})$ definida en la Sección 3.1 y asumimos una teoría ecuacional $(\Sigma_{\mathbb{N}}, E_{\mathbb{N}}, R_{\mathbb{N}})$ para tipos ordenados basada en un tipo especial `Nat` que es una descomposición de la aritmética Presburger $\mathbb{N}$ como se indica también en la Sección 3.1.

A continuación, extendemos el concepto de una extensión válida de la aritmética Presburger para que admita nuevas reglas ecuacionales. Dada una teoría ecuacional para tipos ordenados (Σ, E, R), decimos que es una *teoría extendida* de la aritmética Presburger si cumple las siguiente condiciones:

1. se añaden más símbolos y tipos de datos, es decir, $\Sigma = \Sigma_{\mathbb{N}} \uplus \Sigma'$ donde el conjunto de tipos S asociado a Σ contiene el único tipo Nat permitido en $\Sigma_{\mathbb{N}}$,
2. se añaden más propiedades ecuacionales, es decir, $E = E_{\mathbb{N}} \uplus E'$,
3. se añaden más reglas ecuacionales, es decir, $R = R_{\mathbb{N}} \uplus R'$ tal que R es confluente, terminante y decreciente en tipo modulo E,
4. las reglas R' no contienen símbolos reducibles de $R_{\mathbb{N}}$, es decir, para toda regla $l \to r \in R'$ y para toda sustitución $\sigma : Vars(l) \to \mathcal{T}_{\Sigma,\mathsf{Nat}}$ tal que σ es $R_{\mathbb{N}}, E_{\mathbb{N}}$-irreducible, $l\sigma$ es también $R_{\mathbb{N}}, E_{\mathbb{N}}$-irreducible.
5. la teoría ecuacional protege el algebra inicial de los naturales, es decir, para cada Σ-término t sin variables del tipo Nat, existe un Σ-término t_0 sin variables del tipo Nat tal que $t{\downarrow}_{R,E} =_{E_{\mathbb{N}}} t_0$.

Dada una teoría ecuacional (Σ, E, R) que sea una teoría extendida de la aritmética Presburger $(\Sigma_{\mathbb{N}}, E_{\mathbb{N}}, R_{\mathbb{N}})$ y una conjunción de igualdades $C = \{u_1 = v_1 \wedge \cdots \wedge u_k = v_k\}$ donde los términos $u_1, v_1, \ldots, u_k, v_k$ son términos con variables de $\mathcal{T}_{\Sigma}(\mathcal{X})$, la conjunción C es satisfacible si y sólo si existe una sustitución $\sigma : \mathcal{X}_C \to \mathcal{T}_{\Sigma}$, $\mathcal{X}_C = Vars(\bigwedge_i u_i = v_i)$, tal que para cada i, $(u_i\sigma){\downarrow}_{R,E} =_E (v_i\sigma){\downarrow}_{R,E}$.

En este caso no conseguimos un algoritmo de decisión sobre la satisfacibilidad de una conjunción de igualdades, sólo un algoritmo de semi-decisión, ya que el conjunto de reglas puede ser arbitrario y el uso de narrowing con unificación con SAT-Solver puede no terminar. Queda fuera de esta tesina el estudio de condiciones bajo las cuales existe un algoritmo de decisión. Aunque la condición del *finite variant property* [7] para las reglas extra es una condición suficiente.

El algoritmo asociado al proceso de satisfacibilidad se basa en un conjunto de estados que vamos transformando. Dada la teoría ecuacional (Σ, E, R) donde $R = R_{\mathbb{N}} \uplus R'$, el conjunto de símbolos definidos por las reglas de R, $Def(R) \subseteq \Sigma$, se define como $Def(R) = \{f \in \Sigma \mid f(t_1, \ldots, t_n) \to t \in R\}$.

Un estado se define de la forma $C \mid C'$ donde C y C' son conjunciones de igualdades. Existen dos tipos de estados: *purificados* y *no purificados*. En un estado purificado se guardan en C las igualdades que no tienen ocurrencias de $Def(R)$ y en C' se guardan igualdades de la forma $X = t$ donde t es un término encabezado por un símbolo de $Def(R)$. En un estado no purificado, esas restricciones sobre C y C' no se cumplen. El algoritmo se describe en las siguientes fases:

1. Seleccionamos un estado no purificado $C \mid C'$ del conjunto de estados St y lo eliminamos de St. Si no se puede porque el conjunto St está vacío, se termina el algoritmo y se devuelve falso (es decir, no satisfacible).
2. Se eliminan de C todas aquellas igualdades que sean triviales, es decir, todas las igualdades de la forma $X = t$ (ó $t = X$) tal que la variable X no aparezca en ninguna otra igualdad de C y de C'. El estado resultante se denomina $C_A \mid C'$.
3. Procedemos a la purificación de $C_A \mid C'$ como sigue. Del conjunto $C_A = \{u_1 = v_1 \wedge \cdots \wedge u_k = v_k\}$ se generan dos conjuntos $\widetilde{C_A} = \{\widetilde{u_1} = \widetilde{v_1} \wedge \cdots \wedge \widetilde{u_k} = \widetilde{v_k}\}$ y $C'_A = \{X_1 = t_1 \wedge \cdots \wedge X_n = t_n\}$ donde $X_1, \ldots, X_n$ son variables frescas que no aparezcan en C_A ni en C', $\widetilde{u_1}, \widetilde{v_1}, \ldots, \widetilde{u_k}, \widetilde{v_k}$ no contienen ningún símbolo de $Defs(R)$, y para todo $1 \leq i \leq n$, el término t_i está encabezado por un símbolo en $Defs(R)$ y, además, existe un índice $1 \leq j \leq k$ y una posición $p_{i,j}$ tal que $u_i|_{p_{i,j}} = t_i$ y $\widehat{u_i}|_{p_{i,j}} = X_i$ ó $v_i|_{p_{i,j}} = t_i$ y $\widehat{v_i}|_{p_{i,j}} = X_i$. El estado resultante de esta fase es $\widetilde{C_A} \mid C' \wedge C'_A$.
4. Comprobamos la satisfacibilidad del conjunto $\widetilde{C_A}$ usando el método del Capítulo 4.
   a) Si no es satisfacible, se desecha el estado $\widetilde{C_A} \mid C' \wedge C'_A$ y se vuelve al punto 1.
   b) Si es satisfacible y el conjunto C'_A está vacío, es decir, no se realizó ninguna purificación, se termina el algoritmo y se devuelve cierto (es decir, se ha encontrado una solución y el conjunto inicial de igualdades es satisfacible).
   c) Si es satisfacible y el conjunto C'_A no está vacío, se procede con el siguiente punto del algoritmo.
5. Procedemos a dar un paso de estrechamiento en todas las igualdades de $C' \wedge C'_A$ como sigue. Para cada igualdad de la forma $X = t$ (ó $t = X$)

en $C' \wedge C'_A$, damos un paso de narrowing usando las reglas extras, es decir, dado $R = R_{\mathbb{N}} \uplus R'$, computamos todos los resultantes de t, $Step(t)_{R',E} = \{\langle \sigma, t' \rangle \mid t \rightsquigarrow_{\sigma,R',E} t'\}$.

6. Finalmente, dado el conjunto $Steps = \{\langle \sigma, X = t' \rangle \mid X = t \in (C \wedge C'_A), \langle \sigma, t' \rangle \in Step(t)_{R',E}\}$, añadimos los siguientes estados al conjunto de estados St, es decir, $St' = St \cup \{(C\sigma \wedge X = t' \mid (C' - \{X = t\})\sigma \wedge (C'_A - \{X = t\})\sigma) \mid \langle \sigma, X = t' \rangle \in Steps\}$.

7. Se vuelve al punto 1.

La corrección y completitud de este algoritmo es sencilla de demostrar, aunque no se ha incluído en esta tesina. El algoritmo no es terminante en general, aunque existen casos donde sí terminaría. El estudio de terminación del algoritmo queda fuera de esta tesina.

5.2. Tipos de datos

Como se ha comentado con anterioridad, en este prototipo se define el concepto de *estado* mediante el tipo `PairSet&PairSet&Counter` donde el primer `PairSet` es la conjunto principal, el segundo `PairSet` es el conjunto de restricciones y `Counter` es un `Nat` que indica el próximo identificador (al igual que en tipos anteriores) que debe tener la próxima variable fresca que se genere. Este *estado* se puede concatenar mediante el operador `_;_` formando `StateSet` (conjunto de estados). También se ha definido un tipo de datos `PairSet&PairSet&Counter&PairSet` que se utiliza para la generación de estados. Los datos de los que se compone este tipo son los mismos que los del tipo `PairSet&PairSet&Counter` pero se añade un `PairSet` adicional que contiene las reglas que faltan por procesar. Estas definiciones de tipos se pueden observar en el listado 5.1.

Listing 5.1: Definición de tipos

```
sorts StateSet .
subsort PairSet&PairSet&Counter < StateSet .
sort PairSet&PairSet&Counter&PairSet .
subsort PairSet&PairSet&Counter&PairSet < StateSet .
op emptyState : -> StateSet .
op _;_ : StateSet StateSet -> StateSet [assoc comm id: emptyState prec 73] .
```

5.3. Ejecución del algoritmo

La ejecución de este prototipo se puede dividir en varias partes

- Limpieza trivial de problemas: Se elimina la ecuación en la que aparece únicamente una variable y, ésta no aparece en ninguna ecuación más.
- Expansión del conjunto `PairSet` eliminado ocurrencias de operaciones no conocidas y generando el conjunto de restricciones sobre dicho conjunto.
- Obtener un conjunto de sustituciones sobre las operaciones anteriores utilizando *narrowing*.
- Generación de estados formados por la combinación de las sustituciones encontradas.
- Búsqueda de un estado que satisface tanto la ecuación como sus restricciones.

5.3.1. Expansión de ecuaciones

Este proceso se encarga de encontrar las ocurrencias de ecuaciones definidas en un módulo que aparecen en el conjunto principal de `Pair`, sustituirlas por variables frescas y añadir un nuevo `Pair` al conjunto de restricciones que contiene la sustitución en formato `variable =?= datos_sustituido`. Este proceso es muy similar al realizado en la Sección 4.3.1. En la literatura también es conocido como proceso de "*purificación*".

Este proceso se realiza mediante la función `expand` (listado 5.2). Esta función recibe el módulo que contiene la teoría, el conjunto de ecuaciones de dicho módulo (`EquationSet`), el conjunto de reglas del mismo módulo (`RuleSet`), un `Nat` indicando identificador de variables y un `PairSet` con el conjunto a expandir. El procedimiento seguido es muy similar al realizado en funciones anteriores como `transT` (Sección 3.3) o `transform` (Sección 4.3.1). En primer lugar se descompone cada `Pair` en sus componentes `T1:Term =?= T2:Term` y se delega en la función `convert` para expandir cada uno de los dos términos. El resto del `PairSet` se convierte al llamar a la función `expand` con el `PairSet` restante (en este caso `PS:PairSet`).

Listing 5.2: Función expand

```
***(
Expand
TemList -. Lista de cabeceras de funcion
)
op  expand : Module EquationSet RuleSet Nat PairSet -> PairSet&PairSet&Counter .
eq  expand (M:Module, O:EquationSet, R:RuleSet, N:Nat, emptyPairSet)
        = (emptyPairSet, emptyPairSet, N:Nat) .
ceq expand (M:Module, O:EquationSet, R:RuleSet, N:Nat, T1:Term =?= T2:Term ^ PS:PairSet)
        = (T1':Term =?= T2':Term ^ PS3:PairSet, PS1:PairSet ^ PS2:PairSet ^ PS4:PairSet, N3:Nat)
        if (T1':Term, PS1:PairSet, N1:Nat)
                := convert(M:Module, (T1:Term, emptyPairSet, N:Nat),  O:EquationSet, R:RuleSet)
        /\ (T2':Term, PS2:PairSet, N2:Nat)
                := convert(M:Module, (T2:Term, emptyPairSet, N1:Nat),  O:EquationSet, R:RuleSet)
        /\ (PS3:PairSet, PS4:PairSet, N3:Nat)
                := expand (M:Module, O:EquationSet, R:RuleSet, N2:Nat, PS:PairSet) .
```

Del mismo modo se ha definido una función denominada `canExpand` (listado 5.3) que comprueba si un determinado conjunto `PairSet` puede expandirse (devolviendo `true` o `false` segun el caso). Los argumentos son los mismos que la función `expand`.

Listing 5.3: Función canExpand

```
op canExpand : Module EquationSet RuleSet Nat PairSet -> Bool .
eq canExpand (M:Module, O:EquationSet, R:RuleSet, N:Nat, emptyPairSet) = false .
ceq canExpand (M:Module, O:EquationSet, R:RuleSet, N:Nat, T1:Term =?= T2:Term ^ PS:PairSet)
        = B:Bool or B1:Bool or B2:Bool
        if B:Bool := containSymbol(T1:Term, O:EquationSet, R:RuleSet)
        /\ B1:Bool := containSymbol(T2:Term, O:EquationSet, R:RuleSet)
        /\ B2:Bool := canExpand(M:Module, O:EquationSet, R:RuleSet, N:Nat, PS:PairSet) .
```

La anterior función `canExpand` trabaja conjuntamente con la función `reduceTrivialProblem` (listado 5.4) que comprueba que, en el caso que haya una única variable a uno de los lados de un operador `Pair` (`=?=`), si existe una ocurrencia de dicha variable en alguna otra ecuación. Sino existe ninguna ocurrencia de dicha variable se reemplaza la ecuación en la que aparece por una igualdad siempre cierta (`'0.Zero =?= '0.Zero`). Si dicha variable aparece en otra ecuación, no se modifica el conjunto `PairSet`.

Listing 5.4: Función reduceTrivialProblem

```
op reduceTrivialProblem : PairSet -> PairSet .
eq reduceTrivialProblem(emptyPairSet) = emptyPairSet .
ceq reduceTrivialProblem (T1:Term =?= V:Variable ^ P:PairSet) =
  '0.Zero =?= '0.Zero ^ reduceTrivialProblem(P:PairSet)
  if not VariableIn(V:Variable, P:PairSet) .

ceq reduceTrivialProblem (V:Variable =?= T1:Term ^ P:PairSet) =
  '0.Zero =?= '0.Zero ^ reduceTrivialProblem(P:PairSet)
   if not VariableIn(V:Variable, P:PairSet) .

eq reduceTrivialProblem(P:PairSet) = P:PairSet [owise] .
```

La función `reduceTrivialProblem` se apoya sobre la función `VariableIn` (listado 5.5) que comprueba si una variable esta contenida en un `PairSet`.

Listing 5.5: Función VariableIn

```
op VariableIn : Variable PairSet -> Bool .
eq VariableIn(V:Variable, emptyPairSet) = false .
eq VariableIn(V:Variable, V:Variable =?= T2:Term ^ P:PairSet) = true .
eq VariableIn(V:Variable, T:Term =?= T2:Term ^ P:PairSet)
   = VariableInTerm(V:Variable, T:Term)
   or-else VariableInTerm(V:Variable, T2:Term)
   or-else VariableIn(V:Variable, P:PairSet) [owise] .
```

Finalmente, la función `VariableInTerm` (listado 5.6) es la que realmente comprueba si una variable existe en un término o lista de términos.

Listing 5.6: Función VariableInTerm

```
op VariableInTerm : Variable Term -> Bool .
eq VariableInTerm(V:Variable, V':Variable) = if V:Variable == V':Variable
    then true  else false fi .
eq VariableInTerm(V:Variable, empty) = false .
eq VariableInTerm(V:Variable, C:Constant) = false .
eq VariableInTerm(V:Variable, F:Qid[TL':TermList]) = VariableInTerm(V:
    Variable, TL':TermList) .
eq VariableInTerm(V:Variable, (T:Term, TL:TermList))
  = VariableInTerm(V:Variable, T:Term) or-else VariableInTerm(V:Variable, TL
      :TermList) .

eq VariableInTerm(V:Variable, T:Term) = false [owise] .
```

La función `convert` (listado 5.7) recibe un `Term&PairSet&Counter` (visto en la Sección 4.2) junto con el módulo y el conjunto de ecuaciones y reglas y, si el término (`T:Term`) contiene alguna regla (función `containRls`) entonces se llama a la función `replaceTermRls` que los sustituye. Aunque la función `convert` puede también reemplazar ecuaciones se ha eliminado (*comentado*) dicha opción pues queda fuera del ámbito de la presente teoría. En un futuro podría realizarse también una sustitución de ecuaciones, por este motivo en la presente memoria no se detallarán las funciones de sustitución de ecuaciones sino únicamente las que sustituyen reglas.

Listing 5.7: Función convert

```
***(
Convert
TemList -. Lista de cabeceras de funcion
)
op convert : Module Term&PairSet&Counter  EquationSet RuleSet -> Term&PairSet&Counter .
eq convert(M:Module, (empty, PS:PairSet, N:Nat),  E:EquationSet, R:RuleSet) = (empty, PS:PairSet, N:Nat) .
eq convert(M:Module, (T:Term, PS:PairSet, N:Nat),  E:EquationSet, R:RuleSet)
        = if containRls(T:Term, R:RuleSet) == true
                then replaceTermRls(M:Module, (T:Term, PS:PairSet, N:Nat), R:RuleSet)
        else
                (T:Term, PS:PairSet, N:Nat)
                ***( Codigo para reemplazar ecuaciones
                if containEqs(T:Term, E:EquationSet) == true
                then replaceTermEqs(M:Module, (T:Term, PS:PairSet, N:Nat), E:EquationSet)
                else
                (T:Term, PS:PairSet, N:Nat)
                fi) --- fin comentario
        fi .
```

La función `containRls` (listado 5.8) se ejecuta junto a la función `containRl` (listado 5.9) y se encargan de comprobar si en un término hay alguna ocurrencia de alguna regla definida en un conjunto `RuleSet`. Su ejecución es el siguiente, para cada regla `containRls` delega en `containRl` para que ésta última compruebe si existe dicha regla en el término dado. Si el resultado es `true` entonces `containRls` devuelve `true` pero si el resultado es `false` entonces `containRls` sigue iterando sobre el conjunto restante de reglas.

Listing 5.8: Función containRls

```
***(
Comprueba si el termino contiene algun QID de alguna regla
)
op containRls : Term RuleSet -> Bool .
eq containRls(T:Term, none) = false .
ceq containRls(T:Term, (rl F':Qid[TL:TermList] => T':Term [Ats:AttrSet] .) R:RuleSet)
        = B:Bool or B1:Bool
        if B:Bool := containRl(T:Term, (rl F':Qid[TL:TermList] => T':Term [Ats:AttrSet] .))
         /\ B1:Bool := containRls(T:Term, R:RuleSet) .
```

Listing 5.9: Función containRl

```
***(
Comprueba si los terminos y subterminos tienen el mismo qid que una regla
    dada
)
op containRl :  Term Rule -> Bool .
eq containRl(empty, E:Rule) = false .
eq containRl(F:Qid[TL':TermList], rl F':Qid[TL:TermList] => T':Term [Ats:
    AttrSet] . )
        =if F:Qid == F':Qid
                        then true
        else containRl(TL':TermList, rl F':Qid[TL:TermList] => T':Term [Ats:
            AttrSet] . ) fi .
eq containRl((F:Qid[TL':TermList] , TL:TermList), E:Rule)
        = containRl(F:Qid[TL':TermList], E:Rule) or containRl(TL:TermList, E
            :Rule) .
eq containRl((T:Term , TL:TermList), E:Rule)
        = containRl(TL:TermList, E:Rule) .
eq containRl(T:Term, E:Rule) = false [owise] .
```

Para reemplazar las ocurrencias de las reglas sobre un término se ha definido la función `replaceTermRls` (listado 5.10), en ella se comprueba en primer lugar (utilizando la función `containRls`) si existe alguna ocurrencia de la regla actual. Si el resultado es `true` entonces se llama a la función `replaceTermRl` para que reemplace dicha regla. Si el resultado es `false` entonces recursivamente se vuelve a llamar a la función `replaceTermRls` con el resto de reglas hasta que no haya ninguna (`none`).

Listing 5.10: Función replaceTermRls

```
***(
Reemplaza un termino por una variable si coincide con alguna regla (sobre un
     conjunto)
)
op replaceTermRls : Module Term&PairSet&Counter  RuleSet -> Term&PairSet&
    Counter .
eq replaceTermRls(M:Module, (T:Term, PS:PairSet, N:Nat), none) = (T:Term, PS
    :PairSet, N:Nat) .
eq replaceTermRls(M:Module, (T:Term, PS:PairSet, N:Nat), (rl F':Qid[TL:
    TermList] => T':Term [Ats:AttrSet] .) R:RuleSet )
        = if containRl(T:Term, rl F':Qid[TL:TermList] => T':Term [Ats:
            AttrSet] .) == true
        then replaceTermRl(M:Module, (T:Term, PS:PairSet, N:Nat), rl F':Qid[
            TL:TermList] => T':Term [Ats:AttrSet] .)
        else replaceTermRls(M:Module, (T:Term, PS:PairSet, N:Nat), R:RuleSet
             ) fi .
```

La función `replaceTermRl` (listado 5.11) recibe un `Term&PairSet&Counter` con el término a expandir y una regla (`Rule`) y según la composición del término pueden darse varios casos de expansión. El primero de ellos se da en el caso que el término sea una operación con un `Qid` (en concreto

F:Qid[TL':TermList]) entonces, en primer lugar se comprueba si ese Qid es el mismo que el Qid de la regla (F':Qid) (al mismo tiempo se intenta convertir la lista de términos TL':TermList dando como resultado la lista de términos T1:TermList). Si son el mismo, entonces se crea una variable nueva (método similar al visto en la Sección 4.3.1), se sustituye todo el término por dicha variable y se añade una nueva restricción al conjunto de restricciones ((createVariableT(M:Module, F:Qid[T1:TermList], N1:Nat) =?= F:Qid[T1:TermList])). Si los Qid no son iguales se devuelve el mismo Qid con la lista de términos (posiblemente) convertida (T1:TermList).

Listing 5.11: Función replaceTermRl

```
***(
Reemplaza un termino por una variable si coincide con una regla
)
op replaceTermRl : Module Term&PairSet&Counter  Rule -> Term&PairSet&Counter .
ceq replaceTermRl(M:Module, (F:Qid[TL':TermList], PS:PairSet, N:Nat),
        rl F':Qid[TL:TermList] => T':Term [Ats:AttrSet] .)
        = if F:Qid == F':Qid
                then (createVariableT(M:Module, F:Qid[T1:TermList], N1:Nat),
                        PS1:PairSet ^ (createVariableT(M:Module, F:Qid[T1:TermList], N1:Nat)
                        =?= F:Qid[T1:TermList]), N1:Nat + 1)
        else (F:Qid[T1:TermList], PS1:PairSet, N1:Nat) fi
        if (T1:TermList, PS1:PairSet, N1:Nat)
                := replaceTermRl(M:Module, (TL':TermList, PS:PairSet, N:Nat),
                        rl F':Qid[TL:TermList] => T':Term [Ats:AttrSet] .) .
```

Otro de los posibles casos de la función replaceTermRl es que el término sea una un operador junto a una lista de términos ((F:Qid[TL':TermList], TL:TermList)), en ese caso, primero se intenta convertir cada lista de términos por separado y, finalmente se unen los resultados que proporcionan las llamadas replaceTermRl(M:Module, (F:Qid[TL':TermList], PS:PairSet, N:Nat), R:Rule) y replaceTermRl(M:Module, (TL:TermList, PS:PairSet, N1:Nat), R:Rule) (conjuntos de pares y restricciones).

```
ceq replaceTermRl(M:Module, ((F:Qid[TL':TermList], TL:TermList), PS:PairSet, N:Nat), R:Rule)
        = ((T1:TermList, T2:TermList), PS1:PairSet ^ PS2:PairSet, N2:Nat)
        if (T1:TermList, PS1:PairSet, N1:Nat)
                := replaceTermRl(M:Module, (F:Qid[TL':TermList], PS:PairSet, N:Nat), R:Rule)
        /\ (T2:TermList, PS2:PairSet, N2:Nat)
                := replaceTermRl(M:Module, (TL:TermList, PS:PairSet, N1:Nat), R:Rule) .
```

También puede darse el caso que en la lista de términos el primero sea un término común, en ese caso únicamente hay que convertir la lista de términos (`TL:TermList`).

```
ceq replaceTermRl(M:Module, ((T:Term, TL:TermList), PS:PairSet, N:Nat), R:
    Rule)
        = ((T:Term, T1:TermList), PS1:PairSet, N1:Nat)
        if (T1:TermList, PS1:PairSet, N1:Nat)
                := replaceTermRl(M:Module, (TL:TermList,
                        PS:PairSet, N:Nat), R:Rule) .
```

El último caso es en el que no pueda entrar por ningún caso anterior y, por tanto, el término se devuelve sin modificar.

```
eq replaceTermRl(M:Module, (T:Term, PS:PairSet, N:Nat), R:Rule)
        = (T:Term, PS:PairSet, N:Nat) [owise] .
```

A continuación se describen algunos ejemplos.

Ejemplos

Por ejemplo, definimos el siguiente módulo `MULT` (listado 5.12) en Maude que es una teoría extendida de la aritmética Presburger con la multiplicación.

Listing 5.12: Módulo MULT

```
mod MULT is
pr NAT .
sorts S Nat2 .
subsort Nat < Nat2 < S .
op _;_ : S S -> S [assoc comm prec 75] .
op _**_ : Nat2 Nat2 -> Nat2 [prec 73] .
rl 0 ** S1:Nat2 => 0 .
rl s(S1:Nat2) ** S2:Nat2 => S2:Nat2 + (S1:Nat2 ** S2:Nat2) .
endm
```

Si se da el siguiente ejemplo

```
'_**_['s_^1['0.Zero], 's_^4['0.Zero]] =?= 's_^4['0.Zero]
```

Como se puede observar en el ejemplo no existe ninguna variable, por tanto la función `reduceTrivialProblem` no puede eliminar ninguna ecuación.

Al aplicar el comando de **Maude**

```
red expand(upModule('MULT, false), getEqs(upModule('MULT, false)),
        getRls(upModule('MULT, false)), 0,
        '_**_['s_^1['0.Zero], 's_^4['0.Zero]] =?= 's_^4['0.Zero]) .
```

Se obtiene el siguiente `PairSet&PairSet&Counter`

```
red expand(upModule('MULT, false), getEqs(upModule('MULT, false)),
                getRls(upModule('MULT, false)), 0,
                '_**_['s_^1['0.Zero], 's_^4['0.Zero]] =?= 's_^4['0.Zero]) .
        reduce in METASAT : expand(upModule('MULT, false),
                getEqs(upModule('MULT, false)), getRls(upModule('MULT, false
                    )), 0,
                '_**_['s_^1['0.Zero],'s_^4['0.Zero]] =?= 's_^4['0.Zero]) .
rewrites: 77 in 0ms cpu (0ms real) (~ rewrites/second)
result PairSet&PairSet&Counter:
('#0:Nat2 =?= 's_^4['0.Zero],'#0:Nat2 =?= '_**_['s_^1['0.Zero],'s_^4['0.Zero
    ]],1)
```

Donde el conjunto convertido es

```
'#0:Nat2 =?= 's_^4['0.Zero]
```

Y el conjunto de restricciones es

```
'#0:Nat2 =?= '_**_['s_^1['0.Zero],'s_^4['0.Zero]]
```

Si se toma el siguiente ejemplo

```
 '_+_['Y:Nat2,'_**_['Z:Nat2,'s_^1['0.Zero]]] =?= 's_^1['0.Zero]
```

Al lanzar el comando **Maude**

```
red expand(upModule('MULT, false), getEqs(upModule('MULT, false)),
getRls(upModule('MULT, false)), 0,
'_+_['Y:Nat2,'_**_['Z:Nat2,'s_^1['0.Zero]]] =?= 's_^1['0.Zero]) .
```

Se obtiene el siguiente resultado

```
reduce in METASAT : expand(upModule('MULT, false), getEqs(upModule('MULT,
    false)),
        getRls(upModule('MULT, false)), 0,
        '_+_['Y:Nat2,'_**_['Z:Nat2,'s_^1['0.Zero]]] =?= 's_^1['0.Zero])
```

```
    .
rewrites: 89 in 0ms cpu (0ms real) (89000000 rewrites/second)
result PairSet&PairSet&Counter:
        ('_+_['Y:Nat2,'#0:Nat2] =?= 's_^1['0.Zero],
        '#0:Nat2 =?= '_**_['Z:Nat2,'s_^1['0.Zero]],1)
```

Donde el conjunto convertido es

```
'_+_['Y:Nat2,'#0:Nat2] =?= 's_^1['0.Zero]
```

Y el conjunto de restricciones

```
'#0:Nat2 =?= '_**_['Z:Nat2,'s_^1['0.Zero]]
```

A continuación se muestra un ejemplo un poco más complejo

```
'_**_['Y:Nat2,'_**_['Z:Nat2,'s_^1['0.Zero]]] =?=
        '_**_['s_^3['0.Zero],'_**_['W:Nat2,'X:Nat2]]
```

El comando de **Maude** es

```
red expand(upModule('MULT, false), getEqs(upModule('MULT, false)),
        getRls(upModule('MULT, false)), 0,
        '_**_['Y:Nat2,'_**_['Z:Nat2,'s_^1['0.Zero]]] =?=
                '_**_['s_^3['0.Zero],'_**_['W:Nat2,'X:Nat2]]) .
```

Y el correspondiente resultado es

```
reduce in METASAT : expand(upModule('MULT, false), getEqs(upModule('MULT,
    false)),
        getRls(upModule('MULT, false)), 0,
        '_**_['Y:Nat2,'_**_['Z:Nat2,'s_^1['0.Zero]]] =?=
        '_**_['s_^3['0.Zero],'_**_['W:Nat2,'X:Nat2]]) .
rewrites: 122 in 0ms cpu (0ms real) (~ rewrites/second)
result PairSet&PairSet&Counter:
        ('#1:Nat2 =?= '#3:Nat2,'#0:Nat2 =?= '_**_['Z:Nat2,
        's_^1['0.Zero]] ^ '#1:Nat2 =?= '_**_['Y:Nat2,'#0:Nat2] ^
        '#2:Nat2 =?= '_**_['W:Nat2,'X:Nat2] ^
    '#3:Nat2 =?= '_**_['s_^3['0.Zero],'#2:Nat2]
    ,4)
```

Donde el conjunto convertido es

```
'#1:Nat2 =?= '#3:Nat2
```

Y el conjunto de restricciones

```
'#0:Nat2 =?= '_**_['Z:Nat2,'s_^1['0.Zero]] ^
'#1:Nat2 =?= '_**_['Y:Nat2,'#0:Nat2] ^
'#2:Nat2 =?= '_**_['W:Nat2,'X:Nat2] ^
'#3:Nat2 =?= '_**_['s_^3['0.Zero],'#2:Nat2]
```

5.3.2. Obtención de sustituciones mediante *Narrowing*

Una vez se ha obtenido el conjunto `PairSet` convertido y el conjunto `PairSet` de restricciones, visto en la Sección 5.3.1, se intentan encontrar substituciones sobre las partes derechas de las restricciones utilizando la función de **Maude** llamada `metaNarrowSearch` (vista en la Sección 2.2.11). La función que realiza esa llamada se denomina `callMetaNarrow` (listado 5.13). Esta función recibe el módulo con la teoría ecuacional, un `PairSet` de la forma `Var =?= Term` (contenido en el conjunto de restricciones) y una variable que es la meta-representación del punto de llegada de la función `metaNarrowSearch`. Como se puede observar en la llamada, se le indica a la función `metaNarrowSearch` que busque el resultado en varios pasos `'*` y que únicamente devuelva una solución (con el parámetro `1`). La función `callMetaNarrow` devuelve la tripleta `ResultTripleSet` obtenida (en caso afirmativo) a partir de la llamada a `metaNarrowSearch` o `empty` en caso que el `PairSet` esté vacío (`emptyPairSet`).

Listing 5.13: Función callMetaNarrow

```
***(
        Lanza la funcion metaNarrowSearch
)
op callMetaNarrow : Module PairSet Variable -> ResultTripleSet .
eq callMetaNarrow(M:Module, emptyPairSet, V:Variable) = empty .
eq callMetaNarrow(M:Module, T:Term =?= T2:Term , V:Variable)
        = metaNarrowSearch(M:Module, T2:Term, V:Variable, none, '*, 1,
            unbounded)        .
```

Ejemplos

Si se toman las restricciones obtenidas de un ejmplo anterior

```
'#0:Nat2 =?= '_**_['Z:Nat2,'s_^1['0.Zero]] ^
'#1:Nat2 =?= '_**_['Y:Nat2,'#0:Nat2] ^
```

```
'#2:Nat2 =?= '_**_['W:Nat2,'X:Nat2] ^
'#3:Nat2 =?= '_**_['s_^3['0.Zero],'#2:Nat2]
```

Y se realiza la llamada `callMetaNarrow`

```
red callMetaNarrow(upModule('MULT,false),
        '#0:Nat2 =?= '_**_['Z:Nat2,'s_^1['0.Zero]], '#4:Nat2) .
```

Se obtienen las substituciones del listado 5.3.2, de las cuales tan sólo interesan la primera y tercera, pues la segunda (`'_**_['Z:Nat2,'s_^1['0.Zero]]`) es exactamente el mismo término que se han lanzado en la llamada, por tanto, si se utilizase en la ejecución **se crearían ejecuciones infinitas**.

```
reduce in METASAT : callMetaNarrow(upModule('MULT, false),
        '#0:Nat2 =?= '_**_['Z:Nat2,'s_^1['0.Zero]], '#4:Nat2) .
rewrites: 2207 in 2ms cpu (2ms real) (821667 rewrites/second)
result ResultTripleSet: {'0.Zero,'Zero,
        '#4:Nat2 <- '0.Zero ;
        '#5:Nat2 <- 's_['0.Zero] ;
        'Z:Nat2 <- '0.Zero} |
{'_**_['Z:Nat2,'s_^1['0.Zero]],'Nat2,
        '#4:Nat2 <- '_**_['Z:Nat2,'s_['0.Zero]] ;
        '#6:Nat2 <- 'Z:Nat2} |
{'_+_['s_['0.Zero],'_**_['#8:Nat,'s_['0.Zero]]],'Nat2,
        '#10:Nat <- '#8:Nat ;
        '#4:Nat2 <- '_+_['s_['0.Zero],'_**_['#8:Nat,'s_['0.Zero]]] ;
        '#5:Nat2 <- '#8:Nat ;
        '#6:Nat2 <- 's_['0.Zero] ;
        'Z:Nat2 <- 's_['#8:Nat]}
```

Si se ejecuta la segunda restricción con la llamada

```
red callMetaNarrow(upModule('MULT,false),
        '#1:Nat2 =?= '_**_['Y:Nat2,'#0:Nat2], '#4:Nat2) .
```

Se produce el resultado del listado 5.3.2,y, de estas sustituciones las únicas validas son la primera y la tercera por los mismos motivos que se han definido anteriormente.

```
reduce in METASAT : callMetaNarrow(upModule('MULT, false), '#1:Nat2 =?= '_**
    _[
    'Y:Nat2,'#0:Nat2], '#4:Nat2) .
rewrites: 1856 in 1ms cpu (2ms real) (1022026 rewrites/second)
result ResultTripleSet: {'0.Zero,'Zero,
        '#4:Nat2 <- '0.Zero ;
        '#5:Nat2 <- '#0:Nat2 ;
        '#7:Nat2 <- '#0:Nat2 ;
        'Y:Nat2 <- '0.Zero} |
```

```
{'_**_['Y:Nat2,'#0:Nat2],'Nat2,
        '#4:Nat2 <- '_**_['Y:Nat2,'#0:Nat2] ;
        '#6:Nat2 <- 'Y:Nat2 ;
        '#7:Nat2 <- '#0:Nat2} |
{'_+_['#0:Nat2,'_**_['#8:Nat,'#0:Nat2]],'Nat2,
        '#11:Nat <- '#8:Nat ;
        '#12:Nat2 <- '#0:Nat2 ;
        '#4:Nat2 <- '_+_['#0:Nat2,'_**_['#8:Nat,'#0:Nat2]] ;
        '#5:Nat2 <- '#8:Nat ;
        '#6:Nat2 <- '#0:Nat2 ;
        '#9:Nat2 <- '#0:Nat2 ;
        'Y:Nat2 <- 's_['#8:Nat]}
```

5.3.3. Generación de estados

Esta es, posiblemente, la parte más compleja del prototipo ya que se produce una explosión combinatoria de estados por niveles (semejante a un árbol). Es decir, a partir de un estado inicial (obtenido mediante la función `expand` comentada en la Sección 5.3.3) se producen estados al realizar combinaciones de sustituciones obtenidas mediante la función `callMetaNarrow`. Estos nuevos estados forman un conjunto de estados `StateSet` donde cada uno de ellos será expandido nuevamente y los nuevos estados se añadirán al mismo conjunto. A continuación se detalla el proceso.

La función `nextLevel`, véase listado 5.14, es el punto de entrada de la generación y recibe un conjunto de estados, para cada uno de éstos se llama a la función `convertStates` que genera un nuevo nivel para dicho estado. Posteriormente se realiza una llamada recursiva a la misma función para el resto de estados.

Listing 5.14: Función nextLevel

```
***(
Crea todos los estados de un nivel del arbol
StateSet -> Conjunto de estados existentes
)
op nextLevel : Module StateSet -> StateSet .
eq nextLevel (M:Module, emptyState ) = emptyState .
ceq nextLevel (M:Module, (P:PairSet, Rules:PairSet, C:Nat) ; S:StateSet)
 =  toP&P&C( R:StateSet ; R':StateSet )
 if R:StateSet := convertStates(M:Module, (P:PairSet, emptyPairSet, C:Nat,
     Rules:PairSet))
 /\ R':StateSet := nextLevel(M:Module,  S:StateSet) .
```

La función `convertStates` (listado 5.15) va enviando cada

`PairSet&PairSet&Counter&PairSet` a la función `generateStates` junto al `ResultTripleSet` devuelto por la función `callMetaNarrow` y llama a la función `generateStates` que expande el estado, creando un conjunto de estados (una por cada resultado de la función `callMetaNarrow`), para una restricción dada. Para procesar las restantes restricciones se utiliza la función `remainingStates`.

Listing 5.15: Función convertStates

```
***(
StateSet -> Conjunto de estados existentes
)
op convertStates : Module StateSet -> StateSet .
eq convertStates (M:Module, (Forms:PairSet, ReturnRules:PairSet, N:Nat,
    emptyPairSet))
 = (Forms:PairSet, ReturnRules:PairSet, N:Nat, emptyPairSet) .
ceq convertStates (M:Module, (Forms:PairSet, ReturnRules:PairSet, N:Nat,
        T1:Term =?= T2:Term ^ Remaining:PairSet))
  =       S'':StateSet
 if R:ResultTripleSet := callMetaNarrow(M:Module, T1:Term =?= T2:Term,
    createVariable(M:Module, T1:Term =?= T2:Term, N:Nat))
 /\ S':StateSet  := generateStates(M:Module, R:ResultTripleSet, Forms:
     PairSet,
        ReturnRules:PairSet, Remaining:PairSet, T1:Term =?= T2:Term, N:Nat)
 /\ S'':StateSet := remainingStates(M:Module, S':StateSet)
                         .
```

La función `generateStates` (listado 5.16) recibe un `ResultTripleSet`, el conjunto de fórmulas del estado, el conjunto de sus restricciones y el `Pair` sobre el que se ha realizado la llamada a `callMetaNarrow` y, para cada uno de los resultados (salvo en el que el término resultante es igual al término con el que se ha lanzado realizado *narrowing*, como se ha explicado con anterioridad), sustituye las variables en los conjuntos de ecuaciones y reglas y reemplaza la restricción original por la misma variable y dicha sustitución.

Listing 5.16: Función generateStates

```
***(
Genera los estados a partir de la triple de resultados (omite el propio
    resultado de narrowing)
PairSet -> Formulas
PairSet -> Constrains finales (con sustitucion de narrowing)
PairSet -> Constrains
Pair    -> Termino utilizado para metaNarrowSearch
Nat     -> Indica a partir de que numero se generan variables
)
op generateStates : Module ResultTripleSet PairSet PairSet PairSet Pair Nat
    -> StateSet .
eq generateStates (M:Module, empty, PS:PairSet, Return:PairSet, PS':PairSet,
     T1:Term =?= T2:Term, N:Nat) = emptyState .
ceq generateStates(M:Module, {T:Term,TP:Type,S:Substitution} | R':
    ResultTripleSet, Eq:PairSet,
```

```
        ReturnRules:PairSet, RemainingRules:PairSet, T1:Term =?= T2:Term, N:
            Nat)
 = if T2:Term == Result:Term then S:StateSet
   else
    (Eq':PairSet, ReturnRules':PairSet ^ T1:Term =?= Result:Term,
                getFreshNumber(Result:Term, N:Nat),  RemainingRules':PairSet
                    ) ; S:StateSet fi

if      Result:Term := T:Term
/\      Eq':PairSet := substitutePairs(Eq:PairSet, S:Substitution)
/\      ReturnRules':PairSet := substitutePairs(ReturnRules:PairSet, S:
    Substitution)
/\      RemainingRules':PairSet := substitutePairs(RemainingRules:PairSet, S
    :Substitution)
/\      S:StateSet := generateStates(M:Module, R':ResultTripleSet, Eq:
    PairSet,
                        ReturnRules:PairSet, RemainingRules:PairSet, T1:Term
                            =?= T2:Term, N:Nat) .
```

Finalmente, la función `remainingStates` (listado 5.17) recibe los estados que no son completos (no se ha realizado *narrowing* sobre todas sus restricciones) y va iterando sobre cada restricción restante realizando llamadas a la función `convertStates` explicada con anterioridad. Posteriormente realiza una llamada recursiva para el resto de estados.

Listing 5.17: Función remainingStates

```
op remainingStates : Module StateSet -> StateSet .
eq remainingStates(M:Module, emptyState) = emptyState .

eq remainingStates(M:Module, (Forms:PairSet, ReturnRules:PairSet, N:Nat,
    emptyPairSet) ; S:StateSet)
 = (Forms:PairSet, ReturnRules:PairSet, N:Nat, emptyPairSet) ;
     remainingStates(M:Module, S:StateSet) .

ceq remainingStates(M:Module, (Forms:PairSet, ReturnRules:PairSet, N:Nat, T1
    :Term =?= T2:Term ^ Remaining:PairSet) ; S:StateSet)
 = S'':StateSet ; S''':StateSet
if S':StateSet  := convertStates(M:Module,(Forms:PairSet, ReturnRules:
    PairSet, N:Nat, T1:Term =?= T2:Term))
/\ S'':StateSet := remainingStates(M:Module, S':StateSet)
/\ S''':StateSet := remainingStates(M:Module, S:StateSet) .
```

En este punto, dada la complejidad del algoritmo, se explicará el algoritmo anterior utilizando ejemplos.

Ejemplos

Dado el ejemplo

```
'_**_['0.Zero,'_**_['X:Nat,'s_^2['0.Zero]]] =?= 's_^['0.Zero]
```

La función `reduceTrivialProblem` no puede eliminar la ecuación ya que no existe ninguna variable (como término único) en ninguno de los lados de la igualdad. Por tanto, se sigue con la ejecución del algoritmo.

Se obtiene (mediante la función `expand`) el siguiente estado (`PairSet&PairSet&Counter`) inicial

```
PairSet&PairSet&Counter: ('#1:Nat2 =?= 's_^['0.Zero],'#0:Nat2 =?=
        '_**_['X:Nat,'s_^2['0.Zero]] ^
        '#1:Nat2 =?= '_**_['0.Zero,'#0:Nat2],2)
```

Donde el conjunto "expandido" es

```
'#1:Nat2 =?= 's_^['0.Zero]
```

Y las restricciones son

```
'#0:Nat2 =?= '_**_['X:Nat,'s_^2['0.Zero]] ^ '#1:Nat2 =?= '_**_['0.Zero,'#0:
    Nat2]
```

Si calculamos las sustituciones por `callMetaNarrow` de la primera restricción obtenemos

```
result ResultTripleSet: {'0.Zero,'Zero,
        '#2:Nat2 <- '0.Zero ;
        '#3:Nat2 <- 's_^2['0.Zero] ;
        'X:Nat <- '0.Zero} |
{'_**_['X:Nat,'s_^2['0.Zero]],'Nat2,
        '#2:Nat2 <- '_**_['X:Nat,'s_^2['0.Zero]] ;
        '#4:Nat <- 'X:Nat} |
{'_+_['s_^2['0.Zero],'_**_['#6:Nat,'s_^2['0.Zero]]],'Nat2,
        '#2:Nat2 <- '_+_['s_^2['0.Zero],
                '_**_['#6:Nat,'s_^2['0.Zero]]] ;
        '#3:Nat2 <- '#6:Nat ;
        '#4:Nat2 <- 's_^2['0.Zero] ;
        '#8:Nat <- '#6:Nat ;
        'X:Nat <- 's_['#6:Nat]}
```

Las sustituciones de la segunda restricción serían

```
result ResultTripleSet: {'0.Zero,'Zero,
```

```
        '#2:Nat2 <- '0.Zero ;
        '#3:Nat2 <- '#0:Nat2 ;
        '#5:Nat2 <- '#0:Nat2} |
{'_**_['0.Zero,'#0:Nat2],'Nat2,
        '#2:Nat2 <- '_**_['0.Zero,'#0:Nat2] ;
        '#4:Nat2 <- '#0:Nat2}
```

Para generar un conjunto de estados sobre el primer `ResultTripleSet` se utiliza la función `generateStates`

```
red generateStates(upModule('MULT, false),
{'0.Zero,'Zero,'#2:Nat2 <- '0.Zero ; '#3:Nat2 <- '#0:Nat2 ; '#5:Nat2 <- '#0:
    Nat2} |
{'_**_['0.Zero,'#0:Nat2],'Nat2,'#2:Nat2 <- '_**_['0.Zero,'#0:Nat2] ; '#4:
    Nat2 <- '#0:Nat2},
'#1:Nat2 =?= 's_^['0.Zero],
emptyPairSet,
'#0:Nat2 =?= '_**_['X:Nat,'s_^2['0.Zero]] ^
        '#1:Nat2 =?= '_**_['0.Zero,'#0:Nat2],
'#0:Nat2 =?= '_**_['X:Nat,'s_^2['0.Zero]], 2) .
```

Se generan los estados, uno por cada sustitución

```
result StateSet:
('#1:Nat2 =?= 's_^['0.Zero],'#0:Nat2 =?= '0.Zero,2,
        '#0:Nat2 =?= '_**_['X:Nat,'s_^2['0.Zero]] ^ '#1:Nat2 =?= '_**_['0.
            Zero,'#0:Nat2]) ;

('#1:Nat2 =?= 's_^['0.Zero],'#0:Nat2 =?= '_**_['0.Zero,'#0:Nat2],2,
        '#0:Nat2 =?= '_**_['X:Nat,'s_^2['0.Zero]] ^ '#1:Nat2 =?= '_**_['0.
            Zero,'#0:Nat2])
```

Estos estados serían devueltos a la función `convertStates` que realizaría dos llamadas sobre el segundo `ResultTripleSet`, cada vez para un estado distinto. En primer lugar se realizaría la llamada sobre el estado $('\#1 : Nat2 =? =' s\_ \wedge ['0.Zero],' \#0 : Nat2 =? =' 0.Zero, 2,' \#0 : Nat2 =? =' \_{*}{*}\_['X : Nat,' s\_\wedge 2['0.Zero]]\wedge' \#1 : Nat2 =? =' \_{*}{*}\_['0.Zero,' \#0 : Nat2])$

```
red generateStates(upModule('MULT, false),
{'0.Zero,'Zero,'#2:Nat2 <- '0.Zero ;
'#3:Nat2 <- '#0:Nat2 ; '#5:Nat2 <- '#0:Nat2} |
{'_**_['0.Zero,'#0:Nat2],
'Nat2,'#2:Nat2 <- '_**_['0.Zero,'#0:Nat2] ;
'#4:Nat2 <- '#0:Nat2},

'#1:Nat2 =?= 's_^['0.Zero],
'#0:Nat2 =?= '0.Zero,'#1:Nat2 =?= '_**_['0.Zero,'#0:Nat2],
'#1:Nat2 =?= '_**_['0.Zero,'#0:Nat2],
2) .
```

Generando el estado completo (ya que no hay más restricciones en dicho estado)

```
result PairSet&PairSet&Counter&PairSet:
('#1:Nat2 =?= 's_^['0.Zero],'#0:Nat2 =?= '0.Zero ^ '#1:Nat2 =?=
'0.Zero,2,'#1:Nat2 =?= '_**_['0.Zero,'#0:Nat2])
```

Si se realiza la llamada sobre el segundo estado ($'\#1 : Nat2 =? =' s\_\wedge['0.Zero],' \#0 : Nat2 =? =' \_{*}{*}\_['0.Zero,' \#0 : Nat2], 2,' \#0 : Nat2 =? =' \_{*}{*}\_['X : Nat,' s\_\wedge 2['0.Zero]] \wedge' \#1 : Nat2 =? =' \_{*}{*}\_['0.Zero,' \#0 : Nat2]$) mediante el comando

```
red generateStates(upModule('MULT, false),
{'0.Zero,'Zero,'#2:Nat2 <- '0.Zero ;
'#3:Nat2 <- '#0:Nat2 ; '#5:Nat2 <- '#0:Nat2} |
{'_**_['0.Zero,'#0:Nat2],
'Nat2,'#2:Nat2 <- '_**_['0.Zero,'#0:Nat2] ;
'#4:Nat2 <- '#0:Nat2},

'#1:Nat2 =?= 's_^['0.Zero],
'#0:Nat2 =?= '_**_['0.Zero,'#0:Nat2],
'#1:Nat2 =?= '_**_['0.Zero,'#0:Nat2],
'#1:Nat2 =?= '_**_['0.Zero,'#0:Nat2], 2) .
```

Se genera el estado final completo

```
PairSet&PairSet&Counter:
('#1:Nat2 =?= 's_^['0.Zero],'#0:Nat2 =?= '_**_['0.Zero,'#0:Nat2] ^ '#1:Nat2
    =?= '0.Zero,2)
```

Finalmente, podemos decir que a partir del estado inicial

```
result PairSet&PairSet&Counter&PairSet:
('#1:Nat2 =?= 's_^['0.Zero],'#0:Nat2 =?= '_**_['0.Zero,'#0:Nat2] ^
'#1:Nat2 =?= '0.Zero,2,
'#1:Nat2 =?= '_**_['0.Zero,'#0:Nat2])
```

Se han obtenido los estados

```
('#1:Nat2 =?= 's_^['0.Zero],'#0:Nat2 =?= '0.Zero ^ '#1:Nat2 =?=
'0.Zero,2,'#1:Nat2 =?= '_**_['0.Zero,'#0:Nat2])

('#1:Nat2 =?= 's_^['0.Zero],'#0:Nat2 =?= '_**_['0.Zero,'#0:Nat2] ^
'#1:Nat2 =?= '0.Zero,2,
'#1:Nat2 =?= '_**_['0.Zero,'#0:Nat2])
```

Estos dos estados serían el resultado de la llamada a la función `nextLevel` creando, a partir de un estado, un nuevo nivel en el árbol que contiene dos estados. En este ejemplo, por cada llamada `nextLevel` sucesiva sobre cada estado se generarían estados dependiendo del número de sustituciones que generase la función `callMetaNarrow`, es decir, si hay 2 restricciones cada una con 2 sustituciones válidas para cada una se generarían 4 estados. En el caso en que hubiesen 3 restricciones con 3 sustituciones válidas para cada una se generarían 9 estados, y así sucesivamente.

5.3.4. Control del flujo del programa

Esta parte del prototipo refleja cómo se comporta la ejecución y generación de estados para no entrar en ciclos infinitos de recurrencia ya que podría existir un problema de terminación con la explosión combinatoria de estados que se ha comentado con anterioridad (Sección 5.3.3).

Toda la gestión del prototipo la maneja la función `globalControl` (listado 5.18) en la que se sigue un procedimiento para evitar en lo posible, la creación de bucles infinitos. Esta función en primer lugar comprueba si el conjunto de fórmulas "expandido" se satisface llamando a la función de la segunda parte del prototipo `metasat-trans` (Sección 4). Si esta función devuelve `false` entonces se elimina dicho estado ya que ninguna sustitución sobre las variables hará que la fórmula resulte cierta.

Si por el contrario, la función devuelve `true` entonces significa que *podría* ser cierta y, por este motivo, se genera un nuevo estado temporal uniendo el conjunto de fórmulas "expandidas" con el conjunto de reglas. Si este estado se puede seguir expandiendo (función `canExpand` comentada con anterioridad), entonces continúa la ejecución de la función `globalControl` expandiendo todos los posibles estados al aplicar un nuevo nivel (`nextLevel`) sobre dicho estado temporal. Si dicho estado temporal no se puede expandir y la llamada a la función `metasat-trans` con dicho estado devuelve `false` entonces se elimina el estado principal y continúa la ejecución del algoritmo. Si la función `metasat-trans` devuelve `true` sabemos que las fórmulas y restricciones de ese estado se satisfacen y, por tanto, no hace falta seguir con la ejecución del algoritmo (devolviendo el valor `true`). Este proceso se explica en *pseudocodigo* (listado 5.19).

Listing 5.18: Función globalControl

```
***(
Funcion global de control
)
op globalControl : Module StateSet -> Bool .
eq globalControl (M:Module, emptyState) = false .
eq globalControl (M:Module, P&P&C ; S:StateSet)
 =if metasat-trans(M:Module, getFirstPair(P&P&C), getCounter(P&P&C)) ==
     false
  then globalControl(M:Module, S:StateSet)
 else
  if canExpand(M:Module, getEqs(M:Module), getRls(M:Module), getCounter(P&P&
      C), getFirstPair(P&P&C) ^ getSecondPair(P&P&C)) == false
    then if metasat-trans(M:Module, getFirstPair(P&P&C) ^ getSecondPair(P&P&
        C) , getCounter(P&P&C)) == true
     then true
    else
     globalControl(M:Module, S:StateSet)
    fi
   else
    globalControl(M:Module, S:StateSet ;
     expandAll(M:Module, getEqs(M:Module),
     getRls(M:Module), nextLevel(M:Module, P&P&C)))
   fi
 fi .
```

Listing 5.19: Pseudocodigo globalControl

```
globalControl(conjuntoestados):
 si conjuntoestados == []
  return false;

 si metasat-trans(conjuntoestados[1]->formulas) == false
  eliminar(conjuntoestados[1]);
  globalControl(conjuntoestados);
 si no // metasat-trans == true
  si posibleExpandir(conjuntoestados[1]->formulas + conjuntoestados[1]->
      restricciones) == false
   si metasat-trans(conjuntoestados[1]->formulas + conjuntoestados[1]->
       restricciones) == true
    return true;
   si no //metasat-trans == false
    eliminar(conjuntoestados[1]);
    globalControl(conjuntoestados);
   si no // posibleExpandir == true
    eliminar(conjuntoestados[1]);
    nuevoconjunto=nuevonivel(conjuntoestados[1]->formulas + conjuntoestados
        [1]->restricciones);
    conjuntoexpandido=expandir(nuevoconjunto);
    conjuntoestados.append(conjuntoexpandido);
    globalControl(conjuntoestados);
```

Finalmente, el punto de entrada al prototipo es la función `metasat` (listado 5.20) que delega todo el control a la función `globalControl`.

Listing 5.20: Función metasat

```
op metasat : Module PairSet -> Bool .
eq metasat(M:Module, P:PairSet) = globalControl(M:Module,expand(M:Module,
    getEqs(M:Module), getRls(M:Module), 0, reduceTrivialProblem(P:PairSet)))
      .
```

Ejemplos

A continuación se muestran algunos ejemplos de ejecución donde entran los tres prototipos proporcionando resultados completos.

Ejemplo `false`

El primer ejemplo es

```
'_**_['s_^1['0.Zero], 's_^4['0.Zero]] =?= 's_^5['0.Zero]
```

Cuya expansión genera el estado

```
result PairSet&PairSet&Counter:
('#0:Nat2 =?= 's_^5['0.Zero],'#0:Nat2 =?= '_**_['s_^1['0.Zero],'s_^4['0.Zero
    ]],1)
```

El primer estado que recibiría la función `metasat-trans` sería

```
'#0:Nat2 =?= 's_^5['0.Zero]
```

Cuya transformación en `SATProblem` sería

```
'#4:Nat === '#0:Nat2 ^ '#4:Nat === 's_^5['0.Zero]
```

A su vez, el `iBool` resultante sería

```
c(true) ^ c(true) ^ (c(5) === i(1)) ^ (i(1) === i(2)) ^ i(1) >= c(0) ^ i(2)
    >= c(0)
```

Dando como resultado el valor ***true***, entonces se comprueba si se puede expandir, al ser cierto (por tener la restricción `'#0:Nat2 =?= '_**_['s_^1['0.Zero],'s_^4['0.Zero]]`) se llama a la función `nextLevel` generando el estado

```
'#0:Nat2 =?= '_+_['s_^4['0.Zero],'#1:Nat2] ^ '#0:Nat2 =?= 's_^5['0.Zero] ^ '
    #1:Nat2 =?= '0.Zero
```

Cuyo `SATProblem` generado sería

```
 '#10:Nat === 's_^4['0.Zero] ^ '#8:Nat === '#0:Nat2 ^
 '#8:Nat === 's_^5['0.Zero] ^ '#9:Nat === '#1:Nat2 ^
 '_+_['#9:Nat,'#10:Nat] === '#0:Nat2
```

Y el correspondiente `iBool` sería

```
c(true) ^ c(true) ^ (c(4) === i(1)) ^ (c(5) === i(2)) ^
(i(2) === i(3)) ^ (i(3) === i(1) + i(4)) ^ (i(4) === i(5)) ^
i(1) >= c(0) ^ i(2) >= c(0) ^ i(3) >= c(0) ^ i(4) >= c(0) ^
i(5) >= c(0)
```

Este estado resulta ***false*** y, por tanto, se elimina dicho estado del conjunto de estados y como no existen más estados en el conjunto se sale de la ejecución devolviendo el valor `false`.

Ejemplo `true`

Dado el ejemplo

```
'_**_['Y:Nat2, 's_^1['0.Zero]] =?= '0.Zero
```

Al igual que en los ejemplos anteriores la función `reduceTrivialProblem` no puede realizar ninguna acción ya que no hay ninguna variable sola en ningún lado de la igualdad.

Cuya expansión genera el estado

```
result PairSet&PairSet&Counter:
('#0:Nat2 =?= '0.Zero,'#0:Nat2 =?= '_**_['Y:Nat2,'s_^1['0.Zero]],1)
```

El estado envíado a metasat-trans sería

```
'#0:Nat2 =?= '0.Zero
```

Convirtiéndose en el `SATProblem`

```
'#4:Nat === '#0:Nat2 ^ '#4:Nat === '0.Zero
```

Y el correspondiente `iBool`

```
c(true) ^ c(true) ^ (c(0) === i(1)) ^ (i(1) === i(2)) ^
i(1) >= c(0) ^ i(2) >= c(0)
```

La función devuelve `true`, por tanto se comprueba si el estado se puede expandir. Como tiene la restricción (`'#0:Nat2 =?= '_**_['Y:Nat2, 's_^1['0.Zero]]`) se devuelve cierto y, por tanto, se debe llamar a la generación de un nuevo nivel de estados.

```
result StateSet: ('#0:Nat2 =?= '0.Zero,'#0:Nat2 =?= '0.Zero,1) ;
('#0:Nat2 =?= '0.Zero,'#0:Nat2 =?= '_+_['s_['0.Zero],'_**_['#5:Nat,'s_['0.
    Zero]]],6)
```

El siguiente estado en ser ejecutado es

```
'#0:Nat2 =?= '0.Zero ^ '#0:Nat2 =?= '0.Zero
```

Que genera el `SATProblem`

```
'#6:Nat === '#0:Nat2 ^ '#6:Nat === '0.Zero ^ '#7:Nat === '#0:Nat2 ^ '#7:Nat
    === '0.Zero
```

Y el `iBool`

```
c(true) ^ c(true) ^ (c(0) === i(1)) ^ (c(0) === i(3)) ^ (i(1) === i(2)) ^
(i(2) === i(3)) ^ i(1) >= c(0) ^ i(2) >= c(0) ^ i(3) >= c(0)
```

Dando como resultado `true` y cómo este estado no es puede expandir más (no hay operaciones ** en las restricciones) el prototipo finaliza su ejecución y devuelve el valor `true`.

Otro posible ejemplo vendría dado por la siguiente ecuación

```
'_**_['Y:Nat2, 's_^1['0.Zero]] =?= 'X:Nat2
```

Donde la función `reduceTrivialProblem` resolvería el problema ya que la `'X:Nat` no aparece en otra ecuación. La fórmula se reduciría a

```
'0.Zero =?= '0.Zero
```

Dando como resultado final `true` ya que no hay más estados disponibles.

Otro ejemplo más complejo sería el siguiente

```
'_**_['Y:Nat2, 's_^1['0.Zero]] =?= '_**_['s_^2['0.Zero], 'X:Nat2]
```

Donde el primer estado generado sería

```
result PairSet&PairSet&Counter:
('#0:Nat2 =?= '#1:Nat2,
        '#0:Nat2 =?= '_**_['Y:Nat2,'s_^1['0.Zero]] ^
        '#1:Nat2 =?= '_**_['s_^2['0.Zero],'X:Nat2],2)
```

El término que se enviaría a la función `metasat-trans` sería

```
'#0:Nat2 =?= '#1:Nat2,
```

Que devolvería el valor `true` (se han omitido los pasos intermedios). Por tanto se generaría el siguiente conjunto de estados al llamar a la función `nextLevel`

```
result StateSet:
('#0:Nat2 =?= '#1:Nat2,'#0:Nat2 =?= '0.Zero ^
'#1:Nat2 =?= '_+_['X:Nat2,
'_**_['s_['0.Zero],'X:Nat2]],2) ;

('#0:Nat2 =?= '#1:Nat2,'#0:Nat2 =?= '_+_['s_['0.Zero],
'_**_['#6:Nat,'s_['0.Zero]]] ^
'#1:Nat2 =?= '_+_['X:Nat2,'_**_['s_['0.Zero],'X:Nat2]],7)
```

Y expandidos (función `expand` sobre cada uno de ellos) quedarían

```
result StateSet:
 ('#0:Nat2 =?= '#1:Nat2 ^ '#0:Nat2 =?= '0.Zero ^
 '#1:Nat2 =?= '_+_['X:Nat2,'#2:Nat2],
 '#2:Nat2 =?= '_**_['s_['0.Zero],'X:Nat2],3)

('#0:Nat2 =?= '#1:Nat2 ^
'#0:Nat2 =?= '_+_['s_['0.Zero],'#7:Nat2] ^
'#1:Nat2 =?= '_+_['X:Nat2,'#8:Nat2],
'#7:Nat2 =?= '_**_['#6:Nat,'s_['0.Zero]] ^
'#8:Nat2 =?= '_**_['s_['0.Zero],'X:Nat2],9)
```

El estado que se enviaría a la función `metasat-trans` sería

```
'#0:Nat2 =?= '#1:Nat2 ^ '#0:Nat2 =?= '0.Zero ^
        '#1:Nat2 =?= '_+_['X:Nat2,'#2:Nat2]
```

Transformándose en el `SATProblem`

```
'#11:Nat === '#0:Nat2 ^ '#11:Nat === '#1:Nat2 ^
        '#12:Nat === '#0:Nat2 ^ '#12:Nat === '0.Zero ^
        '#13:Nat === 'X:Nat2 ^ '#14:Nat === '#2:Nat2 ^
        '_+_['#13:Nat,'#14:Nat] === '#1:Nat2
```

Y a su vez en el `iBool`

```
c(true) ^ c(true) ^ (c(0) === i(4)) ^ (i(1) === i(2)) ^
        (i(1) === i(3)) ^ (i(2) === i(4)) ^
        (i(3) === i(5) + i(7)) ^ (i(5) === i(6)) ^
        (i(7) === i(8)) ^ i(1) >= c(0) ^ i(2) >= c(0) ^
        i(3) >= c(0) ^ i(4) >= c(0) ^ i(5) >= c(0) ^
        i(6) >= c(0) ^ i(7) >= c(0) ^ i(8) >= c(0)
```

Que da como resultado `true`, por tanto, éste estado puede ser satisfacible. A continuación se comprueba si puede ser expandido y como sí que puede ser (restricción `'#2:Nat2 =?= '_**_['s_['0.Zero],'X:Nat2]`) se debe crear un nuevo nivel sobre éste y expandirlo. En este caso únicamente se genera un nuevo estado que será añadido al conjunto de estados.

```
PairSet&PairSet&Counter:
('#0:Nat2 =?= '#1:Nat2 ^ '#0:Nat2 =?= '0.Zero ^
'#1:Nat2 =?= '_+_['X:Nat2,'#2:Nat2],
'#2:Nat2 =?= '_+_['X:Nat2,'_**_['0.Zero,'X:Nat2]],3)
```

A continuación se prosigue la ejecución con el siguiente estado del conjunto enviándose a la función `metasat-trans`

```
'#0:Nat2 =?= '#1:Nat2 ^
'#0:Nat2 =?= '_+_['s_['0.Zero],'#7:Nat2] ^
'#1:Nat2 =?= '_+_['X:Nat2,'#8:Nat2]
```

Generando el `SATProblem`

```
'#10:Nat =?= '#1:Nat2 ^ '#10:Nat =?= '_+_['#13:Nat,'#14:Nat] ^
'#11:Nat =?= 's_['0.Zero] ^ '#12:Nat =?= '#7:Nat2 ^
'#13:Nat =?= 'X:Nat2 ^ '#14:Nat =?= '#8:Nat2 ^
'#9:Nat =?= '#0:Nat2 ^ '#9:Nat =?= '#10:Nat ^
'#9:Nat =?= '_+_['#11:Nat,'#12:Nat]
```

Y el `iBool` final

```
c(true) ^ c(true) ^ (i(4) === c(0) + i(1) + i(3)) ^
(i(6) === c(0) + i(2) + i(5)) ^ (i(7) === c(0) + i(3) + i(5)) ^
(i(8) === c(0) + i(1) + i(2) + i(3) + i(5)) ^
(i(9) === c(0) + i(1) + i(2) + i(3) + i(5)) ^
(c(0) + c(1) === c(0) + i(1) + i(2)) ^
(c(0) + i(1) + i(2) + i(3) + i(5) === c(0) + i(1) + i(2) + i(3) + i(5)) ^
(c(0) + i(1) + i(2) + i(3) + i(5) === c(0) + c(0) + c(0) + i(1) + i(2) + i
    (3) + i(5)) ^
(c(0) + i(1) + i(2) + i(3) + i(5) === c(0) + c(0) + c(0) + i(1) + i(2) + i
    (3) + i(5)) ^
i(1) >= c(0) ^ i(2) >= c(0) ^ i(3) >= c(0) ^ i(4) >= c(0) ^ i(5) >= c(0) ^
i(6) >= c(0) ^ i(7) >= c(0) ^ i(8) >= c(0) ^ i(9) >= c(0)
```

La función `metasat-interface` con este `iBool` devuelve `true`, por tanto hay que hacer el mismo proceso anterior. Como el estado se puede expandir, hay que llamar a la función `nextLevel` y expandir los estados resultantes que son añadidos al conjunto de estados global.

```
('#0:Nat2 =?= '#1:Nat2 ^ '#0:Nat2 =?= '_+_['s_['0.Zero],'#7:Nat2] ^
'#1:Nat2 =?= '_+_['X:Nat2,'#8:Nat2],'#7:Nat2 =?= '0.Zero ^
8:Nat2 =?= '_+_['X:Nat2,'_**_['0.Zero,'X:Nat2]],9) ;

('#0:Nat2 =?= '#1:Nat2 ^ '#0:Nat2 =?= '_+_['s_['0.Zero],'#7:Nat2] ^
'#1:Nat2 =?= '_+_['X:Nat2,'#8:Nat2],'#7:Nat2 =?= '_+_['s_['0.Zero],
'_**_['#13:Nat,'s_['0.Zero]]] ^
'#8:Nat2 =?= '_+_['X:Nat2,'_**_['0.Zero,'X:Nat2]],14)
```

Este proceso se realiza recursivamente hasta que se genera el siguiente estado no expansible

```
('#0:Nat2 =?= '#1:Nat2 ^ '#0:Nat2 =?= '0.Zero ^
'#1:Nat2 =?= '_+_['X:Nat2,'#2:Nat2] ^ '#2:Nat2 =?= '_+_['X:Nat2,'#3:Nat2] ^
'#3:Nat2 =?= '0.Zero,emptyPairSet,4)
```

Cuyo resultado final (depués de transformarse a `SATProblem` y `iBool`) es `true` y, como no se puede expandir, el algoritmo acaba y devuelve el valor ***true***. A continuación se puede observar cómo se ejecutaría el presente ejemplo de una forma gráfica (Figura 5.1).

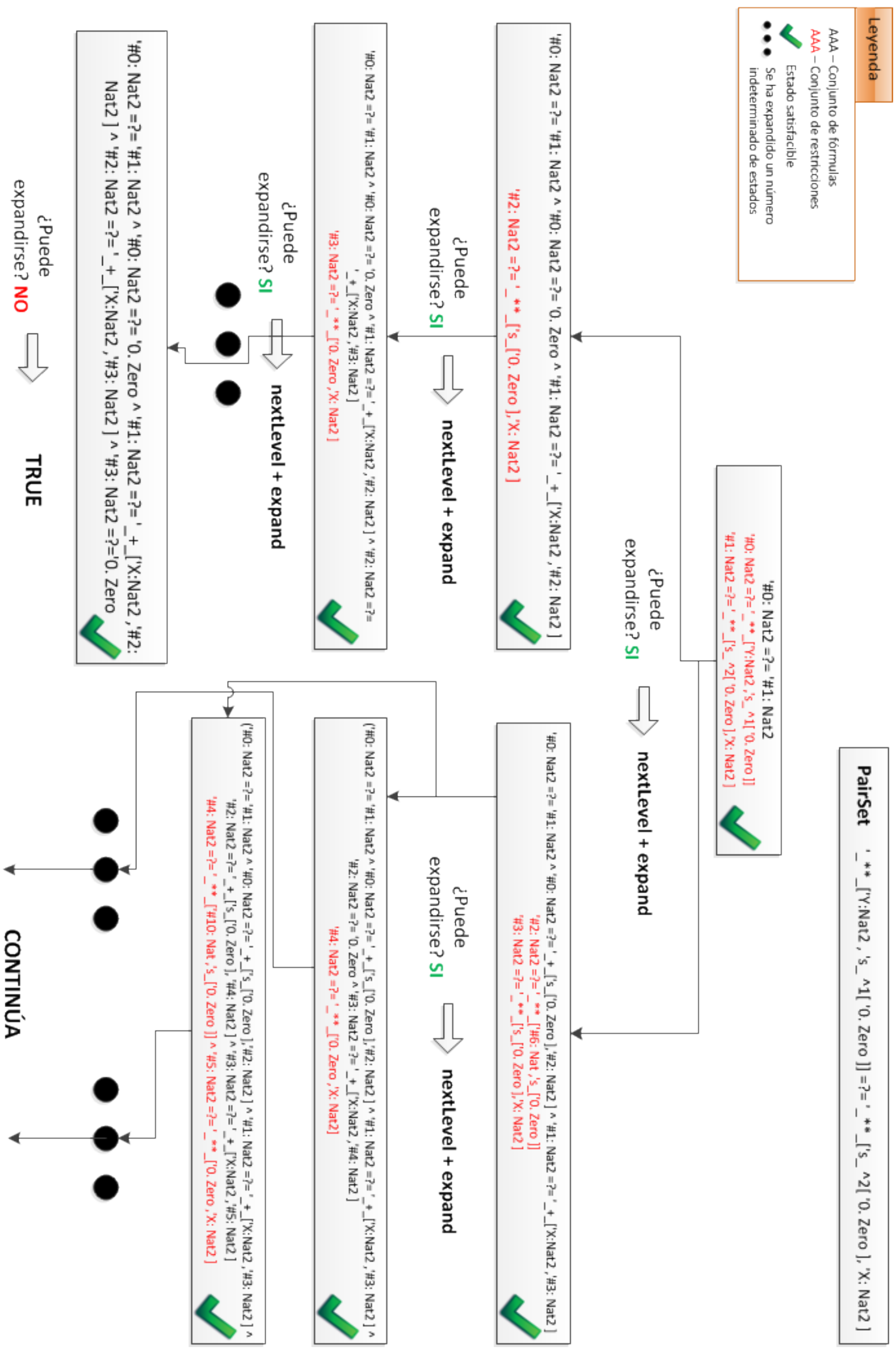

Figura 5.1: Diagrama de ejecución

6

Conclusiones

El desarrollo de la tesina ha sido bastante duro ya que el alumno no poseía apenas conocimientos de programación lógica ni programación funcional. Se disponía de conocimientos mínimos del lenguaje **Maude**, únicamente visto en asignaturas del máster (**FSA**, **DLP** y **ATM**). Por tanto, tampoco se tenían conocimientos de la representación al metanivel de términos, módulos, unificación o narrowing. El alumno tampoco disponía de conocimientos previous sobre satisfacibilidad de restricciones, que no han sido estudiados en al máster. Básicamente, el alumno no tenía ningún conocimiento del tema antes de comenzar a realizar esta tesina.

Durante el transcurso de la implementación de los prototipos se han tenido numerosos problemas, la mayoría de éstos por falta de conocimientos previos, pero se han conseguido solucionar de forma correcta, siempre bajo el visto bueno del tutor. Por tanto, el alumno está muy satisfecho con la labor realizada en la tesina, con el prototipo en Maude obtenido, los buenos resultados obtenidos y, sobre todo, por los nuevos conocimientos adquiridos como la creación de programas `Maude`, manejo del meta-nivel en **Maude**, propiedades de *unificación* y *narrowing*, satisfacibilidad de restricciones, etc.

Aparte, el autor de esta tesina está muy agradecido a los otros dos investigadores interesados en al trabajo de esta tesina, José Meseguer de la *University of Illinois at Urbana-Champaign* y Vijay Ganesh del *MIT*.

Como trabajo futuro, se abren muchísimas posibilidades, ya que hemos realizado una integración de satisfacibilidad de fórmulas en Maude muy rudimentaria, aunque tremendamente útil. Las área de satisfacibilidad de fórmulas (SAT) y satisfacibilidad de fórmulas con una teoría de fondo (SMT) son muy extensas y con múltiples aplicaciones en Maude.

Bibliografía

[1] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[2] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, pages 825–885. 2009.

[3] TeReSe, editor. *Term Rewriting Systems*. Cambridge University Press, Cambridge, 2003.

[4] José Meseguer. Conditional rewriting logic as a united model of concurrency. *Theor. Comput. Sci.*, 96(1):73–155, 1992.

[5] José Meseguer. Membership algebra as a logical framework for equational specification. In Francesco Parisi-Presicce, editor, *WADT*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1997.

[6] Jean-Pierre Jouannaud and Hélène Kirchner. Completion of a set of rules modulo a set of equations. *SIAM J. Comput.*, 15(4):1155–1194, 1986.

[7] Santiago Escobar, Ralf Sasse, and José Meseguer. Folding variant narrowing and optimal variant termination. *J. Log. Algebr. Program.*, 2012. In Press.