Técnicas de Refactorización para Maude

Universidad Politécnica de Valencia

Departamento de Sistemas
Informáticos y Computación

Tesis de Máster

Autor:
Angel Cuenca Ortega

Directores:
María Alpuente Frasnedo
Francisco Frechina Navarro

Valencia, Julio 2013

“ Algunos hombres observan el mundo y se preguntan “¿por qué?”. Otros observan el mundo y se preguntan “¿por qué no?.”

George Bernard Shaw

Resumen

La refactorización (en inglés *refactoring*) es una técnica de transformación propia de la Ingeniería del Software que permite reestructurar el código fuente, alterando su estructura interna sin cambiar su comportamiento externo.

Cuando se realiza la refactorización de forma adecuada, es posible obtener un código más fácil de entender o modificar, con garantías de no haber introducido errores ni alterado la semántica original de los programas.

El objetivo de esta tesis de máster es el desarrollo de técnicas de refactorización para la lógica de reescritura (RWL) y concretamente aplicables al lenguaje Maude.

Para ello investigamos la aplicación de técnicas clásicas de transformación de programas, basadas en plegado/desplegado, cuya adaptación a la RWL resulta no trivial debido a tener que lidiar simultáneamente con reglas, ecuaciones, sorts, subsorts y axiomas algebraicos y/o de pertenencia (memberships).

Como resultado de este trabajo, desarrollamos un catálogo de refactorizaciones aplicables al lenguaje Maude, que hemos implementado en el mismo lenguaje Maude y que ilustramos mediante algunos ejemplos y casos de estudio representativos.

Agradecimientos

Doy gracias a Dios por darme salud y vida para realizar estos estudios.

Agradezco a todos mis compañeros del máster con quienes pude compartir buenos momentos de alegría dentro y fuera de las aulas, y a mis profesores por haberme compartido con entrega y profesionalismo sus conocimientos.

Quiero agradecer a mi país, Ecuador, quien a través de la SENESCYT me ha brindado la oportunidad de poder realizar este máster.

Mis sinceros agradecimientos a María Alpuente y Francisco Frechina porque dedicaron tanto tiempo y paciencia para responder muchos correos, hacerme correcciones, compartir conocimientos, ..., etc., porque estoy seguro que sin ellos no hubiera terminado este proyecto.

Tantos buenos amigos que he podido conocer durante este tiempo, dentro y fuera de la universidad, pero sin duda cuatro personas han sabido estar más cerca de mi en todo momento. Mi gran amiga Priscila, que pese haberme conocido tan poco, junto a su familia me brindaron un apoyo incondicional, que jamás podré olvidar. Edith es sin duda la persona con la que más he pasado dentro de la universidad, sus consejos y apoyo la han convertido en una de mis mejores amigas. Miguel Ángel y su familia me han brindado su amistad desde mi llegada. Por último y no menos importante, sino por el contrario, el mejor amigo que he conocido en este país es Jonathan, quien junto a su familia me brindaron una amistad sincera.

Gracias a todos.

Angel Cuenca.
Valencia, Julio 2013

Índice general

Índice de Figuras

Índice de Tablas

Índice de Ejemplos

Dedicado a mi madre, familia y amigos.

Capítulo 1

Introducción

Esta tesis investiga e implementa un catálogo de refactorizaciones para el lenguaje Maude. Las refactorizaciones se formalizan en el propio lenguaje Maude y se implementan en el mismo Maude también. Las técnicas de definición general de refactorización se presentan en la Sección 1.1. La Sección 1.2 presenta una breve descripción de los lenguajes funcionales para centrarse a continuación en el objetivo de refactorizar programas funcionales, lo que nos permitirá después acometer el problema de refactorizar programas escritos en Maude. En concreto se describen las técnicas de refactorización desarrollado por la Universidad de Kent para lenguajes funcionales y que han sido aplicados a los lenguajes Haskell y Erlang.

1.1. Refactorización de un programa

La refactorización es una técnica para transformar código de tal manera que cambia la estructura y organización interna del programa pero no cambia su comportamiento externo. Es decir, se garantiza que la refactorización no introduce errores o invalida las propiedades existentes que no dependen de la estructura interna del programa. La refactorización se puede usar para mejorar el diseño y calidad del software e incrementar su reutilización.

La refactorización es una práctica común que utilizan algunos programadores que a menudo modifican el código existente, sin llegarlo a llamar "Refactorización". El término refactorización fue introducido por primera vez en el trabajo de W. Opduke y R. Johnson en 1990.

La refactorización ha sido identificada como central al desarrollo y mantenimiento del software, especialmente con las comunidades de ingeniería de software y orientada a objetos *OO*.

La refactorización puede darse en varios niveles dentro del código. Por ejemplo, renombrar una variable local sólo tiene impacto en la entidad donde la variable está declarada; sin embargo renombrar un variable global afecta a todo un programa.

Dos de las principales actividades involucradas dentro de la refactorización son el análisis de programas y la transformación de programas. El análisis de programas comprueba si ciertas condiciones colaterales se cumplen en el programa bajo la refactorización para poder asegurar la corrección del proceso y recoge la información necesaria para la fase de transformación del programa. La transformación de programas lleva a cabo el paso de reestructuración propio de una refactorización. Tanto el análisis de programas como la transformación de programas se prestan a automatización, manifestada por la rica colección de trabajos existentes en muchas áreas de la ingeniería de software, incluyendo construcción de compiladores, comprensión de código, validación y depuración, slicing de programas, mantenimiento de programas e ingeniería inversa, etc.

Una variedad de herramientas de refactorización han sido desarrolladas para proporcionar soporte de refactorización para varios lenguajes de programación, tales como el Browser for Smalltalk, IntelliJ Idea para Java, ReSharper para C#, VB.NET & Eclipse refactorization suppport para C++, Java y muchos más. Para los lenguajes de programación funcional, podemos citar HaRe para Haskell y para Erlang los sistemas Wrangler y RefactorErl [21].

En [21] podemos encontrar una terminología que detallaremos más adelante, pero que de manera resumida clasifica en dos los tipos la refactorización. *Atómica* y *Compuesta*. Una *refactorización atómica* es una refactorización que no se puede descomponer en refactorizaciones más simples. Agregar un parámetro a una definición de función es un ejemplo de una refactorización atómica. Una *refactorización compuesta* puede estar formada de una serie de refactorizaciones atómicas. Un ejemplo de refactorización compuesta es el reemplazamiento de un tipo de datos concreto por un tipo abstracto de dato*s ADT*.

1.2. Programación Funcional

Los lenguajes de programación funcional están diseñados para reflejar la forma de pensar de manera abstracta, en lugar de reflejar la estructura de la máquina subyacente. Fueron basados originalmente en el cálculo Lambda, que incorpora un modelo simple

de computación que proporciona una manera formal para describir evaluaciones de funciones y expresiones. Este tipo de lenguaje de programación considera la computación como la evaluación de expresiones que se construyen mediante la aplicación de funciones. Cada función toma cero o más parámetros como entrada y retorna un único valor como salida. Las funciones son tratadas como ciudadanos de primera clase, que significa que las funciones pueden ser parámetros de otras funciones y pueden ser el resultado de otras funciones, llamándose a éstas *funciones de orden superior* (*higher-order functions*).

A diferencia de los tradicionales lenguajes imperativos como *C* o *Pascal*, en los lenguajes funcionales puros no hay asignación de memoria explícita y el operador '=' se utiliza para definir el valor de una función con argumentos concretos. Este resultado de aplicar una función a un conjunto dado de parámetros será el mismo, con independencia de cuándo, o dónde, la función es evaluada. Esta característica de *transparencia referencial* hace que la verificación formal de programas escritos en lenguajes funcionales sea muy fácil [19].

1.3. Refactorización de Programas Funcionales

Los lenguajes de programación funcional difieren de los imperativos y lenguajes OO tanto en la teoría como en la práctica.

Como es el caso de otros tipos de transformaciones de programas, la refactorización está basada en la semántica formal del lenguaje y la equivalencia de programas. La semántica limpia de los lenguajes de programación funcional y la rica base teórica para el razonamiento sobre este tipo de programas facilita la especificación formal y asegura la corrección de la refactorización. A menudo esto no es posible para lenguajes imperativos existentes y lenguajes OO debido a los efectos laterales y la falta de semántica formal. Desde este punto de vista, los lenguajes de programación funcional resultan más adecuados para la refactorización. El estilo de programación *"primero codificar, después revisar"* (first code, then revise") es ampliamente usado en la práctica por los programadores funcionales por lo que la refactorización funcional tiene muchas oportunidades de producir mejora muy apreciables.

En la programación funcional, la transformación de programas se ha estudiado ampliamente en el contexto de la optimización de código y la derivación eficiente de programas. En este contexto, un sistema de transformación de programas funcionales, por lo general, toma una especificación como un punto de partida. A continuación se transforma la especificación en un programa de eficiencia aceptable mediante la aplicación de una secuencia de transformaciones que preservan la semántica. Este tipo de transformación

es vertical en el sentido que viene conducido por el flujo de control o de datos del programa. La transformación de programas inherente a la refactorización es diferente de la anterior en que opera sobre la estructura del programa, por lo que, a menudo, sus efectos están bien localizados - es horizontal.

1.3.1. El lenguaje de Programación Maude

Maude fue desarrollado en el SRI (Stanford Research Institute) como un lenguaje de dominio público bajo licencia GNU y se halla disponible a través de su propia página web. Desarrollado por José Meseguer y sus colaboradores como lenguaje para especificar y programar sistemas OO concurrentes [15], el objetivo consiste en diseñar una plataforma común para la investigación, docencia y aplicación de los lenguajes declarativos [10].

Maude es un lenguaje y un entorno de programación que da soporte a una lógica denominada *"lógica ecuacional con pertenencia"* (membership equational logic) y a la *"lógica de reescritura"* (rewriting logic), que son extensiones de la lógica ecuacional. Maude fue concebido como un lenguaje lógico en el sentido estricto de la palabra.

Un programa Maude, en efecto, es una teoría en una lógica, y un cómputo en Maude es una derivación en dicha teoría.

La lógica de reescritura propuesta por José Meseguer en 1990 como marco de unificación de modelos de computación concurrente, es una lógica para especificar sistemas concurrentes con estado que evolucionan por medio de transiciones. La lógica ecuacional con pertenencia da cuenta del concepto de subsorting (o subtipo) incorporando una variedad de posibles relaciones entre sorts. La lógica de reescritura, por su parte, es una lógica relativa al cambio concurrente, que da cuenta naturalmente del concepto de estado y de cómputos concurrentes y no deterministas. Esta lógica constituye un marco general para dotar de semántica a una amplia gama de lenguajes y modelos de concurrencia. Provee, en particular, muy buen soporte a cómputos con objetos concurrentes [31].

Maude tiene sus raíces en el lenguaje OBJ3, que implementa una lógica ecuacional. OBJ3 es, en efecto, un sublenguaje del lenguaje de Maude. La principal diferencia de Maude con respecto a OBJ3 es que da soporte a una lógica más rica, que extiende la lógica ecuacional de géneros ordenados de OBJ3.

Maude soporta una lógica reflexiva de forma sistemática y eficiente a través de un módulo META-LEVEL, que permite no sólo la definición y ejecución de estrategias internas mediante reglas de reescritura, sino también muchas otras aplicaciones, incluyendo la metaprogramación, al considerar programas o especificaciones como datos, y un álgebra de módulos extensible. Finalmente, algunas de las aplicaciones más interesantes de

Maude son aplicaciones de metalenguaje, en cual se usa Maude para crear entornos ejecutables para lógicas diferentes, probar teoremas, o definir lenguajes y modelos de computación [22].

A diferencia de otros lenguajes funcionales como Haskell, que utilizan una estrategia de *evaluación perezosa* (evaluación de orden normal, en inglés *lazy*), Maude es un lenguaje con estrategia de *evaluación impaciente* (evaluación de orden aplicativo, en inglés *eager*). Esto quiere decir que primero calcula los operandos de los operadores antes de abordar el cálculo de los mismos; en otras palabras, evalúa todos los argumentos de una función antes de conocer si serán o no necesarios. Este tipo de estrategia de evaluación puede llevarnos fácilmente a que el cómputo sea no terminante, por lo que el programador debe imponer condiciones a las reglas para garantizar la terminación de éstas.

1.3.2. Técnicas de Refactorización de Programas Funcionales

La *Refactorización de Programas Funcionales* fue un proyecto llevado a cabo por el Laboratorio de Computación de la Universidad de Kent. El objetivo de este proyecto fue investigar la refactorización de los lenguajes de programación funcional, como son Haskell y Erlang. Dos herramientas de refactorización para lenguajes funcionales muy diferentes han surgido del proyecto: Hare, y Erlang Wrangler, que son a la vez los resultados de la labor de desarrollo conjunto de Li, Reinke y Thompson [3].

Algunas refactorizaciones funcionales, tal como renombrar un identificador, eliminar un parámetro no usado, etc., son también válidas para los lenguajes imperativos. Hay muchas refactorizaciones, sin embargo, que son específicas o propias de la programación funcional. Añadir un constructor a un tipo de datos, introducir/eliminar patrones y modificar una expresión particular (por ejemplo, envolver una expresión de tipo **a** como **m a**), son todos ejemplos de refactorizaciones específicas de los lenguajes funcionales [3].

Algo adicional y muy importante es mantener la disposición y estética con la que un programador escribe sus programas, ya que ésta debe ser conservada y reconocida tras el proceso de refactorización; esto incluye la preservación de los comentarios que el programa tenga.

Esta tesis de máster trata de la refactorización de programas escritos en el lenguaje funcional Maude. Para ello, analizaremos las peculiaridades del lenguaje como la genericidad, sort ordenados, coexistencia de ecuaciones y reglas, atributos que se usen para expresar axiomas algebraicos, etc.

1.4. Estructura de la Tesis

Esta tesis está estructurada de la siguiente manera:

Capítulo 1 (este capítulo): se introducen los temas de refactorización, enfocándose en el paradigma de la programación funcional.
Capítulo 2: se presentan las notaciones necesarias y definiciones preliminares sobre los formalismos de reescrituras de términos que serán usados en esta tesis.
Capítulo 3: se describe brevemente las herramientas de refactorización para programas funcionales analizadas.
Capítulo 4: se detallan las diferentes teorías y técnicas estudiadas y que han sido usadas de alguna manera en la realización de esta tesis.
Capítulo 5: se presentan las características más representativas del lenguaje Maude, tales como sus módulos, metanivel, estrategias, etc.
Capítulo 6: se muestra el catálogo de refactorizaciones para Maude que se ha definido.
Capítulo 7: se describe el sistema MR, que corresponde a la implementación de algunas de las refactorizaciones del catálogo.
Capítulo 8: se presenta un caso de estudio y la refactorización realizada en este caso utilizando el sistema MR.
Capítulo 9: se resumen las conclusiones generales y se presentan posibles direcciones para trabajos futuros.

Capítulo 2

Preliminares

En esta sección se introducen algunas nociones y terminología básica de la lógica de reescritura y los sistemas de reescritura de términos, que se utilizará en el resto de la tesis de máster.

Terminología y definiciones básicas

Consideramos una signatura de géneros ordenados Σ con un conjunto finito parcialmente ordenado de tipos $(S, \leq)$. Asumimos una familia S-ordenada $V = \{V_s\} s \in S$ de conjuntos disjuntos de variables. Una variable $x \in V$ de tipo s se denota como $x :: s$, mientras que $f :: s_1, ..., s_n \mapsto s$ representa la signatura del operador $f \in \Sigma$ de aridad n y tipo s. $\Gamma_\Sigma(V)_s$ y Γ_{Σ_s} son, respectivamente, los conjuntos de términos y términos (base con variables) de tipo s.

$\Gamma_\Sigma(V)$ y Γ_Σ representan las correspondientes álgebras de términos. El conjunto de ocurrencias de variables en un término t se denota por $Var(t)$. Por simplicidad, escribimos $\overline{o_n}$ para la *lista* de objetos sintácticos $o_1, ..., o_n$.

Las *Posiciones* se representan como secuencias de números naturales. La secuencia vacía es denotada por Λ, que es la posición raíz de un término. Dado $S \subseteq \Sigma \cup V$, $O_s(t)$ representa el conjunto de posiciones de un término t que son enraizados por símbolos de S. Las posiciones pueden ordenarse por el orden de prefijo: $p \leq q$, si $\exists w$ tal que $p.w = q$. Mediante $t|_p$ denotamos el *subtérmino* de t en la posición p, y $t[s]_p$ denota el resultado de *reemplazar el subtérmino* $t|_p$ por el término s. La igualdad sintáctica se representa por $\equiv$.

Una sustitución $\sigma \equiv \{x_1/t_1, x_2/t_2, ...\}$ es una aplicación del conjunto de variables V en el conjunto de términos $\Gamma_\Sigma(V)$ que satisface las siguientes condiciones: (i) $x_i \neq x_j$, para

todo $i \neq j$, (ii) $x_i\sigma = t_i$, $i = 1, ..., n$ y (iii) $x\sigma = x$, para todo $x \in V \setminus \{x_1, ..., x_n\}$. Por ϵ denotamos la sustitución *vacía.* Una sustitución θ es *más general* que una sustitución σ, en símbolos $\theta \leq \sigma$, si $\sigma = \theta\gamma$ para alguna sustitución γ. Dados dos términos s y t, un unificador de s y t es una sustitución σ tal que $s\sigma = t\sigma$. Por $mgu\,(s,t)$ denotamos el *unificador más general* (*mgu* del inglés *most general unifier*) para s y t. Una instancia de un término t se define como $t\sigma$, donde σ es una sustitución. La sustitución identidad se denota por *id.*

Reescritura de Términos

Los Sistemas de reescritura de términos (condicional y no condicional *SRT* y *SRTC* respectivamente) proporcionan un modelo computacional adecuado para los lenguajes funcionales. En esta sección, haremos un breve resumen de los fundamentos de la reescritura de términos.

Una *teoría ecuacional con géneros ordenados (order-sorted)* es un par $E \equiv (\Sigma, \Delta \cup B)$, donde Σ es una *signatura de géneros ordenados*, Δ es un *conjunto de ecuaciones* de la forma $l = r$, donde l, $r \in \Gamma_\Sigma(V)$, $l \notin V$ y B es un *conjunto de axiomas algebraicos* tales como la asociatividad (A), conmutatividad (C) e identidad (U) declarados para las diferentes funciones definidas en la teoría. Asumimos que Σ puede ser siempre considerada como la unión disjunta $\Sigma \equiv \mathcal{C} \uplus \mathcal{D}$ de símbolos $c \in \mathcal{C}$, llamados *símbolos constructores* y símbolos $f \in \mathcal{D}$, llamados *funciones definidas,* cada uno con una aridad fija, donde $\mathcal{D} \equiv \{f \mid f(t) = r \in \triangle\}$ y $\mathcal{C} \equiv \Sigma - \mathcal{D}$. Dado una ecuación $l = r$, los términos l y r son llamados *lado izquierdo* (o *lhs* del inglés left-hand side) y *lado derecho* (o *rhs* del inglés right-hand side) de las ecuaciones, respectivamente y $Var\,(r) \subseteq Var(l)$.

Las ecuaciones en un teoría ecuacional E son consideradas como reglas de simplificación para ser usadas en la dirección de izquierda a derecha sobre algún término t. Mediante la aplicación repetida de las ecuaciones como reglas de simplificación, eventualmente alcanzamos un término al cuál no se pueden aplicar ecuaciones adicionales. El resultado se denomina la *forma canónic*a (canonical form) de t con respecto *a* E. Esto se garantiza por el hecho que a E se le impone ser terminante y Church-Rosser [14]. El conjunto de ecuaciones en Δ junto con las axiomas ecuacionales de B en una teoría ecuacional E inducen una relación de congruencia sobre el conjunto de términos $\Gamma_\Sigma(V)$ la cuál se denota como $=_E$. E es una presentación o axiomatización de $=_E$. Por abuso de notación, hablamos de la teoría ecuacional E para denotar la teoría axiomatizada por E. Dada una teoría ecuacional canónica E, decimos que una sustitución σ es un E-unificador de dos términos t y t' (en símbolos, $t\sigma =_E t'\sigma$) si es posible reducir $t\sigma$ y $t'\sigma$ a la misma forma canónica módulo la teoría ecuacional.

Una *teoría de reescritura de géneros ordenados* es una tripleta $\Re \equiv (\Sigma, \Delta \cup B, R)$, donde R es un conjunto de reglas de reescritura de la forma $l \to r$, donde l, $r \in \Gamma_\Sigma(V)$, $l \notin V$, y Σ es la unión disjunta por parejas $\mathcal{D}_1 \uplus \mathcal{D}_2 \uplus \mathcal{C}$ tal que $(\mathcal{D}_1 \uplus \mathcal{C}, \Delta \cup B)$ es una teoría ecuacional de géneros ordenados y $\mathcal{D}_2 \equiv \{f \mid f(t) \to r \in R\}$. Los símbolos de $\mathcal{D}_2$ se llaman *símbolos definidos* en las reglas de reescritura, mientras que $\mathcal{D}_1$ recoge los símbolos definidos en la teoría ecuacional. Omitimos Σ siempre que no cause confusión. Dado una regla $l \to r$, los términos l y r son llamados *lado izquierdo* (o *lhs* del inglés left-hand side) y *lado derecho* (o *rhs* del inglés right-hand side) de la regla, respectivamente y $Var\,(r) \subseteq Var(l)$. Una ecuación de la forma $t = t'$ o una regla de la forma $t \to t'$ se dicen que son:

1. *Non-erasing*, si $Var\,(t) = Var(t')$.
2. *Sort preserving*, si para cada sustitución σ, tenemos $t\sigma \in \Gamma_\Sigma(V)_s$ si y sólo si $t'\sigma \in \Gamma_\Sigma(V)_s$.
3. *Sort decreasing*, si para cada sustitución σ, $t'\sigma \in \Gamma_\Sigma(V)_s$ implica $t\sigma \in \Gamma_\Sigma(V)_s$.
4. *Lineal por izquierda (o derecha)*, si t (resp. t') es *lineal*; es decir, no hay ocurrencias repetidas de variables en los términos. Se dice lineal si ambos, t y t', son lineales.

Un conjunto de ecuaciones/reglas se llama non-erasing o sort decreasing, o sort preserving o lineal (izq. o der.), si cada ecuación/regla en el conjunto lo es.

Una teoría ecuacional (resp. teoría de reescritura) se dice que es *condicional* si sus ecuaciones (resp. reglas) son de la forma $(l = r \Leftarrow c)$ (resp. $(l \to r \Leftarrow c)$), donde c es un término que representa la condición. Por otra parte, se puede asociar etiquetas (*labels*) tanto a ecuaciones como a reglas con el fin de identificarlas fácilmente, en la forma $(etiqueta : l = r)$ o $(etiqueta : l \to r)$.

Definimos la *relación de reescritura en un paso* (one-step rewrite relation) en $\Gamma_\Sigma(V)$ como sigue: $t \to_R t'$ si hay una posición $p \in O_\Sigma(t)$, una regla $l \to r$ en R, y una sustitución σ tal que $t \mid_p \equiv t[r\sigma]$. Una instancia $l\sigma$ de una regla $l \to r$ se denomina *redex* (del inglés ***red**ucible **ex**pression*). Un término t sin redexes se llama *forma normal.*

La relación $\to_{R/E}$ de reescritura, *modulo* E se define como $=_E \circ \to_R \circ =_E$. Sea $\to \subseteq A \times A$ una relación binaria sobre un conjunto A. Denotamos el cierre transitivo por $\to^+$, el cierre reflexivo y transitivo por $\to^*$ y la reducción hasta la forma normal por $\to^!$.

Una teoría de reescritura es *suficientemente completa* si tiene suficientes reglas/ecuaciones especificadas, de modo que todas las funciones de la teoría están definidas completamente para todos los valores de los argumentos en el dominio de definición de la función.

Ejemplo 1 Consideremos la siguiente teoría de reescritura $(\Sigma, \Delta \cup B, R)$ tal que $\mathcal{C} \equiv \{b, c, e\}$, $\mathcal{D}_1 \equiv \{a, d\}$, $\mathcal{D}_2 \equiv \{f\}$, $\Delta \equiv \{a = b, d = e\}$, $R \equiv \{f(b, c) \to d\}$ donde B contiene el axioma conmutativo para *f*. Entonces podemos reescribir (en R módulo E) el término $f(c, a)$ a e, por medio de la siguiente secuencia de reescritura *módulo E;* $f\,(c, b) =_B f(b, c) \to_R d =_\Delta e$.

Decimos que la teoría de reescritura $R \equiv (\Sigma, \Delta \cup B, R)$ es terminante con respecto a $\to_{R/E}$, si no hay una secuencia infinita de reescritura $t_1 \to_{R/E} t_2 \to_{R/E} \ldots$ Una teoría de reescritura es confluente o Church-Rosser con respecto a $\to_{R/E}$ si para todos los términos s, t_1, t_2, tal que $s \to^*_{R/E} t_1$ y $s \to^*_{R/E} t_2$, existe un término t s.t $t_1 \to^*_{R/E} t$ y $t_2 \to^*_{R/E} t$.

Capítulo 3

Herramientas para Refactorización

Este capítulo describe las herramientas de refactorización que se han analizado para la realización de esta tesis. Dichas herramientas fueron creados para la refactorización de programas escritos en los lenguajes funcionales Haskell y Erlang. Nos centraremos básicamente en el estudio de HaRe y, muy brevemente, en Erlang Wrangler, para destilar los aspectos esenciales que nos sirvan para abordar nuestro objetivo que es la refactorización de programas escritos en el lenguaje Maude.

3.1. Introducción

Como se dijo en el Capítulo 1, el Haskell Refactorer (HaRe) fue originalmente diseñado e implementado por el grupo de *Refactorización de Programas Funcionales* en la Universidad de Kent. El grupo estuvo conformado por Simon Thompson, Huiqing Li y Claus Reinke.

El objetivo del grupo de Kent fue implementar una herramienta con soporte para refactorizar programas Haskell. Como los editores de texto, las herramientas de refactorización dan soporte a la manipulación interactiva; sin embargo, las herramientas de refactorización necesitan consultar y modificar la sintaxis y semántica del programa en lugar de tratar al programa como una cadena de tipo string [3].

Normalmente para realizar una refactorización, el usuario necesita seguir los siguientes pasos:

1. El código de interés ha de ser seleccionado en el editor. Por ejemplo, un identificador es seleccionado ubicando el cursor en el comienzo de algunas de sus ocurrencias; una expresión es seleccionada y remarcada asimismo con el cursor.

2. El usuario escoge el comando de refactorización y puede ingresar algún parámetro.

3. Finalmente el refactorizador chequeará que el código seleccionado es adecuado para la refactorización, que los parámetros son válidos y que las precondiciones son satisfechas.

Si todo va bien, el refactorizador realizará la refactorización. En otro caso emitirá un mensaje de error y abortará el proceso.

3.2. Herramientas analizadas

Las herramientas analizadas para la refactorización de programas Haskell y Erlang, están implementadas en su propio lenguaje. El análisis del código del programa a refactorizar da lugar a lo que se conoce como *árbol sintáctico abstracto* ASA (*abstract syntax tree*); que servirá de base para conocer la estructura del programa y permitir realizar las diferentes refactorizaciones que cada una realiza particularmente.

Tanto Erlang como Haskell son lenguajes de programación de propósito general y con muchas diferencias entre sí. Haskell es un lenguaje de estrategia perezosa (*lazy*), tipado estáticamente, puramente funcional con orden superior, polimorfismo, clases de tipos y efectos monádicos. Erlang es un lenguaje de programación funcional estricto, impuro, de tipos dinámicos con soporte a funciones de orden superior, ajuste de patrones, concurrencia, comunicación, distribución, tolerancia a fallos y con carga dinámica de código. A diferencia de Haskell, que surgió de una iniciativa académica, Erlang fué desarrollado en el Laboratorio de Ciencias de la Computación de Ericson y ha sido activamente usado en la industria tanto en Ericson como fuera de ella [20].

3.2.1. HaRe para Haskell

A diferencia de los editores de texto, que operan sobre cadenas de string, las herramientas de refactorización necesitan tener conciencia de la sintaxis y semántica de un programa. Típicamente, una refactorización sigue los siguientes pasos:

1. El código de programa es analizado para obtener el ASA.

2. El análisis del programa, tal como extraer el alcance y el tipo de información de los identificadores, es llevado a cabo en el ASA para asegurar que las precondiciones de la refactorización son satisfechas.

3. Si las precondiciones son satisfechas, el ASA será transformado para implementar la refactorización y el programa refactorizado será presentado al usuario en forma de código.

4. En otro caso, el mensaje de error es emitido y el programa permanece sin cambios.

Obviamente, aparte del análisis del programa general y la propia transformación, una herramienta de refactorización para Haskell necesita un *analizador léxico* (*lexer*) y un *analizador sintáctico* (*parser*) para obtener el ASA, un tipo de *verificador* (*checker*) para recoger el tipo de información y un *impresor edulcorado* (en inglés, *pretty-printer*) para presentar el ASA en forma de código.

El grupo de Kent seleccionó las herramientas *Programatica* y *Strafunski* como analizador sintáctico y para la programación genérica, respectivamente. El primero mantiene los espacios en blanco y comentarios, así como la posición de la información del código. Esto da una ayuda a la herramienta de refactorización para preservar la distribución del programa y los comentarios; el segundo es un paquete de software para dar soporte a la programación genérica en las áreas de aplicación que incluye recorridos de términos sobre sintaxis abstractas de gran tamaño [3].

3.2.2. Wrangler para Erlang

Wrangler es una herramienta que soporta la refactorización interactiva e inspección "*code smell*" de programas Erlang. Está integrado con (X)Emacs y con Eclipse. Wrangler está implementado en Erlang. Los ASA son expresados como estructuras de datos Erlang y son usados como la representación interna de programas Erlang. La representación ASA está estructurada de tal manera que todos sus nodos tengan una estructura uniforme y cada nodo pueda ser adjuntado con varias anotaciones tales como: localización, comentarios del código, información estática-semántica, etc.

Wrangler ofrece una *plantilla de alto nivel* (*high-level template*) y un *API basado en reglas* (*rule-based API*), tal que los usuarios puedan escribir refactorizaciones que satisfagan sus propias necesidades de una forma concisa e intuitiva, sin tener que entender la representación subyacente del ASA y otros detalles de implementación [21].

Capítulo 4

Refactorización de Programas Funcionales

Este capítulo presenta las diferentes teorías y técnicas estudiadas y que han sido usadas, de una u otra manera, para realizar las refactorizaciones presentadas en esta tesis a través de transformaciones precisas que se describirán en los capítulos posteriores.

La transformación de programas funcionales tiene una larga historia, con trabajos tempranos en el campo en la década de los 80 como se describe en [27]. Un sistema de transformación de programas toma una especificación funcional o programa y reescribe el programa en otro programa (eventualmente más) eficiente usando reglas de transformación. La transformación de programas también se usa automáticamente durante la fase de optimización realizada por los compiladores, ya sea a nivel de código fuente o de representaciones en un lenguaje intermedio. En esta tesis, el objetivo no será tanto optimizar la eficiencia del código como mejorar su estructura, legibilidad y facilidad de mantenimiento.
Trabajos representativos de transformación de programas son los bien conocidos *plegado/desplegado* (*folding/unfoldind*) de Burstall y Darlington [4], la *fragmentación* (*slicing*) de Weiser [33], el *unraveling* de Marchiori [23], la *memorización* (*memoization*) de Michie [24] y la técnica de *recursión de cola* (*tail recursion*), que presentaremos sistemáticamente a lo largo de este capítulo.

4.1. Detección de código duplicado

Los programas están sometidos permanentemente a labores de desarrollo y mantenimiento, lo que a menudo conduce a la generación de código duplicado. Los programas con

grandes fragmentos de código duplicado son muy difíciles de entender y aún más difíciles resulta mantenerlos. El código duplicado también hace que los programas sean más largos. Además cuando se realizan mejoras o correcciones de errores para una instancia de código duplicado, esa modificación también se debe replicar en todas las instancias del mismo código duplicado. Repetir correcciones de errores y modificaciones conduce a menudo a nuevos y numerosos errores.

Copiar una pieza de código es, a menudo, más simple y rápido que escribir el código desde cero, o hacerlo más general. Algunas organizaciones controlan la cantidad de código que un programador escribe, motivando al empleado en algunos casos a duplicar bloques de código. En algunas situaciones, en cambio, el código es más eficiente si contiene duplicados, pues evita tener que poner un montón de llamadas a procedimiento dentro del código.

4.1.1. Uso del Plegado / Desplegado

Las transformaciones de *plegado/desplegado* fueron introducidas por Burstall y Darlington en [4] para los programas funcionales. Por lo general se consideran las técnicas más básicas y poderosas para transformar programas.

Hay seis reglas básicas de transformación, que son:

- *Desplegado (Unfolding)*: Reemplazar una llamada de función con el cuerpo de dicha función, sustituyendo los parámetros actuales por los parámetros formales.
- *Plegado (Folding)*: Reemplazar una expresión con una llamada a una función, si el cuerpo de dicha función puede ser instanciado a la expresión dada con parámetros adecuados.
- *Instaciación (Instantiation):* Introducir una instancia (por aplicación de una sustitución) de una regla o ecuación existente.
- *Abstracción (Abstraction)*: Introducir una cláusula condicional para asignar un nuevo nombre a una expresión. Esto puede ser usado en combinación con el plegado e instanciación.
- *Definición (Definition)*: La adición de una nueva declaración de función.
- *Leyes (Laws):* El uso de leyes acerca de las primitivas del lenguaje.

La ventaja de usar esta metodología es que es simple y muy eficiente y conduce a un amplio repertorio de transformaciones de programas. Por ejemplo, con la técnica de plegado

en el sentido que hemos descrito anteriormente (reemplazar todas las subexpresiones en un programa), es posible eliminar código duplicado ya que identifica instancias comunes dentro de las funciones. Sin embargo, según Brown [2] la desventaja de esta técnica es que la introducción de definiciones puede resultar no terminante y se debe mantener un historial de todas las versiones del programa mientras éste está siendo transformado.

4.2. Eliminación de código

Se llama código muerto (*dead code*) aquél código que no es alcanzable durante la ejecución de un programa. La eliminación de código muerto es una optimización del compilador, usada para reducir el tamaño del programa, al eliminar las partes del programa que no son necesarias.

En esta sección introducimos dos tipos de eliminación de código: eliminación de código muerto y eliminación de código irrelevante. La *eliminación de código muerto* se preocupa de coger una función de nivel más alto y eliminar alguna declaración anidada dentro de esa función que no es necesaria. La *eliminación de código irrelevante* es una generalización de la anterior y se centra en eliminar declaraciones anidadas que no son necesarias para computar una subexpresión en particular en el nivel más alto. La eliminación de código irrelevante puede ser usada para ayudar a los programadores a realizar tareas de depuración (*debugging*).

4.2.1. Eliminar código muerto

Como hemos dicho, se llama código muerto a aquellos fragmentos de código que nunca son ejecutados o cuya ejecución no contribuye al resultado final y que aparecen a menudo como resultado de procesos de optimización, modificación y reutilización de código.

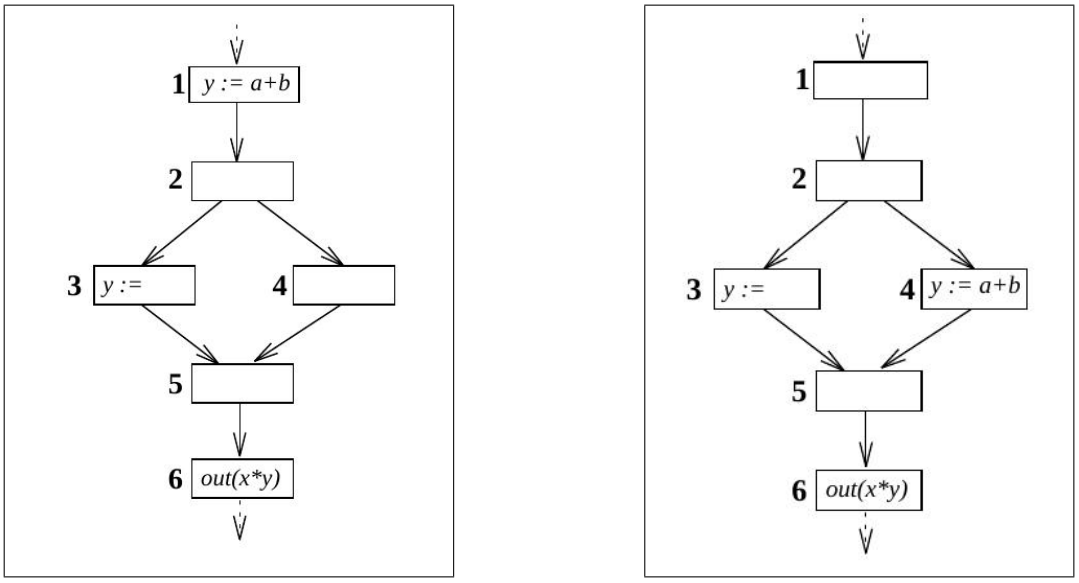
El *código muerto* es código innecesario, inapropiado y que puede ser (o debería) eliminado. Lo opuesto del código muerto es código vivo (*live code*).

Se puede mejorar significativamente la calidad de un programa al eliminar el código muerto. Mientras la funcionalidad no sea afectada, ésto mejorará la calidad interna. También ayudará en el mantenimiento ya que reduce el tamaño del código, lo hace más fácil de entender y previene la introducción de fallos. La eliminación de código muerto también permite mejorar la eficiencia de un programa y evitar ciertas ejecuciones innecesarias.

Usualmente, una asignación se considera innecesaria si es *totalmente* muerta, i.e., si el contenido de la variable que aparece en la parte izquierda de la asignación no se usa en el resto del programa [17].

Consideremos el ejemplo de la Figura 4.1(a), en donde pueden observar en el nodo *1*, aparece una asignación a la variable "y" que está muerta en la rama izquierda (nodos 1-3) pero viva en la rama derecha (nodo 6).

Sin embargo, moviendo en la rama derecha la asignación $y := a + b$ desde el nodo *1* al nodo 4 el problema puede ser solucionado, como se muestra en la Figura 4.1(b).



(a) Analizada (b) Removida

FIGURA 4.1: Asignación parcialmente muerta

4.2.2. Eliminar código irrelevante

La eliminación de código irrelevante es útil para propósitos de depuración y en cierta medida se utiliza en la depuración algorítmica [30]. En la depuración algorítmica, la herramienta de depuración plantea una serie de preguntas al usuario sobre si una subexpresión particular en el código está generando el correcto resultado o no. Como las preguntas se van formando simultáneamente, las partes particulares del programa que la herramienta de depuración ha cuestionado serán claramente focalizadas. Sin embargo, el programador bien podría intuitivamente ubicar el error del código, identificar la subexpresión que supuestamente podría provocar un fallo en el funcionamiento y así acelerar la corrección del mismo.

Como se ha descrito arriba, es posible generalizar la técnica de eliminación de código muerto para facilitar la depuración. En concreto, es posible seleccionar una subexpresión

de interés y en la función asociada podar aquellas declaraciones que no son necesarias para evaluar aquella subexpresión. Esta generalización particular de la eliminación de código muerto no es una verdadera refactorización sino maás bien una transformación que cambia la semántica.

4.2.3. Slicing (Fragmentación)

La fragmentación de programas (*program slicing*) es una técnica que ayuda a la *depuración*, *testing*, *integración*, *comprensión y mantenimiento de programas*, *paralelismo*, *reutilización* e *ingeniería reversa*, etc., por medio de porciones aisladas de código, denominadas *slices* (o fragmentos), de un programa para reducir su complejidad. Estas porciones de código afectan potencialmente a los valores calculados para algún punto de interés del programa, al que se conoce como criterio de *slicing* (*slicing criterion*). Dos formas principales de *slicing* son el *slicing* estático (*static slicing*) introducido por M.D. Weiser [33] para el paradigma imperativo, debido a que no considera ninguna ejecución particular del programa y, el *slicing* dinámico (*dinamic slicing*) de Korel y Laski [18] que tiene en cuenta los valores de las variables de interés durante la ejecución del programa. Sin embargo, existe una aproximación intermedia entre estas dos, denominada *slicing* condicional (*conditional slicing*) [5], que contiene aquellas partes del programa en donde cierta condición aplicada sobre las entradas se cumple.

Un *slicing* estático de un programa P con respecto al *criterio de slicing (p, v)*, donde p es una posición o sentencia determinada del programa y v la variable de interés, es un subprograma P' que contiene aquellas sentencias que afectan al valor de v calculado en p. Por lo tanto, un *slice* estático de un programa puede ser generado para eliminar aquellas declaraciones que no tienen efecto en el criterio de *slicing*.

El *slicing* dinámico fué propuesto para propósitos de depuración, pero su aplicación ha sido extendido mucho más allá. Normalmente los *slices* estáticos son típicamente largos, pero a cambio son independientes de cualquier ejecución posible del programa original; los *slice* dinámicos son mucho más pequeños, pero sólo atienden a una entrada particular.

Mientras que en el *slicing* estático no se tiene información sobre las entradas concretas del programa y en el *slicing* dinámico se emplea un conjunto de valores específicos para los argumentos, en el *slicing* condicional el criterio contiene adicionalmente una formula lógica que caracteriza a un conjunto de valores de entrada. Por lo tanto, el *slice* condicional del programa estará compuesto por aquellas partes del programa que pueden ser (potencialmente) ejecutadas cuando la condición sobre las entradas se cumple.

La fragmentación de programas puede ser usada en una refactorización, al reducir parámetros, variables innecesarias y todo lo que tiene que ver con diseño. Por ejemplo, cuando una función que devuelve una tupla de valores, por ejemplo (f, g), una refactorización podría consistir en hacer que devuelva dichos valores f y g de forma separada.

4.3. De la reescritura condicional a incondicional (*unraveling*)

La reescritura de términos condicional es un paradigma crucial en la especificación algebraica de tipos de datos abstractos ya que un medio natural para ejecutar las especificaciones ecuacionales. Muchos lenguajes, incluido Maude, proveen motores de reescritura condicional para permitir al usuario ejecutar y razonar sobre las especificaciones. La reescritura condicional juega un rol fundamental en la programación lógico-funcional. Muchos autores consideran que el paradigma de reescritura es poderoso, pero la reescritura condicional conduce a programas significativamente lentos comparados con los incondicionales que tienen la misma funcionalidad. La razón principal es que, con el fin de reducir un término, el motor de reescritura necesita mantener el *control del contexto* para cada regla condicional que es seleccionada para ser aplicada. Entiéndase por control de contexto el estado de la evaluación de la condición, más el término en la parte derecha de la regla o ecuación a usar en caso de que la condición se evalúe a *true* [28].

En inglés se conoce como *unraveling* a la transformación de los sistemas de reescritura de términos condicionales *SRTC* en sistemas de reescritura de términos incondicionales *SRT* propuesta por Marchiori [23]. Para reducir un término, un motor de reescritura necesita mantener el control del contexto para cada regla condicional que es tratada. Al transformar un sistema condicional en incondicional será necesariamente garantizar que no se modifica su semántica y, algo muy importante, asegurar la mantenibilidad del sistema. Es decir, si el código resultante de la transformación no es claro ni mucho menos entendible por el programador, difícilmente será posible realizar un mantenimiento o un cambio necesario en el futuro.

Existe una extensa literatura dedicada a las transformaciones de *unraveling* preservando ciertas propiedades de los SRTC, por ejemplo terminación y/o confluencia, que caen fuera del alcance de este trabajo. El lector puede encontrar más detalle en la literatura de Ohlebusch [26].

Dos casos particulares que se han estudiado son los siguientes:

4.3.1. *Primer caso: Ecuaciones condicionales complementarias*

Consideremos dos ecuaciones condicionales como (e1) y (e2) que definen la semántica de algún símbolo de función definido[1], en donde $r1 \neq r2$, $u_1 = s_1 \wedge ... \wedge u_n = s_n$ y $v_1 = t_1 \wedge ... \wedge v_n = t_n$:

$$l \rightarrow r_1 \Leftarrow u_1 = v_1 \wedge ... \wedge u_n = v_n \tag{e1}$$

$$l \rightarrow r_2 \Leftarrow s_1 \neq t_1 \wedge ... \wedge s_n \neq t_n \tag{e2}$$

Consideremos el operador $if\,(\_,\_,\_)$, junto a las siguientes reglas que lo definen:

$$if\,(true, x, y) \rightarrow x \tag{e3}$$

$$if\,(false, x, y) \rightarrow y \tag{e4}$$

En este caso es posible escribir las ecuaciones condicionales (e1) y (e2), en una única ecuación incondicional que tiene la siguiente forma:

$$l \rightarrow if(u_1 = v_1 \wedge ... \wedge u_n = v_n,\, r1,\, r2) \tag{e5}$$

Esto hace que el lhs de (e5) se reescribe a *r1* siempre que las guardas se evalúen a *true* y, se reescribe a *r2* si sus guardas se evalúan a *false*. La ecuación (e5) tiene la misma semántica que las ecuaciones (e1) y (e2) expresaban por separado.

4.3.2. *Segundo caso: Otras ecuaciones condicionales*

Asumimos que el primer caso no se ha cumplido y se debe analizar cada ecuación de la forma:

$$l \rightarrow r \Leftarrow u_1 = v_1 \wedge ... \wedge u_n = v_n \tag{e6}$$

por separado. Hemos hecho un estudio, comparación y análisis de la cobertura que dan cada una de las transformaciones presentadas en [28] y [25] sobre unraveling. En [25]

[1]Las ecuaciones (e1) y (e2) no necesariamente son las únicas ecuaciones que definen la semántica de algún símbolo de función, ya que pueden existir más ecuaciones condicionales y/o incondicionales que lo hagan, como se aprecia en el ejemplo del operador *max* en la Subsección 6.3.1.

los SRTs resultantes del unraveling son sobreaproximaciones del SRTC original w.r.t a la reducción y también son usados para analizar las propiedades de los SRTC originales tales como propiedades sintácticas, modularidad y terminación operacional ya que los SRTs son en general mucho más fáciles de manejar que los SRTC.

En [28], la reescritura es el mecanismo principal para llevar a cabo las transformaciones, en donde los símbolos de función de la signatura original son manipulados debido a que se incrementa la aridad de algunos de ellos.

También aparece un operador binario de igualdad, denotado por $equal?\,(t, t')$ que devuelve *true* sí y sólo sí las formas normales de t y t' son idénticas; de lo contrario el resultado devuelto será *false*. La regla obvia para ese operador es $equal?\,(x, x)$, que será siempre evaluada a *true*. Además se considera el operador $if\,(\_, \_, \_)$ y las reglas descritas en el primer caso (Subsección 4.3.1).

Para un operador $\sigma \in \Sigma$ tenemos:

$$equal?\,(\sigma\,(x_1, ..., x_n)\,, \sigma\,(y_1, ..., y_n)) \rightarrow equal?\,(x_1, y_1) \wedge ... \wedge equal?\,(x_n, y_n),$$

donde $x_1, ..., x_n, y_1, ..., y_n$ son variables distintas. Estas reglas propagan la igualdad de dos términos que tienen el mismo operador como símbolo raíz a la igualdad de sus correspondientes subtérminos. Tenemos que notar que *equal?* necesita evaluarse de forma impaciente (*eager*) sobre sus argumentos.

Para una signatura Σ, sea Σ' la signatura Σ extendida con todas las operaciones auxiliares descritas arriba y sea $I(\Sigma)$ el sistema de reescritura de Σ' que contiene todas las reglas anteriores (las reglas que contiene son incondicionales). Llamamos a $I(\Sigma)$ la "infraestructura de sistema de reescritura" de Σ[2].

4.3.2.1. Agregar el contexto

En esta tesis seguiremos la aproximación de [28] para transformación de reglas condicionales a incondicionales cuando sea factible determinar que ésto mejorará el rendimiento del sistema sin oscurecer su legibilidad. Para adherir el contexto de control consideremos un sistema de reescritura $R = (\Sigma, E)$. Sea $f \in \Sigma_n$ una operación de n argumentos. Para cada n y cada f asociamos un único número entre 1 y k_f a cada regla de reescritura condicional en R, cuyo operador raíz de lhs es f, es decir, la regla de la siguiente forma:

$$f\,(t_1, ..., t_n) \rightarrow r \;\; if \;\; u_1 = v_1, ..., u_m = v_m \qquad \text{(r1)}$$

[2]La teoría completa y demostraciones de teoremas se encuentran en [28]

donde $t_1, ..., t_n, r, u_1, v_1, ..., u_m, v_m$ son términos, $m \geq 1$ y k_f es el número total de reglas. Podemos notar que k_f es 0 si no hay reglas que tengan f como raíz de su lhs o si las reglas son incondicionales.

La idea principal de Rosu se basa en extender la signatura Σ, reemplazando cada $f \in \Sigma_n$ con la adición de k_f argumentos basado en el conjunto de las reglas.

Se define a continuación la signatura $\overline{\Sigma}$, que resulta de reemplazar cada $f \in \Sigma_n$ por un operador de $n + k_f$ argumentos, $\bar{f} \in \overline{\Sigma}_{n+k_f}$. Los k_f argumentos adicionales son escritos a la derecha de los n argumentos y pueden tener sólo dos valores que son *true* o *false*.

Algo muy importante en esta teoría es reemplazar todas las operaciones de Σ por las correspondientes operaciones en $\overline{\Sigma}$. La idea principal de esto es transformar el contexto de control en datos de contexto: el argumento i-ésimo de una operación f indica si la i-ésima regla que tiene f en la raíz de su lhs está o no habilitada en esa posición; si es *true* la regla puede ser aplicada y si es *false* indica que la regla ya ha sido utilizada en esa posición pero que ha fallado, así que ya no es necesario volverla a usar.

Sea $\widetilde{t}^X$ un término que se obtiene transformando el término t de Σ en un término de $\overline{\Sigma}$, reemplazando cada operación $f \in \Sigma$ por $\bar{f} \in \overline{\Sigma}$ y agregando distintas variables para los argumentos adicionales.

Dada una signatura Σ de términos t de la forma $f\,(t_1, ..., t_n)$ y dado un número natural i entre 1 y k_σ, entonces tenemos que $\widetilde{t}^X_{i/true}$ denota los términos de la signatura $\overline{\overline{\Sigma}}$ de la forma $f\left(\widetilde{t_1}^X, ..., \widetilde{t_n}^X, b_1, ..., b_{i-1}, true, b_{i+1}, ..., b_{k_f}\right)$, que reemplaza b_i en $\widetilde{t}^X$ por *true* y donde $b_1, ..., b_{k_f}$ son variables frescas que no ocurren ni en V ni en $\widetilde{t_1}^X, ..., \widetilde{t_n}^X$. De manera similar, $\widetilde{t}^X_{i/false}$ denota $\sigma\left(\widetilde{t_1}^X, ..., \widetilde{t_n}^X, b_1, ..., b_{i-1}, false, b_{i+1}, ..., b_{k_f}\right)$, que reemplaza b_i en $\widetilde{t}^X$ por *false*. Estos $\widetilde{t}^X_{i/true}$ (resp. $\widetilde{t}^X_{i/false}$) contienen la información adicional del control de contexto si la i-ésima regla condicional de f está habilitada.

4.3.2.2. La transformación de eliminación de condiciones

Basándonos únicamente en agregar el contexto de control como datos de contexto, el SRT resultante de la transformación no es confluente como veremos al final de esta sección, por lo que se plantea la siguiente solución al problema.

La transformación correcta (propuesta por Rosu) necesita un mecanismo para informar el término a reducirse después de cada aplicación satisfactoria de una regla de reescritura, ya que algunas reglas condicionales que se han intentado aplicar antes y que han fallado, ahora pueden ser satisfactorias. De una forma más precisa, se necesita atravesar el término a lo largo del camino desde la raíz hasta de la posición actual (donde

la regla se aplicó satisfactoriamente) y forzar que todos los argumentos auxiliares de la operación en este camino sean *true*. Para ello se plantea un nuevo operador unario $\{\_\}$, que denota que el término entre llaves ha sido modificado, junto con las reglas de reescritura apropiadas para propagar la información hacia arriba, modificando los "bits de aplicabilidad". Por ejemplo consideremos una regla

$$\begin{aligned} f\left(x_1,...,x_{i-1},\{x_i\},x_{i+1},...,x_n,b_1,...,b_{k_\sigma}\right) \rightarrow \\ \rightarrow \{f\left(x_1,...,x_{i-1},x_i,x_{i+1},...,x_n,true,...,true\right)\} \end{aligned} \tag{r2}$$

Para mantener la idempotencia debemos considerar la regla:

$$\{\{x\}\} \rightarrow \{x\} \tag{r3}$$

Formalmente, para un sistema de reescritura condicional R con signatura Σ, denotamos $\overline{\Sigma}'_{\{\}}$ la signatura extendida de $\overline{\Sigma}'$ descrita en la sección anterior, con el operador unario $\{\_\}$. Sea $\overline{R}$ el sistema incondicional de $\overline{\Sigma}'_{\{\}}$ en el que se ha extendido $I\left(\overline{\Sigma}\right)$ con el operador $\{\_\}$. Además de las reglas incondicionales antes mencionadas se añaden también las siguientes reglas. Para cada regla condicional $(m \geq 1)\ l \rightarrow r\, if\, u_1 = v_1, \cdots, u_m = v_m$ sobre el conjunto de variables V en R, tal que es la i-ésima regla condicional en R que tiene la operación raíz de l como símbolo raíz de su parte izquierda, se agrega a $\overline{R}$ la regla siguiente;

$$\widetilde{t}^X_{i/true} \rightarrow if\left(equal?\left(\{\overline{u_1}\},\{\overline{v_1}\}\right) \wedge ... \wedge equal?\left(\{\overline{u_m}\},\{\overline{v_m}\}\right),\{\overline{r}\},\widetilde{l}^X_{i/false}\right)$$

Para cada regla incondicional $l \rightarrow r$ en R sobre las variables V, agregamos a $\overline{R}$ una regla de reescritura incondicional $\widetilde{t}^X \rightarrow \{\bar{r}\}$.

A modo de ilustración, consideremos el siguiente ejemplo tomado de [28]:

Ejemplo 1: SRTC de Rosu

$$\begin{aligned} &f\left(g\left(x\right)\right) \rightarrow x\, if\, x = 0, \\ &g\left(g\left(x\right)\right) \rightarrow g\left(x\right) \end{aligned}$$

Su correspondiente SRT generado por la transformación anterior contiene las siguientes reglas:

$$\begin{aligned} &\{\{x\}\} \rightarrow \{x\} \\ &f\left(\{x\},b\right) \rightarrow \{f\left(x,true\right)\} \\ &g\left(\{x\}\right) \rightarrow \{g\left(x\right)\} \\ &f\left(g\left(x\right),true\right) \rightarrow if\left(equal?\left(\{x\},\{0\}\right),\{x\},f\left(g\left(x\right),false\right)\right), \\ &\qquad\qquad g\left(g\left(x\right)\right) \rightarrow \{g\left(x\right)\} \end{aligned}$$

Observamos que el SRT obtenido no es confluente, porque el término admite dos formas normales que son t y $\{t\}$, pero es fácil inferir a partir de $\{t\}$ la forma normal deseada, que es t.

4.4. Memorización de valores computados

El término *memoization* fue acuñado por Donald Michie en 1968 [24] para referirse a la memorización, que es una técnica para optimizar programas basada en el almacenamiento de valores para evitar recomputarlos cada vez que una función es invocada. La idea básica de la memorización es realizar cálculos sólo una vez y, en lugar de volver a calcularlos, buscar en una tabla los valores previamente calculados. En particular, si las funciones son computacionalmente costosas, la memorización acelera la ejecución. Sin embargo, hay un inconveniente en que el consumo de memoria puede llegar a crecer bastante. No obstante, muchas técnicas avanzadas de programación, como la programación dinámica, dependen de la memorización. Aunque la memorización puede aplicarse de manera efectiva en la mayoría de lenguajes de programación, a menudo se asocia a la programación funcional, dado que es sencillo convertir las funciones recursivas en las funciones equivalentes que hacen uso de la memorización.

4.4.1. Memorización automática

La memorización está basada en la idea de guardar los resultados de cada llamada a función, indexada por los argumentos a esa llamada, en una tabla *hash* en memoria. Esta tabla *hash* es consultada antes de cada llamada a dicha función para comparar sus argumentos. Si el argumento ha sido computado previamente, se devuelve el correspondiente valor de la tabla; en otro caso se realiza la llamada y su resultado se agrega a la tabla *hash* con el argumento como clave y a continuación se devuelve ese valor como resultado. En la Figura 4.2 se muestra una gráfica de la memorización, en donde la función de entrada es $f(x)$ con argumento x (x podría ser un conjunto de argumentos).

La semántica e implementación de las búsquedas *memo* son críticas en esta optimización de código funcional. Según [1], debemos proveer control sobre los siguientes tres aspectos que son claves para una implementación correcta de la memorización:

4.4.1.1. Igualdad

El esquema de la memorización necesita buscar en una tabla para ajustar al argumento actual. Tal búsqueda requiere al menos un test de igualdad. Incluso en ciertas estructuras

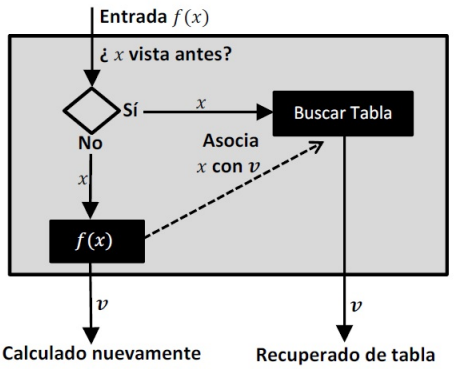


FIGURA 4.2: Memorización

que se utilizan en implementaciones en lenguajes estándar dicho test requiere atravesar la estructura completa. El coste de tal búsqueda puede paliar la ventaja de la memorización y puede incluso cambiar el comportamiento asintótico de la función. Para solventar el problema se han hecho varias propuestas. La primera propuesta es que la memorización se base en el hecho de que no necesita ser exacta, lo que podría dar resultados no iguales cuando dos argumentos son actualmente iguales.

Otro enfoque para reducir el coste del test de igualdad es asegurarse que hay sólo una copia de cada valor, vía técnicas conocidas como "*hash consing*" [Goto y Kanada, 1976, Allen, 1978, Spitzen y levitt, 1978]. En la práctica, el *hash-consing* puede ser caro debido a la gran demanda de memoria y la interacción con el "*garbage collection*" [Pugh, 1988, Appel y Gonalves, 1993, Murphy et al., 2002].

4.4.1.2. Dependencias precisas

El resultado de una llamada a función debe ser almacenado respecto a su dependencia verdadera, lo que hace que se maximice la reutilización de los mismos. El asunto surge cuando la función examina sólo parte o aproximación de su parámetro. Dicho de otra manera, en un test de igualdad aquellas partes no examinadas deben ser ignoradas. Consideremos el siguiente ejemplo:

$$fun\, f\,(x, y, z) = if\,(x > 0)\ then\, fy\,(y)\ else\, fz\,(z)$$

El resultado de f depende de (x, y) o (x, z). Además, depende de forma aproximada de x (cuando es o no positiva) en lugar de su valor exacto. Por ejemplo, la entrada *memo* para $f\,(7, 11, 20)$ se ajustaría a las llamadas $f\,(7, 11, 30)$ y $f\,(4, 11, 50)$, ya que cuando x es positivo, el resultado depende únicamente de y.

Varios investigadores han comentado que el ajuste parcial puede ser muy importante en algunas aplicaciones [Pennings et al., 1992, Pennings, 1994, Abadi et al., 1996, Heydon et al., 2000]. También, sus técnicas pueden cambiar la complejidad asintótica de un programa, hacerlo difícil poder evaluar los efectos de la memorización.

4.4.1.3. Gestión de espacio

Otro problema con la memorización es su elevado requerimiento de espacio. Cuando se ejecuta un programa, sus tablas memo pueden convertirse en gigantescas y esto limita la utilidad de la memorización. Para aliviar este problema, las tablas memo o entradas individuales deberían estar disponibles bajo control del programador.

4.4.2. Aplicaciones de la Memorización

Según Hall-Mayfield en [13], existen cuatro objetivos básicos al realizar una memorización automática:

- Repeticiones dentro de una llamada a función.
- Repeticiones al paso del tiempo.
- Ejecuciones fuera de línea (*off-line*).
- Mejora del rendimiento.

En lo que sigue discutiremos separadamente dichas aplicaciones.

4.4.2.1. Repeticiones dentro de una llamada a función

Este caso ocurre cuando una rutina llama a alguna subrutina (o ella misma recursivamente) más de lo necesario, lo que puede dar lugar a cálculos extra. El punto importante es que las llamadas recursivas forman un grafo y no un árbol y sus cálculos son repetidos, como se puede apreciar en la Figura 4.4 (*Directed Acyclic Graph, DAG*[3] un grafo dirigido acíclico). Es decir, determinar un orden de llamada para evitar estos cálculos repetidos pueden no parecer obvios y esto hace que dichos cálculos sean un candidato ideal para la memorización.

[3]Un DAG es un grafo dirigido donde no hay camino que comienza y termina en el mismo vértice (es decir, que no tiene ciclos). También se conoce como un grafo acíclico orientado. Un DAG tiene al menos un origen (es decir, un nodo que no tenga aristas entrantes) y un sumidero (es decir, un nodo que no tiene aristas salientes).

En este caso la memorización puede resultar en la reducción de tiempo algorítmico exponencial a tiempo polinomial o lineal en la primera invocación, sin necesidad de reescribir el código o diseñar un algoritmo dinámico.

La recursión es fundamentalmente una computación *top-down* (de arriba abajo), donde se observa que un grafo computacional como el de la Figura 4.3 (los nodos son las llamadas a funciones, los arcos son dependencias de llamadas y las flechas se dirigen desde el llamador al llamado), puede ser representado como un árbol de computación (Figura 4.4).

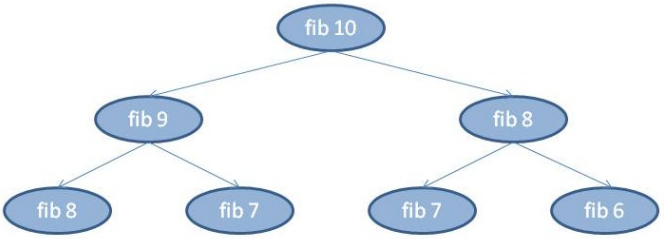


FIGURA 4.3: DAG de fibonacci sin memorización

En consecuencia, la memorización es una optimización de cálculos *top-down*; es decir, primero calcular en profundidad para dar una respuesta. La Figura 4.4 muestra la ejecución de fibonacci haciendo uso de la memorización, en donde apreciamos que las llamadas siguen siendo las mismas, pero los óvalos de puntos son las llamadas que no se computan, cuyos valores son en su lugar buscados hacia arriba (en la tabla *hash*) y sus flechas emergentes muestran que el valor computado fue devuelto por el memorizador.

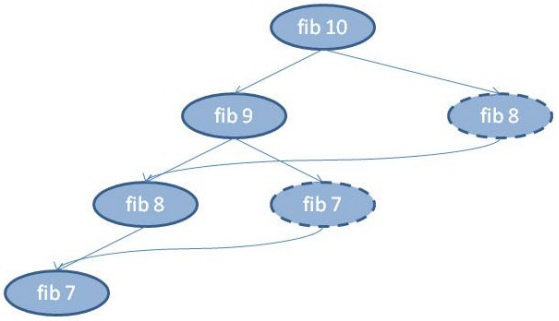


FIGURA 4.4: Árbol de fibonacci con memorización

4.4.2.2. Repeticiones al paso del tiempo

En la sección anterior discutimos una rutina central que invoca funciones de bajo nivel en repetidas ocasiones. Cambiar el algoritmo en este nivel o introducir una estructura de datos local a una rutina central podría ganar muchos de los mismos beneficios de eficiencia que produce la memorización automática, aunque decremente la flexibilidad.

Sin embargo, en un entorno de programación puede ocurrir que diferentes secciones del sistema, escrito por diferentes programadores pueden acceder a una misma función. Alternativamente, en un sistema interactivo, el usuario puede invocar cálculos en diferentes instantes de tiempo que hacen uso de algunas de las mismas piezas. En estos casos, no hay una rutina central que pueda manejar la secuencia de llamadas y la única alternativa para soportar la memorización automática es tener la rutina en cuestión manejando sus propias estructuras de datos globales para recordar resultados previos. En este contexto, la memorización ofrece una eficiente y conveniente alternativa.

4.4.2.3. Ejecuciones fuera de línea (*off-line*)

En la sección anterior hemos discutido cómo la memorización puede ahorrar cálculos de invocaciones repetidas potencialmente largos. Esto deja sin resolver el caso en el que la primera invocación es igualmente muy costosa. Normalmente esto lleva a construir un archivo de datos de propósito especial y llenar los valores con ejecuciones *off-line* de las rutinas más costosas. Entonces, la función que se ejecute se modifica de tal manera que pueda acceder a ese archivo. La memorización automática proporciona un método para hacer lo mismo, manteniendo la transparencia y la facilidad de uso de la memorización y sin obligar a que el programador debe conocer qué rangos de valores se guardan en el archivo de datos y qué valores deben ser recién calculados. La idea es que la función se memorice y se ejecute fuera de línea de la manera normal en los casos de interés. El contenido de la tabla *hash* se guarda en el disco en un archivo con un nombre asociado a la función.

4.4.2.4. Mejora del rendimiento

Sin duda, la memorización vale la pena para los casos de costes de ejecución importantes ya que puede producir mejoras igualmente importantes. En los casos más pequeños, sin embargo, la memorización proporciona un método rápido, pero difícil de usar, para determinar dónde dedicar el esfuerzo de optimización. La simplicidad es la clave: herramientas que tardan mucho tiempo para ser utilizadas serán utilizados sólo ocasionalmente; herramientas que son fáciles de usar para los programadores serán utilizadas con más frecuencia.

4.4.3. Memorización de Operadores

Según las aplicaciones vistas en la Subsubsección 4.4.2.1, vamos a considerar las llamadas recursivas a una función $f\,(x)$ tal que $f \in \mathcal{D}$ de aridad $n \geq 1$ y de argumentos x (x puede

ser un conjunto) y un conjunto de ecuaciones $\triangle_f = e_1, ..., e_n$ que definen la semántica de f. Asumimos también que se cumple la siguiente condición (es el caso comentado por Hall en [13]): existe $e_i \in \triangle_f$ tal que e es de la forma $f(x) = g(f(x))$ o de, manera más general, $f(x) = g(f(x), f(x'), \ldots, f(x''))$, en donde $g \in \mathcal{D}$ es una función de aridad $m \geq 1$. Como g utiliza repetidamente los valores que devuelve la función recursiva f, por cuestiones de eficiencia será conveniente poder hacer uso de valores previamente calculados en lugar de recalcularlos. Esto es justamente lo que logramos a través de la memorización.

4.5. Recursión de Cola (*Tail Recursion*)

La *recursividad* es un concepto clave en ciencias de la computación y matemáticas y juega un rol importante en la adquisición de competencias con respecto a la descomposición de problemas, inducción o abstracción funcional. La recursión es una herramienta poderosa para resolver problemas que constituyen una alternativa atractiva a la iteración, especialmente cuando los problemas pueden resolverse usando el enfoque de *divide y vencerás* [29].

La recursión de cola (*tail recursion*) es una forma de recursividad que puede ser implementada mucho más eficientemente que una recursividad general. En general, una llamada recursiva requiere que el compilador reserve espacio en la pila (*stack*) en tiempo de ejecución para cada llamada aún no resuelta (*almacenanamiento y restaura*ción del contexto). El consumo de memoria puede hacer que la recursividad sea ineficiente e inaceptable para representar algoritmos repetitivos largos o de tamaño ilimitado.

Una función se dice que es recursiva de cola si su definición es de *cola recursiva*. La definición de una función f es de cola recursiva siempre que haya al menos una llamada recursiva a f en el cuerpo de la definición y tal llamada recursiva a f es una cola recursiva [11].

En la siguiente sección recordamos qué significa que una llamada recursiva sea cola recursiva en una definición como la siguiente[4]:

$$(defun\ f\ (x1 \ldots xn\ )$$

$$body)$$

[4]Las funciones de los ejemplos tomados de [11] están escritas en el lenguaje ACL2. Se puede encontrar información amplia de dicho lenguaje en [32].

Asumiremos que el cuerpo (*body*) no contiene macros o aplicaciones lambda. Es decir, en lo que sigue asumimos que se han expandido todas las macros en el cuerpo y se han reducido las aplicaciones lambda usando $\beta - reducción$.

4.5.1. Condiciones de la recursión de cola

Una llamada recursiva a f en el cuerpo de su definición es una cola recursiva en el caso de que estas dos condiciones se cumplan:

1. La llamada a f no está en la rama de prueba de ningún if.
2. En cualquier rama que contiene la llamada a f, sólo el if puede aparecer por encima de la llamada a f.

Ejemplos de recursión de cola

- ```
  (defun f (x)
    (if (f x)
        x
        x))
  ```

  La llamada a `f` en este cuerpo viola la primera condición de arriba, por lo tanto **no** es una cola recursiva.

- ```
  (defun f (x)
    (if (zp x)
       1
       (* x
          (f (- x 1)))))
  ```

  La llamada a `f` en el cuerpo viola la segunda condición de arriba (`*` aparece sobre `f` en la expresión de árbol), por lo tanto **no** es una cola recursiva

- ```
  (defun A (x y)
    (declare
      (xargs :guard
         (and (natp  x)
              (natp  y))))
    (if (zp x)
       (+ y 1)
       (if (zp y)
          (A (- x 1)  1)
          (A (- x 1)
             (A x (- y  1))))))
  ```

  Hay tres llamadas a `A` en el cuerpo. Tanto la llamada `(A (- x 1) 1)` como la más externa `(A (- x 1)(A x (- y 1)))`, son ambas de cola recursiva. La llamada más interna `(A x (- y 1))` **no** es de cola recursiva porque la llamada más externa a `A` aparece sobre la llamada más interna de la expresión árbol.
  ```

- ```
  (defun M91 (x)
    (declare
     (xargs :guard
        (integerp x)))
    (if (> x 100)
       (- x 10)
       (M91 (M91 (+ x 11)))))
  ```

  Hay dos llamadas recursivas a `M91` en el cuerpo. La llamada más externa en `(M91 (M91 (+ x 11)))` es de cola recursiva. La llamada más interna `(M91 (+ x 11))` **no** es de cola recursiva porque la llamada más externa a `M91` aparece sobre la llamada más interna de la expresión árbol.

- ```
  (defun 3x+1 (x)
    (declare
     (xargs :guard (natp x)))
    (if (<= x 1)
       x
       (if (evenp x)
         (3x+1 (/ x 2))
         (3x+1 (+ (* 3 x) 1)))))
  ```

  Las dos llamadas recursivas a `3x+1` en el cuerpo son de cola recursiva.

4.5.2. Recursividad Simple frente a Recursión de Cola

Consideremos el Ejemplo 2 del factorial de un número definido matemáticamente para $g\,(x)$ y $f\,(x, y)$, que son funciones recursiva y de recursión de cola, respectivamente.

Ejemplo 2: Factorial de un número definido matemáticamente.

$$g(x) = \begin{cases} 1 & if\ x = 0, \\ x \cdot g(x-1) & if\ x \geq 1 \end{cases} \qquad f(x,\, y) = \begin{cases} y & if\ x = 0, \\ f(x-1, x \cdot y) & if\ x \geq 1 \end{cases}$$

$$(a) \qquad (b)$$

La definición *(a)* del Ejemplo 2 corresponde a la llamada recursiva de la función $g\,(x) = x!$ cuyo *caso base* o *condición de parada* ocurre cuando x es igual a 0. Además, la simplificación de x *hacia el caso base* es $x-1$. La definición *(b)* corresponde a la definición de una función f basada en recursión de cola, en donde se ha introducido un parámetro adicional "y" que se obtiene como el resultado de aplicar *generalización* y no considerarlo como una variable *acumulador*. Dicha función f evita tener ejecuciones pendientes en la pila de memoria.

En la Tabla 4.1 se observan las trazas de la que sería la ejecución de la función recursiva g para $n = 5$. La condición de parada viene del caso base, que ocurre cuando $n = 0$. Las flechas muestran el orden de ejecución. La regla de ejecución recursiva es muy

Condición de parada *n=0*	**Fase llamada recursiva**	**Fase recolección de salida**
	$g\,(5)$	
0	$5 * g\,(4)$	120
0	$4 * g\,(3)$	24
0	$3 * g\,(2)$	6
0	$2 * g\,(1)$	2
0	$1 * g\,(0)$	1
1	1	1

TABLA 4.1: Trazas de una ejecución recursiva

Condición de parada *n=0*	**Fase llamada recursiva**	**Fase recolección de salida**
	$f\,(5, 1)$	
0	$f\,(4, 5)$	
0	$f\,(3, 20)$	
0	$f\,(2, 60)$	
0	$f\,(1, 120)$	
0	$f\,(0, 120)$	
1	120	

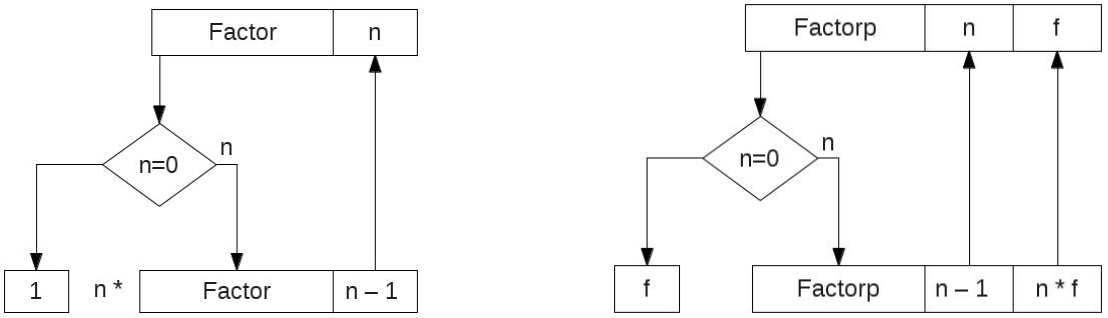
TABLA 4.2: Trazas de la recursión de cola

simple: *lo que no puede ser ejecutado debe ser escrito simbólicamente en la pila* (columna "*fase llamada recursiva*"). *Después se debe buscar la condición de parada que debe ser ejecutada* (columna "*fase de salida*"). La columna "*fase llamada recursiva*" muestra simbólicamente que la llamada recursiva exige al compilador asignar almacenamiento en la pila en tiempo de ejecución para todas las llamadas que aún no han devuelto un resultado.

Por su parte, en la Tabla 4.1 se muestra que, en el caso de la función f, nada se tiene que hacer después de la llamada recursiva, ya que la solución se recoge en el parámetro adicional. La función de cola recursiva consume menos memoria y no hay fase de recolección de salida durante la ejecución. Por lo general, el compilador optimiza la versión de cola recursiva para incluir un simple bucle [12].

Un diagrama de flujo muestra la iteración de las versiones recursivas sin y con recursión de cola.

Ejemplo 3: Flujo de recursividad y recursión de cola para el ejemplo del factorial



(a) (b)

4.5.3. La Transformación en Recursión de Cola

La metodología para diseñar una función con recursión de cola f, que es la optimización de alguna función recursiva genérica g, se puede describir con el siguiente algoritmo que está basado en el propuesto por Rubio-Sánchez en [29] [5]:

1. Obtener una expresión matemática de $g(X)$, donde X es un conjunto de parámetros. También definimos $Y = \varnothing$.
2. Agregar un parámetro al conjunto Y y crear la nueva función $f(X, Y)$, que es la generalización de $g(X)$ y así incorporar los parámetros en Y en la nueva fórmula.
3. Determinar los valores o propiedades de X que corresponden a los casos base para g y establecer los casos base para f.
4. Escoger una simplificación $\tilde{X}$ de X para los valores *hacia los casos base* de g.
5. Formar la ecuación $f(X, Y) = f\left(\tilde{X}, Z\right)$ donde Z reemplaza a Y y es desconocido.
6. Obtener el valor de Z. Si no es posible, retornar al paso 1 (escoger otra expresión para g) o 2.
7. Después de obtener Z, definir f incorporando la recursividad y los casos base.
8. Establecer los valores de los parámetros $\tilde{Y}$ tal que $g(X) = f\left(X, \tilde{Y}\right)$, en base a los valores base de X.

[5] Para la implementación, este algoritmo se someterá a los necesarios ajustes para adaptarlo a las peculiaridades del lenguaje Maude.

Capítulo 5

El Lenguaje Funcional Maude

En este capítulo descubriremos las características más sobresalientes del lenguaje Maude en relación con las necesidades de este proyecto. Discutiremos las ventajas que tiene Maude frente a otros lenguajes funcionales, incluyendo las estrategias de evaluación, módulos y facilidades de metaprogramación.

5.1. Introducción

Maude es un lenguaje de alto nivel y alto rendimiento que soporta especificaciones tanto de la lógica ecuacional de pertenencia como de la lógica de reescritura y la programación de un espectro muy amplio de aplicaciones. En las siguientes secciones se describen las características más importantes del lenguaje, en su versión Maude 2.6, que serán utilizadas en el resto de esta tesis.

El sistema Maude, su documentación, una colección de ejemplos, algunos casos de estudio y varios artículos relacionados están disponibles en la página web de Maude en *http://maude.cs.uiuc.edu*.

5.2. Módulos

El módulo es la unidad base de Maude. Esencialmente, es un conjunto de definiciones que describen una colección de operaciones y la interacción entre ellas o matemáticamente hablando, el *álgebra* que éstas definen. Un álgebra es un conjunto de tipos y operaciones sobre ellos.

Hay tres tipos de módulos en Maude: Los módulos funcionales (*functional module*), los módulos de sistema (*system module*) y los módulos orientados a objetos (*object-oriented module*). La diferencia entre ellos radica en sus distintas capacidades y la forma de especificarlos. Cada tipo de módulo se declara con las siguientes palabras claves:

```
fmod NOMBRE is ... endfm
mod NOMBRE is ... endm
omod NOMBRE is ... endom
```

donde los puntos suspensivos indican todas las declaraciones e instrucciones entre el inicio del módulo y el final del mismo.

Actualmente hay dos niveles separados de Maude: *Core Maude* y *Full Maude*. Todo lo que está escrito en Core Maude está en Full Maude, pero no viceversa.

Llamamos *Core Maude* al intérprete de Maude 2, implementado en C++, que provee todas las funcionalidades básicas de Maude. Los módulos de sistema y funcionales se encuentran ambos en Core Maude. La diferencia entre los módulos funcionales y de sistema reside en las declaraciones que éstos tienen:

- los módulos funcionales admiten ecuaciones, identidad de datos y pertenencia, así como declaraciones de tipos, mientras que
- los módulos del sistema también admiten reglas, que describen transiciones entre estados, ademas de ecuaciones y axiomas de pertenencia.

Usaremos E, E', etc. para denotar conjuntos de ecuaciones y pertenencias, y R, R', etc. para denotar conjuntos de reglas.

Full Maude es una extensión de Maude, escrita en el propio Maude, que dota al lenguaje con un módulo aún más potente y extensible del álgebra que está disponible en Core Maude. El módulo orientado a objetos está dentro de Full Maude. Este módulo soporta notación para objetos, mensajes y herencia. Para usar Full Maude, todos los comandos y módulos deben estar entre paréntesis. Así, la declaración de un módulo orientado a objetos debería ser:

```
(omod NAME ... endom)
```

Al igual que en muchos lenguajes de programación, podemos importar un módulo desde otro. En Maude hay tres formas de hacerlo:

```
protecting NOMBRE-MODULO .
including NOMBRE-MODULO .
extending NOMBRE-MODULO .
```

La palabra clave `protecting` permite importar el módulo sin modificarlo, `including` permite importar y cambiar el significado de las operaciones pero no su interfaz y por último `extending` permite importar y agregar sorts y nuevas operaciones al módulo importado.

Hay que resaltar que todas las operaciones de Maude necesariamente deben terminar con un espacio en blanco y un punto[1].

Para el objetivo de esta tesis vamos a centrarnos en Core Maude y los módulos funcionales (que es el tipo de módulos sobre el que basaremos nuestro desarrollo) y de sistema. El estudio de Full Maude queda fuera de ámbito para los intereses de este trabajo.

5.3. Sorts y Subsorts

Un *sort* es una categoría para los valores. Por ejemplo, un número podría ser un valor de un sort "*integer*" o tal vez sólo "*number*" o quizás uno propio definido por el programador; por ejemplo, "*coordenada*". Es decir un sort define el tipo de valor.

Los nombres de sorts no pueden tener espacios en blanco ni '{', '[', '(', a menos que sean precedidos por un apóstrofe. Un sort se declara en un módulo con la palabra clave *sort* si se trata de uno, o con *sorts* si son varios, seguido de su respectivo terminador "punto".

```
sort integer .
sorts integer decimal .
```

También podemos declarar *subsorts*, que son grupos específicos adicionales que pertenecen todos a un mismo sort. Por ejemplo, el sort de los números reales *(real)* podría tener irracionales (*irrational*) o racionales (*rational*) como subsorts y los racionales a su vez podrían tener enteros (*integer*) o fracciones (*fraction*). Los enteros podrían ser positivos (*positive*), negativos (*negative*), etc. Los subsorts son declarados con la palabra reservada *subsort* si se trata de uno o *subsorts* si son varios.

```
sorts Real Irrational Rational Integer Fraction Positive Negative .
subsorts Irrational Rational < Real .
```

[1] El olvidar el espacio y punto es uno de los errores típicos al programar en Maude.

```
subsort Fraction < Rational .
subsorts Positive Negative < Integer < Rational .
```

5.4. Operadores

En un módulo de Maude, un operador se declara con la palabra reservada *op* seguida por su nombre, seguida por dos puntos "`:`", seguida por la lista de sorts para sus argumentos (llamado la *aridad* de los operadores o *dominio de sort*), seguida por `->`, seguida por el sort de su resultado (llamado la *co-aridad de operador* o *rango de sort*) y opcionalmente, seguido por los atributos que van entre corchetes `[...]`.

`op` $\langle nombre \rangle$: $\langle Sort - 1 \rangle$ ... $\langle Sort - k \rangle$ `->` $\langle Sort \rangle$ `[` $\langle Atributos \rangle$ `]`

Maude entiende ambos tipos de notaciones, *prefija* y *mixfija (mezcla de notación),* para las operaciones. La *notación prefija* consta del nombre del operador, seguido por sus argumentos entre paréntesis y separados por comas. Para llamar al operador `+` en las variables `x` e `y` usando notación prefija, deberíamos escribir `+(x, y)`.

La notación alternativa es *mixfija,* la cual no es como la notación prefija en Maude, ya que a ésta se declara especificando los lugares fijos para las variables. Por ejemplo, la declaración en notación prefija del operador `+` sería `op +`. mientras que en notación mixfija debería ser `op _+_`. Tanto la declaración de la notación mixfija como prefija se ilustran en la siguiente declaración del operador de adición de dos números naturales.

```
op + : Nat Nat -> Nat .
op _+_ : Nat Nat -> Nat .
```

De manera similar, podemos declarar dos operaciones con el mismo número de argumentos y sort resultante usando la palabra `ops`.

```
ops + * : Nat Nat -> Nat .
op _+_ _*_ : Nat Nat -> Nat .
```

5.5. Tipos

Como ya hemos dicho, la lógica ecuacional subyacente en Maude es la lógica ecuacional con pertenencia. En esta lógica, los sorts se agrupan en clases de equivalencia llamada *kinds*. Los sorts de Maude son definidos por el usuario, mientras que los kinds son

asociados implícitamente con las componentes conexas de los sorts y son considerados como "*error supersorts*".

Un tipo es identificado con su clase equivalente de sort y se representa con un nombre tras encerrar el nombre de uno o más sorts en corchetes, separados por coma (si son varios sorts).

Por ejemplo, consideremos las siguientes declaraciones:

```
sorts Nat Zero NatSeq NzNat .
subsort Zero NzNat < Nat < NatSeq .
op zero : -> Zero .
op p : NzNat -> Nat .
```

En este caso, Maude ajustará el término `p(zero)` y le asignará el tipo `[Nat]`, o equivalentemente `[NatSeq]` o también `[Nat, NatSeq]`, ya que los sorts `Nat` y `NatSeq` pertenecen a la misma componente conexa.

5.6. Sobrecarga de Operadores

Los operadores en Maude pueden ser sobrecargados, esto es, podemos tener varias declaraciones para el mismo operador con diferentes aridad y co-aridad. Por ejemplo, consideremos el operador de adición de naturales y que queremos extender para que realice la adición sobre un nuevo sort de "números naturales módulo 3" (`Nat3`). Consideremos también que tenemos las constantes `0`, `1`, `2` del sort `Nat3` y dos tipos adicionales de declaración para el operador de adición.

```
op _+_ : NzNat Nat -> NzNat .
sort Nat3 . ps 0 1 2 : -> Nat3 .
op _+_ : Nat3 Nat3 -> Nat3 .
```

Ahora el operador de adición `+` se encuentra sobrecargado y tiene tres declaraciones (considerando la declaración de suma de naturales de la sección anterior). Sin embargo, hay dos tipos diferentes de sobrecarga presentes en el ejemplo. El primero se produce cuando el primer argumento es un `NzNat` y que como resultado devuelve un `NzNat`. En este caso, el comportamiento del operador `+` en `Nat` y `NzNat` se supone que es el mismo para ambos argumentos. Sin embargo, el `NzNat` restringe a que sus valores sean naturales diferentes a cero. El segundo recibe en ambos argumentos un `Nat3` y su resultado es un `Nat3`, que semánticamente no está relacionado con la adición de números naturales.

Si consideramos la constante `0`, la siguiente expresión `0+0` podría ser ambigüa con la adición de `Nat` y `Nat3`, por lo que podemos desambigüar dicha expresión colocando junto al valor el sort al que pertenecen; por ejemplo `0+(0).Nat3` o `(0+0).Nat` y `(0+0).Nat3`.

5.7. Variables

Una *variable* es un valor indefinido para un elemento de un sort. Las variables se declaran con la palabra *var* o *vars* dependiendo si es una única variable o varias a la vez, seguidas de dos puntos ":", el sort al que pertenecen y el punto.

```
var x : number .
vars c1 c2 c3 : color .
```

A diferencia de los lenguajes como C++, Java y otros lenguajes comunes de programación, las variables de Maude nunca tienen un valor asignado a ellas.

Maude también permite declaraciones de variables "a vuela pluma"(*on-the-fly declarations*), lo cuál es útil por ejemplo si una variable debe ser usada una sola vez. La sintaxis consiste de un identificador (nombre de la variable), los dos puntos ":", el sort o kind al que pertenece. Por ejemplo, `N:Nat` declara una variable `N` de sort `Nat` y `X:[Nat]` declara la variable llamada `X` de kind `[Nat]`.

5.8. Constructores y Atributos de Operadores

En toda álgebra hay operaciones básicas, llamadas constructores, que son las que definen la estructura básica de la misma. Para designar a una operación como constructora, agregamos `[ctor]` después del sort asociado y antes del punto.

Ejemplo 4: Módulo de operadores básicos de `Nat`

```
fmod Nat is
    sort Nat .
    op 0 : -> Nat [ctor] .
    op suc_ : Nat -> Nat [ctor] .
endfm
```

Hay otros atributos que pueden ser agregados a las operaciones y que nos permiten "ahorrar" la escritura de ciertas ecuaciones. Estos son: `assoc`, `comm`, `id:oper`. Dichos

atributos indican que la operación a la cuál están vinculados es *asociativa*, *conmutativa* y/o tiene como *elemento neutro* al operador `oper`, respectivamente[2].

Una lista es un ejemplo ideal para entender el uso de los tres atributos `assoc`, `comm` e `id`; por ello consideremos el siguiente módulo básico de listas. En el módulo tenemos `Elt` que es el sort al que pertenecen los elementos de lista y `nil` que corresponde al elemento neutro de la lista, como se muestra en la siguiente declaración.

Ejemplo 5: Módulo de Listas

```
fmod LISTAS is
    sorts List Elt .
    subsort Elt < List .
    op nil : -> List [ctor] .
    op __ : List List -> List [ctor] .
    vars E1 E2 : Elt .
    vars L1 L2 : List .
endfm
```

La *asociatividad* implica que la lista "(formada por `L1 L2`) `L3`" es la misma lista formada por "`L1 (L2 L3)`" y también es igual a "`L1 L2 L3`". La asociatividad se declara añadiendo la palabra reservada `assoc` después de `ctor` entre corchetes.

```
op __ : List List -> List [ctor assoc] .
```

La *identidad* constituye un atributo extremadamente importante para las listas. Por ejemplo, la propiedad de identidad de la adición es `n + 0 = n`. De manera similar, la propiedad de identidad de las listas "`L1 nil`" es la misma que "`L1`". La identidad es declarada con la palabra reservada `id:` seguida por la constante de nulidad que satisface la propiedad de identidad, en este caso `nil`.

```
op __ : List List -> List [ctor assoc id: nil] .
```

La *conmutatividad* es una propiedad que técnicamente no pertenece a las listas, sino a los conjuntos (o más bien a multi-conjuntos, pero no va ser un problema su uso aquí). La conmutatividad significa que el orden no importa; así, "`S1 S2 S3`" es lo mismo que "`S1 S3 S2`" y es lo mismo que "`S2 S1 S3`". [3]

[2] Todos estos atributos sólo pueden ser colocados en operadores binarios.
[3] El orden entre los atributos de las ecuaciones entre corchetes no importa.

```
op __ : Set Set -> Set [ctor assoc comm id: none] .
```

Hay otro atributo a mencionar que es `idem`, que denota el axioma ecuacional de *idempotencia*, la propiedad que descarta los elementos repetidos. Esta puede ser una propiedad muy importante para conjuntos.[4]

Un atributo usado para operadores unarios (operadores que toman sólo un argumento) que deben ser repetidos una y otra vez es `iter`, que permite expresar iteraciones como una simple instancia del operador, elevado al número de iteraciones. Por ejemplo, se puede aplicar `iter` al operador "sucesor de un número" representado por `s_` (en notación de Peano), donde el cinco toma la forma de `s s s s s 0`. El mismo número cinco utilizando el atributo `iter`, debería escribirse como `s^5(0)`. Esta notación debe ser usada tanto para la entrada de datos como la salida. Se declara igual que los otros atributos, entre corchetes.

```
op s_ : Nat -> Nat [ctor iter] .
```

5.9. Módulos Funcionales

Los módulos funcionales contienen definiciones de sorts y definiciones de operaciones y ecuaciones, pero no contienen reglas de reescritura. Es decir, describen la estructura estática del problema pero no los cambios de estado del mismo, lo cuál se define en los módulos de sistema a través de reglas de reescritura.

Los módulos funcionales de Maude asumen que sus ecuaciones tienen las propiedades adecuadas porque éstas son consideradas como reglas de simplificación y usadas en la dirección de izquierda a derecha sin pérdida de completitud. Dichas propiedades son *Church-Rosser* y *terminación*.

5.9.1. Ecuaciones incondicionales

Las ecuaciones sirven para proveer a Maude de ciertas reglas para simplificar una expresión a su forma más simple, también llamada forma canónica o irreducible, que es una expresión equivalente en la que los operadores son todos constructores. Se declaran con la palabra *eq*, seguida por dos *expresiones* separadas por el símbolo *"="* y el *punto final*.

$$\texttt{eq}\ \langle Término-1\rangle\ \texttt{=}\ \langle Término-2\rangle\ \texttt{[}\ \langle Atributos\rangle\ \texttt{]}\ .$$

[4] Los atributos `assoc` e `idem` no pueden ir juntos en Maude

Consideremos nuevamente el módulo de los número naturales, cuyas ecuaciones son todas ellas incondicionales.

Ejemplo 6: Módulo de números naturales con ecuaciones

```
fmod PEANO-NAT is
    sort Nat .
    op 0 : -> Nat [ctor].
    op s : Nat -> Nat [ctor] .
    op _ + _ : Nat Nat -> Nat .
    vars M N : Nat .
    eq [eq1] : 0 + N = N .
    eq [eq2] : s(M) + N = s(M + N) .
endfm
```

5.9.2. Ecuaciones condicionales

Las ecuaciones condicionales son ecuaciones que dependen de una condición booleana. Se escriben con la palabra reservada `ceq` y después la *ecuación*, una *condición* que empieza con la palabra reservada `if`[5] y el punto final.

`ceq` $\langle Término - 1\rangle$ `=` $\langle Término - 2\rangle$
`if` $\langle EqCondición - 1\rangle$ `/\` ... `/\` $\langle EqCondición - k\rangle$
`[` $\langle Atributos\rangle$ `]` .

La sintaxis concreta de las ecuaciones en las condiciones de una ecuación condicional tiene tres variantes, esto es:

- ecuaciones ordinarias `t = t'`,
- ecuaciones de ajuste (matching) `t := t'`, y
- ecuaciones booleanas que se abrevian usando la forma `t`, con `t` un término en el `kind [Bool]` y que representan la ecuación `t = true`.

Los términos booleanos aparecen en ecuaciones booleanas abreviadas y pueden utilizar los predicados de igualdad `_==_` y desigualdad `_=/=_`, además de conectores booleanos

[5] No debemos confundirlo con el operador `if_then_else_fi`, que también existe en Maude.

tales como `_and_`, `_or_` y `not_`[6]. Una ecuación condicional ejecutará una reducción sólo si su condición se reduce a `true`.

```
op diferentes?(_,_) : Nat Nat -> Bool .
ceq diferentes?(N,M) = true if N =/= M .
```

5.9.3. Axiomas de Pertenencia

Los *axiomas de pertenencia* (membership axioms) simplemente especifican que ciertos términos son "*miembros*" de determinados sorts.

Existen dos tipos de axiomas de pertenencia: los *incondicionales* y los *condicionales*, que se declaran con la palabra reservada *mb* y *cmb,* respectivamente. A continuación se escribe el *término*, seguido por *":"*, seguido por el *sort* y por último el punto. Como las ecuaciones, los axiomas de pertenencia pueden tener opcionalmente atributos[7].
La sintaxis es la siguiente:

`mb` ⟨*Término*⟩ : ⟨*Sort*⟩ [⟨*Atributos*⟩] .
`cmb` ⟨*Término*⟩ : ⟨*Sort*⟩
`if` ⟨*EqCondición* − 1⟩ / \ ... / \ ⟨*EqCondición* − *k*⟩
`[` ⟨*Atributos*⟩ `]` .

Los *axiomas condicionales de pertenencia* son similares a las ecuaciones condicionales, ya que la parte condicional tiene la misma estructura. Por ejemplo, si queremos expresar que la suma de dos naturales en `Nat` es otro elemento de `Nat`, siempre y cuando cumpla que sus dos valores sean estrictamente de `Nat`, lo haríamos de la siguiente manera:

```
cmb N + M : Nat if N : Nat and M : Nat .
```

Así también podríamos expresar que el cociente entre dos valores expresados por `N` y `M` debe ser diferente de cero, lo cual se puede definir con un *axioma de pertenencia incondicional*:

```
vars N M : NzNat .
mb N / M : NzNat .
```

o su equivalente con un axioma de pertenencia condicional que es:

[6] Maude viene con un sort `BOOL` ya definido, que tiene las constantes `true`, `false` y las operaciones mencionadas.

[7] Los axiomas de pertenencia no son utilizados para la simplificación, a diferencia de las ecuaciones.

```
var N : Nat .
var M : NzNat .
cmb N / M : NzNat if ( N =/= 0 ) .
```

5.9.4. Operadores y Declaración de Atributos

Aparte de lo que ya conocemos de constructores y atributos, se puede emplear varias técnicas potentes para declarar operadores, que fijan cómo se comportará el operador en ciertas expresiones, ecuaciones y reducciones.

5.9.4.1. Etiquetas

Las etiquetas (*labels*) de un atributo deben estar seguidas por un identificador. Las declaraciones de etiquetas pueden usarse para trazas y depuración.

```
eq N ; N = N [label natset-idem] .
```

El *azúcar sintáctico (syntactic sugar)* de Maude para las etiquetas, permite escribirlas al estilo de las etiquetas de las reglas (en los módulos del sistema). Entonces la ecuación anterior podría escribirse como sigue:

```
eq [label natset-idem] : N ; N = N .
```

5.9.4.2. Memorización

Un atributo de operador extremadamente poderoso es `memo`, que instruye a Maude para memorizar los resultados de la simplificación ecuacional (que están en forma canónica) para aquellos subtérminos que tienen dicho operador en la raíz.

Por ejemplo, la definición recursiva de la función fibonacci es como sigue:

Ejemplo 7: Sucesión de fibonacci

```
fmod FIBONACCI is
    protecting NAT .
    op fibo : Nat -> Nat .
    var N : Nat .
    eq fibo(0) = 0 .
    eq fibo(1) = 1 .
```

```
    eq fibo(s s N) = fibo(N) + fibo(s N) .
endfm
```

Debido a la naturaleza altamente recursiva de esta definición de `fibo`, la evaluación de la expresión `fibo(50)` computará muchas llamadas a la misma instancia de la función una y otra vez. El número exponencial de llamadas repetidas a funciones hace que la evaluación de `fibo` con la ecuación de arriba sea muy ineficiente; concretamente requiere sobre 61 millones de pasos de reescritura para `fibo(50)`.

Si agregamos a la función fibonnacci el atributo `memo`,

```
op fibo : Nat -> Nat [memo] .
```

el cambio es bastante dramático (cambio para mejora), ya que cambiaría el número de reescrituras a 148.

5.9.4.3. Precendencia de operadores

En ocasiones se puede tener algún tipo de temor de ambigüedad en las expresiones. Por ejemplo, consideremos la expresión `3 + 3 * 3`, que puede ser entendida por Maude como `(3 + 3) * 3`, que es `24`, `3 + (3 * 3)`, que es, `12`. Entonces se puede declarar una *precedencia* para los operadores `+` y `&` haciendo uso del atributo `prec` y un número natural. Por ejemplo, si declaramos:

```
op _+_ : Nat Nat -> Nat [prec 35] .
op _*_ : Nat Nat -> Nat [prec 25] .
```

entonces `3 + 3 * 3` será siempre `12`. Si no se escoge una precedencia para una operación definida por el usuario, Maude de forma implícita y automática asigna el valor de precedencia igual a `41`.

5.9.4.4. Patrones de asociación - *Gathering patterns*

En Maude no sólo se puede especificar la asociatividad de los operadores, sino también el *patrón de asociación* (*gathering pattern*) de cada operador.

El patrón de asociación de un operador restringe la precedencia de los términos que el operador admite como argumentos. Lo declaramos usando el atributo `gather` y los símbolos claves `e`, `E` y `&`. El atributo puede declararse como `gather`, donde el número de símbolos en los paréntesis es el número de argumentos que el operador toma.

- `E` indica que el argumento debe tener un valor de precedencia menor o igual que el valor de precedencia del operador,
- `e` indica que el argumento debe tener un valor de precedencia estrictamente menor que el valor de precedencia del operador, y
- `&` indica que el operador permite cualquier valor de precedencia para el correspondiente argumento.

Por ejemplo podemos especificar los operadores de adición y producto dándoles un patrón de asociación `(E e)`.

```
op _+_ : Nat Nat -> Nat [prec 33 gather (E e)] .
op _*_ : Nat Nat -> Nat [prec 31 gather (E e)] .
```

De esta manera, forzamos que el segundo argumento de estos operadores tenga una precedencia estrictamente menor que el primero.

5.9.4.5. Otherwise

A veces, en la definición de una operación mediante ecuaciones, hay determinados casos en que éste puede ser fácilmente definido mediante las ecuaciones y luego cierto caso o casos restantes que resultan difíciles o engorrosos de definir. En tal situación, se podría decir *de otra manera* (*otherwise*), que indica qué hacer en todos los casos restantes no cubiertos por las ecuaciones anteriores.

Por ejemplo, consideremos el problema de la pertenencia de números naturales en un conjunto finito de números.

```
op _in_ : Nat NatSet -> Bool .
```

La parte fácil es definir cuándo un número pertenece a un conjunto:

```
var N : Nat .
var NS : NatSet .
eq N in N ; NS = true .
```

Es algo más complicado definir cuándo no pertenecen. Una simple forma es usar el atributo `otherwise` (abreviado `owise`) y dar una ecuación adicional:

```
eq N in NS = false [owise] .
```

La operación es clara: si la llamada no encaja en primera ecuación (no hace *matching*), entonces el número no está en el conjunto y el predicado debería ser falso.

5.10. Módulos de Sistema

Un *módulo de sistema* en Maude especifica una *teoría de reescritura*. Una teoría de reescritura tiene sorts, tipos y operadores (pero con argumentos "*congelados*" o *frozen*) y pueden tener tres tipos de declaraciones: ecuaciones, relaciones de pertenencia y reglas, todas las cuales pueden ser condicionales. Por lo tanto, cualquier teoría de reescritura tiene una teoría ecuacional subyacente, que contiene las ecuaciones y relaciones de pertenencia, *más* las reglas.

Computacionalmente, las reglas especifican las *transiciones locales concurrentes* que pueden tener lugar en un sistema si el patrón en el lado izquierdo de la regla coincide con un fragmento del estado del sistema y la condición de dicha regla se satisface.

5.10.1. Reglas incondicionales

Matemáticamente, una regla de reescritura incondicional tiene la forma $l : t \to t'$, donde t y t' son términos del mismo tipo, que pueden contener variables y l es la etiqueta de la regla.

Una regla incondicional es introducida en Maude con la siguiente sintaxis:[8]

$$\texttt{rl [}\langle Etiqueta\rangle\texttt{] : }Term-1\texttt{ => }Term-2\texttt{ [}\langle Atributos\rangle\texttt{] .}$$

Como ejemplo de un módulo de sistema, consideremos la siguiente especificación del *juego de la grúa*. Tenemos una caja de vidrio, rellena de figuras de animales apiladas una sobre otra y un brazo robot controlado con una garra que intenta atraparlos.

Para las figuras de animales hay tres estados constructores necesarios: 1) la figura está sobre el piso de la máquina, no sobre alguna otra figura, 2) la figura está sobre otra figura y 3) la figura está limpia, es decir, no hay ninguna figura ni sobre ni bajo ella. Para el brazo robot, hay dos estados: 1) la garra del brazo sostiene a la figura y 2) la garra

[8] Maude permite declaraciones de reglas no etiquetadas, omitiendo simplemente el atributo [⟨*Etiqueta*⟩].

está vacía. Además hay cuatro transiciones: *recoger* (pick up), *dejar* (put down), *desapilar* (unstack) o *apilar* (stack). Las dos últimas se podrían interpretar como parecidas a las dos primeras, pero la diferencia radica en que recoger y dejar una figura debe ser sobre la mesa, mientras que apilar y desapilar se da sobre una figura (arriba de ésta).

```
mod ARCADE-CRANE is
    protecting QID .
    sorts ToyID State .
    subsort Qid < ToyID .
    op floor : ToyID -> State [ctor] .
    op on : ToyID ToyID -> State [ctor] .
    op clear : ToyID -> State [ctor] .
    op hold : ToyID -> State [ctor] .
    op empty : -> State [ctor] .
    op 1 : -> State [ctor] .
    op _&_ : State State -> State [ctor assoc comm id: 1] .
    vars X Y : ToyID .
    rl [pickup] : empty & clear(X) & floor(X)
                => hold(X) .
    rl [putdown] : hold(X)
                => empty & clear(X) & floor(X) .
    rl [unstack] : empty & clear(X) & on(X,Y)
                => hold(X) & clear(Y) .
    rl [stack] : hold(X) & clear(Y)
                => empty & clear(X) & on(X,Y) .
endm
```

5.10.2. Reglas Condicionales

Las reglas de reescritura condicionales pueden tener condiciones muy generales que involucren ecuaciones, relaciones de pertenencia y otras reescrituras. En su notación matemática, pueden tomar la forma:

$$l : t \rightarrow t' \, if \left(\bigwedge_i u_i = v_i \right) \wedge \left(\bigwedge_j w_j : s_j \right) \wedge \left(\bigwedge_k p_k \rightarrow q_k \right)$$

con restricciones en las que no pueden aparecer variables nuevas en el lado derecho ni en la condición. No hay necesidad, sin embargo, de listar primero las ecuaciones, luego las

relaciones de pertenencia y luego las reescrituras: Esto es sólo una abreviación notacional, ya que pueden ser listados en cualquier orden. Sin embargo, en Maude, las condiciones son evaluadas de izquierda a derecha y, por lo tanto, el orden en el cual aparecen, aunque matemáticamente no sea esencial, es muy importante operacionalmente.

La representación en Maude de las reglas condicionales tiene la siguiente sintaxis:

`crl` $[\langle Etiqueta \rangle]$: $\langle Término - 1 \rangle$ `=>` $\langle Término - 2 \rangle$
`if` $\langle EqCondición - 1 \rangle$ / \ ... / \ $\langle EqCondición - k \rangle$
`[` $\langle Atributos \rangle$ `]` .

donde las etiquetas de las reglas pueden omitirse en lugar de ser declaradas como atributo.

Como en las ecuaciones condicionales, la condición puede consistir en una simple declaración o puede ser una conjunción formada usando la conectiva /\. Pero ahora las condiciones son más generales ya que adicionalmente a las ecuaciones y relaciones de pertenencia, pueden contener expresiones de reescritura, para las que se usa la sintaxis concreta `t => t'`.

El uso de las reglas condicionales no es muy común porque usualmente los patrones de estados en el lado izquierdo de una regla de reescritura suministran todos los *"ifs"* de la transición. Sin embargo, muchas veces son sin lugar a duda útiles. Consideremos el ejemplo del *juego de la grúa* de la Subsección 5.10.1; suponiendo que necesitamos mencionar el peso de la figura, podríamos escribir la siguiente regla condicional.

```
crl [pickup] : empty & clear(X) & floor(X) => hold(X)
    if weight(X) < 10 .
```

Esto, obviamente, asume que hay una operación `weight` (peso) que devuelve el peso de la figura. Como en el caso de las ecuaciones condicionales, la declaración `if` puede tomar la forma de una declaración booleana o de ajuste de patrón, pero adicionalmente a esto, la condición de una regla de reescritura puede ser también otra regla de reescritura.

5.11. Reflexión y Computación en el Metanivel

Informalmente, una lógica reflexiva es una lógica en la que aspectos importantes de su metateoría pueden ser representados a nivel de objeto de manera consistente, de forma que la representación a nivel de objeto simula correctamente los aspectos relevantes de la metateoría. En otras palabras, una lógica reflexiva es una lógica que puede ser fielmente

interpretada en sí misma. El diseño e implementación del lenguaje Maude hace uso sistemático del hecho de que la lógica de reescritura es reflexiva. Esto hace a la meta-teoría de la lógica de reescritura accesible al usuario de manera clara.
La lógica de reescritura es reflexiva en una forma matemática; es decir, decimos que hay una teoría de reescritura finita representada por $\mathcal{U}$ que es *universal*, en el sentido que podemos representar en $\mathcal{U}$ alguna teoría de reescritura $\mathcal{R}$ (incluida la misma $\mathcal{U}$) como un término $\overline{\mathcal{R}}$, algunos términos t, t' en $\mathcal{R}$ como términos $\overline{t}$, $\overline{t'}$ y algún par de términos $(\mathcal{R}, t)$ como un término $\langle \overline{\mathcal{R}}, \overline{t} \rangle$, de tal manera que tenemos la siguiente equivalencia

$$\mathcal{R} \vdash t \to t' \Leftrightarrow \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \to \langle \overline{\mathcal{R}}, \overline{t'} \rangle$$

Ya que $\mathcal{R}$ se puede representar a sí misma, podemos lograr una "torre reflexiva" con un número arbitrario de niveles de reflexión:

$$\mathcal{R} \vdash t \to t' \Leftrightarrow \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \to \langle \overline{\mathcal{R}}, \overline{t'} \rangle \Leftrightarrow \mathcal{U} \vdash \left\langle \overline{\mathcal{U}} \vdash \overline{\langle \overline{\mathcal{R}}, \overline{t} \rangle} \right\rangle \to \left\langle \overline{\mathcal{U}} \vdash \overline{\langle \overline{\mathcal{R}}, \overline{t'} \rangle} \right\rangle \cdots$$

En esta cadena de equivalencias, decimos que la primera computación de reescritura toma el nivel 0, la segunda el nivel 1 y así sucesivamente.
En Maude, la funcionalidad primaria de la teoría $\mathcal{U}$ ha sido implementada eficientemente en el módulo funcional `META-LEVEL`. Este módulo incluye los módulos `META-MODULE` y `META-TERM` [10]. En resumen,

- en el módulo `META-TERM`, los términos de Maude son metarepresentados como elementos de un tipo de dato `Term` de términos.
- en el módulo `META-MODULE`, los módulos de Maude son metarepresentados como términos en un tipo de dato `Module` de módulos.
- el proceso de reducción de un término a su forma normal en un módulo funcional y de determinar si tal forma normal tiene un tipo dado se representan por una función `meta-reduce`;
- el proceso de aplicación de una regla de un módulo de sistema a un término se representa por medio de una función `meta-apply`;
- el proceso de reescritura de un término en un módulo de sistema utilizando el intérprete por defecto de Maude se representa por una función `meta-rewrite`; y
- el análisis sintáctico y la impresión edulcorada (en inglés, *pretty-printing*) de un término en un módulo, así como operaciones sobre géneros; tales como la comparación de géneros en el orden de subgénero de una signatura, son representadas también por funciones adecuadas al metanivel.

5.11.1. Sintaxis de términos y módulos

Los términos se representan como elementos del tipo `Term`, construidos con la siguiente signatura:

```
subsort Qid < Term .
subsort Term < TermList .

op {_}_ : Qid Qid -> Term .
op _[_] : Qid TermList -> Term .
op _,_ : TermList TermList -> TermList [assoc] .
op error* : -> Term .
```

La primera declaración, que convierte a `Qid` en un subtipo de `Term`, se utiliza para representar las variables como identificadores con comilla. El operador `{_}_` se utiliza para representar las constantes como pares, donde el primer argumento es la constante y el segundo su tipo, siendo ambos identificadores con comilla. El operador `_[_]` corresponde a la construcción recursiva de términos a partir de subtérminos, con el operador más externo como primer argumento y la lista de sus subtérminos como segundo argumento, donde la concatenación de listas se denota por `_,_`. La última declaración para el tipo de datos de los términos es una constante `error*` utilizada para representar valores erróneos.
Para ilustrar esta sintaxis, utilizaremos un módulo `NAT` de números naturales [9] con cero y sucesor y, operadores de suma y multiplicación conmutativos.

Ejemplo 8: Módulo NAT de suma y producto en notación Peano.

```
fmod NAT is
    sorts Zero Nat .
    subsort Zero < Nat .
    op 0 : -> Zero .
    op s_ : Nat -> Nat .
    op _+_ : Nat Nat -> Nat [comm] .
    op _*_ : Nat Nat -> Nat [comm] .
    vars N M : Nat .
    eq 0 + N = N .
    eq s N + M = s (N + M) .
```

[9] El módulo `NAT` del Ejemplo 8 es una variante del módulo `PEANO-NAT` del Ejemplo 6. En donde declaramos el sort `Zero` para el operador `0` y que es subsort de `Nat`. Además se define un nuevo operador de producto (`*`) y su correspondiente semántica.

```
    eq 0 * N = 0 .
    eq s N * M = M + (N * M) .
endfm
```

El término `s s 0 + s N` de tipo `Nat` en el módulo `NAT` se metarepresenta como

```
_+_['s_['s_[{'0}'Zero]],,'s_['N]].
```

Ya que los términos del módulo `META-LEVEL` se pueden metarepresentar como los términos de cualquier otro módulo, la representación de términos puede iterarse. Por ejemplo, la meta-metarepresentación $\overline{\overline{\texttt{s 0}}}$ del término `s 0` en `NAT` es el término [10]

```
'_[_][{''s_}'Qid,'{_}_[{''0}'Qid,{''Zero}'Qid]].
```

Los módulos funcionales y de sistema se metarepresentan con una sintaxis muy similar a la original. Las principales diferencias son que: (1) los términos en ecuaciones, axiomas de pertenencia y reglas de reescritura aparecen metarepresentados, según la sintaxis ya explicada; (2) en la metarepresentación de módulos se sigue un orden fijo en la introducción de las diferentes clases de declaraciones; y (3) los conjuntos de identificadores utilizados en las declaraciones de los sorts se representan como conjuntos de identificadores con apóstrofe construidos con el operador asociativo y conmutativo `_;_`.
La sintaxis de los operadores más externos para la representación de módulos funcionales y de sistema es la siguiente:

```
sorts FModule Module .
subsort FModule < Module .

op fmod_is_sorts_.____endfm : Qid ImportList SortDecl SubsortDeclSet
      OpDeclSet VarDeclSet MembAxSet EquationSet -> FModule .
op mod_is_sorts_.____endm : Qid ImportList SortDecl SubsortDeclSet
      OpDeclSet VarDeclSet MembAxSet EquationSet RuleSet -> Module .
```

El primer aspecto a resaltar es el lugar especial para la declaración de los sorts. A diferencia del resto de argumentos, que son todos conjuntos de declaraciones realizadas con palabras reservadas definidas, el tercer argumento toma un conjunto de sorts metarepresentados, que son los nombres de los sorts como identificadores con apóstrofe: `'Nat`, `'Bool`, `'State`, etc. Por ejemplo, la declaración de

[10] Para simplificar la presentación utilizamos la metanotación $\overline{t}$ para denotar la metarepresentación del término t e $\overline{Id}$ para denotar la metarepresentación del módulo con nombre Id.

```
sorts Zero Nat Bool .
```

aparecería como

```
'Zero ; 'Nat ; 'Bool
```

en el tercer argumento.
La segunda cosa a resaltar es que carece de un conjunto de declaraciones de variables. En efecto, no hay sintaxis para metarepresentar una declaración de variables. Simplemente cada variable está metarepresentada a vuelo pluma con su respectivo sort o kind. Por ejemplo, la ecuación

```
eq s(M) + N = s (M + N) .
```

se metarepresenta así:

```
eq '_+_['s['M:Nat], 'N:Nat] =
    's['_+_['M:Nat, 'N:Nat]]
              [none] .
```

La palabra reservada `none` (entre corchetes) se coloca cuando existen valores nulos. Es decir, cuando no se ha definido un valor, como en este caso, en donde la ecuación no tiene asociada una etiqueta[11]. Además, la palabra reservada `none` no va en el mismo lugar que la etiqueta. Es decir, no aparece en la parte frontal, sino al final.
La definición completa de esta sintaxis en el módulo `META-LEVEL` se puede encontrar en [10].
La representación $\overline{\texttt{NAT}}$ del módulo `NAT` es el siguiente término de tipo `FModule`:

```
fmod 'NAT is
    nil
    sorts 'Zero ; 'Nat .
    subsort 'Zero < 'Nat .
    op '0 : nil -> 'Zero [none] .
    op 's_ : 'Nat -> 'Nat [none] .
    op '_+_ : 'Nat 'Nat -> 'Nat [comm] .
    op '_*_ : 'Nat 'Nat -> 'Nat [comm] .
```

[11] Para los valores nulos de las listas se utiliza **nil** en lugar de none.

```
    var 'N : 'Nat .
    var 'M : 'Nat .
    none
    eq '_+_[{'0}'Zero, 'N] = 'N .
    eq '_+_['s_['N], 'M] = 's_['_+_['N, 'M]] .
    eq '_*_[{'0}'Zero, 'N] = {'0}'Zero .
    eq '_*_['s_['N], 'M] = '_+_['M, '_*_['N, 'M]] .
endfm
```

Obsérvese que, cualquiera de los términos de tipo `Module` puede ser metarepresentado de nuevo, dando lugar a términos de tipo `Term` y que esta metarepresentación puede iterarse un numero arbitrario de veces. Esto es de hecho necesario cuando los cómputos al metanivel tienen que operar a niveles más altos

5.11.2. Funciones de descenso

El módulo `META-LEVEL` tiene tres funciones predefinidas (implementadas internamente por el sistema) que proporcionan formas eficientes y útiles de reducir cómputos del metanivel a cómputos del nivel objeto (nivel 0): `meta-reduce`, `meta-apply` y `meta-rewrite`. Estas funciones se denominan *funciones de descenso* [7].

La operación `meta-reduce` tiene sintaxis

```
op meta-reduce : Module Term -> Term .
```

en donde toma como argumentos la representación de un módulo M y un término t y devuelve la forma completamente reducida del término t utilizando las ecuaciones en M. Por ejemplo,

```
Maude> red meta-reduce(NAT, s 0 + s 0) .
result Term: s s 0
```

La operación `meta-rewrite` tiene sintaxis

```
op meta-rewrite : Module Term MachineInt -> Term .
```

y es análoga a `meta-reduce` pero, en lugar de utilizar sólo la parte ecuacional de un módulo, utiliza tanto ecuaciones como reglas para reescribir un término utilizando la estrategia por defecto de Maude. Sus primeros dos argumentos son las representaciones

de un módulo M y un término t, y su tercer argumento es un número natural n. La función devuelve la representación del término obtenido a partir de t después de, como mucho, n aplicaciones de reglas de M, utilizando la estrategia por defecto de Maude, que aplica las reglas de forma justa de arriba abajo. Si en el tercer argumento damos el valor 0, el número de reescrituras no estaría acotado.
Si consideramos la metarepresentación del siguiente módulo de sistema:

```
mod A-TRANSITION-SYSTEM is
    sort State .
    ops n1 n2 n3 n4 n5 : -> State .
    rl [a] : n1 => n2 .
    rl [b] : n1 => n3 .
    rl [c] : n3 => n4 .
    rl [d] : n4 => n2 .
    rl [e] : n2 => n5 .
    rl [f] : n2 => n1 .
endm
```

podemos reescribir (utilizando como mucho 10 aplicaciones de reglas) la metarepresentación del estado `n3` de la siguiente manera:

```
Maude> red meta-rewrite(A - TRANSITION - SYSTEM, n3, 10) .
result Term: n5
```

La operación `meta-apply` tiene la sintaxis

```
op meta-apply : Module Term Qid Substitution MachineInt
    -> ResultPair .
```

en donde los primeros cuatro argumentos son las representaciones de un módulo M, un término t en M, una etiqueta l de alguna(s) regla(s) en M y un conjunto de asignaciones (posiblemente vacío) que definen una sustitución parcial σ de las variables en esas reglas. El último argumento es un número natural n. Esta función devuelve una pareja de tipo `ResultPair` formada por un término y una sustitución. La sintaxis de las sustituciones y de los resultados es

```
sorts Assignment Substitution ResultPair .
subsort Assignment < Substitution .
op _<-_ : Qid Term -> Assignment .
```

```
op none : -> Substitution .
op _;_ : Substitution Substitution
    -> Substitution [assoc comm id: none] .
op {_,_} : Term Substitution -> ResultPair .
```

La operación **meta-apply** se evalúa de la siguiente manera:

1. el término t se reduce completamente utilizando las ecuaciones en M;
2. el término resultante se intenta encajar con los lados izquierdos de todas las reglas con etiqueta l parcialmente instanciadas con σ, descartando aquellos ajustes que no satisfagan la condición de la regla;
3. se descartan los primeros n ajustes que hayan tenido éxito; si existe un $n+1$-*ésimo* ajuste, su regla se aplica utilizando este encaje y se realizan los pasos siguientes 4 y 5; en caso contrario, se devuelve el par {`error*, none`};
4. el término resultante se reduce completamente utilizando las ecuaciones de M;
5. se devuelve el par formado utilizando el constructor `{_,_}`, cuyo primer elemento es la representación del término resultante completamente reducido y cuyo segundo elemento es la representación del encaje utilizado en la reescritura.

Utilizando la operación `meta-apply` y el módulo `A-TRANSITION-SYSTEM` metarepresentado, se pueden obtener todas las reescrituras secuenciales en un paso del estado `n2`:

```
Maude> red meta-apply('A - TRANSITION - SYSTEM', 'n2, 'e, none, 0) .
result ResultPair: {'n5, none}

Maude> red meta-apply('A - TRANSITION - SYSTEM', 'n2, 'f, none, 0) .
result ResultPair: {'n1, none}
```

Además de estas tres funciones, el módulo `META-LEVEL` proporciona otras funciones pertenecientes a la teoría universal y que podrían haber sido definidas ecuacionalmente, pero que por razones de eficiencia se han implementado directamente. Estas funciones incluyen el análisis sintáctico, la impresión edulcorada de términos de un módulo al metanivel y operaciones sobre los tipos declarados en la signatura de un módulo.

La operación `meta-parse` tiene como sintaxis

```
op meta-parse : Module QidList -> Term .
```

cuyos argumentos son: (i) la representación de un módulo M y (ii) la representación de una lista de *tokens* (componentes sintácticas) dada por una lista de identificadores con apóstrofe y devuelve el término analizado a partir de la lista de *tokens* para la signatura de M . Si la lista no se puede analizar, se devuelve la constante `error*`. Por ejemplo, dado el módulo `NAT` y la entrada `'s '0' + 's '0`, obtenemos el siguiente resultado:

```
Maude> red meta-parse(NAT, 's '0 '+ 's '0) .
result Term: '_+_['s_[{'0}'Zero], 's_[{'0}'Zero]]
```

La operación `meta-pretty-print` tiene como sintaxis

```
op meta-pretty-print : Module Term -> QidList .
```

en donde sus argumentos son la representación de un módulo M y de un término t y devuelve la lista de identificadores con comilla que codifican la cadena de *tokens* producida al imprimir t de forma edulcorada, utilizando la sintaxis concreta dada por M. Si se produce un error, se devuelve la lista vacía. Por ejemplo,

```
Maude> red meta-pretty-print(NAT, '_+_['s_[{'0}'Zero], 's_[{'0}'Zero]]) .
result QidList: 's '0 '+ 's '0
```

5.11.3. Estrategias internas

Como hemos visto, los módulos de sistema de Maude son teorías de reescritura que no tienen por qué ser confluentes ni terminantes. Por lo tanto, necesitamos controlar el proceso de reescritura, que en principio, podría ir en muchas direcciones, mediante *estrategias* adecuadas. Gracias a las propiedades reflexivas de Maude, dichas estrategias pueden hacerse internas al sistema; es decir, pueden definirse en un módulo Maude, con el que se puede razonar como con cualquier otro módulo. De hecho, hay una libertad absoluta para definir diferentes lenguajes de estrategias dentro de Maude, ya que los usuarios pueden definir sus propios lenguajes, sin estar limitados a un lenguaje fijo y cerrado. Para ello se utilizan las operaciones meta-reduce, meta-apply y meta-rewrite como expresiones de estrategias básicas, extendiéndose el módulo `META-LEVEL` con expresiones de estrategias adicionales y sus correspondientes reglas. A continuación tomamos un ejemplo de [8], que sigue la metodología para definir lenguajes de estrategias internos para lógicas reflexivas introducido en [6].

Para ilustrar las ideas utilizaremos el módulo de sistema `SORTING` para ordenar vectores de enteros. En este módulo, los vectores se representan como conjuntos de pares de enteros, cuya primera componente de cada par representa una posición (o índice) del vector y la segunda el valor en dicha posición.

```
mod SORTING is
   protecting MACHINE-INT .
   sorts Pair PairSet .
   subsort Pair < PairSet .
   op <_;_> : MachineInt MachineInt -> Pair .
   op empty : -> PairSet .
   op __ : PairSet PairSet -> PairSet [assoc comm id: empty] .
   vars I J X Y : MachineInt .
   crl [sort] : < J ; X > < I ; Y > => < J ; Y > < I ; X >
   if (J < I) and (X > Y) .
endm
```

Antes de explicar algunas de las estrategias que se pueden definir, obsérvese que la estrategia por defecto del intérprete Maude para módulos de sistema puede ser utilizada, de forma sencilla y eficiente, por medio de la operación de descenso meta-rewrite:

```
Maude> rew meta-rewrite(SORTING,
        '__['<_;_>[{'1}'MachineInt, {'3}'MachineInt],
        '<_;_>[{'2}'MachineInt, {'2}'MachineInt],
        '<_;_>[{'3}'MachineInt, {'1}'MachineInt]],
        0) .
result Term: '__['<_;_>[{'1}'NzMachineInt,{'1}'NzMachineInt],
        '<_;_>[{'2}'NzMachineInt,{'2}'NzMachineInt],
        '<_;_>[{'3}'NzMachineInt,{'3}'NzMachineInt]]
```

Los lenguajes de estrategias pueden definirse en Maude en extensiones del módulo `META-LEVEL`. El módulo que definimos es el siguiente módulo `STRATEGY`, del que primero mostramos su sintaxis, para luego introducir sus ecuaciones que serán ilustradas con ejemplos.

```
fmod STRATEGY is
    including {\tt META-LEVEL} .
    sorts MetaVar Binding BindingList Strategy StrategyExpression .
    subsort MetaVar < Term .
    ops I J : -> MetaVar .
    op binding : MetaVar Term -> Binding .
    op nilBindingList : -> BindingList .
    op bindingList : Binding BindingList -> BindingList .
```

```
op rewInWith : Module Term BindingList Strategy
        -> StrategyExpression .
op set : MetaVar Term -> Strategy .
op rewInWithAux : StrategyExpression Strategy
        -> StrategyExpression .
op idle : -> Strategy .
op failure : -> StrategyExpression .
op and : Strategy Strategy -> Strategy .
op apply : Qid -> Strategy .
op applyWithSubst : Qid Substitution -> Strategy .
op iterate : Strategy -> Strategy .
op while : Term Strategy -> Strategy .
op orelse : Strategy Strategy -> Strategy .
op extTerm : ResultPair -> Term .
op extSubst : ResultPair -> Substitution .
op update : BindingList Binding -> BindingList .
op applyBindingListSubst : Module Substitution BindingList
        -> Substitution .
op substituteMetaVars : TermList BindingList -> TermList .

op SORTING : -> Module .

var M : Module .
vars V V' F S G L : Qid .
vars T T' : Term .
var TL : TermList .
var SB : Substitution .
vars B B' : Binding .
vars BL BL' : BindingList .
vars MV MV' : MetaVar .
vars ST ST' : Strategy .

eq SORTING = SORTING .
```

En el módulo `STRATEGY` la operación `rewInWith` computa expresiones de estrategias. Los dos primeros argumentos son las metarepresentaciones de un módulo M y un término t. El cuarto argumento es la estrategia S a computar y el tercer argumento guarda información que puede ser relevante para S, como veremos más adelante. La definición de `rewInWith` es tal que, al ir computando la estrategia, t se reescribe mediante una

aplicación controlada de las reglas en M, se actualiza la información en el tercer argumento y la estrategia S se reescribe a la estrategia que queda por ser computada. En caso de terminación, se acaba con la estrategia `idle`. La expresión de estrategia `failure` se devuelve si una estrategia no puede ser aplicada.

Una primera estrategia básica que podemos definir es la aplicación de una regla una vez y al nivel más alto de un término y con el primer encaje encontrado. Para esta estrategia básica se introduce el constructor `apply`, cuyo único argumento es la etiqueta de la regla a aplicar. La siguiente ecuación define el valor de `rewInWith` para esta estrategia:

```
eq rewInWith(M, T, BL, apply(L)) =
  if meta-apply(M, T, L, none, 0) == {error*, none}
  then failure
  else rewInWith(M, extTerm(meta-apply(M, T, L, none, 0)), BL, idle)
  fi .
```

Las operaciones `extTerm` y `extSubst` devuelven la primera y segunda componente, respectivamente, de un par construido con `{_,_}`.

```
eq extSubst({T, SB}) = SB .
eq extTerm({T, SB}) = T .
```

Podemos ilustrar el cómputo de una expresión de estrategia `apply` con el siguiente ejemplo:

```
Maude> rew rewInWith(SORTING,
        '__['<_;_>[{'1}'MachineInt, {'3}'MachineInt],
        '<_;_>[{'2}'MachineInt, {'2}'MachineInt],
        '<_;_>[{'3}'MachineInt, {'1}'MachineInt]],
        nilBindingList,
        apply('sort')).
result StrategyExpression:
rewInWith(SORTING,
        '__['<_;_>[{'1}'NzMachineInt,{'2}'NzMachineInt],
        '<_;_>[{'2}'NzMachineInt,{'3}'NzMachineInt],
        '<_;_>[{'3}'NzMachineInt,{'1}'NzMachineInt]],
        nilBindingList,
        idle)
```

La información relevante para el cómputo de una estrategia se almacena en una lista de ligaduras de valores a metavariables, donde los valores son de tipo `Term` y las metavariables se introducen por el usuario como constantes del tipo `MetaVar`.

El cómputo de la estrategia `set` actualiza la información almacenada asociando a una metavariable `MV` un término `T'`. Esto se lleva a cabo mediante la operación `update`. Antes se utiliza `substituteMetaVars` para sustituir las posibles metavariables en `T'` por su valor actual.

```
eq rewInWith(M, T, BL, set(MV, T')) =
   rewInWith(M, T,
      update(BL,
         binding(MV, meta-reduce(M, substituteMetaVars(T', BL)))),
      idle) .
eq substituteMetaVars(T, nilBindingList) = T .
eq substituteMetaVars(MV, bindingList(binding(MV', T'), BL)) =
   if MV == MV' then T' else substituteMetaVars(MV, BL) fi .
eq substituteMetaVars(F, BL) = F .
eq substituteMetaVars({F}S, BL) = {F}S .
eq substituteMetaVars(F[TL], BL) = F[substituteMetaVars(TL, BL)] .
eq substituteMetaVars((T, TL), BL) =
   (substituteMetaVars(T, BL), substituteMetaVars(TL, BL)).

eq update(bindingList(binding(MV, T), BL), binding(MV', T')) =
   if MV == MV' then bindingList(binding(MV, T'), BL)
            else bindingList(binding(MV, T),
                   update(BL, binding(MV', T')))
   fi .
eq update(nilBindingList, B) = bindingList(B, nilBindingList) .
```

El cómputo de la estrategia `applyWithSubst` aplica una regla, parcialmente instanciada con un conjunto de asignaciones, una única vez al nivel más alto de un término, utilizando el primer ajuste consistente con la sustitución parcial dada. Las representaciones de los términos asignados a las variables pueden contener metavariables que deben ser sustituidas por las representaciones a las que están ligadas en la lista actual de ligaduras. La operación `applyBindingListSubst` hace exactamente eso.

```
eq rewInWith(M, T, BL, applyWithSubst(L, SB)) =
   if meta-apply(M, T, L, applyBindingListSubst(M, SB, BL), 0)
```

```
        == {error*, none}
    then failure
    else rewInWith(M, extTerm(meta-apply(M, T, L,
              applyBindingListSubst(M, SB, BL), 0)),
         BL, idle)
    fi .
  eq applyBindingListSubst(M, none, BL) = none .
  eq applyBindingListSubst(M, ((V <- T); SB), BL) =
    ((V <- meta-reduce(M, substituteMetaVars(T, BL))) ;
    applyBindingListSubst(M, SB, BL)) .
```

Muchas de las estrategias interesantes se definen como concatenación o iteración de estrategias básicas. Para representar estos casos, extendemos el lenguaje de estrategias con los constructores `and`, `orelse`, `iterate` y `while`.

Las ecuaciones para las estrategias `and`, `orelse` e `iterate` se definen a continuación:

```
eq rewInWith(M, T, BL, and(ST, ST')) =
  if rewInWith(M, T, BL, ST) == failure
  then failure
  else rewInWithAux(rewInWith(M, T, BL, ST), ST')
  fi .
eq rewInWith(M, T, BL, orelse(ST, ST')) =
  if rewInWith(M, T, BL, ST) == failure
  then rewInWith(M, T, BL, ST')
  else rewInWith(M, T, BL, ST)
  fi .
eq rewInWith(M, T, BL, iterate(ST)) =
  if rewInWith(M, T, BL, ST) == failure
  then rewInWith(M, T, BL, idle)
  else rewInWithAux(rewInWith(M, T, BL, ST), iterate(ST))
  fi .
```

donde la operación `rewInWithAux` se define mediante la ecuación

```
eq rewInWithAux(rewInWith(M, T, BL, idle), ST) = rewInWith(M, T, BL, ST) .
```

lo que fuerza a que el cómputo de una secuencia de estrategias se realice paso a paso, en el sentido de que una estrategia solo se considerará cuando la anterior ya haya terminado completamente. Podemos ilustrar el cómputo de estas estrategias con los siguientes ejemplos:

```
Maude> rew rewInWith(SORTING,
        '__['<_;_>[{'1}'MachineInt, {'3}'MachineInt],
            '<_;_>[{'2}'MachineInt, {'2}'MachineInt],
            '<_;_>[{'3}'MachineInt, {'1}'MachineInt]],
        nilBindingList,
        and(set(I, {'3}'MachineInt),
            applyWithSubst('sort, ('I <- I)))) .
result StrategyExpression:
rewInWith(SORTING,
        '__['<_;_>[{'1}'NzMachineInt,{'1}'NzMachineInt],
            '<_;_>[{'2}'NzMachineInt,{'2}'NzMachineInt],
            '<_;_>[{'3}'NzMachineInt,{'3}'NzMachineInt]],
        bindingList(binding(I, {'3}'NzMachineInt), nilBindingList),
        idle)
Maude> rew rewInWith(SORTING,
        '__['<_;_>[{'1}'MachineInt, {'3}'MachineInt],
            '<_;_>[{'2}'MachineInt, {'2}'MachineInt],
            '<_;_>[{'3}'MachineInt, {'1}'MachineInt]],
        bindingList(binding(J, {'2}'MachineInt), nilBindingList),
    orelse(applyWithSubst('sort, ('J <- {'4}'MachineInt)),
        applyWithSubst('sort, ('J <- J)))).
result StrategyExpression:
rewInWith(SORTING,
        '__['<_;_>[{'1}'NzMachineInt,{'3}'NzMachineInt],
            '<_;_>[{'2}'NzMachineInt,{'1}'NzMachineInt],
            '<_;_>[{'3}'NzMachineInt,{'2}'NzMachineInt]],
        bindingList(binding(J, {'2}'MachineInt), nilBindingList),
            idle)
Maude> rew rewInWith(SORTING,
        '__['<_;_>[{'1}'MachineInt, {'3}'MachineInt],
            '<_;_>[{'2}'MachineInt, {'2}'MachineInt],
            '<_;_>[{'3}'MachineInt, {'1}'MachineInt]],
          nilBindingList,
          iterate(apply('sort))).
```

```
result StrategyExpression:
rewInWith(SORTING,
        '__['<_;_>[{'1}'NzMachineInt,{'1}'NzMachineInt],
            '<_;_>[{'2}'NzMachineInt,{'2}'NzMachineInt],
            '<_;_>[{'3}'NzMachineInt,{'3}'NzMachineInt]],
        nilBindingList, idle)
```

Finalmente, la estrategia `while` hace que el cómputo de una estrategia dada dependa de que se cumpla una condición. Esta condición debería ser la representación de un término de tipo `Bool`.

```
eq rewInWith(M, T, BL, while(T', ST)) =
    if meta-reduce(M, substituteMetaVars(T', BL)) == {'true}'Bool
    then (if rewInWith(M, T, BL, ST) == failure
        then rewInWith(M, T, BL, idle)
        else rewInWithAux(rewInWith(M, T, BL, ST), while(T', ST))
        fi)
    else rewInWith(M, T, BL, idle)
    fi .
```

El lenguaje de estrategias anterior se puede extender para definir, por ejemplo, el algoritmo de *ordenación por inserción*. La siguiente estrategia `insertion-sort(n)` puede utilizarse para ordenar un vector de enteros de longitud n.

```
   op insertion-sort : MachineInt -> Strategy .
   ops X Y : -> MetaVar .
   var N : MachineInt .
   eq insertion-sort(N) =
      and(set(Y, {'2}'MachineInt),
      while(' <= [Y, {index(', N)}'MachineInt],
         and(set(X, Y),
         and(while(' > [X, {'1}'MachineInt],
               and(applyWithSubst('sort,
                     (('I <- X);
                     ('J <- ' - [X, {'1}'MachineInt]))),
               set(X, ' - [X, {'1}'MachineInt]))),
         set(Y, ' + [Y, {'1}'MachineInt])))))) .
```

Por ejemplo, podemos utilizar la estrategia `insertion-sort` para ordenar un vector de enteros de longitud 10:

```
Maude> rew rewInWith(SORTING,
    '__['<_;_>[{'1}'MachineInt, {'10}'MachineInt],
        '<_;_>[{'2}'MachineInt, {'9}'MachineInt],
        '<_;_>[{'3}'MachineInt, {'8}'MachineInt],
        '<_;_>[{'4}'MachineInt, {'7}'MachineInt],
        '<_;_>[{'5}'MachineInt, {'6}'MachineInt],
        '<_;_>[{'6}'MachineInt, {'5}'MachineInt],
        '<_;_>[{'7}'MachineInt, {'4}'MachineInt],
        '<_;_>[{'8}'MachineInt, {'3}'MachineInt],
        '<_;_>[{'9}'MachineInt, {'2}'MachineInt],
        '<_;_>[{'10}'MachineInt, {'1}'MachineInt]],
    nilBindingList,
    insertion-sort(10)) .
result StrategyExpression:
rewInWith(SORTING,
    '__['<_;_>[{'1}'NzMachineInt,{'1}'NzMachineInt],
        '<_;_>[{'2}'NzMachineInt,{'2}'NzMachineInt],
        '<_;_>[{'3}'NzMachineInt,{'3}'NzMachineInt],
        '<_;_>[{'4}'NzMachineInt,{'4}'NzMachineInt],
        '<_;_>[{'5}'NzMachineInt,{'5}'NzMachineInt],
        '<_;_>[{'6}'NzMachineInt,{'6}'NzMachineInt],
        '<_;_>[{'7}'NzMachineInt,{'7}'NzMachineInt],
        '<_;_>[{'8}'NzMachineInt,{'8}'NzMachineInt],
        '<_;_>[{'9}'NzMachineInt,{'9}'NzMachineInt],
        '<_;_>[{'10}'NzMachineInt,{'10}'NzMachineInt]],
    bindingList(binding(Y, {'11}'NzMachineInt),
    bindingList(binding(X, {'1}'NzMachineInt), nilBindingList)),
idle)
```

En [9] se utiliza una variación del módulo `SORTING` que ilustra cómo se puede controlar el proceso de ordenación de un vector utilizando estrategias diferentes a las que hemos presentado aquí.

Capítulo 6

Catálogo de Refactorizaciones

Este capítulo describe el catálogo de refactorizaciones para Maude que se ha definido en esta tesis, un subconjunto de las cuales ha sido además implementado en el propio sistema Maude. Para cada refactorización que presentaremos, tendremos la siguiente estructura:

- Una **descripción** general de la refactorización, indicando de qué manera la refactorización podría cambiar el programa.
- Un **ejemplo** (generalmente escrito en Maude) que muestre los efectos de la refactorización y su función inversa compensación (si es que existe). Entiéndase por compensación, el hecho de tratar de dar una solución alternativa, cuando la refactorización no se realice completamente.
- Un conjunto de **comentarios** que detallen las propiedades de la transformación de la refactorización. Podría discutirse aquí también de cuestiones generales y/o problemas que surjan, en su caso, durante la eventual implementación.
- Un conjunto de **condiciones** que debe cumplir el programa a transformar para que la refactorización preserve la semántica, o de lo contrario terminar con error.
- Algunas **cuestiones de diseño** de la refactorización. Tener un problema durante el proceso de la refactorización, da lugar a la compensación[1] o generar un fallo; otra alternativa podría ser permitir que sea el propio usuario quién especifique la refactorización.
- Una descripción de la **refactorización inversa** (en el caso de existir) de la realizada. Es decir, si una refactorización transforma la función f en la función f'

[1] Entiéndase por compensación, el hecho de tratar que de dar una solución alternativa, cuando la refactorización no se cumpla completamente.

porque cumple cierta condición c, su inversa será transformar la función f' en f porque no cumple cierta condición c.

Las siguientes son las refactorizaciones que detallaremos en este capítulo:

- Añadir un atributo constructor;
- Eliminar una definición no usada;
- Unraveling de ecuaciones condicionales a no condicionales;
- Añadir atributos de memorización (*memoization*);
- Transformación recursión de cola (*tail recursion*).

6.1. Añadir un atributo constructor

Asumiendo que las ecuaciones de un módulo funcional son *ground* (es decir, no contienen variables), Church-Rosser y terminantes, entonces cada término en el módulo se puede simplificar a una forma canónica. Los constructores son los operadores que aparecen en los términos que están en forma canónica. Los operadores que desaparecen tras la simplificación usando las ecuaciones del programa son llamados *funciones definidas*. Por ejemplo, en el sort `Nat` de Maude el `zero` y sucesor (`s_`) son operadores constructores.

En Maude podemos declarar a un constructor con el atributo `ctor`. Esto es bastante usado para diferentes propósitos tales como para la depuración (*debugging*) y demostración de teoremas, para especificar cuándo un operador dado es un constructor.

6.1.1. Ejemplo de añadir un atributo constructor

Consideremos el módulo `BASIC-NAT` [2] del Ejemplo 9 (Variante del módulo `PEANO-NAT`, Ejemplo 6) escrito en el lenguaje Maude, en donde agregamos el operador de resta $\langle \_ - \_ \rangle$ y su respectivo comportamiento:

Ejemplo 9: Números naturales, módulo original.

```
fmod BASIC-NAT is
    sort Nat .
```

[2] Otra definición del módulo `BASIC-NAT` sería simplemente importar el módulo `PEANO-NAT`, evitando así definir nuevamente los operadores `0` y `s` $\langle \_ + \_ \rangle$, junto con las ecuaciones `eq1` y `eq2`. Tal definición no se ha usado porque esta refactorización no considera código de módulos externos.

```
    op 0 : -> Nat .
    op s : Nat -> Nat .
    op _+_ : Nat Nat -> Nat .
    op _-_ : Nat Nat -> Nat .
    vars N M X Y : Nat .
    eq [eq1] : 0 + N = N .
    eq [eq2] : s(M) + N = s(M + N) .
    eq [eq3] : X - 0 = X .
    eq [eq4] : s(X) - s(Y) = X - Y .
endfm
```

Este módulo introduce un sort llamado `Nat`, con cuatro operadores. El primero es `op 0 : -> Nat` que es una función con aridad cero (sin argumentos); es decir una constante. Matemáticamente hablando, una función con aridad cero siempre devuelve el mismo valor. El segundo operador definido por `op s : Nat -> Nat .` es la función sucesor. El tercer operador definido por `op _+_ : Nat Nat -> Nat .` y el cuarto por `op _-_ : Nat Nat -> Nat .` son las funciones de suma y resta, respectivamente (en notación *mixfix*), de aridad dos (sus argumentos representados por los guiones bajos "_").

Primer caso:

Notamos que las ecuaciones `eq1` y `eq2` definen el comportamiento para el operador de adición, del mismo modo que lo hacen las ecuaciones `eq3` y `eq4` para el operador de resta. Obviamente, no hay ecuaciones que definan el comportamiento para los operadores `0` y `s`. Estrictamente hablando, estos dos operadores generan datos o, por decirlo de otra manera, permiten construir términos constructores. Además, usando `0` y `s` se pueden generar todos los elementos del sort `Nat` y los operadores de adición y resta se aplican a ellos; es decir, un resultado de una adición o resta puede ser `0` o de la forma `s(s(...s(0)...))`.

Entonces la definición correcta para los operadores sucesor y cero debe acompañarles el atributo `[ctor]`.

```
op s : Nat -> Nat [ctor] .
op 0 : -> Nat [ctor].
```

Segundo caso:

Un operador que no tiene un semántica asociada (cumple el primer caso) y es de aridad cero (una constante), debe declararse como un operador constructor. En el módulo, el único que cumple esta condición es el operador `0`.

Entonces la definición correcta para este operador constructor la declararía acompañado por el atributo `[ctor]`.

```
op 0 : -> Nat [ctor].
```

Tercer caso:

Si sólo nos basamos en el primer caso, podríamos decir que los operadores suma y resta no son constructores porque tienen una semántica propia. Sin embargo, eso no se puede concluir de forma directa para estos dos operadores. Así que los analizaremos por separado.

El operador suma (`+`) tiene un conjunto de ecuaciones que definen su semántica (`eq1` y `eq2`) y cualquiera que sean los valores dentro del dominio de los números naturales, el resultado siempre será un número natural; es decir siempre va a reducirse a una forma normal constructora; por consiguiente este operador no es un constructor.

El operador resta (`-`) tiene un conjunto de ecuaciones que definen su semántica (`eq3` y `eq4`), pero no todos los valores dentro del dominio de los números naturales dan como resultado un valor basado en `0` o `s`. Citemos el ejemplo de restar los números 2 - 3 (número enteros), cuyo resultado es un número que no es natural (-1) y que se puede interpretar como un error en la secuencia de cómputo no manejado por la teoría, ya que un término ground sólo se asocia a un sort y no a un kind; es decir, siempre debe retornar un valor sort correspondiente. Ante esto, *¿el operador debe evaluarse al kind del sort definido, en este caso* `[Nat]`*?*. La respuesta es un negativa, ya que, al declarar que el operador se evalúa a un kind, no tenemos una solución completa; es decir, aún hay error. Lo que se busca es tratar de que esta situación no sea considerada un error, sino un tipo especial de dato. Pero entonces *¿cómo se logra ésto?*. La forma más fácil es: i) evaluar el operador a su kind y ii) declararlo como un constructor. Con esto conseguimos que cualquier valor resultante que no se puede reducir a una forma normal sea considerado como un tipo especial de dato y no un error.

Entonces la definición correcta para este operador es:

```
op _-_ : Nat Nat -> [Nat] [ctor] .
```

El módulo refactorizado del Ejemplo 9 quedaría escrito de la siguiente manera:

Ejemplo 10: Números naturales refactorizado.

```
fmod BASIC-NAT is
    sort Nat .
    --- Operador (0) definido constructor
    op 0 : -> Nat [ctor] .
    --- Operador  (s) definido constructor
    op s : Nat -> Nat [ctor] .
    op _+_ : Nat Nat -> Nat .
    --- Operador (-) definido constructor y kind
    op _-_ : Nat Nat -> [Nat] [ctor] .
    vars N M X Y : Nat .
    eq [eq1] : 0 + N = N .
    eq [eq2] : s(M) + N = s(M + N) .
    eq [eq3] : X - 0 = X .
    eq [eq4] : s(X) - s(Y) = X - Y .
endfm
```

6.1.2. Comentarios

Los dos primeros casos descritos en la sección anterior, han sido analizados en la herramienta de verificación de completitud SCC (del inglés *Sufficient Completeness Checker*) de Maude. Esta herramienta verifica que todas las operaciones definidas (no declaradas como constructores), hayan sido definidas completamente[3], es decir para todos los posibles valores de sus argumentos.

A diferencia de SCC, la refactorización implementada en este apartado modifica el código del programa (su sintaxis); es decir, se agrega el atributo `ctor` a todos los operadores que cumplen los casos analizados en la sección anterior. También agrega un comentario explícito de la refactorización realizada en la línea anterior a la que se ha modificado.

El *paso 2* descrito en la Subsubsección 6.1.4 podría provocar ejecuciones largas, ya que tiene una relación directa con el número de argumentos que una función pueda tener y con el número de simbolos contructores que pertenezcan a cada uno de los sort de cada argumento.

[3]Podemos encontrar este concepto en la sección 21.1.5 de [10]

6.1.3. Condiciones de aplicación

Esta refactorización considera únicamente las instrucciones que han sido escritas en el módulo, pasando por alto todo código de módulos externos que han sido importados.

6.1.4. Cuestiones de diseño

La refactorización se realiza en dos pasos que cubren los tres casos que se presenta en el ejemplo de la Subsección 6.1.1.

El *paso 1* es automático y cubre los dos primeros casos de la refactorización.

El *paso 2* es semiautomático y cubre el tercer caso analizado, requiere de la confirmación del programador que realiza la refactorización ya que cambia un *sort* por un *kind* a ciertos operadores (los que estén dentro del tercer caso).

Paso 1:

1. Una teoría ecuacional de entrada E.
2. De la signatura Σ de E, coger el conjunto de símbolos de función $\mathcal{D} = f_1, ..., f_n$ y el conjunto de ecuaciones $\triangle = e_1, ..., e_m$ de la forma $l = r \Leftarrow c$, en donde c es un término que representa la condición (si $c = \varnothing$ se trata de una ecuación no condicional), y donde $n \geq 0$ y $1 \leq m$.
3. Para cada símbolo $f_i \in \mathcal{D}$ donde $1 \leq i \leq n$, verificar si no existe $e_i \in \triangle$ tal que $l|_\Lambda = f_i$; es decir, no existe una ecuación del conjunto de ecuaciones $\triangle$ que definan el comportamiento de f_i (no tiene una semántica definida).
   a) Si la verificación para f_i tiene éxito, éste deja de ser un símbolo definido ($f_i \notin \mathcal{D}$) y se convierte en un símbolo constructor $f \in \mathcal{C}$.
   b) Si la verificación para f_i fracasa, continuamos con la verificación de f_{i+1} hasta que $i \geq n$.

Paso 2:

1. Una teoría ecuacional de entrada E.
2. De la signatura Σ de E, coger el conjunto de símbolos de función $\mathcal{D} = f_1, ..., f_n$, donde $f :: s_1, ..., s_p \mapsto s$ es la signatura del operador $f \in \mathcal{D}$, de aridad p y tipo s, el conjunto de ecuaciones $\triangle = e_1, ..., e_m$ de la forma $l = r \Leftarrow c$ en donde c

es un término que representa la condición (si $c = \varnothing$ se trata de una ecuación no condicional) y el conjunto de símbolos constructores $\mathcal{C} = c_1, ..., c_q$ de igual tipo que los argumentos de f.

3. Para cada símbolo f_i, con $1 \leq i \leq n$, evaluar con cada símbolo constructor c_j para $1 \leq j \leq q$, hasta obtener una forma normal.

   a) Si la forma normal no es del mismo tipo que el operador f_i, informar al usuario de que se procederá a realizar un cambio y continuar con f_{i+1}.

      1) La modificación a realizar es cambiar el tipo de operador de sort a kind y hacer que pertenezca al conjunto de símbolos constructores.

   b) Si la forma normal es del mismo tipo que el operador f_i, continuar con c_{k+1} hasta que $k \geq q$.

   c) Cuando se ha evaluado f_i con cada símbolo constructor de $\mathcal{C}$, entonces se continúa analizando a partir de f_{i+1} y el f_i queda sin cambios.

6.1.5. Refactorización inversa

La refactorización inversa de esta refactorización sería *"eliminar un atributo constructor"* de aquellos operadores que han sido declarados como contructores (tienen el atributo `ctor`); sin embargo, no pertenecen a los tres casos que hemos presentado en esta sección.

6.2. Eliminar una definición no usada

Dentro de un programa fuente existe código que se ejecuta pero cuyo resultado nunca se usa, llamado *código muerto* (*dead code*). Esto implica el uso de recursos en algo que jamás se usa. Detectarlo y eliminarlo es una tarea no trivial. Concretando en el lenguaje Maude, cuando importamos un módulo dentro de otro, el código externo (del módulo importado) llega a ser parte del módulo que realiza la importación, y si en el módulo importado eliminamos algún código que creemos que no está siendo utilizado, podría causar algún tipo de error en el módulo que realizó la importación.

El tipo de código muerto en el que nos centraremos es detectar y eliminar variables que no están siendo utilizadas dentro del módulo. Además se ve la necesidad de mejorar la legibilidad del código de un programa cuando se tiene situaciones que generen una mala comprensión por parte del programador que lo lee; concretamente nos referimos a mantener una única declaración de una variable dentro de un módulo en particular[4],

[4]Maude permite declaraciones de una variable en la sección `vars` que es a nivel de módulo y a vuelo pluma, incluso para diferentes sorts y kinds. Este tipo de declaraciones no altera el comportamiento de las ejecuciones pero tener una única declaración para cada variable es más legible para el programador.

que ha sido declarada más de una vez en el módulo.

Otro caso que podría agregarse a esta refactorización es eliminar operadores que no están siendo referenciados dentro de un módulo; sin embargo, eso no es tan trivial, por ejemplo consideremos tener dos módulos M_1 y M_2. Supongamos que en M_1 existe un operador definido f que no está siendo utilizado en ese módulo, pero M_2 hace una extensión de M_1 y en éste se define el comportamiento de f con un conjunto de ecuaciones $\triangle_{M_2}$. Por tanto, si f se elimina de M_1 por no estar siendo utilizada en ese módulo, esto haría que cambie el comportamiento de M_2, lo que genera un error de "*operador no definido*". Por el hecho de no considerar la refactorización de módulos extendidos y por lo que acabamos de mencionar, no se realiza la refactorización de operadores no utilizados.

6.2.1. Ejemplo de eliminar una definición no usada

Consideremos el siguiente ejemplo que es una variante del Ejemplo 10[5] que hemos ajustado para apreciar la transformación de esta refactorización.

Ejemplo 11: Variante de los números naturales.

```
fmod BASIC-NAT1 is
    sort Nat .
    op 0 : -> Nat [ctor] .
    op s : Nat -> Nat [ctor] .
    op _+_ : Nat Nat -> Nat .
    op _-_ : Nat Nat -> [Nat] [ctor] .
    op conj : Bool Bool -> Bool .
    op disy : Bool Bool -> Bool .
    vars N X Y Z : Nat .
    var M P : Bool .
    eq [eq1] : 0 + N:Nat = N:Nat .
    eq [eq2] : s(M:Nat) + N:Nat = s(M:Nat + N:Nat) .
    eq [eq3] : X:Nat - 0 = X:Nat .
    eq [eq4] : s(X) - s(Y) = X - Y .
    eq [eq5] : conj(M,P) =
            if M == P and M == true
            then true
            else false fi .
```

[5]Se omiten las líneas de comentarios y se agregan los operadores `con` y `disy` de conjunción y disyunción respectivamente, con su respectiva semántica.

```
    eq [eq6] : disy(V:Bool,true) = true .
    eq [eq7] : disy(V:Bool,false) = V:Bool .
endfm
```

En el módulo notamos la declaración global de las variables `N` y `M` que no están siendo utilizadas en ninguna ecuación, ya que en las ecuaciones para el operador `+` éstas se encuentran definidas a vuelo pluma. Esto implica que dichas variables no se manipulan en ninguna reescritura. También se puede apreciar que la variable `X` está declarada para poder ser utilizada en todo el módulo como lo hacemos en la ecuación `eq4`; sin embargo, en la `eq3` existe la declaración de la misma variable `X` a vuelo pluma por lo que debemos analizar y decidir cuál de las dos declaraciones debe permanecer. Tambien existe la variable `Z` que no es referenciada en ninguna línea de código y debería ser eliminada.

Para solventar este inconveniente se lo presenta dentro de cuatro casos, en donde en segundo y tercero son complementarios entre sí. Los casos son los siguientes:

Primer caso

Variables nunca referenciadas como la variable `Z` que ha sido declarada en la sección de variables `vars` para poder ser usada en todo el módulo. Sin embargo, esta variable no ha sido utilizada en ninguna ecuación definida en el programa, lo que implica que no afecta al comportamiento del mismo y debería ser eliminada.

Segundo caso

Una misma variable declarada en más de una ocasión, en la sección `vars` *y a vuelo pluma, para un mismo sort.* Es el caso de la variable `N` de sort `Nat` que puede usarse en todo el módulo; sin embargo, en las ecuaciones `eq1` y `eq2` nuevamente se declara a vuelo pluma la variable `N` también de sort `Nat`. Por ello deberíamos considerar tan sólo una de las dos declaraciones de las variables ya que, por la necesidad de que el código sea legible, una de ellas es irrelevante. Se puede realizar una de las siguientes opciones:

1. Eliminar la declaración de `N` en la sección `vars` y dejar únicamente las declaraciones a vuelo pluma, así:

```
vars X Y Z : Nat .
eq [eq1] : 0 + N:Nat = N:Nat .
eq [eq2] : s(M:Nat) + N:Nat = s(M:Nat + N:Nat) .
```

2. Eliminar la declaración de `N` a vuelo pluma y dejar únicamente las declaración de la sección `vars`, así:

```
vars N X Y Z : Nat .
eq [eq1] : 0 + N = N .
eq [eq2] : s(M:Nat) + N = s(M:Nat + N) .
```

Tercer caso

Es una variante del segundo caso que se puede apreciar con la variable `X` que igual que `N` es declarado en la sección `vars` y también a vuelo pluma en la ecuación `eq3`. Sin embargo, la diferencia con `N` está en la ecuación `eq4,` en donde la variable `X` declarada en la sección vars es referenciada para el sort `Nat`. Para tratar este problema podemos elegir una de las siguientes opciones:

1. Extender la opción *1* del *segundo caso* y eliminar la declaración de `X` de la sección `vars` y dejar las declaraciones que están hechas a vuelo pluma. Además, en toda ecuación que haga referencia a `X`, ésta se declara a vuelo pluma, así:

```
vars N M Y Z : Nat .
eq [eq3] : X:Nat - 0 = X:Nat .
eq [eq4] : s(X:Nat) - s(Y) = X:Nat - Y .
```

   Sin embargo, esto va a depender de la cantidad de ecuaciones que usen la variable `X`, ya que si son muy numerosas ésta no podría ser la mejor opción.

2. Igual que la opción *2* del *segundo caso*, es decir eliminar todas las declaraciones de `X` a vuelo pluma y dejar únicamente la declaración de la sección `vars`, así:

```
vars N M X Y Z : Nat .
eq [eq3] : X - 0 = X .
eq [eq4] : s(X) - s(Y) = X - Y .
```

Cuarto caso

Una misma variable declarada más de una vez para dos sorts diferentes. Es el caso de la variable `M` que en la sección `vars` está definida para el sort `Bool` mientras que en la ecuación `eq2` esta variable nuevamente ha sido declarada a vuelo pluma para el sort `Nat`. Este no sería un inconveniente si la variable no estuviera siendo referenciada en

las diferentes ecuaciones para ambos sorts, como son el caso de `eq2` y `eq5` (`Bool` y `Nat`, respectivamente). Lo correcto sería tener una única declaración de la variable pero sin afectar la semántica del programa, para lo cual podríamos realizar una de las siguientes opciones:

1. Agregar una variable fresca `M'` para el sort que esté definido con `M` a vuelo pluma, en este caso para el sort `Nat` en donde $\mathtt{M}' \bigcap \mathtt{V} = \varnothing$. A continuación, reemplazar cada declaración de `M` de sort `Nat` por `M'`. Esto hace que al final tenga únicamente `M` definido en la sección `vars` para el sort `Bool`, así:

```
var M P : Bool .
eq [eq2] : s(M':Nat) + N:Nat = s(M':Nat + N:Nat) .
eq [eq5] : conj(M,P) =
        if M == P and M == true
        then true
        else false fi .
```

2. Eliminar la declaración `M` en la sección `vars` y declarar `M` a vuelo pluma en cada ecuación donde se haga referencia a `M` con el sort correspondiente, así:

```
var P : Bool .
eq [eq2] : s(M:Nat) + N:Nat = s(M:Nat + N:Nat) .
eq [eq5] : conj(M:Bool,P) =
            if M:Bool == P and M:Bool == true
            then true
            else false fi .
```

6.2.2. Comentarios

Las variables que son declaradas dentro de un módulo de Maude únicamente existen dentro de éste, por lo que no tendrá efectos laterales si el módulo que se está refactorizando ha sido incluido en algún otro módulo. Es decir, este tipo de refactorización a nivel de variables es totalmente transparente a módulos ajenos que hagan uso del módulo refactorizado.

6.2.3. Condiciones de aplicación

Esta refactorización considera una lógica ecuacional con pertenencia, ya que las variables son para un sort en concreto.

6.2.4. Cuestiones de diseño

Esta refactorización se realiza dentro de un mismo procedimiento y cubre los cuatro casos analizados. Es también automática y agrega a cada línea anterior a la que se realiza la refactorización un comentario de la acción realizada.

Procedimiento

1. Un sistema de reescritura SRT de entrada.
2. Tomamos el conjunto de variables definidas en el sistema $V_S = x_1 :: s_1, ..., x_m :: s_h$, en donde $m \geq 1$, $h \geq 1$ y $S = s_1, ..., s_n$ es el conjunto de sorts.
3. Consideramos el conjunto de términos $\Gamma_\Sigma(V)$ con variables.
4. Para cada variable $x_i :: s$, se verifica si está siendo referenciada dentro de algún término $t \in \Gamma_\Sigma(V)$.
   a) Si $x_i :: s$ no es referenciada en ningún término t, entonces la variable está muerta y debe eliminarse y el proceso continúa con $x_{i+1} :: s$.
   b) Si $x_i :: s$ no es referenciada en ningún término t pero existe una declaración a vuelo pluma de $x_i :: s$, entonces se conserva únicamente la variable declarada a vuelo pluma y el proceso continúa con $x_{i+1} :: s$.
   c) Si $x_i :: s$ es referenciada en algún término t y no existe una declaración a vuelo pluma de $x_i :: s$, entonces la variable debe permanecer en el código sin cambios y el proceso continúa con $x_{i+1} :: s$.
   d) Si $x_i :: s$ es referenciada en algún término t y existe en un término t' una declaración a vuelo pluma de $x_i :: s$ de igual sort s, entonces se eliminan todas las declaraciones a vuelo pluma y el proceso continúa con $x_{i+1} :: s$.
   e) Si $x_i :: s$ es referenciada en algún término t y existe en un término t' una declaración a vuelo pluma de $x_i :: s'$ para un sort distinto s', entonces se declara a vuelo pluma una nueva variable $x' :: s'$ para el sort s' y se reemplaza $x_i :: s'$ por $x' :: s'$ en t'. Esto hace que tengamos para cada sort s y s' distintas variables; el proceso continúa con $x_{i+1} :: s$.

6.2.5. Refactorización inversa

Lo que se busca con esta transformación es mejorar la legibilidad del código, por lo tanto para esta refactorización no consideramos la transformación inversa para ninguno de los casos analizados.

6.3. De ecuaciones condicionales a incondicionales

El enfoque de esta refactorización es generar sistemas incondicionales SRT computacionalmente equivalentes a sistemas condicionales SRTC. El unraveling a realizar en esta refactorización consiste en transformar las ecuaciones condicionales en ecuaciones incondicionales escritas de la forma $l \rightarrow r \Leftarrow u_1 = v_1 \wedge ... \wedge u_n = v_n$ y $l \rightarrow if(u = v_1 \wedge ... \wedge u_n = v_n, r)$ [6], respectivamente.

Como se ha descrito en Sección 4.3 hay dos casos que se contemplan en esta refactorización:

1. Dos ecuaciones condicionales complementarias entre sí de la forma de (e1) y (e2) ($l \rightarrow r_1 \Leftarrow u_1 = v_1 \wedge ... \wedge u_n = v_n$ y $l \rightarrow r_2 \Leftarrow s_1 \neq t_1 \wedge ... \wedge s_n \neq t_n$, respectivamente).
2. Otras ecuaciones condicionales de la forma (e6) ($l \rightarrow r \Leftarrow u_1 = v_1 \wedge ... \wedge u_n = v_n$).

6.3.1. Ejemplo de ecuaciones condicionales a incondicionales

Para mostrar la refactorización que se lleva a cabo, se presentan dos ejemplos para cada caso, que resaltan diferencia a la hora de detectar el primer caso y cuándo el segundo.

Primer caso

Consideremos el siguiente ejemplo escrito en el lenguaje Maude que recupera el valor máximo de una lista de naturales.

Ejemplo 12: Máximo de una lista.

```
fmod MAX-LISTA is
    pr NAT .
    sort NatList .
    subsort Nat < NatList .
    op nil : -> NatList .
    op _:_ : NatList NatList -> NatList[assoc id: nil] .
    op max : NatList -> Nat .
    var N : Nat .
```

[6] Por abuso de sintaxis y simplemente por representación, se asume que el operador *if* permite la reescritura de l por r únicamente al evaluarse a *true*, caso contrario no hace nada.

```
    var NL : NatList .
    eq [eq1]: max(nil) = 0 .
    eq [eq2]: max(N) = N .
    ceq [ceq1]: max(N : NL) = N if N > max(NL) .
    ceq [ceq2]: max(N : NL) = max(NL) if N <= max(NL) .
endfm
```

En este módulo tenemos el operador definido `max` de aridad 1 y que es evaluado al sort `Nat`. El argumento que recibe es una `NatList` que está formada por la concatenación de valores por medio del operador $\langle \_ : \_ \rangle$ de aridad 2 y notación infija. Sus argumentos son dos `NatList` y se evalúa a `NatList`.

Se han especificado cuatro ecuaciones que definen la semántica del operador `max`, de las cuales `eq1` y `eq2` son incondicionales y `ceq1` y `ceq2` son condicionales. Las ecuaciones que son consideradas para este tipo de refactorización (primer caso) son las ecuaciones condicionales `ceq1` y `ceq2` y nuestro objetivo es convertirlas en ecuaciones incondicionales.

Al analizar las ecuaciones `ceq1` y `ceq2` notamos que están dentro del primer caso analizado. Con una especificación semiformal decimos que `ceq1` y `ceq2` cumplen la forma de las ecuaciones (e1) y (e2), en donde `e1` $\equiv$ `ceq1` y `e2` $\equiv$ `ceq2` detalladas en la Subsección 4.3.1, de forma que si detallamos la relación entre estas tendremos:

$l = \texttt{max}(\texttt{N} : \texttt{NL})$
$r1 = \texttt{N}$
$r2 = \texttt{max}(\texttt{NL})$
$u1 = s1 = \texttt{N}$
$v1 = t1 = \texttt{max}(\texttt{NL})$

Por tanto podemos aplicar la ecuación (e5) $l \rightarrow if(u = v_1 \wedge ... \wedge u_n = v_n, r1, r2)$ y tendremos:

$$\texttt{max}(\texttt{N} : \texttt{NL}) \rightarrow \texttt{if}(\texttt{N} > \texttt{max}(\texttt{NL}), \texttt{N}, \texttt{max}(\texttt{NL})) \tag{e7}$$

Ahora al escribir la ecuación (e7) en el lenguaje Maude, tendremos lo siguiente:

```
eq [ceq3]: max(N : NL) = if N > max(NL) then N else max(NL) fi .
```

La nueva ecuación `ceq3` reemplaza a las dos ecuaciones que anteriormente estaban escritas, de tal manera que el módulo del Ejemplo 12 quedaría escrito de la siguiente manera:

```
fmod MAX-LISTA is
    pr NAT .
    sort NatList .
    subsort Nat < NatList .
    op nil : -> NatList .
    op _:_ : NatList NatList -> NatList [assoc id: nil] .
    op max : NatList -> Nat .
    var N : Nat .
    var NL : NatList .
    eq [eq1]: max(nil) = 0 .
    eq [eq2]: max(N) = N .
    --- Unraveling caso 1 ecuaciones operador (max)
    eq [ceq3]: max(N : NL) = if N > max(NL)
                             then N
                             else max(NL) fi .

endfm
```

Semánticamente este módulo expresa lo mismo que las ecuaciones `ceq1` y `ceq2` juntas, pero la diferencia es que el tiempo que se tarda en realizar una reducción de términos para el operador `max` es significativamente menor, ya que el cómputo de las ecuaciones incondicionales es menor que las ecuaciones condicionales como vimos en la Sección 4.3.

Segundo caso

Para este caso consideremos el mismo ejemplo de SRTC que se ha estudiado en la Subsección 4.3.2:

Ejemplo 13: Módulo TEST-2

```
fmod TEST-2 is
    pr NAT .
    op f : Nat -> Nat .
    op g : Nat -> Nat .
    var X : Nat .
```

```
    ceq [ceq4]: f(g(X)) = X if X = 0 .
    eq [eq3]: g(g(X)) = g(X) .
endfm
```

En el módulo tenemos una ecuación condicional y una incondicional que son `ceq4` y `eq3`, respectivamente.

Según la teoría que se ha presentado para poder transformar este tipo de ecuación, el resultado del Ejemplo 13 sería:

```
fmod TEST-2REF is
    pr NAT .
    pr BOOL .
    --- Unraveling caso 2: aridad operador (f) extendida
    op f : Nat Bool -> Nat .
    op g : Nat -> Nat .
    --- Unraveling caso 2: operador (equal) agregado
    op equal : Nat Nat -> Bool .
    --- Unraveling caso 2: operador ({_}) agregado
    op {_} : Nat -> Nat .
    var X, Y : Nat .
    --- Unraveling caso 2: ecuacion agregada
    eq {{X}} = {X} .
    --- Unraveling caso 2: ecuacion agregada
    eq [eqfin] : {X} = X .
    --- Unraveling caso 2: ecuacion agregada
    eq f({X},B:Bool) = {f(X,true)} .
    --- Unraveling caso 2: ecuacin agregada
    eq g({X}) = {g(X)} .
    --- Unraveling caso 2: ecuacion agregada
    eq equal(X,X) = true .
    --- Unraveling caso 2: ecuacin agregada
    eq equal(X,Y) = false .
    --- Unraveling caso 2: ecuacin modificada
    eq f(g(X),true) = if equal({X},{0}) == true
                      then {X}
                      else f(g(X),false) fi .
    --- Unraveling caso 2: ecuacion modificada
    eq g(g(X)) = {g(X)} .
```

```
endfm
```

Podemos notar que se ha incrementado de 1 a 2 la aridad del operador `f`, se han agregado los operadores unario `{_}` y binario `equal`. Asimismo se han agregado 6 nuevas ecuaciones que definen el comportamiento de los dos operadores definidos (3 ecuaciones), así como las ecuaciones propias de la teoría investigada (las otras 3). Finalmente, se ha modificado la ecuación del operador `g` y cambiado la ecuación condicional del operador `f` a una ecuación incondicional (que era el objetivo).

6.3.2. Comentarios

La decisión final de realizar o no esta transformación es del usuario, ya que los cambios sintácticos que se realizan en el código son bastante significativos: la aridad de los operadores de las ecuaciones condicionales se extiende, se agregan ecuaciones, etc.

Por nuestra parte, tratamos de hacer que la transformación sea lo más explícita posible, ya que detallamos por medio de comentarios cada uno de los cambios que se han realizado en el código.

6.3.3. Condiciones de aplicación

Un detalle muy importante es que no se considera que el módulo a refactorizar esté siendo importado[7] desde otro módulo. Si fuera así, podría posteriormente generar errores porque las aridades de los símbolos de función definidos que se han refactorizado han sido modificadas.

6.3.4. Cuestiones de diseño

Esta refactorización se realiza en dos pasos que cubren los dos casos analizados.

El *procedimiento caso 1* descrito más abajo es automático y cubre el *caso 1* de la refactorización.

El *procedimiento caso 2* es semiautomático y cubre el *caso 2* analizado. Requiere de la confirmación del programador que realiza la refactorización ya que introduce modificaciones en el código tales como extender la aridad del operador de la ecuación condicional, agregar nuevas ecuaciones y modificar otras.

[7] Cualquiera de los tres tipos de importaciones: *protecting, including o extending.*

Procedimiento caso 1

1. Un sistema de reescritura condicional SRTC de entrada.

2. Tomamos el conjunto de símbolos de función definidos $D = f_1, ..., f_n$, donde $n \geq 1$ es el total de símbolos definidos.

3. Definimos un subconjunto de pares de ecuaciones de $\triangle = e_1, ..., e_z$[8] que se denota como $\triangle'_w = e'_1, ..., e'_w$, $\left(\triangle'_w \subseteq \triangle_z\right)$ formado por los pares de las ecuaciones condicionales de la forma $e : l = r \Leftarrow u_1 = v_1 \wedge ... \wedge u_m = v_m$ que cumplen las condiciones del primer caso descrito en la Subsección 4.3.1, con $m \geq 1$, z el número total de ecuaciones del sistema, y w el total de pares de ecuaciones, siendo w un número par mayor o igual a 2 y $w \leq z$. Debemos tener en cuenta que si $z < 2$ todo este proceso termina, ya que no existirá ningún símbolo de función con un par de ecuaciones condicionales que definan su comportamiento y que esté dentro del primer caso de esta refactorización.

4. Del conjunto $\triangle'_w$ se seleccionan los pares de ecuaciones para cada operador.

5. Para cada par de ecuación e'_i y e'_j con $i, j \leq w$ e $i \neq j$, tomamos sus lados *lhs*, *rhs* y sus respectivas guardas (l, $r1$, $r2$, $u_1, v_1, ..., u_m, v_m$)[9].

6. Se considera ahora el operador $if\ (\_, \_, \_)$ y las reglas $if\ (true, x, y) \rightarrow x$ y $if\ (false, x, y) \rightarrow y$ para definir la nueva ecuación $e'_{i,j}$ de la forma de (e5) con los términos tomados en el paso anterior y en el orden definido por la misma (e5), que reemplaza a e'_i y e'_j en el sistema.

7. El proceso continúa analizando todos los pares de ecuaciones e'_i y e'_j del conjunto $\triangle'_w$ y, para cada par, los pasos 5 y 6 de este proceso[10].

Procedimiento caso 2

1. Consideramos un sistema de reescritura condicional SRTC de entrada.

2. Tomamos el conjunto de símbolos de función definidos $D = f_1, ..., f_n$, donde $n \geq 1$ es el total de símbolos definidos.

3. Definimos un subconjunto de $\triangle = e_1, ..., e_h$ que denotamos como $\triangle''_q = e''_1, ..., e''_q$, $\left(\triangle''_q \subseteq \triangle_z\right)$ formado por las ecuaciones condicionales de la forma $e : l = r \Leftarrow$

[8] Cuando se quiere hacer explícito el total de ecuaciones que forman el conjunto $\triangle$ se usa con el subdíndice z ($\triangle_z$); es decir, $\triangle$ y $\triangle_z$ expresan lo mismo.

[9] Ya que se cumple la condición $u_1 = s_1 \wedge ... \wedge u_n = s_n$ y $v_1 = t_1 \wedge ... \wedge v_n = t_n$, los términos s_i y t_i son iguales a los términos u_j y v_j respectivamente, donde $i = j$ para $j \leq n$, por lo que son omitidos.

[10] El sistema resultante no necesariamente es un SRT, ya que sólo se han analizado las ecuaciones que cumplen el *caso 1*.

$u_1 = v_1 \wedge ... \wedge u_m = v_m$ que no pertenezcan a $\triangle_w^{'}$ (si $w \geq 2$), donde $\triangle_w^{'} \cup \triangle_q^{''} = \triangle_z$ y $w + q{=}z$, con $z \geq 1$ y $q \leq z$.

4. Por cada ecuación e_j donde $j \leq q$ del conjunto $\triangle_q^{''}$, tomamos un subconjunto de variables $V' \subseteq V$, que pertenecen al conjunto de términos $\Gamma(V)_s$ que están en las ecuaciones condicionales de $\triangle_q^{''}$.

5. En V' se agregan nuevas variables k_f para la signatura Σ; es decir, agregamos el total de variables nuevas que son igual al total de reglas de reescrituras condicionales.

6. Se genera $\overline{\Sigma}$ agregando a cada operador $f \in \Sigma$ los k_f argumentos necesarios a la derecha de los n argumentos iniciales.

7. Se genera $I\left(\overline{\Sigma}\right)$, la infraestructura del sistema de reescritura incondicional $\overline{\Sigma}^{'}$, agregando a $\overline{\Sigma}$ los operadores $if\,(\_,\_,\_)$ y *equal*? y sus correspondientes reglas.

8. Se genera $\overline{R}$, el sistema de reescritura incondicional $\overline{\Sigma}^{'}$, agregando a $I\left(\overline{\Sigma}\right)$ la operación $\{\,\_\,\}$ a sus reglas, así como las reglas de reescritura incondicionales asociadas a las reglas R, como se muestra en *la transformación* de la Subsección 4.3.2.

9. Se agregan las ecuaciones necesarias para poder reducir los términos $\{\bar{t}\}$ a una forma normal t. La ecuación necesaria para esta transformación es $\{\bar{t}\} \rightarrow t$; sin embargo, esta ecuación debe ser usada de manera cautelosa ya que puede causar confusión con la regla (r3) ($\{\{x\}\} \rightarrow \{x\}$). Por tanto, su uso debe hacerse únicamente al final del proceso; es decir, cuando todas las reescrituras se han realizado y nos queda sólo la expresión $\{\bar{t}\}$.

10. El proceso termina cuando $j \geq q$.

6.3.5. Refactorización inversa

Después de presentar las mejoras que se obtienen al momento de transformar un SRTC en un SRT, y su dificultad, difícilmente se puede plantear una refactorización inversa. En un hipotético caso de querer implementar una refactorización inversa, el grado de complejidad para ésta sería muy elevado, ya que esta refactorización genera un número determinado de nuevas ecuaciones y extiende la aridad del operador refactorizado.

6.4. Añadir atributos de memorización (*memoization*)

Como ya vimos, la memorización para los programas funcionales es una técnica poderosa que permite reutilizar resultados y no volver a computarlos. La semántica e implementación de búsqueda en la tabla de memoria son críticos; sin embargo, en Maude esta refactorización no precisa implementar la memorización en si misma, sino únicamente detectar aquellos operadores que deberían utilizarla. El hecho de sólo detectar y no implementar la memorización se debe a que Maude posee un mecanismo de memorización propio, que se realiza a través del atributo `memo` (comentado en la Subsubsección 5.9.4.2). Por tanto, el programador puede hacer uso de ella (la memorización) en cualquier operador de manera fácil, únicamente agregándole dicho atributo (al operador).

6.4.1. Ejemplo de memorización

Para visualizar esta refactorización, vamos a presentar dos ejemplos que son: (i) el factorial de un número y (ii) la sucesión de fibonacci, que corresponden a dos formas distintas de recursión. Posteriormente haremos una breve comparación entre estos dos casos.

6.4.1.1. Factorial de un número

Consideremos el Ejemplo 14 del factorial de un número escrito en Maude.

Ejemplo 14: Factorial de un número

```
fmod FACTORIAL is
    protecting INT .
    op _! : Int -> Int .
    var N : Int .
    eq [eq1] : 0 ! = 1 .
    eq [eq2] : N ! = (N - 1)! * N [owise] .
endfm
```

Este módulo funcional define el operador factorial `_!` de aridad 1 evaluado a `Int`. Es un ejemplo que nos permite ver claramente la necesidad del uso de memorización. Por ejemplo, si evaluamos la expresión `50!` en Maude, el número de reescrituras es de *151* durante las n repeticiones que hagamos. Sin embargo, si agregamos el atributo `memo` al operador de factorial

```
op _! : Int -> Int [memo] .
```

y nuevamente evaluamos la expresión `50!`, notamos que, en su primera evaluación, el número de reescrituras no ha variado; es decir, sigue siendo *151*, lo que no sucede en las siguientes evaluaciones que hagamos. Por ejemplo, al hacer una segunda, tercera e i-ésima evaluación de esa expresión, el número de reescrituras disminuye a *1*; es decir, el valor de dicha computación ha sido almacenado y reusado en lugar de ser recalculado. Si probamos ahora evaluando las expresiones `40!` y `60!`, notaremos que la reescritura será de *1* y *31* respectivamente, ya que tanto para el primer y segundo caso utiliza los valores que calculó en la expresión *50 !* y, específicamente en el segundo caso, sólo busca los valores que faltan por calcularse en base a los que ya conoce. En este caso, Maude detecta que faltan los valores desde el *50* hasta el *60*, por lo que realiza el cálculo respectivo, memoriza el valor en la tabla hash y, por último, lo devuelve.

Tras aplicar la refactorización al módulo `FACTORIAL` del Ejemplo 14, se obtendrá como resultado el módulo `FACTORIAL-M`, que muestra en el Ejemplo 15.

Ejemplo 15: Factorial de un número refactorizado

```
fmod FACTORIAL-M is
    protecting INT .
    --- Operador _! agregado atributo memo
    op _! : Int -> Int [memo] .
    var N : Int .
    eq [eq1] : 0 ! = 1 .
    eq [eq2] : N ! = (N - 1)! * N [owise] .
endfm
```

6.4.1.2. Sucesión de fibonacci

En el Ejemplo 7 de la Subsubsección 5.9.4.2 se ha analizado la sucesión de fibonacci escrita en el lenguaje Maude (ejemplo a mostrar a continuación). Al agregar el atributo memo a la función *fibo,* logramos que la evaluación de esta función realice a un menor número de reescrituras con respecto a cuando no tiene dicho atributo, cuyo efecto se verá reflejado en el rendimiento de la misma.

```
fmod FIBONACCI is
    protecting NAT .
    op fibo : Nat -> Nat .
```

```
    var N : Nat .
    eq fibo(0) = 0 .
    eq fibo(1) = 1 .
    eq fibo(s s N) = fibo(N) + fibo(s N) .
endfm
```

Al aplicar la refactorización al módulo `FIBONACCI` obtendríamos como resultado el módulo `FIBONACCI-M`, que mostramos en el Ejemplo 16.

Ejemplo 16: Sucesión de fibonacci refactorizado

```
fmod FIBONACCI-M is
    protecting NAT .
    --- Operador fibo agregado atributo memo
    op fibo : Nat -> Nat [memo] .
    var N : Nat .
    eq fibo(0) = 0 .
    eq fibo(1) = 1 .
    eq fibo(s s N) = fibo(N) + fibo(s N) .
endfm
```

6.4.1.3. Comparación

En la Sección 6.6 presentamos una comparación más detallada entre la memorización y la recursión de cola. Estas dos técnicas optimizan la computación de las funciones recursivas e iterativas. Sin embargo, podemos notar que entre estos dos módulos existe ejecución recursiva e iterativa (módulos `FACTORIAL` y `FIBONACCI` respectivamente). Básicamente la diferencia entre una versión y otra (recursiva e iterativa) es que la recursión tiene que trabajar con valores de funciones que aún no devuelven valor y éstos debe ser almacenado en la pila de memoria, lo que no sucede en las variantes iterativas.

6.4.2. Comentarios

Para los lenguajes de evaluación perezosos la forma de implementar la memorización es limitada. Ésta es una ventaja de Maude frente a otros lenguajes funcionales, porque su estrategia de evaluación para el caso de las ecuaciones es impaciente [16].

En la Subsubsección 5.9.4.2 comentamos que Maude cuenta con el operador `memo` que nos permite memorizar resultados anteriores y que simplemente se debe agregar como un atributo en el operador.

Cada vez que se realiza esta refactorización, se agrega un comentario en la línea precedente en el código para que el programador conozca exactamente lo que se ha realizado.

6.4.3. Condiciones de aplicación

- Consideramos únicamente aquellos operadores que tienen estrategia de evaluación por defecto *impaciente* ya que, para otras estrategias, los diferentes términos que tienen la misma forma canónica pueden almacenarse y pueden hacer que la tabla de memorización sea muy grande.

- Hablando en términos de Maude, esta refactorización considera únicamente operadores con evaluación de argumentos en posición estricta. Es decir, operadores que tengan el atributo `strat(1,2,...,n 0)`[11] o, lo que es lo mismo, cuando no se haya especificado la posición de los argumentos[12] (no hacer uso del atributo `strat`). En consecuencia, aquellos operadores que tienen estrategia `strat(0)`,[13] que indican que se comportan de forma perezosa, no son considerados.

- Consideramos únicamente aquellos operadores con tipo de recursión directa; es decir, funciones como $f(a) \rightarrow f(a')$ y no recursiones indirectas como por ejemplo, $f(a) \rightarrow g(b) \rightarrow h(c) \rightarrow \ldots \rightarrow f(a')$ en donde, para llamarse f a sí misma, se requiere llamar previamente a las funciones g y h.

- Funciones recursivas como $f(x) = g(f(x), f(x'), \ldots, f(x''))$, donde f no es lineal, son consideradas un caso que requiere aplicar la memorización (comentado en la Subsubsección 4.4.2.1).

- Asumimos que $f(x)$ tiene aridad n y que toma como argumentos x (x puede ser un conjunto) y consideramos un conjunto de ecuaciones $\triangle_f = e_1, ..., e_n$ que definen la semántica de f en donde se cumple la siguiente condición: existe $e_i \in \triangle_f$ tal que e es de la forma $f(x) = g(f(x))$, donde g es una función de aridad m.

    1. La condición 1 hace que, implícitamente, f no sea de cola recursiva por tener g como una operación más externa[14].

[11] Consideramos una función f de n argumentos, en donde los números diferentes de 0 denotan la posición de los argumentos y el 0 indica la evaluación del nivel más alto de la función.

[12] En su metanivel se representa de igual forma que cuando no se usa el atributo `strat`. Además, una estrategia de evaluación de argumentos que no es de orden estricto no puede definirse junto al atributo commutativo `comm`, ya que Maude lo cambia a orden estricto. Nuestro sistema no soporta ningún otro atributo que no sea `strat` (eg. `assoc`, `comm`, etc.).

[13] Si la estrategia definida para un operador es incorrecta, Maude de forma implícita la evalúa con `strat(0)`.

[14] Condiciones de cola recursiva vistas en la Subsección 4.5.1.

2. Las ejecuciones de la función f se realizan de forma iterativa, sin llamadas recursivas que generen ejecuciones sin valor de retorno en la pila de memoria y que deberían ser tratadas con recursión de cola[15].

6.4.4. Cuestiones de diseño

La transformación de refactorización estudiada en este apartado realiza una fase que consiste en detectar aquellos casos en que una función f pueda aplicar memorización ya que es de la forma $f(x) = g(f(x))$ o más generalmente de la forma $f(x) = g(f(x), f(x'), \ldots, f(x''))$ presentada en la Subsección 4.4.3.

1. Consideremos una teoría ecuacional de entrada E.
2. De la signatura Σ de E, coger el conjunto de símbolos de función $\mathcal{D} = f_1, ..., f_n$ y el conjunto de ecuaciones $\triangle = e_1, ..., e_m$ de la forma $l = r \Leftarrow c$, en donde c es un término que representa la condición (si $c = \varnothing$ se trata de una ecuación no condicional), siendo $n \geq 0$ y $1 \leq m$.
3. Para cada símbolo $f_i \in \mathcal{D}$ donde $1 \leq i \leq n$, verificar si su estrategia de evaluación es por defecto (impaciente) o de orden estricto.
   a) Si la verificación para f_i tiene éxito, se continúa al paso 4.
   b) Si la verificación para f_i fracasa, continuamos en el paso 3 la verificación desde f_{i+1} hasta que $i \geq n$.
4. Para la función $f_i \in \mathcal{D}$, verificar si existe $e_i \in \triangle$ tal que $l|_\Lambda = f_i$ y $r|_p$ se ajusta a la forma recursiva $f(x) = g(f(x), f(x'), \ldots, f(x''))$; es decir, existe al menos una función $g \in \mathcal{D}$ que se evalúa en base a alguna llamada recursiva de $f_i \in \mathcal{D}$.
   a) Si la verificación para f_i tiene éxito, se añade el atributo `memo` a la función f_i y se retorna al paso 3 con la verificación de la función f_{i+1} hasta que $i \geq n$.
   b) Si la verificación para f_i fracasa, continuamos la verificación desde f_{i+1} hasta que $i \geq n$.

6.4.5. Refactorización inversa

La refactorización inversa a esta refactorización sería *"eliminar atributos de memorización"* para aquellos operadores que cumplan las condiciones que hemos presentado en esta sección.

[15] Las recursiones de cola son tratadas en la Sección 4.5.

6.5. Transformación a recursión de cola (*tail recursion*)

Tal y como estudiamos en la Sección 4.5, la recursión de cola es una técnica que permite mejorar la recursividad ya que la nueva función agregada, f, nos permite almacenar iterativamente el valor resultante de la llamada recursiva, sin necesidad de que el compilador reserve espacio en memoria en la pila para aquellas funciones que aún no devuelven un valor.

El enfoque de esta refactorización es transformar definiciones recursivas en recursiones de cola computacionalmente equivalentes.

6.5.1. Ejemplo de recursión de cola

Para visualizar la refactorización diseñada en este trabajo, consideremos el Ejemplo 2 del factorial de un número, definido en el lenguaje Maude tal y como se vio en el Ejemplo 14:

```
fmod FACTORIAL is
    protecting INT .
    op _! : Int -> Int .
    var N : Int .
    eq [eq1] : 0 ! = 1 .
    eq [eq2] : N ! = (N - 1)! * N [owise] .
endfm
```

En el módulo `FACTORIAL`, observamos que el operador $\langle \_! \rangle$ no es cola recursiva ya que viola la segunda condición definida en la Subsección 4.5.1; es decir, la llamada recursiva debe ser la última operación a realizarse o, como textualmente dice la segunda condición: *"En cualquier rama que contiene la llamada a f, sólo el if puede aparecer por encima de la llamada a f* [16]*"*.

Según la teoría de [29] en la que nos estamos basando, el módulo refactorizado para transformar la recursividad a cola recursiva sería:

Ejemplo 17: Factorial de un número refactorizado

```
fmod FACTORIAL-TR is
```

[16] No debemos confundir la función `f` de la condición con la función auxiliar agregada de la recursión de cola.

```
        protecting INT .
        op _! : Int -> Int .
        op f : Int Int -> Int .
        vars N Y : Int .
        eq [eq1.1] : N ! = f(N, 1) .
        eq [eq2.1] : f(0, Y) = Y .
        eq [eq3.1] : f(N, Y) = f(N - 1, Y * N) .
endfm
```

Podemos notar que en el módulo `FACTORIAL-TR` se introducen cambios en las ecuaciones, ya que se agrega la función con recursión de cola f y el "parámetro de generalización" de la función original (como se denomina en [29]). En `eq1.1` tenemos la cola recursiva para el operador $\langle \_! \rangle$, en donde su rhs es la llamada a la función `f` con el valor del caso base del módulo original (rhs de `eq1`). Luego se agregan las ecuaciones para la función `f` donde `eq2.1` es el caso base para esta función, que está basada en el caso base del módulo original (valor de `N=0`). Además se incluye el parámetro que representa la cola recursiva `Y`, cuyo valor es el resultado del factorial al terminar las reescrituras. La ecuación `eq3.1` reemplaza la `eq2` del módulo original con dos parámetros que son la expresión *hacia el caso base*, que en este caso es `N-1`, y la operación adicional que se realiza en esa ecuación, que en este caso es el producto de la cola recursiva con el parámetro recursivo `Y` y `N`, respectivamente.

6.5.2. Comentarios

En la Sección 6.6 se hace un análisis más detallado de cómo utilizar la técnica de recursión de cola junto a la técnica de memorización. Sin embargo, en estos casos puntuales analizados, la técnica de recursión de cola no puede ir de la mano con la memorización[17], ya que los valores computados uno a uno se obtienen con la función auxiliar y no con el operador principal. En tal caso podríamos preguntarnos si *sería conveniente agregar la memorización a la función auxiliar o no.* En este caso la respuesta es negativa, porque de lo contrario no tendría sentido esta refactorización, que trata de evitar el almacenamiento excesivo en la pila de memoria. Por consiguiente, podríamos mantener el atributo `memo` en el operador $\langle \_! \rangle$ como hicimos en el Ejemplo 15 de la refactorización: *"Añadir atributos de memorización (memoization)"*. Sin embargo, este caso no tiene mucho sentido porque este operador únicamente almacenaría los valores resultantes de lo que la función auxiliar `f` compute. Por ejemplo, si buscamos el factorial de 5, el número de reescrituras es 17 y el resultado es 120, que es lo que se almacenaría en

[17] En la Sección 6.6 analizaremos casos en que sí se pueden usar las dos técnicas conjuntamente pero bajo ciertas condiciones.

la pila, pero si ahora computamos el factorial de *6* el número de reescrituras son 20. Es decir, concluimos que no ha mejorado en nada el rendimiento al agregar el atributo memo que únicamente funcionaría si por segunda, tercera o i-ésima vez computamos un mismo valor para un operador. Es decir, si nuevamente computamos el factorial de 5 o 6 (previamente computados) podemos notar que el número de reescrituras es 1.

6.5.3. Condiciones de aplicación

- La aplicación de la transformación a recursión de cola se aplica a problemas recursivos y nunca a iterativos ya que la idea básica es convertir una ejecución recursiva en una iterativa porque es mayor el rendimiento en tiempo de ejecución, consume menos memoria y el procesamiento es más rápido al implementar un proceso iterativo.

- Añadir una variable cuando la fórmula consiste en un producto, o multiplicar por el nuevo parámetro cuando la fórmula incluye una suma, conduce a la recursión anidada. Esto implica que no sea tan trivial la aplicación de recursión de cola en estos casos. Por ejemplo, si tuviéramos el caso de $x!/y$, $y + x!$, $(x!)^y$, etc., necesitaríamos varias generalizaciones para llegar a un algoritmo correcto. Por este motivo, este caso no es considerado en nuestro catálogo de refactorizaciones.

- Sólo se considera las recursiones lineales; es decir, debe existir sólo una llamada recursiva a una función. Por ejemplo, la definición $f(x) = g(f(x))$ representa un tipo de recursión lineal ya que la función f es llamada sólo una vez recursivamente. Por el contrario, no se consideran recursiones del tipo $f(x) = g(f(x), f(x'), \ldots, f(x''))$ en donde la linealidad no se cumple, porque f tiene más de una llamada a sí misma.

6.5.4. Cuestiones de diseño

La metodología para diseñar una función con recursión de cola f, que es la optimización de alguna función recursiva genérica g, se puede describir con el siguiente algoritmo que está basado en el algoritmo de Rubio-Sánchez en [29], aunque con variantes adaptadas a las peculiaridades del lenguaje Maude:

1. Consideramos una teoría ecuacional de entrada E.

2. De la signatura Σ de E, coger el conjunto de símbolos de función $\mathcal{D} = g_1, ..., g_n$ y el conjunto de ecuaciones $\triangle = e_1, ..., e_m$ de la forma $l = r \Leftarrow c$, en donde c

es un término que representa la condición (si $c = \varnothing$ se trata de una ecuación no condicional), siendo $n \geq 0$ y $1 \leq m$.

3. Dada la función $g_i \in \mathcal{D}$ de sort $s \in S$ y con argumentos $x = x_1 : s_1, \cdots, x_m : s_m$ donde $s_1, \cdots, s_m \in S$; $x_1, \cdots, x_m \in \mathcal{C}$, verificar que g es recursiva lineal y que no es de cola recursiva. Es decir, verificar si existe $e_i \in \triangle$ tal que $l|_\Lambda = g_i$ y $r|_p$ se ajusta a la forma recursiva $g(x) = h_j(h_{j+1} \cdots (h_k(g(x))))$, donde $g(x)$ tiene una única llamada recursiva, $h_i \in \mathcal{D}$ representa la(s) función(es) que se aplica(n) después de $g(x)$, lo que hace que la función g no sea de cola recursiva y $j \geq 1 \leq k$.

   a) Si la verificación para g_i tiene éxito, se continúa con el paso 4.

   b) Si la verificación para g_i fracasa, retornamos al paso 3 para proceder con la verificación de g_{i+1} hasta que $i \geq n$.

4. Definimos el nuevo parámetro $y : s \in V$, como una nueva variable del mismo sort que g_i.

5. Creamos la función $f(x, y)$ en donde se incorpora el nuevo parámetro y.

6. Determinamos los casos base de g y establecemos los casos base para f_i.

7. Seleccionamos una simplificación $\ddot{x}$ de x hacia el caso base, que será un término t cada vez más pequeño en cada llamada recursiva, hasta llegar a ser el valor del caso base.

8. Formamos la ecuación $f(x, y) = f(\ddot{x}, z)$ donde z reemplaza el elemento y, y es desconocido.

9. Resolvemos el valor de z, a partir de las operaciones que se están aplicando a la llamada recursiva de g en su rhs.

10. Definimos la función f del caso base y la recursividad de todo el conjunto x.

11. Establecemos el parámetro de $\ddot{y}$ para $g(x) = f(x, \ddot{y})$. Para determinar el valor de $\ddot{y}$, del conjunto x no se considera el caso base, sino la rhs de la ecuación de la que se ha determinado éste (el caso base).

6.5.5. Refactorización inversa

La refactorización inversa a esta refactorización no tiene mucho sentido por las ventajas que brinda la recursión de cola, inclusive para casos pequeños. Por ello no consideramos conveniente plantear un hipotético caso de refactorización inversa.

6.6. Memorización y Recursión de cola

La memorización, como ya hemos visto, es una técnica que se puede aplicar a sistemas recursivos para acelerar procesos de computación. Por su parte, la recursión de cola es otra técnica que necesariamente se implementa en sistemas recursivos y que, al igual que la memorización, hacen que se acelere el proceso de computación. Por lo importante y poderosas que son estas dos técnicas, se tiene esta sección justamente para hacer una comparación entre ellas ya que, si bien no son iguales, tal vez podrían unificarse o de alguna manera la una compensar a la otra.

En esta tesis nos hemos basado en la recursividad para las dos técnicas y las hemos analizado y aplicado a los ejemplos por separado. En lo que sigue analizaremos los siguientes casos:

1. Casos en donde se puede aplicar *cualquiera de las dos técnicas*; sin embargo, sólo una de ellas produce los mejores resultados.
2. Casos en donde es posible aplicar *sólo una de las técnicas*.
3. Casos en donde sería conveniente de alguna manera utilizar *las dos técnicas conjuntamente*, ya que obtendríamos mejores resultados.

Todos los ejemplos están escritos en el lenguaje Maude.

6.6.1. Caso 1

En esta sección analizaremos los casos en los que es posible aplicar cualquiera de las dos técnicas; es decir, memorización o recursión de cola. Así mismo, comparamos los resultados y presentamos nuestras conclusiones.

Ejemplo 18: Recursion de cola de sucesión de fibonacci

Consideremos el Ejemplo 7 de sucesión de fibonacci definida con recursión de cola.

```
fmod FIBONACCI-TR is
    pr NAT .
    op fibo : Nat -> Nat .
    op f : Nat Nat Nat -> Nat .
    vars N Y Z : Nat .
    eq [eq1] fibo(N) = f(N, 0, s 0) .
    eq [eq2] f(0, Y, Z) = Y .
```

```
    ceq [eq3] f(N, Y, Z) = f(sd(N, 1), Z, Y + Z)
                                if N > 0 .
endfm
```

En la Subsubsección 4.4.2.1 hemos comentado el uso de la memorización para la sucesión de fibonacci (Ejemplo 7). En concreto, el árbol generado tras la ejecución de la expresión `fibo(10)` es el mostrado en la Figura 4.3. Podemos notar que se generan varios nodos y éstos a su vez generan ramas (subárboles) con iguales computaciones. Esto justifica el uso de la memorización y sus ventajas.

La ejecución del ejemplo de la sucesión de fibonacci para la misma expresión `fibo(10)` se recoge en la Tabla 6.1, en donde los valores computados[18] para la función `fibo` siempre son evaluados conforme la función avanza (cuando es invocada); es decir, no resulta necesario ir hasta el nivel más bajo para poder devolver el resultado para un determinado valor. Por ejemplo, en la versión original, la evaluación de esta expresión es dada por la sumatoria de las llamadas a `fibo(9)` y `fibo(8)`. A su vez, `fibo(9)` necesitaba llamar a `fibo(8)` y `fibo(7)` y así sucesivamente hasta llegar a su último nivel para poder devolver el valor. Esto implica la generación de un grafo DAG, lo que no sucede con la recursión de cola que propicia que las evaluaciones sean mucho más rápidas.

Valor de `N`	***Función Caller***	***Función Called*** `f(N,Y,Z)`	***Ecuación usada***	***Valor de*** `Y`
	`fibo` (10)	`f` (10, 0, 1)	`eq1`	0
10	`f` (10, 0, 1)	`f` (9, 1, 1)	`eq2`	1
9	`f` (9, 1, 1)	`f` (8, 1, 2)	`eq2`	1
8	`f` (8, 1, 2)	`f` (7, 2, 3)	`eq2`	2
7	`f` (7, 2, 3)	`f` (6, 3, 5)	`eq2`	3
6	`f` (6, 3, 5)	`f` (5, 5, 8)	`eq2`	5
5	`f` (5, 5, 8)	`f` (4, 8, 13)	`eq2`	8
4	`f` (4, 8, 13)	`f` (3, 13, 21)	`eq2`	13
3	`f` (3, 13, 21)	`f` (2, 21, 34)	`eq2`	21
2	`f` (2, 21, 34)	`f` (1, 34, 55)	`eq2`	34
1	`f` (1, 34, 55)	`f` (0, 55, 89)	`eq2`	55
0	`f` (0, 55, 89)	-	`eq3`	55

TABLA 6.1: Trazas ejecución recursiva

En la Tabla 6.2 mostramos una comparación basada en el número de reescrituras que se generan al ejecutar el programa de fibonacci, sin memorización, con memorización y con recursión de cola, para diferentes valores de `N`.

[18] Por expresividad se van a representar los números en notación convencional y no en notación de Peano.

Iteración	`fibo(N)`	***Número de reescrituras***		
		Original	***Memorización***	***Recursión de Cola***
1	`fibo(10)`	265	28	42
2	`fibo(8)`	100	1	34
3	`fibo(9)`	163	1	38
4	`fibo(15)`	2959	16	62
5	`fibo(100000)`	$Stack\,over flow$	$Stack\,over flow$	400002

Tabla 6.2: Comparación de la ejecución de fibonacci en Maude

Podemos concluir que el coste va a depender del tamaño de los datos que serán evaluados para las diferentes funciones ya que observamos que los valores más bajos a nivel de reescrituras se dan en la función cuando implementa memorización; sin embargo, cuando es un valor grande, por ejemplo el caso de la 5ª iteración, podemos ver que la memorización también genera un desbordamiento de la pila de memoria. Eso no resta todas las ventajas que conlleva el uso de la memorización, sobre todo en lenguajes algebraicos como Maude.

6.6.2. Caso 2

En esta sección analizaremos los casos en los que se puede aplicar sólo una de las dos técnicas; es decir, únicamente memorización y no recursión de cola o viceversa.

Ejemplo 19: Caminos reticulares

Consideremos el ejemplo de caminos reticulares *"lattice paths"*, que consiste en contar el número de rutas que existen en una matriz cuadrada $N \times N$, desde una posición $P\,(X, Y)$ donde X e Y corresponden a las coordenadas de fila y columna, respectivamente, hasta la última posición de la matriz, es decir a la posición $P'\,(N, N)$.

```
fmod LATTICE-PATH is
    pr NAT .
    op contar : Nat Nat Nat -> Nat .
    var N X Y : Nat .
    eq [eq1] : contar(X, Y, N) = 1 [owise] .
    ceq [ceq1] : contar(X, Y, N) =
                 contar(s(X), Y, N) + contar(X, s(Y), N)
                 if X < N /\ Y < N .
endfm
```

La Figura 6.1 ilustra en qué consiste el Ejemplo 19, considerando una matriz con $N = 2$ y la posición $P\,(0, 0)$, en donde el número de rutas desde el punto P al punto P' es exáctamente 6.

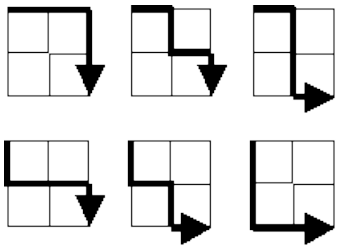


FIGURA 6.1: Caminos reticulares para matriz 2×2

La computación del programa del Ejemplo 19 para la misma matriz 2×2 representada en forma de un grafo DAG, se muestra en la Figura 6.2. Los óvalos representan los nodos generados para cada llamada recursiva con los diferentes valores de los argumentos. Los círculos representan los nodos sumideros o de último nivel, que son los valores computados. Las flechas unen los nodos que son llamados recursivamente. El grafo siempre termina en nodos sumideros. En la figura notamos los siguientes aspectos.

1. Los nodos 1 y 2 generan computaciones iguales en sus ramas derecha e izquierda, respectivamente. Si miramos la figura, cuando la función analizada toma los valores en sus argumentos `contar(1,1,2)`, se crean los nodos 4 y 5. Estos dos nodos son iguales y todos los nodos que se generen bajo éstos serán los mismos. Podemos mirar las ramas a y b, que tienen por raíz los nodos 4 y 5, respectivamente. Estas ramas representan computaciones iguales, que deberían de alguna manera ser calculadas una única vez (ya sea a o b, indistintamente) y, a la siguiente llamada a la función *contar* con dichos argumentos, reutilizarlos y devolver su resultado.

2. La guarda $\langle \texttt{X} < \texttt{N} / \backslash \texttt{Y} < \texttt{N} \rangle$ hace que se ejecuten las llamadas recursivas. Si nos fijamos en la guarda y en los valores para cada variable, notamos que la conjunción permitiría un ajuste parcial al aplicar memorización, como comentamos en Subsubsección 4.4.1.2, ya que es suficiente que una de las dos variables se evalúe a false para que la recursividad ya no se ejecute. Observando los nodos 3, 6, 7, 8, 9 y 10 vemos que esto se cumple, al ser suficiente que la variable `X` o `Y` tomen el valor de 2 para que la reescritura ya no se ajuste a la ecuación `ceq1` sino a la ecuación `eq1`.

La ejecución en Maude da los siguientes resultados:

Sin memorización el número de reescrituras para este ejemplo con los mismos valores es de 35 en las i-ésimas ejecuciones. Si ahora consideramos un valor de $N = 3$, el número incrementa a 126.

Con memorización el número de reescrituras para este ejemplo en la primera vez es de 27 y en las i-ésimas ejecuciones siguientes para los mismos valores basta con 1 reescritura. Ahora si consideramos el valor de $N = 3$, el número de reescrituras es de 55, que corresponde a un 56.35 % más eficiente.

Ahora queda la pregunta de si *es conveniente aplicar recursión de cola a este ejemplo*. La respuesta sería muy similar a la que expusimos en los comentarios en la Subsección 6.5.2. Es decir, no sería posible implementar la recursión de cola porque este ejemplo en particular es iterativo y la recursión de cola es aplicable a la recursión.

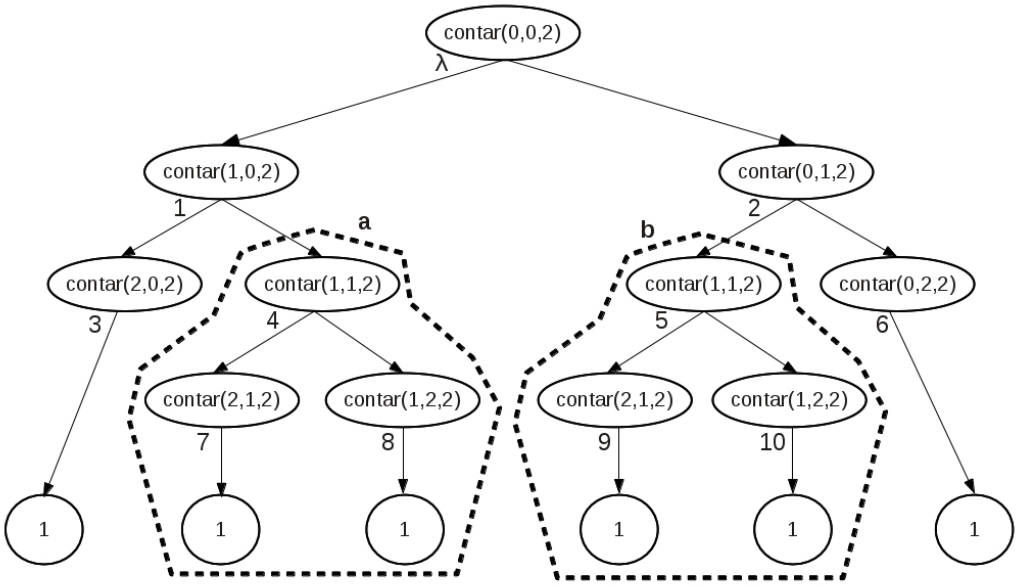


FIGURA 6.2: DAG de ejecución para matriz 2×2

Ejemplo 20: Potencia de un número

Consideremos el ejemplo de la potencia de un número. Es decir, un número base B elevado a un exponente E, así B^E:

```
fmod POWER is
    pr INT .
    op pow : Int Int -> Int .
    var B E : Int .
    eq pow(B, 0) = 1 .
    eq pow(B, E) = B {*} pow(B, E - 1) .
endfm
```

Éste es un ejemplo relativamente similar al Ejemplo 14 del factorial de un número, ya que la manera de ejecutarse es mediante recursión. Por ejemplo, una ejecución del

ejemplo de la potencia de un número (Ejemplo 20) con valores $B = 2$ y $E = 5$ sería: $pow(B, E) = B^E; B = 2, E = 5 \Rightarrow 2^5 = 32$. Sin embargo, la diferencia radica en que la base B va a ser siempre la misma y depende exclusivamente del valor que ésta almacene para su evaluación. En cambio, en el factorial de un número la computación de un valor N[19] siempre implica utilizar los resultados de sus números predecesores en las siguientes llamadas recursivas con el argumento `N-1` (como se puede ver en la Tabla 4.1). Es por ello que, para el factorial de un número, el uso de la técnica de memorización es ideal y mejora el rendimiento de las computaciones (el análisis en el Caso 3). Entonces, *¿conviene el uso de la técnica de memorización para el Ejemplo 20?;* la respuesta sería un negativa porque los valores computados que se almacenen en la tabla de memorización dependen directamente del valor de B, para las evaluaciones posteriores a la primera. Es decir, si no es el mismo valor, poco o nada ayuda tener estos valores en la tabla. Sin embargo, sí va a depender de los valores que puede recibir la función, ya que si siempre se trabaja con valores repetidos puede ser conveniente en cierto modo el uso de la memorización.

La única técnica que en este caso es ideal es la recursión de cola, ya que va a permitir computaciones de valores grandes, evitando así los desbordamientos de pila de memoria. El Ejemplo 20 escrito en formato de cola recursiva es el siguiente:

```
fmod POWER-TR is
    pr INT .
    op pow : Int Int -> Int .
    op f : Int Int Int -> Int .
    var B E Y : Int .
    eq pow(B, Y) = f(B, Y, 1) .
    eq f(B, 0, Y) = Y .
    eq f(B, E, Y) = f(B, E - 1, Y {*} B) .
endfm
```

6.6.3. Caso 3

En esta sección analizaremos los casos en los que conviene aplicar las dos técnicas simultáneamente; es decir, memorización y recursión de cola, ya que los cálculos computacionales son más eficientes en comparación cuando sólo se aplica una de las dos técnicas. Para ello vamos a utilizar el Ejemplo 14 del factorial de un número escrito en Maude[20].

[19] La variable N de factorial de un número es equivalente a la variable B del ejemplo de la potencia de un número.

[20] Este caso puede ser analizado con el ejemplo de fibonacci; sin embargo, en este ejemplo es más fácil apreciar tanto la recursión de cola como el uso de la memorización.

```
fmod FACTORIAL is
    protecting INT .
    op _! : Int -> Int .
    var N : Int .
    eq [eq1] : 0 ! = 1 .
    eq [eq2] : N ! = (N - 1)! * N [owise] .
endfm
```

La Tabla 6.3 muestra los valores de las ejecuciones del ejemplo de factorial de un número sin el uso de la memorización, con la memorización y recursión de cola.

Iteración	`N !`	***Reescrituras***		
		Original	***Memorización***	***Recursión de Cola***
1	100!	301	301	302
2	1000!	3001	2701	3002
3	10000!	30001	27001	30002
4	8000!	24001	1	24002
5	100000!	$Stack\ over flow$	$Stack\ over flow$	300002

TABLA 6.3: Comparación de la ejecución de factorial de un número en Maude

De forma similar al *Caso 1* de la sucesión de fibonacci, en valores grandes la recursión de cola es más conveniente, pese a que el número de reescrituras con valores más pequeños es mayor que en un módulo que utiliza la memorización. Por ejemplo, en la 4ª iteración notamos que el número de reescrituras tras la evaluación de la expresión `8000!` en el módulo de memorización es 1 y en el de recursión de cola es de 24002. Esto justamente se debe a que los valores calculados se encuentran en la tabla de la memorización.

En base a este análisis, cabe preguntarnos si sería factible tratar de unificar estas dos técnicas de alguna manera; es decir, a la vez que se aplique una recursión de cola, memorizar los valores bajo ciertos criterios, de tal manera que para futuras computaciones no sólo tengamos la rapidez que nos brinda la memorización, sino también la eficiencia de la recursión de cola. Bajo nuestro punto de vista y basándonos en los resultados obtenidos tendríamos las siguientes ventajas y desventajas:

6.6.3.1. Ventajas

- Tendríamos un conjunto de valores prestos a ser utilizados, gracias a la memorización y la forma en que almacena los datos previamente calculos.

- Los valores más grandes podrían ser computados más rápido con la memorización, ya que si deseamos evaluar expresiones con valores mayores que algunos previamente calculados (si existieran), simplemente se parte de esos valores y se calcula los valores faltantes, se almacena esos nuevos valores calculados y se retorna el resultado (Sección 4.4).

6.6.3.2. Desventajas

- La tabla de valores en la que se basa la memorización podría crecer extremadamente, lo que a largo plazo generaría problemas de desbordamiento de pila.
- Es difícil decidir en qué función aplicar la memorización, ya que sabemos que al utilizar recursión de cola, la función recursiva $g(x)$ utiliza una función auxiliar $f(x, y)$, en donde y es el parámetro adicional que se genera al aplicar la recursión de cola (Sección 4.5). Entonces el único valor que al final obtiene $g(x)$ es el de y, cuyo valor está en base a iteraciones de $f(x, y)$. Por otra parte, si bien $f(x, y)$ trabaja con todos los valores iterativamente, es un sólo resultado a la vez el que el parámetro y almacena. Por lo tanto, la decisión de implementar la memorización en una de las dos funciones, es algo no trivial.

Capítulo 7

El sistema MRS

Este capítulo describe el sistema **MRS** (de sus siglas en inglés "*Maude Refactorer System*"), que implementa toda la selección de refactorizaciones presentadas en el catálogo del Capítulo 6, haciendo uso de las técnicas estudiadas en el Capítulo 4.

La arquitectura del sistema y la descripción de cada uno de los módulos que la componen son descritas en la Sección 7.1. La Sección 7.2 presenta el funcionamiento del sistema MRS al momento de refactorizar una especificación Maude.

Las siguientes refactorizaciones son las que se implementan en el sistema MRS:

- Añadir un atributo constructor;
- Añadir atributos de memorización (*memoization*);
- Recursión de cola (*tail recursion*);
- Transformar ecuaciones condicionales a incondicionales (*unravelling*).

7.1. Arquitectura del sistema MRS

El sistema MRS está escrito en Maude y posee alrededor de 100 operadores de función (aproximadamente 1K líneas de código), que corresponde a la aplicación del sistema. Puede ser usado desde la consola de Maude (haciendo uso de Core Maude) o mediante el uso de una interfaz web que se encuentra publicada en *http://safe-tools.dsic.upv.es/mrs/*.

La Figura 7.1 muestra la arquitectura que soporta MRS y que consiste en dos módulos principales llamados **Analizador** y **Transformador**, y un submódulo llamado **Catálogo**.

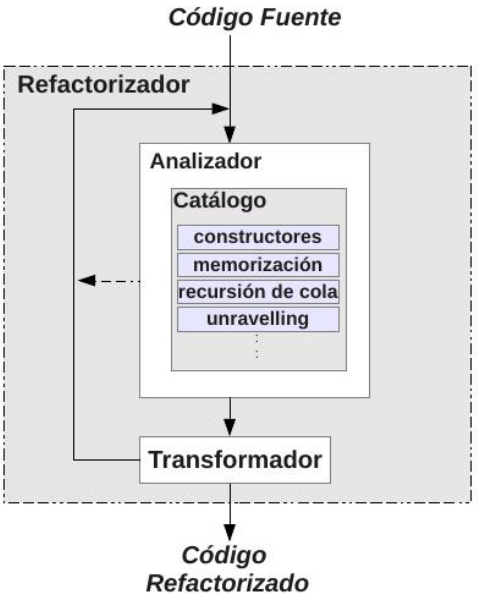


FIGURA 7.1: Arquitectura del sistema MRS

El módulo **Analizador** es aquel que recibe una especificación Maude (concretamente un módulo funcional), lo lleva a una especificación de metanivel y posteriormente solicita a su submódulo *Catálogo* las respectivas operaciones que debería aplicarse en el código de acuerdo con el análisis realizado. Es recursivo, ya que por cada una de las refactorizaciones que existen en el catálogo, debe invocarse un número finito de veces e interactuar con los módulos siguientes. La función del submódulo **Catálogo** es analizar si es posible aplicar alguna refactorización en la especificación dada, de ser así, devuelve la función respectiva que el módulo Transformador debe aplicar, de lo contrario retorna nulo. Cada una de las refactorizaciones que el catálogo ofrece se encuentran bajo un orden de aplicación y el catálogo sirve para cuando el usuario no especifica la refactorización que desea aplicar sobre una especificación. El orden de refactorizaciones del catálogo es la siguiente:

1. Añadir un atributo constructor;
2. Añadir atributos de memorización (*memoization*);
3. Recursión de cola (*tail recursion*);
4. Transformar ecuaciones condicionales a incondicionales (*unravelling*).

Mantener un orden de refactorizaciones para aplicar a una definición que se quiere refactorizar influye directamente sobre el código resultante. En el Capítulo 8 mostramos el efecto que produce el orden del catálogo.

Por su parte, el módulo **Transformador** es aquel que al recibir una función, debe aplicarla a la especificación Maude que se está refactorizando y posteriormente retorna al módulo Analizador (si es necesario) o devuelve al usuario la especificación resultante. La especificación resultante puede estar refactorizada si se ha aplicado alguna refactorización, o de lo contrario se obtiene la misma especificación ingresada por el usuario.

7.2. MRS en funcionamiento

Para describir el funcionamiento de MRS, vamos a refactorizar el módulo de factorial de un número, que es una de las definiciones por defecto que el sistema nos brinda. Tenemos dos alternativas de refactorización, la primera alternativa es seleccionar la refactorización que queremos aplicar y la segunda es aplicar todas las refactorizaciones posibles sobre una especificación Maude dada (un módulo funcional).

Haremos uso de MRS Online, cuya interfaz corresponde a la Figura 7.2. Como podemos apreciar en la figura, la interfaz tiene dos listas de selecciones desplegables. La primera corresponde al catálogo de refactorizaciones disponible que podemos seleccionar y la segunda nos brinda la opción de cargar algunas definiciones por defecto con propósito demostrativo. Posteriormente tenemos dos áreas de texto. La del lado izquierdo es editable y es en donde el usuario debería escribir o cargar la especificación a refactorizar y la del lado derecho, en cambio es en donde se visualiza (automáticamente) la especificación refactorizada (si ha sido posible refactorizar) tras haber presionado el botón "Refactorizar" que se encuentra ubicado debajo de estas dos áreas de texto (parte inferior central).

Refactorizando

Para refactorizar una especificación debemos realizar ordenadamente las siguientes tareas:

Seleccionar la refactorización a aplicar

De la primera lista de selección desplegable (catálogo de refactorización) se debe seleccionar la refactorización que se desea aplicar en la especificación (posteriormente ingresada). Dentro de esta lista se encuentra la opción "Todas", que permite aplicar todas las refactorizaciones posibles (que el sistema detecte) en la especificación. Por defecto se encuentra seleccionada la opción "– Seleccione –", esto implica que ninguna opción de

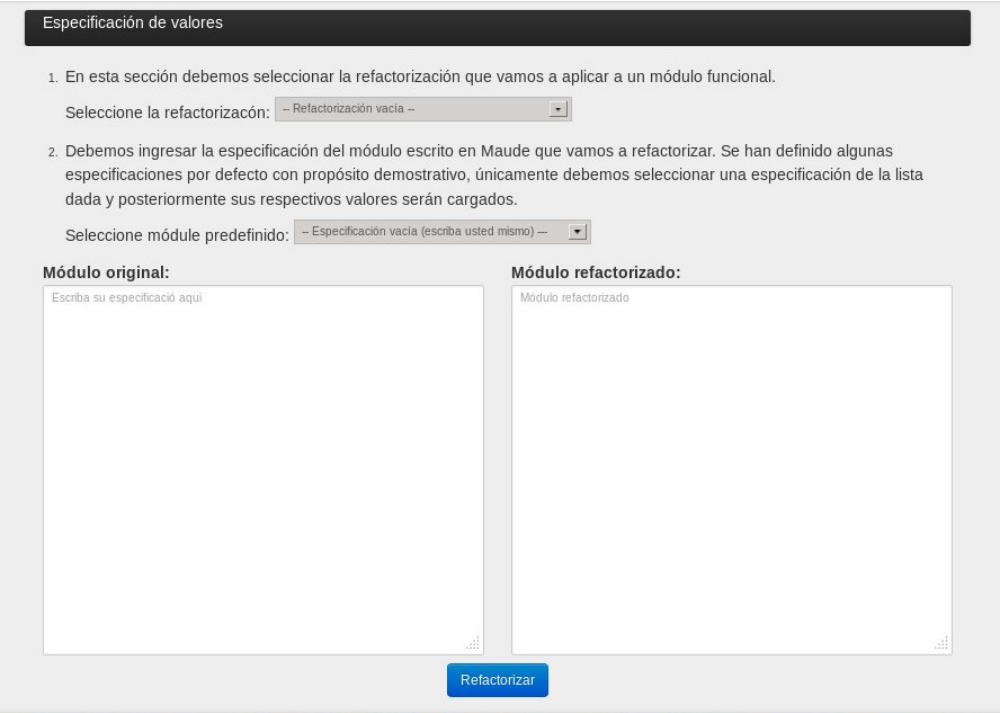


FIGURA 7.2: MRS Online

refactorización está seleccionada. En nuestro caso seleccionaremos "Recursión de Cola" (Figura 7.3).

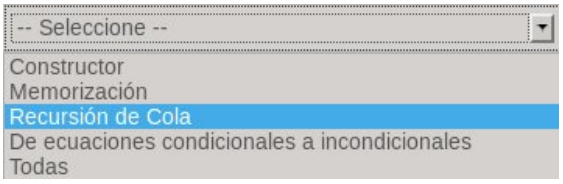
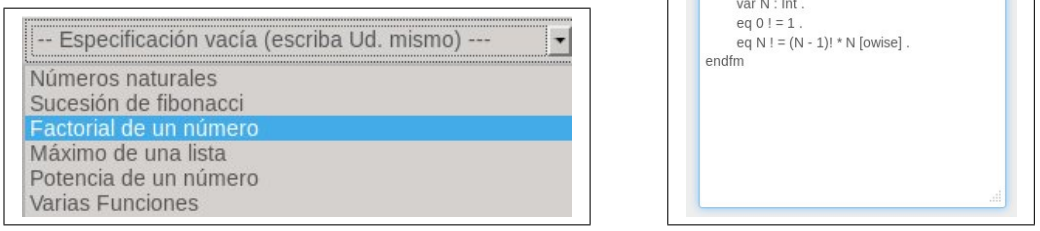


FIGURA 7.3: Selección de refactorización

Introducir la especificación

El usuario debe introducir la especificación Maude (módulo funcional) que desea refactorizar en el área de texto de la izquierda (Figura 7.4(a)). También existe la opción de agregar una especificación por defecto de la segunda lista de selección desplegable. Estas definiciones por defecto, cuyo objetivo es demostrativo, una vez que han sido cargadas en el área de texto (de la izquierda) su código no puede ser modificado. En la lista desplegable, por defecto se encuentra seleccionada la opción "Especificación vacía (escriba Ud. mismo)" lo cual implica que se debe escribir o pegar la especificación. Como hemos dicho, para este caso utilizaremos la especificación por defecto del "Factorial de un número" (Figura 7.4(a)).



(a) Lista por defecto (b) Especificación

FIGURA 7.4: Definición de especificación

Ejecutar la refactorización

Cuando ya los datos anteriores han sido ingresados correctamente, se procede a ejecutar la refactorización pulsando botón "Refactorizar", cuya función es enviar los parámetros introducidos al módulo **Refactorizador**. Si todo ha sido correcto, el resultado de la refactorización se podrá apreciar en el área de texto de la derecha (Figura 7.5), en donde notamos el nuevo operador definido [`op !-tr : Int Int -> Int .`], junto con las ecuaciones que definen la semántica para este operador y que son [`eq !-tr(0, Y) = Y .`] y [`eq !-tr(N, Y) = !-tr(N - 1, Y * N) .`]. Además, las ecuaciones definidas para el operador `_!` han sido reducidas a una única ecuación [`eq N ! = !-tr(N, 1) .`] o, dicho de otra manera, dichas ecuaciones han sido modificadas para dar lugar a las ecuaciones del operador `!-tr` (Sección 6.5).

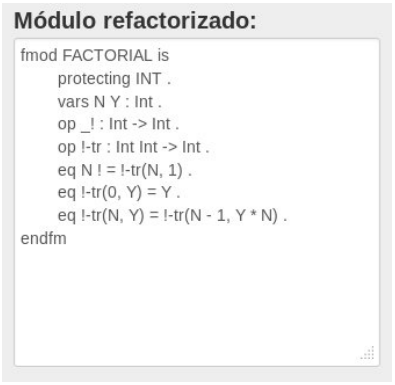


FIGURA 7.5: Especificación refactorizada

El nuevo módulo refactorizado es equivalente al módulo original en su semántica, aunque difiera en su sintaxis y puede ser usado por el usuario para reemplazar al módulo original. Las ventajas del nuevo módulo frente al original son descritas en nuestro caso de estudio.

Capítulo 8

Caso de Estudio

El objetivo de este capítulo es presentar un caso de estudio de refactorización más concreto; la aplicación de todas las refactorizaciones en una misma especificación de Maude (*módulo funcional*).

En este caso de estudio se ha seleccionado y agrupado varias de las funciones que hemos estudiado a lo largo de este trabajo, con el objetivo de aplicar todas las refactorizaciones que el sistema MRS ofrece en su catálogo.

8.1. Especificación a refactorizar

La especificación a usar es el módulo `FUNCIONES` del Ejemplo 21. Las funciones definidas son las siguientes:

- Factorial de un número;
- Sucesión de fibonacci;
- Potencia de un número y
- Valor máximo de una lista de números naturales.

Ejemplo 21: Módulo `FUNCIONES`

```
fmod FUNCIONES is
    protecting INT .
    sort NatList .
    subsort Nat < NatList .
```

```
    vars N B E : Int .
    var NL : NatList .
    ---Factorial de un número
    op _! : Int -> Int .
    eq 0 ! = 1 .
    eq N ! = (N - 1)! * N [owise] .
    ---Sucesión de fibonacci
    op fibo : Int -> Int .
    eq fibo(0) = 0 .
    eq fibo(1) = 1 .
    eq fibo(N) = fibo(N - 2) + fibo(N - 1) .
    ---Potencia de un número
    op pow : Int Int -> Int .
    eq pow(B, 0) = 1 .
    eq pow(B, E) = B * pow(B, E - 1) .
    ---Valor máximo de una lista de números naturales
    op nil : -> NatList .
    op _:_ : NatList NatList -> NatList [assoc id: nil] .
    op max : NatList -> Nat .
    eq max(nil) = 0 .
    eq max(N) = N .
    ceq max(N : NL) = N if N > max(NL) .
    ceq max(N : NL) = max(NL) if N <= max(NL) .
endfm
```

8.2. Secuencia de refactorización

La secuencia de refactorización que el sistema MRS aplicará en el módulo `FUNCIONES` se basa en el orden de las refactorizaciones que tiene el catálogo (Sección 7.1). Por tanto, la secuencia a seguir es la siguiente:

1. Agregar el atributo constructor en los operadores que cumplen con la teoría en la que se basa el sistema MRS (Sección 6.1). En este caso el único operador a modificarse es `nil`, cuyo resultado es el siguiente:

```
op nil : -> NatList [ctor] .
```

2. La siguiente refactorización que el sistema MRS busca aplicar es la memorización. En este caso los operadres que van a ser modificados para que hagan uso de la memorización son `_!`, `fibo` y `pow`. El resultado es el siguiente:

```
op _! : Int -> Int [memo] .
op pow : Int Int -> Int [memo] .
op fibo : Int -> Int [memo] .
```

3. La recursión de cola es la siguiente refactorización que el sistema MRS tratará de aplicar. Para este caso, los operadores a ser modificados son `_!` y `pow`, que corresponden a dos de los tres operadores que han sido modificados por la refactorización anterior (memorización). El resultado tras aplicar la refactorización de recursión de cola es el siguiente:

```
op _! : Int -> Int [memo] .
op !-tr : Int Int -> Int .
op pow : Int Int -> Int [memo] .
op pow-tr : Int Int Int -> Int .
```

   Apreciamos que la sintaxis de los operadores `_!` y `pow` no ha sido modificada. Lo que se ha hecho es agregar los nuevos operadores de la recursión de cola siguiendo la teoría (Sección 4.5). Es en este punto en donde se refleja el efecto del orden que mantiene el sistema MRS al aplicar las refactorizaciones, ya que al haber aplicado en primer lugar la memorización y después la recursión de cola, los tres operadores mantienen el atributo `memo`. Esto se debe a que las refactorizaciones del catálogo en ningún caso elimina atributos que tienen los operadores en su definición original. El efecto que produce tener la memorización y recursión de cola juntas, se la detalla en el "Caso 3" de la Subsección 6.6.3.

   Las ecuaciones que definen la semántica de los nuevos operadores (`pow-tr` y `_!`), son también creadas e insertadas. Además el sistema modifica las ecuaciones de los operadores `_!` y `pow` para que se adapten a la recursión de cola. El resultado es el siguiente:

```
eq N ! = !-tr(N, 1) .
eq !-tr(0, Y) = Y .
eq !-tr(N, Y) = !-tr(N - 1, Y * N) .
eq pow(B, E) = pow-tr(B, E, 1) .
eq pow-tr(B, 0, Y) = Y .
eq pow-tr(B, E, Y) = pow-tr(B, E - 1, Y * B) .
```

4. La última refactorización por aplicar es transformar las ecuaciones condicionales en incondicionales. El operador `max` es el único operador al que se puede aplicar esta refactorización, ya que tiene dos ecuaciones condicionales complementarias entre sí y pueden ser reemplazadas por una única ecuación incondicional que mantiene su semántica (Subsección 4.3.1). El resultado de dicha refactorización es la siguiente:

```
eq max(nil) = 0 .
eq max(N) = N .
eq [UR1] : max(N : NL) = if N > max(NL) then N else max(NL) fi .
```

   Podemos notar que la aplicación de esta refactorización ha afectado únicamente a las ecuaciones condicionales que cumplen con la condición de ser complementarias, es decir, queda al margen de cualquier modificación el resto de ecuaciones para el mismo operador. Algo adicional es que todas las ecuaciones incondicionales nuevas llevan la etiqueta `UR1`, que se refiere a que es el resultado de una transformación del "Caso 1" de *unravelling*.

5. Los operadores y sus respectivas ecuaciones (si las tienen) que no han podido ser refactorizados son devueltos en su forma original. Este es el caso del operador `_:_`.

6. La última tarea que realiza el sistema MRS es devolver al usuario la versión refactorizada de la definición. Para el módulo `FUNCIONES` (que es nuestro caso de estudio) es la siguiente:

```
fmod FUNCIONES-REF is
    protecting INT .
    sort NatList .
    subsort Nat < NatList .
    vars N B E Y : Int .
    var NL : NatList .
    op _! : Int -> Int [memo] .
    op !-tr : Int Int -> Int .
    op pow : Int Int -> Int [memo] .
    op pow-tr : Int Int Int -> Int .
    op nil : -> NatList .
    op _:_ : NatList NatList -> NatList [assoc id: nil] .
    op max : NatList -> Nat .
    op fibo : Int -> Int [memo] .
    eq N ! = !-tr(N, 1) .
    eq !-tr(0, Y) = Y .
```

```
        eq !-tr(N, Y) = !-tr(N - 1, Y * N) .
        eq fibo(0) = 0 .
        eq fibo(1) = 1 .
        eq fibo(N) = fibo(N - 2) + fibo(N - 1) .
        eq max(nil) = 0 .
        eq max(N) = N .
        eq [UR1] : max(N : NL) = if N > max(NL) then N else max(NL) fi .
        eq pow(B, E) = pow-tr(B, E, 1) .
        eq pow-tr(B, 0, Y) = Y .
        eq pow-tr(B, E, Y) = pow-tr(B, E - 1, Y * B) .
endfm
```

El nombre del módulo refactorizado se mantiene igual al original, es decir `FUNCIONES`. Lo hemos cambiado a `FUNCIONES-REF` para diferenciar el módulo original del refactorizado en las diferentes reescrituras que detallaremos en la sección siguiente.

8.3. Comparación de eficiencia

Cambiar la estructura del código y mantener su semántica lo hemos conseguido al momento, resta entonces demostrar que el módulo refactorizado es más eficiente que el módulo original (`FUNCIONES-REF` y `FUNCIONES`, respectivamente). En secciones anteriores, hemos colocado de manera breve y para ciertas refactorizaciones, las ganancias en eficiencia obtenidas tras aplicar una refactorización en concreto sobre alguna definición Maude. Por ejemplo, la Tabla 6.2 y la Tabla 6.3 visualizan las ganancias obtenidas (medidas en número de reescrituras) tras haber aplicado la refactorización para la funciones de sucesión de fibonacci y factorial de un número, respectivamente. Sin embargo, estos valores pueden variar en el módulo `FUNCIONES-REF` porque en el módulo existen varias funciones definidas y no solo una, como en el caso anterior cuando se tabularon los valores de tablas antes mencionadas[1].

Las pruebas y computaciones de rendimiento que vamos a describir a continuación han sido realizadas en **Core Maude 2.6** bajo **Eclipse**, instalado en el sistema operativo **Linux openSUSE 12.2**, hardware **Intel Core I7** con **6GB** de memoria RAM.

La Tabla 8.1 muestra el número de reescrituras y porcentajes ganados para las diferentes reducciones realizadas en el módulo refactorizado y original sobre la función `fibo` que

[1] La eficiencia ganada la hemos medido en número de reescrituras y no en tiempo, ya que este último varía de acuerdo al procesador en el que se ejecute, mientras que las reescrituras serán las mismas en cualquier procesador. Además, mientras mayor número de reescrituras tenga que hacer el sistema mayor será el tiempo que tarde.

corresponde a la sucesión de fibonacci. Podemos observar que para las iteraciones 2 y 3 en la especificación refactorizada el número de reescrituras es 1; esto se debe a que los valores para las llamadas `fibo(8)` y `fibo(9)` han sido computadas en la llamada `fibo(10)`, y se han almacenados en la tabla de memorización que Maude ha creado para la función `fibo` tras la refactorización.

Iteración	`fibo(N)`	***Número de reescrituras***		***Mejora***
		`FUNCIONES`	`FUNCIONES-REF`	
1	`fibo` (10)	441	46	89,57 %
2	`fibo` (8)	166	1	99,38 %
3	`fibo` (9)	271	1	99,63 %
4	`fibo` (15)	4931	26	99,47 %
5	`fibo` (1000)	*Stack overflow*	4926	>100 %

TABLA 8.1: Comparación de la ejecución de fibonacci entre el módulo original y refactorizado

En la Tabla 8.2 apreciamos el número de reescrituras realizadas para las diferentes iteraciones efectuadas para la función del factorial de un número (`_!`). Si bien, el número de reescrituras no se reduce entre las ejecuciones del módulo original y el refactorizado como sucede con la memorización, la diferencia se aprecia al momento de ejecutar expresiones de valores grandes. Por ejemplo, en la 5ª iteración la ejecución de `100000!` en el módulo original genera un desbordamiento en la pila de memoria (*Stack overflow*), debido a las sucesivas llamadas recursivas. Al contrario del original, el módulo refactorizado sí devuelve un valor resultante; esto se debe a las ventajas que ofrece la recursión de cola, que hemos comentado en la Sección 4.5.

Iteración	`N!`	***Número de reescrituras***		***Mejora***
		`FUNCIONES`	`FUNCIONES-REF`	
1	10!	31	32	
2	100!	301	302	
3	1000!	3001	3002	
4	10000!	30001	30002	
5	100000!	*Stack overflow*	300002	✓

TABLA 8.2: Comparación de la ejecución del factorial de un número entre el módulo original y refactorizado

La siguiente comparación de resultados se realizará para la función `max`, que obtiene el máximo valor numérico de una lista de números naturales y se describe en la Tabla 8.3. La refactorización que el sistema MRS ha aplicado sobre esta función es la de transformar ecuaciones condicionales en incondicionales.

En lineas generales, todas las transformaciones que se han realizado sobre el módulo `FUNCIONES` han sido para bien. En este caso, podemos decir que las operaciones del

Iteración	***Longitud de lista***	***Número de reescrituras***		***Mejora***
		`FUNCIONES`	`FUNCIONES-REF`	
1	5	81	33	59,26 %
2	10	2085	68	96,74 %
3	15	65593	103	99,84 %
4	20	2097229	138	$\sim$ 99,99 %
5	50	*Stack overflow*	348	>100 %

Tabla 8.3: Comparación de la ejecución del valor máximo de una lista de números naturales entre el módulo original y refactorizado

módulo `FUNCIONES-REF` son más eficientes que las del módulo `FUNCIONES`. Aunque no sea éste el único criterio que puede justificar la calidad de una refactorización puesto que la legibilidad o facilidad de mantenimiento del código pueden ser igualmente importantes en otras aplicaciones de las técnicas descritas en esta tesis.

Capítulo 9

Conclusiones y Trabajo Futuro

En este trabajo se presenta el primer sistema de refactorización que se ha desarrollado para el lenguaje Maude, cuya implementación ha sido realizada en el mismo Maude que implementa la lógica de reescritura.

Las ventajas en cuanto a calidad y concretamente eficiencia que ofrece el proceso de refactorización se ven reflejados en los resultados presentados en el Capítulo 8. Las ganancias obtenidas con las refactorizaciones de "'*Añadir atributos de memorización*", "*Transformar ecuaciones condicionales en incondicionales*" y "'*Recursión de cola*" son claramente notables. En las dos primeras, sus ganancias son $\sim 97\,\%$ y $\sim 88\,\%$, respectivamente. En cuanto a la recursión de cola, si bien el número de reescrituras no disminuye, sino que se incrementa en una llamada adicional, esto se debe a que el mismo número de llamadas recursivas simples han sido transformadas en llamadas iterativas y la función original debe llamar a la nueva función de cola recursiva (la llamada adicional). La ganancia obtenida se aprecia al momento de computar valores grandes, debido a que requiere menos espacio en la pila de memoria, obteniéndose resultados concretos que no se podía obtener en la versión no recursiva.

La refactorización "'*Transformar ecuaciones condicionales en incondicionales*", además de la ganancia en eficiencia que se produce cuando las ecuaciones condicionales son complementarias, da lugar a que tengamos una especificación más corta, legible e incluso más mantenible. Sin embargo, no podemos decir lo mismo cuando las ecuaciones no son complementarias, donde ganamos en rendimiento pero perdemos legibilidad. Esto sucede porque la teoría de [28] en la que nos hemos basado extiende la aridad de la función, agregando nuevos operadores y nuevas ecuaciones.

Finalmente, el objetivo de mejorar la legibilidad del código se consigue enteramente con la refactorización "'*Agregar un atributo constructor*", ya que tras su aplicación,

el programador sabrá exáctamente los operadores constructores que han de aparecer siempre en un término en forma normal.

El número de refactorizaciones implementadas en el sistema MRS son cuatro de las seis refactorizaciones que hemos presentado, que hemos elegido con criterios de representatividad, por lo que uno de los primeros trabajos futuros sería implementar las refactorizaciones restantes.

Actualmente el sistema MRS es capaz de refactorizar módulos funcionales, es decir únicamente operadores ecuacionales. Por tanto, otro de los trabajos futuros sería extender MRS para que pueda trabajar también con módulos de sistema que contengan reglas.

Otro aspecto a considerar en un trabajo futuro será ampliar el catálogo de refactorizaciones del sistema para cubrir otros casos que no se hayan considerado.

Bibliografía

[1] Acar, U. A., Blelloch, G. E., and Harper, R. Selective Memoization. *CoRR abs/1106.0447* (2011).

[2] Brown, C. M. *Mini Thesis - Refactoring Functional Programs.* PhD thesis, University Of Kent, 2005.

[3] Brown, C. M. *Tool support for Refactoring Haskell Programs.* PhD thesis, University Of Kent, 2008.

[4] Burstall, R. M., and Darlington, J. A Transformation System for Developing Recursive Programs. *Journal of the ACM 24*, 1 (1977), 44–67.

[5] Canfora, G., Cimitile, A., and Lucia, A. D. Conditioned Program Slicing. *Information & Software Technology 40*, 11-12 (1998), 595–607.

[6] Clavel, M. Reflection in General Logics, Rewriting Logic, and Maude, 1998.

[7] Clavel, M., Durán, F., Eker, S., Lincoln, P., Marí-Oliet, N., and Meseguer, J. Metalevel Computation in Maude. In *2nd International Workshop on Rewriting Logic and its Applications (WRLA'98)* (1998), vol. 15 of *Electronic Notes in Theoretical Computer Science*, Elsevier.

[8] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Quesada, J. F. Maude: Specification and Programming in Rewriting Logic. *Computer Science Laboratory, SRI International* (Jan. 1999).

[9] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Quesada, J. F. Maude: Specification and Programming in Rewriting Logic. *Theoretical Computer Science 285(2)* (2002), 187–273.

[10] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Talcott, C. L. *All About Maude- A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, 2nd ed. Springer, 2007.

[11] Cowles, J., and Gamboa, R., Eds. *Contributions to the theory of Tail Recursive Functions.* (Nov. 2004). Department of Computer Science, University of Wyoming.

[12] Foltynowicz, I. Recursion versus Iteration with the List as a Data Structure. *Informatics in education 6*, 2 (Jan. 2007), 283–306.

[13] Hall, M., and Mayfield, J. Improving the Performance of AI Software: Payoffs and Pitfalls in Using Automatic Memoization. In *Proceedings of the Sixth International Symposium on Artificial Intelligence* (1993), Megabyte, pp. 178–184.

[14] Isaza, F. A. *Aplicaciones de la Lógica al Desarrollo del Software.* Universidad Nacional de Colombia, 2008.

[15] J. M. Caete Valdeón, F. J. G. M. Métodos Formales Orientados a Objetos. *Revista de la Asociación de Técnicos de Informática 165* (2003), 62–64.

[16] Jones, S. L. P. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[17] Knoop, J., Rthing, O., and Steffen, B. Partial Dead Code Elimination. In *PLDI* (1994), V. Sarkar, B. G. Ryder, and M. L. Soffa, Eds., ACM, pp. 147–158.

[18] Korel, B., and Laski, J. W. Dynamic Program Slicing. *Inf. Process. Lett. 29*, 3 (1988), 155–163.

[19] Li, H. *Refactoring Haskell Programs.* PhD thesis, University Of Kent, 2006.

[20] Li, H., and Thompson, S. Comparative Study of Refactoring Haskell and Erlang Programs. *Source Code Analysis and Manipulation, IEEE International Workshop on 0* (2006), 197–206.

[21] Li, H., and Thompson, S. A Domain-Specific Language for Scripting Refactorings in Erlang. In *15th Fundamental Approaches to Software Engineering(FASE2012)* (Tallinn, Estonia, Marzo 2012), J. de Lara and A. Zisman, Eds., Lecture Notes in Computer Science. Springer, p. 15pp.

[22] López, J. A. V. *Maude como Marco Semántico Ejecutable.* PhD thesis, Universidad Complutense De Madrid, 2003.

[23] Marchiori, M. Unravelings and ultra-properties. In *Proceedings of the Fifth International Conference on Algebraic and Logic Programming (ALP'96), volume 1139 of LNCS* (1996), Springer, pp. 107–121.

[24] Michie, D. "Memo" Functions and Machine Learning. *Nature 218*, 5136 (Apr. 1968), 19–22.

[25] Nishida, N., Sakai, M., and Sakabe, T. Soundness of Unravelings for Conditional Term Rewriting Systems via Ultra-Properties Related to Linearity. *Logical Methods in Computer Science 8*, 3 (2012).

[26] Ohlebusch, E. *Advanced Topics in Term Rewriting.* Springer, 2002.

[27] Partsch, H., and Steinbrggen, R. Program Transformation Systems. *ACM Comput. Surv. 15*, 3 (1983), 199–236.

[28] Rosu, G. From Conditional to Unconditional Rewriting. In *WADT* (2004), J. L. Fiadeiro, P. D. Mosses, and F. Orejas, Eds., vol. 3423 of *Lecture Notes in Computer Science*, Springer, pp. 218–233.

[29] Rubio-Sánchez, M. Tail Recursive Programming by Applying Generalization. In *ITiCSE* (2010), R. Ayfer, J. Impagliazzo, and C. Laxer, Eds., ACM, pp. 98–102.

[30] Silva, J., and Chitil, O. Combining Algorithmic Debugging and Program Slicing. *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming* (2006), 157–166.

[31] University of Illinois at Urbana-Champaign, D. The System Maude. At: http://maude.cs.uiuc.edu/.

[32] University of Texas at Austin, M. ACL2. At: http://www.cs.utexas.edu/ moore/acl2/.

[33] Weiser, M. D. *Program Slices: Formal, Psychological, and Practical investigations of an Automatic Program Abstraction Method.* PhD thesis, Ann Arbor, MI, USA, 1979. AAI8007856.