Departamento de Informática
Universidad de Valladolid
Valladolid - España

Refactorizaciones de Especialización sobre el Lenguaje Modelo MOON
Versión 1.0

Raúl Marticorena Sánchez, Yania Crespo González-Carvajal

Departamento de Informática, Universidad de Valladolid, España.
{ rmartico@ubu.es, yania@infor.uva.es }

Resumen La búsqueda de la definición de refactorizaciones con una cierta independencia del lenguaje de programación es un campo de interés actual. El presente trabajo define y analiza refactorizaciones de especialización sobre el lenguaje modelo MOON con el objetivo de conseguir la definición de refactorizaciones sobre lenguajes orientados a objetos para su posterior soporte en una solución basada en frameworks.
Se valida MOON como lenguaje para la definición de refactorizaciones, centrándose en las refactorizaciones de especialización, con el objetivo de obtener un diseño más claro y eficiente a través de la especialización de propiedades de clases con y sin genericidad.

Palabras clave: refactorización, especialización, independencia del lenguaje, lenguajes estáticamente tipados, genericidad, lenguaje modelo minimal.

Informe Técnico DI-2003-02

1. Introducción

El creciente auge de los trabajos realizados en el área de la refactorización de software está marcando una serie de tendencias, entre las que se encuentra, como línea abierta de investigación, la búsqueda de una cierta independencia del lenguaje a la hora de definir, analizar y desarrollar herramientas que asistan al proceso de refactorización.

La definición de refactorizaciones, de manera independiente del lenguaje, ofrece una solución a las posibilidades de reutilización en el desarrollo de herramientas de refactorización cuando se adapten a nuevos lenguajes y nuevas operaciones de refactorización. En este sentido, el esfuerzo de la definición de refactorizaciones de especialización de manera general, garantiza una recuperación del esfuerzo inicial y su aplicación futura a nuevos lenguajes.

En trabajos realizados previamente se ha propuesto el lenguaje modelo minimal MOON[1] [Cre00], para representar las construcciones abstractas necesarias en el análisis y definición de refactorizaciones. Dicho lenguaje debe servir como base para una solución a las posibilidades de reutilización en el desarrollo de herramientas que ejecutan refactorizaciones cuando se adapten a nuevos lenguajes y nuevas operaciones de refactorización.

Para lograr esto se hace necesario, por una parte, la validación del modelo frente a la definición de refactorizaciones y, por otra, avanzar en su soporte en una solución basada en frameworks.

El presente trabajo, se centra en la validación de refactorizaciones de especialización sobre el lenguaje modelo MOON, con el objetivo de llegar a lograr la mayor independencia posible del lenguaje original en la realización de refactorizaciones.

Por otro lado, el actual interés de la programación genérica, dentro de la mayoría de lenguajes orientados a objetos, hacen de ésta un área relevante en la definición y aplicación de refactorizaciones. En particular la existencia de propuestas para su inclusión en lenguajes como Java [BCK+01] y en el .Net Common Language Runtime [KS01], en cuya especificación inicial no aparece, determinan la futura importancia de la definición de refactorizaciones en dicho ámbito.

Existen catálogos de refactorizaciones y diferentes definiciones de operaciones de refactorización que pueden considerarse de especialización o de generalización [Opd92, Fow00]. Sin embargo en estos trabajos y catálogos no existen refactorizaciones relacionadas con la genericidad. En este sentido previamente en [Cre00] se definió una refactorización de generalización llamada **parameterize** mientras que en este trabajo se presenta una colección de refactorizaciones de especialización en cuanto a genericidad definidas sobre MOON.

A través de dicho proceso se debe lograr identificar aquellas propiedades generales a los lenguajes y contenidas en MOON e implicadas en la definición de refactorizaciones, frente a aquellas propiedades particulares de cada lenguaje y no recogidas en el lenguaje modelo. Esto permite reducir los esfuerzos de definición

[1] MOON es acrónimo de Minimal Object-Oriented Notation.

de las refactorizaciones de especialización, al no tener que particularizar para cada lenguaje o o bien, permitir diferir la solución a una extensión concreta.

En lo que sigue, el documento se estructura de la siguiente forma. En la **Sección 2. Trabajos Relacionados** (pág. 2) se describe el estado del arte actual, en la **Sección 3. Refactorizaciones de Especialización** (pág. 10) se detalla la definición de refactorizaciones de especialización sin tener en cuenta aspectos de genericidad mientras que en la **Sección 4. Refactorizaciones de Especialización en cuanto a Genericidad** (pág. 32) se definen refactorizaciones de especialización en cuanto a tipos paramétricos y clases genéricas. Por último se concluye con la **Sección 5. Conclusiones y Líneas de Trabajo Futuro** (pág. 57), mostrando las conclusiones obtenidas y las líneas de trabajo futuro.

2. Trabajos Relacionados

En la presente sección se describen los trabajos previos y actuales líneas de investigación en lo referente a la definición de refactorizaciones en general, y más concretamente, a la búsqueda de independencia del lenguaje respecto a la definición de refactorizaciones.

Con este enfoque se pueden distinguir las siguientes líneas principales:

- Definición de refactorizaciones
- Definición independiente del lenguaje
- Especialización de programas
- Refactorizaciones en cuanto a genericidad

2.1. Definición de Refactorizaciones

Esta sección hace un recorrido de la evolución de los trabajos sobre refactorizaciones, desde los trabajos iniciales y asentamiento del concepto [Opd92] hasta la definición de catálogos que son referencia obligada en cualquier trabajo sobre refactorizaciones [Fow00] [2].

El estudio del distinto enfoque tomado por los autores en la definición de refactorizaciones y construcción de catálogos más o menos amplios y centrados en un paradigma y lenguaje concreto, justificarán posteriormente, en la **Sección 3. Refactorizaciones de Especialización** (pág. 10) y en la **Sección 4. Refactorizaciones de Especialización sobre Genericidad** (pág. 32), la estructura seguida en la definición de refactorizaciones.

Evolución y Reestructuración en Sistemas Orientados a Objeto Los trabajos de [Cas90] [Cas92] [Cas94] presentan el problema de falta de metodologías y herramientas en la modificación de código. Sus trabajos describen operaciones de cambio sobre sistemas orientados a objetos. En la definición de

[2] Disponible el catálogo de refactorizaciones en http://www.refactoring.com. Última visita realizada a 29 de diciembre de 2003

estas operaciones, el autor propone la definición de primitivas de una manera informal.

A partir de dichas operaciones de bajo nivel define transformaciones y reestructuraciones de alto nivel (la aplicación de la Ley de Demeter, reestructuración de jerarquías de herencia, etc...). Para llevar esto a cabo, se propone una solución basada en un algoritmo. El algoritmo se describe sobre un modelo formal, en el que toda acción lleva asociada un conjunto de condiciones a cumplir (el autor no llega a hablar de pre y postcondiciones).

En concreto el trabajo práctico de aplicación de los algoritmos se realiza sobre clases implementadas en EIFFEL.

Refactorización de Frameworks Orientados a Objeto En este trabajo [Opd92] se expone el problema de mantenimiento de software, centrándose en la modificación del mismo. Partiendo de trabajos previos, sobre evolución de software, evolución en esquemas de bases de datos orientadas a objeto y reutilización de clases, se presenta el concepto de refactorización como *"planes de restructuración que soportan cambios a nivel intermedio"*.

Frente al problema del diseño y su complejidad de mantenimiento, se aboga por el soporte automático a las modificaciones sobre el software a través de herramientas (desde un punto de vista tanto de diseño como de código), describiendo una aproximación al soporte automático de la restructuración de código orientado a objeto.

Un hecho fundamental, es que las refactorizaciones deben preservar el comportamiento del programa. La refactorización soporta el cambio y evolución de tal forma que se facilita la comprensión y evolución. De especial interés es la apreciación del hecho de que la aplicación de refactorizaciones de forma aleatoria, pese a preservar el comportamiento, puede llevar a una peor solución final. En concreto, el autor apunta a la necesidad de una cierta inferencia e interacción por parte del diseñador o programador.

En lo referente a la clasificación y estructura de refactorizaciones, el autor plantea una división entre las de bajo nivel (low-level), de escasa dificultad, y las de alto nivel (high-level), más expresivas y teóricamente más interesantes basadas a su vez en las de bajo nivel.

El autor aplica dichos conceptos en la definición de refactorizaciones de alto nivel. De interés particular es la definición de refactorizaciones tanto desde el punto de vista de la generalización (*creación de una superclase abstracta*) como de la especialización (*subclasificando y simplificando condicionales*). Estas refactorizaciones se soportan en subconjuntos de refactorizaciones de bajo nivel. Para asegurar la correción de dichas operaciones, el autor, propone la definición de funciones que expresen la semántica del código.

Planteadas dichas bases, la definición de refactorizaciones se realiza de la siguiente forma:

- Descripción no formal, hablando en términos generales, de la refactorización
- Pasos en la realización de refactorizaciones:
  - Descripción breve

- Argumentos de entrada
- Chequeo de precondiciones
- Refactorizaciones de bajo nivel implicadas
- Justificación de la preservación de comportamiento

El enfoque tomado es semiformal, en tanto en cuanto la definición de refactorizaciones se hace de forma textual junto a un conjunto de funciones. El trabajo se centra en un único lenguaje como C++, estáticamente tipado, no cubriendo aspectos avanzados del mismo en la definición de refactorizaciones como herencia múltiple, genericidad (*templates*) o conversiones de tipos explícitas (*casts*). Como líneas abiertas se plantea el estudio de la aplicación práctica de dichas definiciones sobre un conjunto más amplio de lenguajes.

Catálogos de Refactorizaciones En [Fow00] se retoma el concepto de refactorizaciones. La definición del autor expone que: *"Refactorización es el proceso de evolucionar un sistema software de tal forma que no altere el comportamiento externo del código mejorando su estructura interna"*. De nuevo se propone las ideas de preservación del comportamiento y mejora de la estructura.

La obra se propone como un catálogo de refactorizaciones de menor a mayor nivel, dividiendo el catálogo en una clasificación basada en los elementos sobre los que opera la refactorización.

La definición de refactorizaciones, al igual que en [Opd92], sigue una estructura:

- Descripción
- Motivación
- Mecanismos
- Ejemplos

Los ejemplos se plantean sobre código Java, aunque la definición de las refactorizaciones (gráficamente expresada en una notación similar a UML) debe ser válida para cualquier LOO[3], estáticamente tipado, que incluya conceptos comunes a los definidos en Java.

El principal problema en la definición de refactorizaciones del catálogo es la definición no formal. Las operaciones son descritas de manera libre, en forma de recomendaciones a las acciones a realizar sobre el código fuente. El planteamiento es similar al seguido en otros libros que siguen la estructura de *"libros de cocina"* como [GHJV95], en donde la descripción y solución del problema se plantea desde una descripción no formal.

Definición Formal de Refactorizaciones En [Rob99] se expone el problema de automatizar la transformación de código que preserve el comportamiento. El objetivo básico es la definición formal de refactorizaciones, fácil de automatizar a través de una lógica de predicados de primer orden.

[3] Lenguaje Orientado a Objeto

Retomando la propuesta de [Opd92] y su definición de operaciones de bajo nivel, el autor propone la definición de precondiciones y postcondiciones en cada refactorización. La demostración de la conservación del comportamiento descansa sobre el cumplimiento de las postcondiciones.

El trabajo se centra en el estudio sobre SMALLTALK (lenguaje LOO no estáticamente tipado) y su traducción para poder tratar el código transformado a predicados. La implementación de su trabajo se recoge en la herramienta ***Refactoring Brower***[4].

En esta misma línea, y basándose en los trabajos de [Rob99], en [CN00] se propone la idea de composición de refactorizaciones de bajo nivel, siguiendo con el estudio de pre y postcondiciones en la definición de refactorizaciones.

2.2. Definición de Refactorizaciones Independientes del Lenguaje sobre Modelos

Una vez presentado el concepto de refactorización y distintas propuestas de definición de las mismas podemos concluir que en todas ellas se ha tomado un lenguaje como objetivo concreto. Sin embargo existen propuestas en la definición de refactorizaciones independientes del lenguaje.

El Meta-Modelo FAMIX y MOOSE En [Tic01] se presenta la definición de un meta-modelo, FAMIX, para el soporte de refactorizaciones con independencia del lenguaje. El propósito es definir un conjunto de herramientas sobre el modelo, entre ellas, MOOSE [DLT00], como herramienta de refactorizaciones.

Sin entrar en detalles sobre dicho modelo, fuera del ámbito del presente trabajo, la definición de refactorizaciones se realiza con una cierta independencia del lenguaje. Inicialmente se trabaja sobre un subconjunto de refactorizaciones del catálogo de [Fow00] básicamente, y se sigue una estructura similar en la definición:

- Descripción y motivación
- Precondiciones
- Análisis de precondiciones
- Ejemplos
- Trabajos relacionados
- Discusión

Dentro de las precondiciones, indicar la diferenciación en las refactorizaciones de precondiciones independientes del lenguaje (generales) y aquellas particulares de un lenguaje concreto. Los lenguajes concretos sobre los que se trabaja son SMALLTALK y JAVA. El hecho de trabajar sobre lenguajes estática y dinámicamente tipados, carentes del concepto de tipos paramétricos y clases genéricas, hace que dichos temas no se traten.

[4] Disponible en http://wiki.cs.uiuc.edu/RefactoringBrowser. Última visita realizada a 29 de diciembre de 2003

Definición Genérica de Refactorizaciones En [Läm02], se presenta una definición de refatorizaciones de manera genérica, entendiendo ésta, como la definición de la refactorización para un conjunto amplio de paradigmas de programación, soportado a través de una solución de framework de programación funcional, con puntos de extensión para elementos particulares de cada lenguaje.

La propuesta cubre la instanciación a través de un conjunto de lenguajes como PROLOG, JAVA, HASKELL o XML SCHEMA abarcando un amplio abanico de paradigmas de programación y lenguajes.

En el trabajo se expone como ejemplo la refactorización *abstract extraction* (equivalente a *Extract Method (110)*) en [Fow00] dentro de un paradigma orientado a objeto) usando HASKELL como lenguaje de especificación e implementación.

De todos los trabajos, aquí mostrados, éste es el más ambicioso, en el sentido de la definición de refactorizaciones de una manera general para cualquier paradigma de programación.

2.3. Especialización de Programas

La especialización de programas, surge de la posible pérdida de eficiencia de construir software general y reutilizable. Aplicando estas técnicas se intenta garantizar el comportamiento inicial simplificando la estructura. Esto llevado a la definición de refactorizaciones hace que se definan operaciones entre cuyos objetivos pueden estar: obtención de código más eficiente, aumento en la facilidad de comprensión de la solución dada o adaptación y preparación para una modificación posterior.

Troceado de Programas El concepto de troceado de programas se puede definir como *"la técnica de descomposición que extrae de los programas las sentencias relevantes para un cálculo."*[BG96]. Dentro de sus múltiples aplicaciones, una de ellas es asistir en las tareas de mantenimiento del software, dando la posibilidad de:

- comprender mejor el código actual
- hacer cambios sin afectar negativamente al código actual

Aunque inicialmente surge su uso en el paradigma de la programación estructurada, pronto surgen trabajos en los que se aplican dichas técnicas a jerarquías de clases [TCFR96].

Análisis de Programas Las dependencias entre propiedades (atributos y métodos) y clases, (a partir del empleo de técnicas de ***Troceado de Programas***) han sido estudiadas en [Bac98] con el desarrollo del algoritmo *Rapid Type Analysis (RTA)*.

Este algoritmo a partir de la creación de estructuras de datos como *Grafos de Herencia* y *Grafo Virtual de Llamadas*, analiza el código teniendo en cuenta las instanciaciones reales producidas a lo largo de una ejecución para, tomando

una polítca pesimista, deducir todas aquellos métodos que pueden participar en la ejecución (resultado del polimorfismo y ligadura dinámica) sin tomar en cuenta las sentencias de flujo de control. Su trabajo se centra en la eliminación de código *"muerto"* en los lenguajes C++ y JAVA.

En [SRVRH99] se plantean extensiones a *RTA* que mejoran sus resultados: el algoritmo de *Variable Type Analysis (VTA)*, que toma en cuenta las cadenas de asignaciones que se producen entre la instanciación de un tipo y su receptor; por otro lado *Declared-Type Analysis (DTA)* que modifica al anterior en que su análisis toma como representante de las variables su tipo y no su nombre, colocando todas las clases con igual tipo en la misma clase de equivalencia.

Dichas técnicas, normalmente aplicadas u orientadas al campo de la optimización de compilación para la resolución de llamadas virtuales, pueden servir como apoyo a las refactorizaciones, ya que el conjunto de estructuras almacenadas y parte de los resultados obtenidos pueden ser reutilizados como elemento validador de las operaciones de refactorización realizadas.

Extracción de Código La evolución de las técnicas de troceado y análisis de programas ha llevado a la investigación en la extracción de código. En [TLSS99], trabajando sobre código JAVA, se expone la utilidad de utilizar estas técnicas en la transformación de código como:

- remover propiedades redundantes
- transformación de jerarquías de clases
- renombrado de paquetes, clases y propiedades

Sus objetivos básicos son la reducción de tamaño del código generado finalmente, con las ventajas correspondientes en su envío por la red. Básicamente la idea es *"extraer"* el código necesario para obtener un comportamiento equivalente.

Además de la optimización, uno de los resultados obtenidos es que se facilita la comprensión del código (objetivo relacionado con las refactorizaciones) a la hora de señalar aquellas clases y propiedades no utilizadas.

Análisis de Conceptos Formales En [ST97] se presenta un nuevo método de análisis de programas basado en la técnica formal de *Análisis de Conceptos* previamente utilizada en Ingeniería del Software y Representación del Conocimiento, que aplicado a una jerarquía de herencia entre clases se obtiene una nueva, en la que se han eliminado las propiedades no requeridas (estrechamente relacionado con los trabajos de [TLSS99]).

En la misma línea de este trabajo, en [AM02] se propone el uso de *Análisis de Conceptos* para el análisis de clases en una jerarquía orientada a objetos. Se muestra el trabajo sobre auto referencias (*self*) y referencias paternas (*super*), y el envío de mensajes a través de dichas referencias. Aunque su análisis se orienta a ayudar a entender el uso del software, también sirve para la detección de puntos débiles en las jerarquías y posibles puntos de extensión. El trabajo se centra sobre jerarquías de herencia en SMALLTALK.

Esta técnica también se ha utilizado en [PCML00] como soporte para facilitar las tareas de creación, evolución e instanciación de un framework. Un punto de interés, común con el estudio de refactorizaciones, es como a través de estos análisis se puede llegar a la transformación y adaptación de frameworks. Dicho análisis se basa en la definición de elementos (clase y selector) y propiedades (predicados) definidos en base a las relaciones entre clases y selectores.

Esta caracterización de software OO a través de predicados definidos sobre la información del código de las clases se ha planteado también en trabajos sobre refactorizaciones ya mencionados [Rob99] [Opd92].

Entre sus conclusiones se encuentra la posibilidad de aplicar dicho tipo de análisis en la compresión del software y reingeniería, dando guías para la adaptación de clases.

Especialización en Refactorizaciones En algunos programas se encuentran abstracciones generales y un número de casos especiales. Las posibilidades de la programación orientada a objetos, nos pueden llevar a la implementación de una clase que cubra un número excesivo de casos especiales.

En esta situación parece que, la especialización de clases, tal y como la define [Opd92], nos conduce a implementar dichos casos especiales en las subclases. El trabajo de [Opd92] se centra en una especialización en la que se sustituyen los bloques de instrucciones asociadas a las sentencias condicionales, por una jerarquía de herencia y uso del polimorfismo.

Dicha especialización (u operación de refactorización) también ha sido definida como *Replace Conditional with Polymorphism (255)* [Fow00]. En el caso de este autor, la refactorización se puede vincular a su vez a una serie de refactorizaciones definidas en su catálogo como *Replace Type Code with Subclasses (223)* y *Replace Type Code with State Strategy (227)* como pasos previos necesarios. Una vez aplicadas se puede llevar a cabo la refactorización descomponiéndolas en una serie de refactorizaciones de menor tamaño com *Extract Method (110), Move Method (142)*, etc.

2.4. Refactorizaciones en cuanto a Genericidad

Como ya se señaló en [Opd92], la definición de refactorizaciones puede ser extendida a otras características más avanzadas como herencia múltiple o tipos paramétricos, aunque el autor lo propuso desde la perspectiva concreta de un lenguaje como C++.

En trabajos previos como [Cre00] se ha definido una operación de refactorización de generalización (**parameterize**). El objetivo de dicha refactorización es introducir genericidad en una clase, añadiendo un parámetro formal a la clase, teniendo en cuenta los efectos resultantes en las clases dependientes. La definición de la operación se realiza sobre el lenguaje modelo MOON, buscando una definción común para la familia de lenguajes soportados por MOON.

Sin embargo en el sentido de la especialización, en la medida de lo que conoce el autor del presente trabajo, no se han encontrado definiciones de refactoriza-

ciones de especialización sobre los tipos parámetricos y las clases genéricas que los implementan.

La ausencia de definiciones en esta línea, deja abierta la línea de trabajo tomada en la **Sección 4. Refactorizaciones de Especialización sobre Genericidad** (pág. 32) en la definición de un catálogo inicial de refactorizaciones sobre genericidad.

2.5. Líneas Abiertas en las Propuestas Actuales

Con excepción del trabajo de [Läm02], que cubre un amplio conjunto de lenguajes, la mayoría de trabajos mencionados se centran en el estudio de refactorizaciones o técnicas relacionadas sobre un conjunto de lenguajes reducido y concreto. No existe una reutilización de la definición de refactorizaciones o algoritmos desarrollados, ni de su soporte sobre un modelo. La única excepción a estos puntos se encuentra en FAMIX [DTS99] y MOOSE [DLT00], pero el vacío frente a aspectos como la genericidad, deja abierto un gran campo.

En concreto, el presente trabajo se pretende definir un conjunto de refactorizaciones de especialización sobre el lenguaje modelo MOON. En dicha definición, se intenta aglutinar todos aquellos elementos que, a juicio del autor y apoyándose en otros trabajos, describen el concepto de refactorización de una manera completa. La estructura seguida en la definición de cada refactorización es: descripción, motivación, precondiciones, operaciones, postcondiciones y en algunos casos, por mayor claridad, un ejemplo.

Para ello es necesario definir un conjunto de predicados, a partir de la información contenida en las clases escritas en código MOON almacenadas en un repositorio, que nos permitan definir las precondiciones y postcondiciones.

En este proceso se debe determinar que predicados son soportados directamente por la información contenida en MOON, frente a aquellos particulares de familias de lenguajes o de un lenguaje concreto. Por otro lado, se deben definir las acciones a realizar en cada refactorización presentando las modificaciones a realizar, como primitivas de edición sobre código MOON.

Como se ha señalado en esta sección, no existe un catálogo de refactorizaciones de especialización sobre aspectos vinculados a tipos paramétricos y las clases genéricas. En este punto, se plantea continuar con la línea de definición de refactorizaciones de generalización [Cre00], trabajando en la dirección contraria de especialización. La posibilidad de iniciar un catálogo en esta línea parece, en vista a la próxima incorporación de clases genéricas a lenguajes muy difundidos, de claro interés.

El resultado final esperado es obtener es una definición de refactorizaciones de especialización general, independiente del lenguaje (siempre en el contexto de LOO, estáticamente tipados), modificando aspectos como la genericidad, con la consecución de la validación sobre el lenguaje modelo y detectando aquellos puntos de extensión para su soporte final sobre una solución basada en frameworks.

3. Refactorizaciones de Especialización

En esta sección se presenta la validación de la definición de refactorizaciones de especialización sobre el lenguaje modelo MOON. Para ello se definen un conjunto de predicados, un conjunto de operaciones de transformación sobre la gramática de MOON para finalmente describir las refactorizaciones en base a esos predicados y primitivas. La gramática de MOON extraída de [Cre00] se puede consultar en el **Anexo A. Sintaxis Concreta de MOON** (pág. 60).

3.1. Notación e Identificación de Propiedades

La identificación de propiedades de las clases (atributos y métodos) debe realizarse a través de un identificador único en el sistema (al igual que en un sistema OO[5] a cada objeto se le proporciona un único *Object ID* (OID), solución también adoptada en bases de datos orientadas a objeto).

A lo largo del presente trabajo se tomará como base, que las propiedades serán identificadas por letras (e.g. p para propiedades, m para métodos, a para atributos), por simplificación. De hecho, dichas letras identificativas toman un valor genérico, de tal forma que representan un identificador único de una propiedad, sin tomar en cuenta en las definiciones, de qué propiedad concreta se habla. Para las propiedades también se utilizará la notación (id_clase, _, id_propiedad) siguiendo la nomenclatura ya definida en MOON para las entidades.

En la identificación de clases se ha simplificado, y al igual que en trabajos previos [Cre00], se toma el nombre de la clase como identificador único en el sistema. Se adopta también la distinción entre la notación utilizada para indicar el tipo C frente a la clase C que lo implementa.

3.2. Estudio de Predicados sobre el Lenguaje Modelo MOON

En esta sección se estudia un conjunto de predicados definidos sobre la información disponible en el lenguaje modelo MOON. Debe ser necesario evaluar parte de ese conjunto, para poder definir las precondiciones y postcondiciones a cumplir en un grupo de refactorizaciones, verificando que la información necesaria está disponible en el marco establecido por MOON [Cre00]. El objetivo es clasificar las predicados necesarios como verificables con la información de la que se dispone.

Definiciones Previas Inicialmente se parte de un conjunto de conceptos básicos que definen las clases, sus propiedades (atributos y métodos) y signatura de las mismas. El propósito es establecer una base sólida sobre la que definiremos el resto de predicados. Para dicha definición se tomará por un lado definiciones ya establecidas en [Cre00] y por otro, nuevas definiciones, basándose en los conceptos existentes.

[5] Orientado a Objeto

Definiciones en MOON

Classes como el conjunto de todas las clases en un repositorio, denotado como $\mathcal{C}$.

Attributes(C) siendo C una clase se define como el conjunto de propiedades de una clase C, pertenecientes a $Names_C$[6](tanto atributos intrínsecos como heredados), obtenidos a partir de la regla de producción ATTRIBUTE_DECS (regla 14) de la gramática que describe MOON, como sustitución a ATT_DEC (regla 16), en la propia clase y ancestros.

Methods(C) siendo C una clase se define como el conjunto de propiedades de una clase C pertenecientes a $Names_C$ (tanto métodos intrínsecos como heredados), obtenidos a partir de la regla de producción METHOD_DECS (regla 15) de la gramática que describe MOON, como sustitución a METH_DEC (regla 17), en la propia clase y ancestros.

CD(T) siendo T un tipo y dada una expresión de tipo en MOON determinada por la regla 24:
24 CLASS_TYPE $\triangleq$ CLASS_ID [REAL_PARAMETERS])
se define que C es clase determinante (CD) del tipo T si C es el nombre de la clase que se corresponde con la reducción de CLASS_ID en dicha regla.
Sea X una anotación de tipo en el texto de una clase. El tipo T al que hace referencia X está dado por:

- Si X es una reducción de la parte FORMAL_GEN_ID de la regla 23, entonces T =FORMAL_GEN_ID.
- Si X es una reducción de la parte CLASS_TYPE de la regla 23, entonces X es una reducción de CLASS_ID[REAL_PARAMETERS]. Sea C=CD(X), la clase determinante de X y [X_1,...,X_n] la reducción, si la hubiera, de la parte opcional REAL_PARAMETERS, el tipo C implementado por la clase C está dado por $S_C \triangleq TC$(C). Si X = C, el tipo T al que hace referencia X es C. Si X=[X_1,...,X_n], el tipo T al que hace referencia X es $C[T_1, \ldots, T_n]$ donde T_i es el tipo al que hace referencia X_i.

Nuevas Definiciones

Name(p) siendo p una propiedad se define como el texto asignado en el código fuente a dicha propiedad.

- Si la propiedad es un atributo, su nombre es el texto asociado a VAR_ID en la regla de producción 22:

  22 VAR_DEC $\triangleq$ VAR_ID':'TYPE

  Ej: a:X; – Name(p_a)=a, donde p_a es el identificador del atributo.
- Si la propiedad es un método, su nombre es el texto asociado a METHOD_ID en la regla de producción 20:

[6] La definición de $Names_C$ se dará posteriormente, en la Sección 3.2.

20 WITHOUT_RESULT ≙ [**deferred**] METHOD_ID [FORMAL_ARGUMENTS]

Ej: m(c:C):X; – Name(p_m)=m, donde p_m es el identificador del método.

Signature(m) siendo m un método se define como concatenación del nombre (**Name(m)**) de un método a la lista ordenada de tipos de los argumentos formales de su declaración.

En las rutinas se corresponde con la reducción de las reglas de producción:
18 ROUTINE_DEC ≙ WITHOUT_RESULT
20 WITHOUT_RESULT ≙ METHOD_ID [FORMAL_ARGUMENTS]
21 FORMAL_ARGUMENTS ≙ '(' { VAR_DEC ';' ...}+ ')'
22 VAR_DEC ≙ VAR_ID ':' TYPE
tomando la parte TYPE de cada una de las declaraciones de argumentos formales en la regla 22 y como tipo de retorno para la signatura el tipo predefinido NONE.

En las funciones se corresponde con la reducción de las reglas de producción:
19 FUNCTION_DEC ≙ WITHOUT_RESULT ':' TYPE
tomando la parte TYPE de cada una de las declaraciones de argumentos formales en la regla 22 junto con la sustitución de TYPE en la regla 19 como tipo de retorno.

Ej: r(c:C, d:D); – signatura de la rutina es $r : C\ x\ D \rightarrow NONE$

Ej: f(c:C, d:D):E; – signatura de la función es $f : C\ x\ D \rightarrow E$

EqualName(p,q) siendo p y q propiedades se define como el resultado de la comparación de cadenas literales asociadas al nombre de dos propiedades (atributos o métodos).

$$EqualName(m, n) \triangleq Name(m) = Name(n)$$

EqualSignature(m,n) siendo m y n métodos se define como el resultado de la comparación de las signaturas de dos métodos, nombre a nombre, elemento de la lista de tipos a elemento de la lista de tipos y tipo de retorno con tipo de retorno.

$$EqualSignature(m, n) \triangleq Signature(m) = Signature(n)$$

Relaciones en Herencia Establecidas las definiciones básicas debemos en primer lugar repasar las definiciones ya establecidas en [Cre00] respecto a las relaciones de herencia entre clases para, analizar posteriormente los nuevos predicados que se introducen.

Herencia en MOON

Anc(B) siendo B una clase se define como el conjunto de clases de $\mathcal{C}$ tal que existe un camino de herencia desde B a dichas clases:

$$Anc(B) \triangleq \{A \in \mathcal{C} / cam_h(B, A) \neq \emptyset\}$$

DAnc(B) siendo B una clase se define como los ancestros directos o padres de la clase B tal que la longitud del camino de herencia es uno:

$$DAnc(B) \triangleq \{A \in \mathcal{C} / \exists c \in cam_h(B, A) \wedge long(c) = 1\}$$

Desc(A) siendo A una clase se define como el conjunto de clases de $\mathcal{C}$ tal que existe un camino de herencia con origen en B a dichas clases:

$$Desc(A) \triangleq \{B \in \mathcal{C} / cam_h(B, A) \neq \emptyset\}$$

DDesc(A) siendo A una clase se define como los descendientes directos o hijos de la clase tal que la longitud del camino de herencia es uno:

$$DDesc(A) \triangleq \{B \in \mathcal{C} / \exists c \in cam_h(B, A) \wedge long(c) = 1\}$$

EP(C) siendo C una clase se define como las propiedades intrínsecas[7] definidas como aquellas propiedades de la clase resultado de la reducción de la regla de producción SIGNATURES (regla 5) y que no aparecen en las reducciones de las reglas de herencia (regla 10).

IP(C) siendo C una clase se define como las propiedades heredadas no definidas en la clase, definidas en un ancestro, y en su caso, modificadas a través de los operadores MODIFIER (regla 12).
La estructura de las clases de MOON ha sido definida de forma tal que si la propiedad aparece como identificador de un atributo o método en la estructura generada por la regla SIGNATURES (regla 5) y no aparece en las estructuras que son resultado de reducciones de la regla INHERITANCE_CLAUSE (regla 10), entonces se sabe que la propiedad es intrínseca, en caso contrario es heredada.

Names$_C$ siendo C una clase se define como el conjunto de propiedades intrínsecas y heredadas de una clase C.
Se cumple que:

$$Names_C = EP(C) \cup IP(C)$$

$$EP(C) \cap IP(C) = \emptyset$$

Rename$_{B,A}(p)$ siendo A y B clases y p una propiedad se define como el conjunto de nombres resultado de los renombrados de una propiedad p que hayan ocurrido en los caminos de herencia desde B hasta A. Cuando hay herencia repetida desde B hasta A puede ocurrir que $\mid Rename_{B,A}(p) \mid > 1$.
Si $q \in Rename_{B,A}(p)$, quiere decir que la propiedad p en la clase A, ha sufrido en el camino de herencia hacia B algún renombrado, resultado de aplicar el modificador **rename** (regla 12) en alguna de las cláusulas de herencia, de tal forma que finalmente el nombre de la propiedad en la clase B es el nombre de la propiedad q.

[7] Intrínsecas o esenciales. Se elige el término inglés *Essential Properties* (EP).

Nuevas Definiciones

MakeEffective(C) siendo C una clase se define como las propiedades p pertenecientes a $Names_C$, tal que la propiedad se ha hecho efectiva en la clase, siendo diferida en un ancestro directo.
Resultado de la reducción de INHERITANCE_ CLAUSE (regla 10), con un modificador **makeeffective** PROP ID (regla 12) donde *Name(p) = PROP_ID*.

Deferred(C) siendo C una clase se define como el conjunto de propiedades de la clase C, cuya implentación no se proporciona en el cuerpo de la misma.
Marcadas con la palabra terminal **deferred** en la regla de producción 20 o bien p perteneciente a $Names_C$ tal que aparece en la reducción de INHERITANCE_CLAUSE, con un modificador **makedeferred** PROP_ID donde Name(p) = PROP_ID.

Redefine(C) siendo C una clase se define como el conjunto de propiedades de una clase C en la que se redeclara la propiedad dando una nueva implementación.
La propiedad p está marcada con la palabra clave **redefine** en la regla de producción 12 con PROP_ID donde *Name(p) = PROP_ID*.

Relaciones de Cliente

Definiciones en MOON

Client(B) siendo B una clase se definen como clientes el conjunto de clases de $\mathcal{C}$ tal que existe un camino de cliente desde B a dichas clases:

$$Client(B) \triangleq \{A \in \mathcal{C} / cam_c(B, A) \neq \emptyset\}$$

DClient(B) siendo B una clase se define como los clientes directos tal que la longitud del camino de cliente es uno:

$$DClient(B) \triangleq \{A \in \mathcal{C} / \exists c \in cam_c(B, A) \wedge long(c) = 1\}$$

Prov(A) siendo A una clase se definen como proveedores el conjunto de clases de $\mathcal{C}$ tal que existe un camino de cliente con origen en B a dichas clases.

$$Prov(A) \triangleq \{B \in \mathcal{C} / cam_c(B, A) \neq \emptyset\}$$

DProv(A) siendo A una clase se define como los proveedores directos el conjunto de clases tal que la longitud del camino de cliente a ellas desde A es uno:

$$DProv(A) \triangleq \{B \in \mathcal{C} / \exists c \in cam_c(B, A) \wedge long(c) = 1\}$$

DDepend(C) siendo C una clase se define como el conjunto de clases que referencian directamente a la clase C. Para obtener dicho conjunto, partiendo del grafo de dependencias de la clase, tomamos el conjunto de clases clientes directos e hijos de la clase.

$$DDepend(C) \triangleq DClient(C) \cup DDesc(C)$$

Nuevas Definiciones

ReferenceProperty(p,C) siendo p una propiedad y C una clase se define como el conjunto de métodos que referencian dicha propiedad de la clase C.
Distinguimos dos casos:

- **ReferenceAttribute** si la propiedad es un atributo:
  $ReferenceAttribute(p, C) \triangleq \{x \in ReferenceProperty(p, C)/p \in Attributes(C)\}$
- **ReferenceMethod** si la propiedad es un método:
  $ReferenceMethod(p, C) \triangleq \{x \in ReferenceProperty(p, C)/p \in Methods(C)\}$

ReferenceSelf(p,C) siendo p una propiedad y C una clase se define como el conjunto de propiedades (métodos) de la clase C que auto referencian a la propiedad p.
Toda propiedad perteneciente a *ReferenceSelf(p,C)* tiene que cumplir que la clase en la que está definida sea C (pertenece a $Names_C$).

$$\forall p_i \in ReferenceSelf(p, C) \Rightarrow p_i \in Names_C$$

Se corresponde con:

- Regla de producción 37 para rutinas.
  Ej: rou(a); – la rutina *rou* se referencia a través de **self** implícitamente.
- Regla de producción 42 para atributos y funciones.
  Ej: x := a; – el atributo *a* se referencia a través de **self**
  Ej: x := func(a,b); – la función *func* se referencia a través de **self** implícitamente.
- Regla de producción 38 y 43 para envíos de mensaje de longitud 2 en los que la regla de producción 42 se reduce a **self** por la regla 46.
  Ej: x := self.a; – el atributo *a* se referencia a través de **self**.
  Ej: x := self.func(a,b); – la función *func* se referencia a través de **self**.
  Ej: self.rou(a,b); – la rutina *rou* se referencia a través de **self**.
  Ej: at.rou(a,b); – donde *at* es atributo de la clase y *rou* una rutina, se referencia a *at* a través de **self** implícitamente.
  Ej: x : = at.func(a,b); – donde *at* es atributo de la clase y *func* una función, se referencia a *at* a través de **self** implícitamente.

ReferenceSuper(p,C) siendo p una propiedad y C una clase se define como el conjunto de propiedades (métodos) de las clases en *DDesc(C)* que referencian a la propiedad p de C.
Toda propiedad perteneciente a *ReferenceSuper(p,C)* tiene que cumplir que la clase en la que está definida pertenezca a *DDesc(C)*.

$$\forall p_i \in ReferenceSuper(p, C) \Rightarrow p_i \in Names_D \wedge D \in DDesc(C)\}$$

No existe una correspondencia directa con la gramática en MOON al carecer del concepto de referencia paterna.
Como solución se modifica la gramática con las siguientes reglas:

38 CALL_INSTR_LONG2 ≜ ['{' TYPE '}'] TARGET '.' CALL_INSTR_LONG1

43 CALL_EXPR_LONG2 ≜ ['{' TYPE '}'] TARGET '.' CALL_EXPR_LONG1

Además se introduce una nueva regla:
51 TARGET ≜ **super** | ENTITY
Desde el punto de vista de la corrección semántica para las reglas modificadas (reglas 38 y 43), depende de que haya una sustitución a {TYPE}. En tal caso TARGET tiene que ser **super** y la clase determinante de la sustitución a TYPE tiene que ser un ancestro directo de la clase (*DAnc(C)*) donde se encuentra la instrucción o expresión.
Teniendo una instrucción o expresión de longitud 2 de la forma $\{T\}e_1.e_2$ en el cuerpo de un método de la clase C donde $e_1 = (C, \_, \textbf{super})$ se debe cumplir:

$$CD(T) \in DAnc(C)$$

En el caso de necesitar resolver la ambigüedad en la referencia a la propiedad, si se hereda varias veces (en presencia de herencia múltiple), si TARGET se reduce a **super**, tiene que haberse anotado TYPE para deshacer la ambigüedad.
Además, la corrección de la llamada de longitud 2 (instrucción o expresión), cuando la sustitución de TARGET sea **super**, depende de que METHOD_ID o ENTITY generados a partir de las reglas 37 y 42 (llamadas de longitud 1) pertenezcan al conjunto de propiedades heredadas y no sean diferidas.
Entonces para que la invocación sea correcta se debe cumplir que, siendo la clase determinante A del tipo A que sustituye a TYPE:

$$\exists p \in Names_A / (Name(p) = METHOD\_ID \vee Name(p) = VAR\_ID) \wedge p \in IP(C) \wedge p \notin Deferred(A)$$

Ejemplo:

```
class C
    inherit A
          redefine f;
    end
    inherit B
          redefine f;
    end
signatures
body
...
    f
    do
```

```
        ...
        {A} super.f; -- elegimos f del ancestro A por regla 38
         -- en regla 43 equivalente recogiendo el valor enviado
        ...
    end
...
end
```

Con esta modificación se cubre:

- Lenguajes que incluyen el uso de referencias paternas como: EIFFEL (**precursor**), JAVA (**super**), SMALLTALK (**super**), C# (**base**), SELF (**resend**), CLOS (**call_next_method**),etc.
- Solución a la ambigüedad en la referencia, en presencia de herencia múltiple (EIFFEL, C++,etc.)

Dadas las definiciones previas se debe cumplir que:

$$ReferenceSelf(p, C) \subset ReferenceProperty(p, C)$$

$$ReferenceSuper(p, C) \subset ReferenceProperty(p, C)$$

CandidatesProperties(p,C) siendo p una propiedad y C una clase se define como el conjunto de propiedades candidatas a ser invocadas a través del mecanismo de ligadura dinámica, cuando se invoca una determinada propiedad p de una clase.
Estará formado por el conjunto de propiedades en clases descendientes, heredadas, redefinidas o renombradas, a partir de una determinada propiedad

$$CandidatesProperties(p, C) \triangleq \{p_i \in Names_{C'} / C' \in Desc(C) \wedge (p_i \in IP(C') \wedge Name(p_i) = Name(p) \vee p_i \in Rename_{C',C}(p) \vee (p_i \in Redefine(C') \wedge p_i \in Rename_{C',C}(p)))\}$$

UnreferencedProperty(p,C) siendo p una propiedad y C una clase se define como el predicado que evalúa si una determinada propiedad no es referenciada directamente y no es candidata de una propiedad referenciada.
En caso de que el conjunto resultante no sea el conjunto vacío, quiere decir que la propiedad se utiliza bien directamente o bien como candidata a una propiedad.

$$UnreferencedProperty(p, C) \triangleq ReferenceProperty(p, C) = \emptyset \wedge \nexists p_i \in Names_{C'} / ReferenceProperty(p_i, C') \neq \emptyset \wedge p \in CandidatesProperties(p_i, C')$$

Se pueden distinguir dos variantes:

- **UnreferencedAttribute** si la propiedad es un atributo: $UnreferencedAttribute(a, C) \triangleq UnreferencedProperty(a, C) / a \in Attributes(C)$
- **UnreferencedMethod** si la propiedad es un método: $UnreferencedMethod(m, C) \triangleq UnreferencedProperty(m, C) / m \in Methods(C)$

Desarrollo de Relaciones de Cliente de una Propiedad En la estructura definida en [Cre00] como *Grafo de Dependencias* se recogían las relaciones de cliente y herencia (con o sin genericidad, teniendo en cuenta cuando entra en juego una instanciación genérica) entre las clases.

Para el cálculo de nuestros predicados del tipo **ReferenceProperty(p,C)**, **ReferenceAttribute(a,C)**, etc ... se necesita una información más detallada. No basta con conocer la existencia de una relación de cliente entre dos clases, sino que se necesita conocer a través de qué propiedades se establece dicha relación.

Así pues se definen los predicados implicados en el cálculo de estos conjuntos. Partiendo de un conjunto de definiciones básicas ya presentadas en [Cre00]:

MessageSending una llamada o envío de mensajes se define como una forma de encadenar más de un envío de mensajes en una misma construcción. Las llamadas estarán formadas por $e_1 \ldots e_{n-1}.e_n$, donde e_i es entidad, $e_1 \ldots e_{n-1}$ es una expresión y e_n un mensaje. Si el mensaje denotado por e_n se corresponde con un atributo o función se tiene una expresión, en caso contrario (rutina) se tiene una instrucción. En MOON los envíos serán sólo de la forma e_1 o $e_1.e_2$. Se denota como *ms*.

Length longitud de un envío de mensaje (*ms*) se define como la cantidad de llamadas encadenadas que forman la construcción, al margen de los argumentos que pueda tener. Se denota como L(ms).

En MOON se simplifica, eliminado los envíos en cascada por lo que sólo tendremos envíos de longitud 1 y 2.

Ejemplo:

```
t1:=e1;    -- envío de longitud 1
t1:=e1.e2; -- envío de longitud 2
```

Tomando la definición previa de [Cre00] de envío de mensaje (*ms*) y con la simplificación de la no existencia de llamadas en cascada definimos:

MethodBody(m,C) siendo m un método y C una clase se define como el conjunto de instrucciones (INSTR) contenidas en el cuerpo de un método m. Resultado de la reducción de la regla 30:
30 METHOD_BODY $\triangleq$ **body** INSTR **end**

MessageInMethodBody(m,C) siendo m un método y C una clase se define como el conjunto de envío de mensajes (expresiones o instrucciones) que tenemos en el cuerpo de un método m (expresiones o instrucciones).
Se obtiene como resultado de las sustituciones a partir de las reglas de producción 32, 35 y 36 de la gramática definida en MOON.

$MessageInMethodBody(m, C) \triangleq \{ms \in MethodBody(m, C)/m \in Methods(C)\}$

FeatureCall(m,C) siendo m un método y C una clase se define como el conjunto de llamadas a propiedades de las clases clientes, realizadas a través de envío de mensajes (*ms*) desde un método m.

$FeatureCall(m, C) \triangleq \{p \in Names_{C'} \wedge C' \in \mathcal{C} / \forall ms \in MessageInMethodBody(m, C)$ con
$(L(ms) = 1 \Rightarrow p = e_1) \vee (L(ms) = 2 \Rightarrow p = e_2)\}$

Definiendo de nuevo ReferenceProperty(p,C) en base a los nuevos predicados introducidos:

ReferenceProperty(p,C) se define como:

$$ReferenceProperty(p, C) \triangleq \{m \in Names_D / D \in DDepend(C) \wedge p \in FeatureCall(m, D)\}$$

Ejemplo: dado el siguiente código fuente en MOON

```
class Lista ...
    insertar(elem:Int)
        i:Int;
        ultimo:Nodo;
        fin:Bool;
    do
        i := 1;     -- expresion de asignacion con constante manifiesta
        ...
        fin := i.>=(total); -- expresion
        i := i.+(1);        -- expresion y asignacion
        create ultimo;      -- creacion
        ultimo.crear(elem)  -- instruccion
        ...
    end
...
end
```

Tenemos que:

$MessageInMethodBody(insertar, Lista) = \{i. >= (total), i. + (1), ultimo.crear(elem)\}$

Donde:

$FeatureCall(insertar, Lista) = \{(Int, \_, >=), (Int, \_, +), (Nodo, \_, crear)\}$

Si se calcula el conjunto que referencia a la propiedad $(Int, \_, +)$ obtendremos:

$$ReferenceProperty(+, Int) = \{(Lista, \_, insertar)\}$$

La conclusión es que a partir de la información de envío de mensajes en expresiones y instrucciones se puede determinar a nivel detallado, no sólo la relación de cliente entre clases, sino la dependencia concreta entre clases a través de sus métodos y atributos.

3.3. Primitivas de Edición sobre el Lenguaje Modelo MOON

Partiendo de la clasificación de refactorizaciones de bajo nivel (*low-level*) y alto nivel (*high-level*) definida en [Opd92] o bien de la diferenciación entre refactorizaciones (*refactoring*) y refactorizaciones grandes (*big refactoring*) en [Fow00], se estudia, en primer lugar, las primitivas de edición que trabajan directamente sobre las clases MOON.

Esta distinción permite separar lo que son operaciones de manipulación a nivel del lenguaje (*primitivas*), de aquellas operaciones más complejas (*refactorizaciones*), las cuales se suelen caracterizar por una serie de precondiciones, acciones y postcondiciones para poderse llevar a cabo la refactorización.

A continuación se va a describir una serie de primitivas que manipulan directamente sobre el lenguaje, la estructura de las clases. Es necesario observar que simplemente se define la operación realizada sin tomar en cuenta que la clase resultante sea incorrecta de cara a su utilización por parte de las clases dependientes directas.

AddAttribute(a,C) dado un atributo a y una clase C se define como el resultado de añadir a la sustitución de la regla de producción ATTRIBUTE_DECS (regla 14), que contiene la lista de atributos, la declaración del nuevo atributo a.

AddMethod(m,C) dado un método m y una clase C se define como el resultado de añadir a la sustitución de la regla de producción METHOD_DECS (regla 15) que contiene la lista de métodos la declaración del método m y añadir a la sustitución de la regla de producción METHOD_IMPL (regla 26) el cuerpo del método m.

RemoveAttribute(a,C) dado un atributo a y una clase C se define como el resultado eliminar el atributo:

1. Si $a \in EP(C)$: eliminar en la sustitución de la regla de producción ATTRIBUTE_DECS (regla 14), la declaración ATT_DEC (regla 16) que contiene un VAR_ID con *Name(a) = VAR_ID*.
2. Si $a \in IP(C)$: eliminar los modificadores de herencia sobre PROP_ID donde *Name(a)=PROP_ID*. En el caso de utilizar el modificador **redefine** o **makeeffective** sobre PROP_ID con Name(a) = PROP_ID eliminar en la sustitución de la regla de producción ATTRIBUTE_DECS (regla 14), la declaración ATT_DEC (regla 16) que contiene un VAR_ID con *Name(a) = VAR_ID*.

RemoveClass(C) dada una clase C se define como el resultado de borrar la clase del repositorio, eliminado a partir de la regla de producciťon CLASS_DEF (regla 2), con CLASS_ID (regla 3) igual a la clase C, todos los elementos contenidos en dicho módulo.

RemoveMethod(m,C) dado un método m y una clase C se define como el resultado de eliminar un método de una clase:

1. Si $m \in EP(C)$: eliminar en la sustitución a la regla de producción METHOD_DECS (regla 15) la declaración de signatura del método m y eliminar el cuerpo del método en la sustitución a la regla METHOD_IMPL (regla 26) para dicho método m.

2. Si $m \in IP(C)$: eliminar de la sustitución a la regla de producción MODIFIER (regla 12), los textos correspondientes a los modificadores de herencia con PROP_ID donde *Name(m)=PROP_ID*. En el caso de utilizarse **redefine** o **makeeffective** debe eliminarse el cuerpo del método en la sustitución a la regla METHOD_IMPL (regla 26) para dicho método m.

RemoveRedefineClause(m,C) dado un método m y una clase C se define como el resultado eliminar la cláusula de redefinición del método m en la clase C, correspondiente a la regla 12, con $Name(m) = PROP\_ID$.

3.4. Definición de Refactorizaciones de Especialización

Entendiendo como especialización, el proceso contrario a la generalización y abstracción, analizaremos el conjunto de refactorizaciones implicadas según [Opd92] y [Fow00].

Las refactorizaciones serán definidas de forma similar a [Fow00] o [TB99] a través de una plantilla. La plantilla estará formada por un nombre, una breve descripción de la refactorización, motivo, entradas, precondiciones, acciones y postcondiciones. Las precondiciones y postcondiciones serán definidas en base al conjunto de predicados analizados en la Sección 3.2.

Las acciones de las refactorizaciones atómicas se definen directamente sobre las primitivas definidas en la Sección 3.3. La nomenclatura elegida para las refactorizaciones sigue el planteamiento de [Opd92].

Nota Por motivos de brevedad no se ha incluido un ejemplo en código MOON puesto que la funcionalidad de cada refactorización es sencilla y existen varias obras donde consultar dichas refactorizaciones sobre lenguajes concretos, como ya se comentó en la **Sección 2. Trabajos Relacionados**.

- ***Delete Unreferenced Class***
- ***Create Member Function***
- ***Delete Member Function***
- ***Create Member Variable***
- ***Delete Unreferenced Variable***
- ***Moving Member Variable To Subclasses***
- ***Moving Member Method to Subclasses***

Delete Unreferenced Class

Descripción Eliminar una clase del repositorio no utilizada.

Motivación La clase no es referenciada por ninguna otra (relación de cliente o de herencia).

Entradas Una clase C a eliminar.

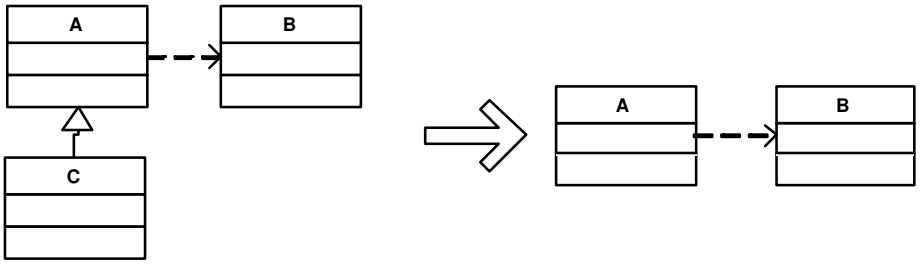

Figura1. Delete Unreferenced Class

Precondiciones

1. La clase pertenece al repositorio.

$$C \in Classes$$

Análisis de la precondición: para poder eliminar la clase debe pertenecer inicialmente al repositorio.

2. La clase no es referenciada, ni desde descendientes ni clientes, excepto desde la propia clase.

$$DDepend(C) = \emptyset \vee DDepend(C) = \{C\}$$

Análisis de la precondición: para poder eliminar una clase del repositorio es necesario que ninguna otra clase tenga dependencias con dicha clase. Las dependencias se definen a través del concepto de *cliente*, *herencia* y *sustitución* (con genericidad). Dichas dependencias se pueden extraer del *Grafo de Dependencias del Repositorio* [Cre00].
La clase eliminada no debe ser referenciada (relación de cliente, incluida la de herencia) desde ninguna otra clase.

Acciones

1. *DeleteClass(C)*

Postcondiciones

1. La clase no pertenece al conjunto de clases del repositorio.

$$C \notin Classes$$

Create Member Function

Descripción Añadir un nuevo método a una clase (Figura 2).

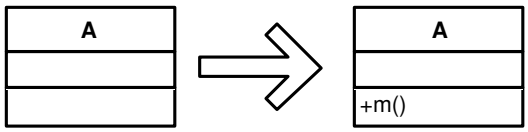

Figura2. Create Member Function

Motivación La clase necesita la definición de un nuevo método para añadir nuevos comportamientos o como resultado de otra refactorización.

Entradas Un método nuevo m y una clase C a la que se añade.

Precondiciones

1. No existe un método con la misma signatura.

$$\nexists n \in Methods(C)/EqualsSignature(m, n)$$

Análisis de la precondición: para poder añadir un método a una clase se debe cumplir que no existe un método con la misma signatura. Esto surge del invariante, en el que no se pueden tener dos propiedades efectivas con la misma signatura.

Acciones

1. *AddMethod(m,C)*

Postcondiciones

1. El método pertenece a la clase como propiedad intrínseca.

$$m \in EP(C)$$

Delete Member Function

Descripción Eliminar un método no referenciado de una clase (Figura 3).

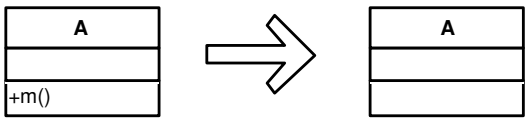

Figura3. Delete Member Function

Motivación El método no es utilizado ni desde la propia clase ni desde otras.

Entradas Un método m a eliminar y una clase C en la que se define dicho método.

Precondiciones

1. El método está implementado en la clase.

$$m \in EP(C) \vee m \in Redefine(C)$$

   Análisis de la precondición: para poder eliminar un método de una clase, dicho método tiene que estar definido (o redefinido) en la misma, puesto que no se pueden eliminar propiedades heredadas.
2. No existen invocaciones (o candidatas) a dicho método excepto desde el propio método (llamadas recursivas).

$$UnreferencedMethod(m, C) \vee (ReferenceMethod(m, C) = \{m\} \wedge \forall n \in CandidatesProperties(m, C) \Rightarrow ReferenceProperty(n, C) = \emptyset)$$

   Análisis de la precondición: el método podrá ser eliminado si no es utilizado por ninguna otra clase, o bien sólo se usa desde el propio método de la clase. Si se elimina el método y se requiere su uso desde algún otro punto, se obtendría un error en compilación.

Acciones

1. *RemoveMethod(m,C)*
2. Si $m \in Redefine(C) \Rightarrow RemoveRedefineClause(m, C)$

Postcondiciones

1. El método no pertenece como propiedad intrínseca o redefinida a la clase.

$$m \notin EP(C) \wedge m \notin Redefine(C)$$

Create Member Variable

Descripción Añadir un nuevo atributo a una clase (Figura 4).

Motivación La clase necesita un nuevo atributo.

Entradas Un atributo a y una clase C a la que se añade el atributo.

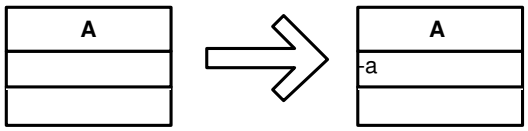

Figura4. Create Member Variable

Precondiciones

1. La clase no contiene un atributo (intrínseco o heredado) con el mismo nombre que el atributo a añadir.

$$\nexists x \in Attributes(C)/EqualName(x, a)$$

Análisis de la precondición: al añadir un nuevo atributo a la clase, no se permite que exista ya un atributo con ese mismo nombre, heredado o intrínseco a la clase. Esto que es conocido como ocultación de atributos en algunos lenguajes no está soportado en MOON.

Acciones

1. *AddAttribute(a,C)*

Postcondiciones

1. El atributo pertenece a la clase.

$$a \in Attributes(C)$$

Delete Unreferenced Variable

Descripción Eliminar un atributo de una clase (Figura 5).

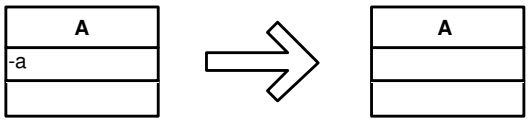

Figura5. Delete Unreferenced Variable

Motivación La clase contiene un atributo que no es utilizado.

Entradas Un atributo a y una clase C.

Precondiciones

1. El atributo está definido en la clase.

$$a \in EP(C) \vee a \in Redefine(C)$$

Análisis de la precondición: para poder eliminar el atributo, éste debe estar definido en la clase de la que se elimina, por lo tanto debe ser una propiedad intrínseca o bien redefinirse en la clase.

2. El atributo no es accedido ni desde la propia clase ni desde otras.
$UnreferencedAttribute(a, C) \vee \forall m \in ReferenceAttribute(a, C) \Rightarrow m \in Methods(C)$
Análisis de la precondición: para poder eliminar un atributo con seguridad, éste no debe ser referenciado desde ninguna clase a excepción de la propia clase.

Acciones

1. *DeleteAttribute(a,C)*

Postcondiciones

1. El atributo no pertenece a la clase.

$$a \notin Attributes(C)$$

Moving Member Variable to Subclasses

Descripción Mover el atributo a los descendientes directos (Figura 6).

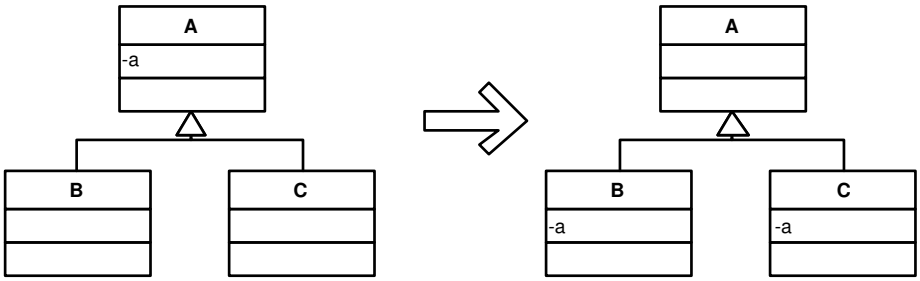

Figura6. Moving Member Variable to Subclasses

Motivación Un atributo es sólo utilizado por alguno de los descendientes directos.

Entradas Un atributo a y una clase C.

Precondiciones

1. Precondiciones generales a todos los lenguajes OO estáticamente tipados definidas sobre MOON.
   a) El atributo no debe ser accedido por otras clases ni por la propia clase. $ReferenceAttribute(a, C) = \emptyset$
   Análisis de la precondición: si se elimina un atributo que es utilizado por otras clases o desde la propia clase, al compilar se obtendría un error.
2. Lenguajes que permiten ocultación de atributos [AGH01]
   a) Las subclases directas no contienen un atributo con el mismo nombre.

$$\forall C' \in DDesc(C) \Rightarrow \neg\exists b \in Attributes(C')/EqualName(a, b)$$

   Análisis de la precondición: al intentar bajar el atributo a las subclases, si éstas ya contienen un atributo con ese nombre, existiría una colisión de nombres.

Acciones

1. $RemoveAttribute(a, C)$
2. $\forall D \in DDesc(C) \Rightarrow AddAttribute(a, D)$

Postcondiciones

1. El atributo no pertenece a la clase C.

$$a \notin Attributes(C)$$

2. El atributo es propiedad intrínseca de alguno de los descendientes directos de C.

$$\exists D \in DDesc(C)/a \in EP(D)$$

Moving Member Method to Subclasses

Descripción Mover el método a los descendientes directos (Figura 7).

Motivación El comportamiento en un ancestro es relevante sólo para algunas de los descendientes directos.

Entradas Un método m y una clase C.

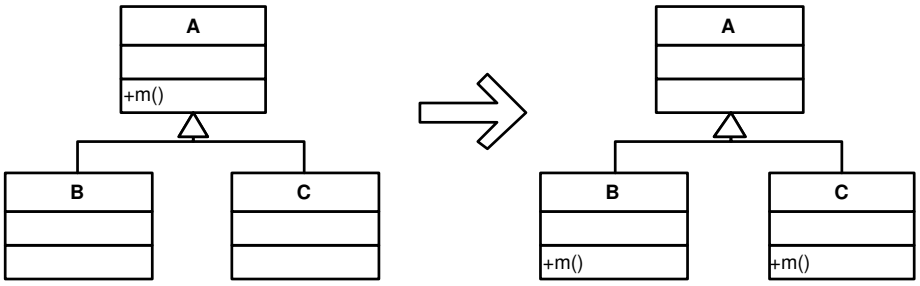

Figura7. Moving Member Method to Subclasses

Precondiciones Las precondiciones para esta refactorización se pueden clasificar a varios niveles.

1. Precondiciones generales a todos los lenguajes OO estáticamente tipados definidas sobre MOON.
   a) El método no debe ser invocado desde ninguna clase (incluida la propia) o bien la invocación se produce desde el mismo método de la clase a través de **self**/**this** (*llamadas recursivas.*)

   $$ReferenceMethod(m, C) = \emptyset \vee ReferenceMethod(m, C) = \{m\}$$

   Análisis de la precondición: no se permiten usos del método desde otras clases o métodos de la propia clase diferentes al actual. Si se elimina el método, dichas referencias serían detectadas como un fallo por parte del compilador.
   b) Al menos un descendiente directo de la clase que define el método no redefine el método.

   $$\exists D \in DDesc(C)/m \notin Redefine(D)$$

   Análisis de la precondición: al efectuar la refactorización alguna de las clases hijas no redefine el método. Aquellas clases hijas que redefinen el método no participan en la refactorización. Por lo tanto es necesario que como mínimo haya una clase que pueda participar.
   c) No existen invocaciones a través de **super** del método m, en los descendientes directos.

   $$ReferenceSuper(m, C) = \emptyset$$

   Análisis de la precondición: si se elimina el método en el ancestro directo las referencias a través de super en la clase C fallarán o bien se llamará a un método heredado en el ancestro directo, no garantizándose la semántica.
2. Lenguajes que permiten ocultación de atributos [AGH01]

a) No se permiten accesos a través de **self** en el método a atributos que también están definidos en una o más de las subclases.
$\nexists p \in ReferenceSuper(m, C) \wedge p \in Attributes(C) / \exists C' \in DDesc(C)$
con $p_2 \in Names_{C'} \wedge p_2 \in Attributes(C') \wedge EqualName(p, p_2)$
Análisis de la precondición: al bajar el método se accedería a un atributo "distinto" (atributo del hijo, no el del padre) al que originalmente se accedía.

b) No se permiten accesos con **super** a atributos también definidos en la clase desde el método m.
$\nexists p \in ReferenceSuper(m, C) / \exists p_2 \in Names_C \wedge p_2 \in Attributes(C)$
Análisis de la precondición: al bajar el método se accedería a un atributo "distinto" (atributo del padre, no el del ancestro directo del padre) al que originalmente se accedía.

3. Lenguajes con control de acceso a través de modificadores

a) No existen accesos a propiedades no exportadas a los descendientes directos en el método.

$$\nexists p \in Names_C / m \in ReferenceProperty(p, C) \wedge \forall D \in DDesc(C) \Rightarrow \neg IsAccesible(p, C, D)$$

El predicado *IsAccesible* no ha sido definido previamente por no reflejarse en MOON, los niveles de acceso. Sin embargo al analizar esta refactorización surge la necesidad de indicar precondiciones que dependen de este tipo de predicados. El análisis y solución final a dicho problema se analizará posteriormente en la Sección 3.5.
Análisis de la precondición: al bajar el método los accesos a propiedades no exportadas a descendientes de la clase padre provocarían un error en compilación.

4. Lenguajes con sintaxis concreta para los constructores

a) El método no puede ser un constructor (específica de Java, C++, C#, etc ...)

$$\neg IsConstructor(m, C)$$

El predicado *IsConstructor* se cumple si el método puede ser invocado como instrucción de incialización de valores en el proceso de instanciación. Este predicado es dependiente de cada lenguaje particular por lo que no se definió previamente en la Sección 3.2. El análisis y solución final a dicho problema se analizará posteriormente en la Sección 3.5.
Análisis de la precondición: en estos lenguajes el constructor cumple una serie de requisitos sintácticos (sin tipo en el retorno, nombre igual al de la clase) que hacen que el método no sea válido en otra clase.

Acciones

1. $RemoveMethod(m, C)$
2. $\forall C' \in DDesc(C) \wedge m \notin Redefine(C') \Rightarrow AddMethod(m, C')$
3. $\forall C' \in DDesc(C) \wedge \forall m \in Methods(C') / m \in Redefine(C') \Rightarrow RemoveRedefineClause(m, C')$

Postcondiciones

1. El método no pertenece a la clase C.

$$m \notin Methods(C)$$

2. El método es propiedad intrínseca de alguno de los descendientes directos de C.

$$\exists D \in DDesc(C)/m \in EP(D)$$

3.5. Refactorizaciones con Precondiciones No Generales

La mayoría de las refactorizaciones estudiadas hasta el momento pueden ser definidas sobre MOON como marco de trabajo, dado que las precondiciones, acciones y postcondiciones están reflejadas.

El problema surge de aquellas refactorizaciones en las que las precondiciones no pueden ser evaluadas con la información disponible en el lenguaje modelo MOON y por lo tanto no se puede asegurar la corrección de la operación en cuanto a que sea respetuosa con el resto de clases y objetos.

Análisis de las Precondiciones relativas a los Niveles de Acceso Si tomamos el catálogo de [Opd92] en su apartado de refactorizaciones de pequeño nivel se pueden encontrar una serie de precondiciones díficiles de verificar a partir de la información guardada de las clases en MOON.

En la refactorización **change_access_control_mode** es necesaria la información de control de acceso entre propiedades, que en MOON no se encuentra reflejada. Sin embargo en la mayoría de los lenguajes existen modificadores de este tipo (C++, C#, Java, etc ...) o mecanismos similares como la exportación selectiva (EIFFEL).

Las precondiciones para dicha refactorización marcadas por [Opd92] del tipo *isPrivate* (accesible sólo desde la propia clase), *isPublic* (accesible desde cualquier clase) y *isProtected* (accesible por clases en la misma agrupación o descendientes), son particulares de lenguajes que proporcionan modificadores de acceso explícitos en el lenguaje.

De cara a una formulación más acorde con las precondiciones definidas previamente, se puede establecer el predicado:

IsAccesible(p,C,C') siendo p una propiedad y C,C' clases se define como la posibilidad de acceso a la propiedad p de la clase C desde una clase C'.

Cada lenguaje define sus propias reglas en el acceso a propiedades en base a relaciones de cliente y herencia. Estas relaciones sí que están recogidas en MOON a través de los *Grafos de Cliente* y *Grafos de Herencia* respectivamente. El problema clave es que no en todos los lenguajes existen dichos modificadores de acceso definidos a nivel sintáctico. El ejemplo más claro es EIFFEL, donde el descendiente hereda todas las propiedades y de nuevo selecciona respecto a sus clientes, los accesos permitidos a otras clases.

Además, en un futuro, se podría encontrar lenguajes en los que la forma de especificar los niveles de acceso fuese muy diferente.

La solución propuesta es que dicha precondición sea clasificada como particular de cada lenguaje, siguiendo el enfoque de [Tic01], clasificando las precondiciones en dos categorías: generales (definidas sobre MOON) y particulares a cada lenguaje.
Ejemplo:

- En JAVA un atributo público siempre devolverá verdadero en la precondición **IsAccessible**, pero con la información disponible en MOON no se puede saber si es público o no.
- En EIFFEL una propiedad puede haber sido exportada a otra clase cliente, por lo tanto **IsAccessible** se cumplirá si la clase cliente pertenece a la lista de exportación. Esta información no está disponible en MOON.

En ambos casos la información se debería extraer del código original. Por lo tanto, la implementación de dicha precondición no puede ser implementada a nivel del lenguaje modelo, sino a nivel particular del lenguaje utilizado.

Refactorizaciones Dependientes del Control de Flujo En las refactorizaciones de alto nivel [Opd92] se puede ver como la información necesaria tampoco está disponible. En concreto la refactorización de especialización **Subclassing and Simplifying Conditionals** necesita comprobar los condicionales del código y extraer la lista de sentencias correspondientes a las ramas **if**, **then** y **else**.

Esta refactorización es similar a la planteada en [Fow00] como **Extract Hierarchy (375)** cuyos casos particulares más simplificados se tratan en refactorizaciones de nivel medio como **Replace Type Code with Subclasses (223)**, **Replace Type Code with State/Strategy (227)** y **Replace Conditional with Polymorphism (255)**.

Dado que en MOON los condicionales han sido eliminados (simplificados) no se puede tomar decisiones en refactorizaciones de este tipo. El problema surge, no por el incumplimiento de precondiciones, puesto que el conjunto básico de las mismas es verificable en MOON, sino por el hecho de carecer en el lenguaje de dichas estructuras de control de flujo que permitan llevar a cabo la refactorización.
Ejemplo: partiendo de código Java

```
public void metodo(int discriminante){
    if (discriminante == VALOR1)
        return (valor + porcentaje(10));
    else return this.valor;
}
```

Se traduce en MOON a:

```
metodo(discriminante:Int) local
```

```
    c1:Bool;
    c2:Bool;
    suma:Int;
do
    c1 := discriminante.=(VALOR1);
    c2 := c1.=(true);
    suma := porcentaje(10);  -- instr1: porcentaje del valor, rama del if
    result := valor.+(suma); -- instr2: rama del if
    result := valor;         -- instr3: rama del else
end;
```

La refactorización sugiere que se separen las distintas instrucciones correspondientes a cada rama del **if** (o **switch**) implementando subclases y delegando en llamadas por ligadura dinámica o tardía.

Sin embargo en el ejemplo en MOON, se comprueba como se ha perdido dicha información:

1. No se puede diferenciar la sentencia condicional (**if**) de la evaluación de expresiones con invocaciones a métodos de las entidades de tipo *(Bool,_,=)* o del tipo *(Int,_,=)*.
2. No se puede distinguir qué instrucciones pertenecen a cada rama de la condición (en nuestro ejemplo *instr1, instr2 o instr3*) para separar el código a través de una refactorización del tipo **Extract Method (110)** [Fow00] o **convert_code_segment_to_function** [Opd92].

Básicamente todas las refactorizaciones que parten del estudio de sentencias de control de flujo (condicionales e iteraciones), como por ejemplo las refactorizaciones *Replace Iteration with Recursion* y *Replace Recursion with Iteration*[8], no pueden ser llevadas a cabo con la información mínima almacenada en MOON.

4. Refactorizaciones de Especialización en cuanto a Genericidad

En esta sección se presenta la validación de la definición de refactorizaciones de especialización en cuanto a genericidad sobre el lenguaje modelo MOON. Para ello, al igual que en la anterior sección, se definen un conjunto de predicados, un conjunto de operaciones de transformación sobre el código para finalmente detallar un catálogo de refactorizaciones en base a esos predicados y primitivas.

4.1. Estudio de Predicados en cuanto a Genericidad sobre el Lenguaje Modelo MOON

En esta sección se continua con el trabajo previo del estudio de refactorizaciones en cuanto a genericidad definidas sobre el lenguaje modelo MOON iniciado con la definición de la refactorización de generalización **parameterize** en [Cre00].

[8] Aunque dichas refactorizaciones no aparecen en [Fow00], en el catálogo en web (http://www.refactoring.com/catalog/) mantenido por el autor, se añaden estas refactorizaciones como contribución de Dave Whipp e Ivan Mitrovic respectivamente

En concreto se definen refactorizaciones relacionadas con la especialización sobre genericidad a través de predicados y primitivas definidos sobre el lenguaje modelo MOON, que soporten un conjunto de refactorizaciones.

Como ya se indicó en la **Sección 2. Trabajos Relacionados** (pág. 2) la ausencia de un catálogo de refactorizaciones en este sentido obliga a la definición completa de un nuevo catálogo de refactorizaciones de especialización.

Definiciones previas

Concepto básicos Dado un tipo paramétrico:

$$S_T \triangleq \text{type } T[T_1, ..., T_n] = \forall(T_1, ..., T_n).ET(T_1, ..., T_n)$$

Se utiliza como $T[A_1, \ldots, A_n]$, donde $A_1, \ldots, A_n$ son tipos que sustituyen a las variables libres $T_1, \ldots, T_n$ en la expresión de tipo, ET. A los tipos $A_1, \ldots, A_n$ se les denomina parámetros genéricos reales o parámetros reales. $T[A_1, \ldots, A_n]$ se suele llamar una instanciación del tipo paramétrico T [Cre00].

Dependiendo de la sustitución que se proponga para los parámetros genéricos formales de un tipo paramétrico, se puede obtener un **tipo completamente instanciado**. Se dice que el tipo resultante es un tipo completamente instanciado si, en el contexto de la sustitución, los parámetros genéricos reales utilizados no contienen una variable de tipo libre para ser instanciada [Cre00].

Se entiende que un tipo no parámetrico es un tipo completamente instanciado al no depender de ningún parámetro formal, puesto que estos no existen en su definición.

En base a dichas definiciones se pueden establecer los siguientes predicados.

Comp(T) siendo T un tipo se define como el conjunto de tipos completamente instanciados a partir de la definición de un tipo paramétrico T.

$$Comp(T) \triangleq \{T[A_1, \ldots, A_n] / \text{ la sustitución } A_1, ..., A_n \text{ hace de } T[A_1, \ldots, A_n] \text{ un tipo completamente instanciado.}$$

Entidades(C) o ($\mathcal{E}(C)$) siendo C una clase se define como el conjunto de todas las entidades que aparecen en la clase C.

Variantes de acotación Respecto a la acotación en MOON, se definen tres variantes de acotación: por subtipado, por cláusulas ***where*** y por conformidad. Dentro de la acotación por subtipado se puede distinguir el caso especial de acotación F, en la que la acotación se define de manera recursiva sobre el propio parámetro formal.

A continuación se define cada una de estas variantes, incluyendo el caso particular de acotación F.

Bound_S se define como la acotación por subtipado consistente en indicar un tipo asociado al parámetro genérico formal G. Todo parámetro real debe ser un subtipo del tipo utilizado como acotación.

$$S_T \triangleq \text{type } T[T_1, \ldots, T_i, \ldots, T_n] =$$
$$\forall(T_1, \ldots, T_i \overset{isa}{\mapsto} R_i, \ldots, T_n).ET(T_1, \ldots, T_i, \ldots, T_n)$$

un tipo S es una sustitución válida para un parámetro genérico formal T que esté acotado por R si y sólo si $S \overset{isa}{\mapsto} R$. Esta definición se puede generalizar para cada uno de los parámetros formales (T_i) de la definición del tipo paramétrico (T) que pueden estar acotados igualmente por subtipado.
Dado que la acotación por subtipado introduce un cierto conjunto de problemas en la definición de tipos recursivos, la acotación F, modificación de la acotación por subtipado, viene a resolver estos problemas [Cre00].

Bound_F o acotación F, en un tipo paramétrico se define como:

$$S_T \triangleq \text{type } T[T_1, \ldots, T_i, \ldots, T_n] = \forall(T_1, \ldots, T_i \overset{isa}{\mapsto} F_i[T_i], \ldots, T_n).ET(T_1, \ldots, T_i, \ldots, T_n)$$

una sustitución S es válida para un parámetro formal T que esté F-acotado si y sólo si $S \overset{isa}{\mapsto} F[S]$. Esta definición se puede generalizar para cada uno de los parámetros formales (T_i) de la definición del tipo paramétrico (T), que pueden estar acotados a su vez por un tipo paramétrico instanciado genéricamente con sustituciones en su parámetros formales de uno o varios de los parámetros genéricos formales ($T_1, \ldots, T_n$) del tipo T.

Bound_W o acotación por cláusulas ***where***

$$S_T \triangleq \text{type } T[T_1, \ldots, T_i, \ldots, T_n] =$$
$$\forall(T_1, \ldots, T_i, \ldots, T_n).ET(T_1, \ldots, T_i, \ldots, T_n) \; where \; T_i \; has \; \{TF_1, \ldots, TF_n\}$$

Donde $\{TF_1, \ldots, TF_n\}$ son tipos funcionales. La definición se puede generalizar a la declaración de cláusulas ***where*** para cada uno de lo parámetros genéricos T_i de la declaración del tipo paramétrico T.

La noción de conformidad se puede expresar como que toda clase conforma con sus clases ancestros respetando las reglas de la herencia donde se exige covarianza en la redefinición de atributos, argumentos y resultados de funciones. En el caso particular de EIFFEL (donde se introduce este concepto [Mey97]) se considera que una clase genérica B instanciada conforma con otra clase genérica A instanciada si la clase B conforma con la clase A y los parámetros reales de B instanciada conforman con los parámetros reales de A instanciada. La noción de conformidad de clases genéricas instanciadas opera tanto fuera como dentro de los corchetes de la definición de un tipo paramétrico [Cre00].

Bound_C o acotación por conformidad, consiste en indicar un tipo asociado al parámetro genérico formal. Todo parámetro real debe conformar con el tipo utilizado como acotación.

$$S_T \triangleq \text{type } T[T_1, \ldots, T_i, \ldots, T_n] =$$
$$\forall(T_1, \ldots, T_i - \triangleright R_i, \ldots, T_n).ET(T_1, \ldots, T_i, \ldots, T_n)$$

un tipo S es una sustitución válida para un parámetro formal T que esté acotado por R si y sólo si $S - \triangleright R$. Esta definición se puede generalizar para cada uno de los parámetros formales (T_i) de la definición del tipo paramétrico (T) que pueden estar acotados igualmente por conformidad.

Definición de Subtipado en Función de las Variantes de Acotación

Subtype en el sistema de tipos MOON, se mantienen las reglas **reflexivo, transitivo, antisimétrico** y se hace una salvedad a las reglas **records, paramétricos, paramétrico y no paramétricos** tal que siendo C_1 y C_2 tipos de objetos que son implementados por clases MOON C_1 y C_2 tales que $S_{C_1} = TC(C_1)$ y $S_{C_2} = TC(C_2)$, para que se cumpla que $C_2 \overset{isa}{\mapsto} C_1$ se añade la condición que C_2 debe ser descendiente de C_1 [Cre00].
Además la regla **records** se reescribe para tener en cuenta los renombrados.
Variantes en la relación de subtipado:
Subtipado se mantiene la definición dada.
Cláusulas *where* se cambia la regla de **paramétrico acotados (subtipado)** por **paramétrico acotados (*where*)**.
Conformidad se cambia $\overset{isa}{\mapsto}$ por $-\triangleright$ (subtipado por conformidad) y se define la regla **funcionales** con covarianza en los argumentos.
Por lo tanto *Subtype(A)* de un tipo A se define como, el conjunto de subtipos según una de las variantes definidas en MOON (subtipado, clásulas *where* o conformidad), donde además en el sistema de tipos de MOON se debe cumplir la relación de herencia a través de las clases determinantes.

Nuevas definiciones

Predicados relacionados con la instanciación de tipos paramétricos

$\mathcal{T}$ se define como el conjunto total de tipos declarados a partir del conjunto de clases $\mathcal{C}$. Está formado por el conjunto de reducciones de la regla 24 de la gramática definida en MOON, contenidas en las clases de $\mathcal{C}$.

$$24\ \text{CLASS\_TYPE} \triangleq \text{CLASS\_ID[REAL\_PARAMETERS]}$$

Incomp(T) siendo T un tipo se define como el conjunto de tipos no completamente instanciados a partir de la definición de un tipo T paramétrico.

$$Incomp(T) \triangleq \{T[A_1, \ldots, A_n] / \exists A_i \in \{A_1, ..., A_n\} \wedge A_i \text{ no está completamente instanciado } \}$$

Types(T) siendo T un tipo se define como el conjunto de tipos (completamente instanciadado e incompletos) instanciados a partir de un tipo paramétrico.

$$Types(T) \triangleq Comp(T) \cup Incomp(T)$$

Dadas las anteriores definiciones se debe cumplir que:
$Comp(T) \subset Types(T)$
$Incomp(T) \subset Types(T)$
$Comp(T) \cap Incomp(T) = \emptyset$

Es necesario aclarar la diferencia entre $\mathcal{T}$ y $Types$. Mientras que el primero es el conjunto de tipos que se pueden extraer del análisis estático del conjunto de clases del repositorio $\mathcal{C}$, el segundo es un conjunto infinito de posibles tipos, completos e incompletos, instanciables genéricamente a partir de una definición de tipo paramétrico realizado a través de su implementación en una clase genérica.

IsCompleted(T) siendo T un tipo se define que el predicado se cumple cuando un CLASS_TYPE de una entidad está completamente instanciado.
Todo tipo cuya propiedad *IsCompleted* sea verdadera, pertenece a *Comp(T)* siendo T el tipo paramétrico origen de la instanciación genérica.
Ejemplo:

```
class C[H]
    ...
    signatures
        attributes ...
            a:A[Integer];
            b:A[A[H]];
    ...
end --class
```

Teniendo $e_1 = (C, \_, a)$ con tipo $A[Integer]$ y $e_2 = (C, \_, b)$ con tipo $A[A[H]]$, y $e_1, e_2 \in \mathcal{E}(C)$ entonces:
$IsCompleted(TE(e_1))$ es verdadero
$IsCompleted(TE(e_2))$ es falso

Se puede definir de nuevo:

$$Comp(T) = \{\forall T_i \in Types(T)/IsCompleted(T_i)\}$$
$$Incomp(T) = \{\forall T_i \in Types(T)/\neg IsCompleted(T_i)\}$$

IsGeneric(C) siendo C una clase se define que el predicado se cumple si en la reducción de la regla 7, [FORMAL_PARAMETERS] para dicha clase, ésta no es vacía, es decir, la clase es genérica con uno o varios parámetros formales.
Las clases genéricas son implementaciones de los tipos paramétricos y sólo a partir de ellas se pueden instanciar genéricamente nuevos tipos.

Predicados relacionados con la declaración de parámetros genéricos y sus sustituciones

FormalParam(C) siendo C una clase genérica se define como el conjunto de parámetros formales (FORMAL_GEN_ID) de la reducción de la regla 8 de la gramática MOON de la clase C.
Ejemplo:
class[G,H,I]is ... end $\Rightarrow FormalParam(C) = \{G, H, I\}$

FormalParamName(C,i) siendo C una clase genérica de la forma C[T_1, T_2, ..., T_n] e i una posición, se define como el texto asociado al parámetro formal (FORMAL_GEN_ID), asociado por su posición i, en las reglas 24, 25 y 23 de la gramática de MOON. El identificador tiene un ámbito local en el contexto de la clase.
Ejemplo:

```
class C[G,H,I] is ... end --class
```

Entonces:
$FormalParamName(C, 1)$=G
$FormalParamName(C, 2)$=H
$FormalParamName(C, 3)$=I

RealParamName(C,i) donde C es una clase e i es una posición, se define como dado un tipo instanciado genéricamente a partir de una declaración de clase genérica C de la forma $T[T_1, T_2, ..., T_n]$, donde T_i son los parámetros reales, se define como la posición del tipo T_i (TYPE) asociado al parámetro formal en la posición i. El tipo que sustituye al parámetro genérico puede ser a su vez un identificador genérico en la clase cliente directa.
Ejemplo 1:

```
class C[G,H,I] ...
    ...
    a:C[Integer,Float,Number];
    ...
end
```

Entonces:
$RealParamName(CD(TE(a)), 1)$=Integer
$RealParamName(CD(TE(a)), 2)$=Float
$RealParamName(CD(TE(a)), 3)$=Number
Ejemplo 2:

```
class C[G,H,I] ... end --class
    ...
class C' inherit C[Integer,Float,Number] ... end --class
```

Entonces:
$RealParamName(C[Integer, Float, Number], 1)$=Integer
$RealParamName(C[Integer, Float, Number], 2)$=Float
$RealParamName(C[Integer, Float, Number], 3)$=Number

SubstFormalPar(C,G) de una clase genérica C y un parámetro formal G se define como el conjunto de sustituciones reales con tipos completos a un parámetro formal G de una clase genérica C.
Dado que una instanciación puede depender a su vez de un identificador de tipo genérico, recursivamente habrá que añadir al conjunto los tipos resultantes de las sustituciones.
Ejemplo:

```
class A[G] ... end

class B[H]
    ...
    attributes
        b:A[H];
        b2:A[Integer];
    ...
```

```
end

class C[I]
    ...
    attributes
        c:B[Cliente];
        c2:B[D[I]];
    ...
end

class D[J]
    attributes
        d:J;
...
end

class E[K]
    attributes
        c:C[Proveedor];
        d:K;
...
end
```

En nuestro ejemplo se obtendría:

$$SubstFormalPar(A, G) = \{Integer, Cliente, D[Proveedor]\}$$

Definiendo formalmente:
$SubstFormalPar(C, G) \triangleq \forall C' \in DDepend(C), \forall e \in \mathcal{E}(C')$ con
$TE(e) = C[A_1, \ldots, A_i, \ldots, A_n] \wedge RealParamName(TE(e), i) =$
$A_i \wedge FormalParamName(C, i) = G \Rightarrow$
si $IsCompleted(A_i)$ añadimos A_i a $SubstFormalPar(C, G)$
en caso contrario: $FormalParamName(C', i)$ tal que depende su instanciación de $[H_1, ..., H_n]$, se añade al conjunto los tipos instanciados completos resultado de sustituir con $SubstFormalParam(C', H_i)$ para todo $H_i \in FormalParam(C')$.

Aplicando la definición a nuestro ejemplo se tienen los siguientes pasos:

1. Se toman las sustituciones del parámetro formal G de la clase A. La clase dependiente directa es B.
   En B, las instanciaciones de A, pueden ser completamente instanciados (usando $Integer$)o bien si no es completamente instanciado (depende de un parámetro de la clase desde donde se instancia, como H de B) repetimos el proceso recursivamente en la clase actual (B) sobre el correspondiente parámetro formal (H).
   La definición recursiva de este predicado, contempla la transitividad entre las sustituciones reales. Volviendo a nuestro ejemplo inicial se va a aplicar la defínción dada para calcular el conjunto de sustituciones.

En una primera iteración se tendrá:
$SubstFormalPar(A, G) = \{Integer\} \cup SubstFormalPar(B, H)$

2. Al obtener los tipos en $SubstFormalPar(B, H)$ se revisan las entidades en la clase dependiente directa C, teniendo como sustituciones los tipos $\{Cliente, D[I]\}$.
   Dado que $D[I]$ no es un tipo completamente instanciado (depende de un parámetro genérico formal local en la clase C, el parámetro I), se invoca al cálculo de $SubstFormalPar(C, I)$:

   $SubstFormalPar(A, G) =$
   $\{Integer, Cliente, \} \cup SubstFormalPar(C, I)$

3. Los tipos que sustituyen a I de C en las instanciaciones genéricas, en la clase dependientes directas, como E, son $\{Proveedor\}$ que genera en la clase C, una entidad de tipo completamente instanciado $D[Proveedor]$:
   El resultado final es:
   $SubstFormalPar(A, G) = \{Integer, Cliente, D[Proveedor]\}$

Predicados Relacionados con la Acotación

BoundType(C,G) siendo C una clase genérica y G un parámetro genérico formal se define como el CLASS_TYPE que acota al parámetro formal G, en las variantes de MOON de subtipado (1) y conformidad (3). Se obtiene de la sustitución del CLASS_TYPE en la regla 48S de la gramática de MOON.
Ejemplo:

class $A[G \rightarrow Comparable] \Rightarrow BoundType(A, G) = Comparable$

BoundSigs(C,G) siendo C una clase genérica y G un parámetro genérico formal se define como el conjunto de signaturas que acotan al parámetro formal G en la clase C, en la variantes de MOON de acotación por cláusulas *where* (2). Se obtiene de la sustitución de SIG_LIST en la regla 50W cuando FORMAL_GEN_ID es igual a G, en la gramática de MOON.
Ejemplo:

class $A[G]$ where G has
$Gf; g(G : a) \Rightarrow BoundSigs(A, G) = \{f : NONE \rightarrow G, g : G \rightarrow NONE\}$

SignatureWithSubstitution(s,C,T) siendo s una signatura de método en una cláusula **where**, G un parámetro genérico formal cuya acotación por cláusulas where contiene a s y T un tipo, se define como el resultado de sustituir en la signatura de un método de la cláusula **where**, todas las apariciones de G por T.
Ejemplo:
class $A[G$ where G has metodo$(a : G)]$ ... end
donde $s_1 = metodo : G \rightarrow G$ entonces
$SignatureWithSubstitution(s_1, G, T) = metodo : T \rightarrow T$

4.2. Primitivas de Edición en cuanto a Genericidad

En esta sección se estudia un conjunto de primitivas definidas sobre el lenguaje MOON, implicadas en la modificación de las clases genéricas.

Teniendo en cuenta que dichas primitivas operan directamente a nivel sintáctico, se define a continuación el conjunto de las mismas, directamente sobre la gramática de MOON.

Primitivas Generales

DeleteFormalParameter(C,G) dada una clase genérica C y un parámetro formal G, se define como el resultado de eliminar en la reducción de la regla de producción 7, la sustitución de FORMAL_PAR correspondiente a G (regla 8).
Ejemplo:

```
class A[G,H]
...
end
```

Si se aplica la primitiva *DeleteFormalParameter(A,H)* se obtiene:

```
class A[G] ... end
```

DeleteRealParameter(C,G,B) dada una clase genérica C, un parámetro formal G y una clase B en cuyo cuerpo se declaren instanciaciones genéricas de la clase C, se define como el resultado de eliminar en el cuerpo de la clase B, en las reducciones de la regla de producción 25, la sustitución a TYPE correspondiente al parámetro formal G de la clase C.
Ejemplo:

```
class A[G,H] ... end

class B
    ...
    attributes
        a:A[Integer,Real];
        ...
end
```

Si se aplica la primitiva *DeleteRealParameter(A,B,H)* se obtiene:

```
class B
    ..
    attributes
        a:A[Integer];
        ...
end
```

SubstituteFormalParameter(C,G,T) dada una clase C, un parámetro formal G y un nuevo tipo T completamente instanciado, se define como el resultado de sustituir en todas las reducciones de la regla de producción 23 en la clase C, la sustitución de FORMAL_GEN_ID correspondiente a G por el CLASS_TYPE T.
Ejemplo:

```
class A[G]
    signatures
        attributes
            a:G;
        methods
            f(x:G):G is do ... end;
        ...
...
end
```

Si se aplica la primitiva *SubstituteFormalParameter(A,G,Integer)* se obtiene:

```
class A[G] -- pendiente de aplicar DeleteFormalParameter(A,G)
    signatures
        attributes
            a:Integer;
        methods
            f(x:Integer):Integer is do ... end;
        ...
...
end
```

Primitivas en Acotación por Subtipado y Conformidad

ReplaceBoundType(C,G,T) dada una clase C con un parámetro formal G acotado por subtipado o conformidad con V, se define como el resultado de reemplazar V por T como sustitución de CLASS_TYPE en la regla 48S.
Ejemplo:

```
class A[G -> Number] ... end
```

Si se aplica la primitiva *ReplaceBoundType(A,G,Integer)* se obtiene:

```
class A[G -> Integer] ... end
```

Primitivas en Acotación por Cláusulas *where*

AddWhereClause(C,G,W) dada una clase C, un parámetro formal G y un conjunto de tipos funcionales W, se define como la inclusión de la cláusula

where W para ese parámetro formal. Se corresponde con la modificación de la regla 49W correspondiente a dicho parámetro genérico G.
Ejemplo:

```
class A[G,H]
    where H has H methodZ(b:H)
    signatures
        attributes
            a:G;
            b:H;
        ...
...
end
```

Si se aplica la primitiva *AddWhereClause(A,G,W)* donde:

$W = \{methodX : NONE \rightarrow G, methodY : G \rightarrow NONE\}$

se obtiene:

```
class A[G,H]
    where G has G methodX; methodY(a:G),
          H has H methodZ(b:H)
    signatures
        attributes
            a:G;
            b:H;
        ...
...
end
```

AddSignatureInWhereClause(C,G,F) dada una clase C y un parámetro formal G sobre el que existe acotación por cláusulas **where**, se define como la inclusión del tipo funcional F en la cláusula **where** de ese parámetro formal. Se corresponde con la inclusión en la sustitución en la regla 50W, del tipo funcional F en la lista de signaturas para dicho parámetro genérico G.
Ejemplo:

```
class A[G,H]
    where G has G methodX,
          H has H methodZ(b:H)
    signatures
        attributes
            a:G;
            b:H;
        ...
...
end
```

Si se aplica la primitiva *AddSignatureInWhereClause(A,G,F)* donde:

$F = methodY : G \rightarrow NONE$

se obtiene:

```
class A[G,H]
    where G has G methodX; methodY(a:G),
          H has H methodZ(b:H)
    signatures
        attributes
            a:G;
            b:H;
        ...
...
end
```

4.3. Definición de Refactorizaciones de Especialización en cuanto a Genericidad

El presente apartado presenta un conjunto de refactorizaciones de especialización que afectan a aspectos vinculados con la definición de clases genéricas. Para ello se definirán los elementos de cada una de las refactorizaciones, siguiendo la estructura ya planteada en [Fow00] [TB99].

El conjunto de refactorizaciones definidas está formado por:

- ***Specialize Bound S***
- ***Specialize Bound F***
- ***Replace Formal Parameter with Type***
- ***Replace Formal Parameter with Type Bound***
- ***Add Where Clause***
- ***Add Signature in Where Clause***

Dado que la variante de acotación por conformidad en MOON puede ser vista como un tipo particular de acotación por subtipado, las refactorizaciones **Specialize Bound S**, **Replace Formal Parameter with Type** y **Replace Formal Parameter with Type Bound** pueden aplicarse en el caso de la variante de acotación por conformidad en MOON, cambiando el concepto de subtipado por el de conformidad, como se definió previamente.

Specialize Bound S

Descripción La refactorización consiste en especializar el tipo de la acotación de un parámetro formal en una declaración de clase genérica por un subtipo (Figura 8). Esto es sólo aplicable en aquellos lenguajes que permitan acotación por subtipado.

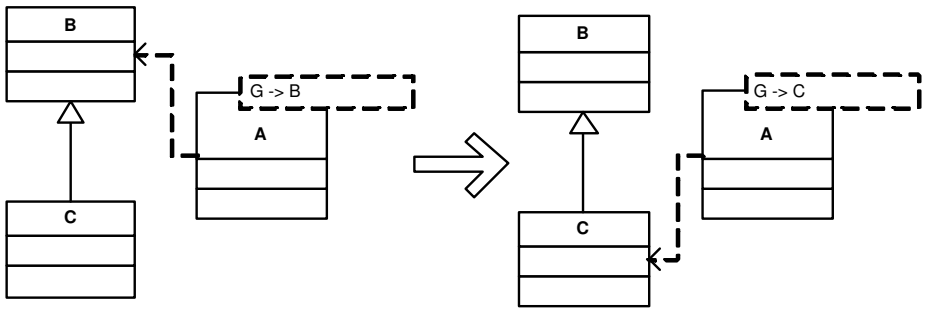

Figura8. Specialize Bound S

Motivación En genericidad con acotación mediante subtipado, se limitan las instanciaciones genéricas válidas, a los subtipos del tipo con el que se acota y se restringe el conjunto de propiedades utilizadas a través de entidades declaradas con el parámetro genérico G de la acotación. En la práctica se puede encontrar que el conjunto de sustituciones reales de las instanciaciones genéricas se limita a un subconjunto de descendientes.

Si se elimina la actual acotación, y se sustituye por el ancestro común a las clases que se están utilizando como parámetros reales, se pierde genericidad, pero se gana claridad respecto al conjunto de sustituciones reales, así como la posibilidad futura de modificar el código con las propiedades intrínsecas definidas en el nuevo tipo de la acotación. Esta refactorización se podría aplicar a la hora de acotar parámetros no acotados inicialmente, en los que la cota es el tipo universal, permitiendo aplicarse esta transformación a lenguajes que inicialmente no incorporaban acotación.

Entradas Una clase C, un parámetro genérico formal G y un nuevo tipo como acotación A.

Precondiciones Para poder llevar a cabo la refactorización ***SpecializeBound-S(C,G,A)*** se deben cumplir:

1. G es un identificador genérico formal en la clase C.

   $$G \in FormalParam(C)$$

   Análisis de la precondición: el identificador de parámetro formal que se ha pasado como parámetro debe pertenecer a la lista de parámetros de la clase genérica.
2. Todas las sustituciones completas al parámetro genérico G son subtipo de A.

   $$\forall T \in SubstFormalPar(C, G) \Rightarrow T \in Subtype(A)$$

Análisis de la precondición: todas las sustituciones al parámetro G de la clase C en sus instanciaciones genéricas, deben ser subtipo del tipo por el que se quiere sustituir en la acotación. Esto asegura que las sustituciones una vez realizada la refactorización siguen siendo correctas.

3. El tipo A es un subtipo de la acotación del parámetro formal.

$$BoundType(C, G) = C'/A \in Subtype(C')$$

Análisis de la precondición: se asegura que las la operación de refactorización se realiza en el sentido de la especialización acotando por un subtipo de la acotación inicial.

4. La acotación del parámetro formal en los descendientes de las clase genérica C es un subtipo de A.

$$\forall C' \in Desc(C)/Bound(C', G) = T \wedge IsCompleted(T) \Rightarrow T \in Subtype(A)$$

Análisis de la precondición: se asegura que las acotaciones en las clases descendientes son correctas en tipos respecto a la nueva acotación.

Acciones

1. $ReplaceBoundType(C, G, A)$

Postcondiciones

1. La acotación actual del parámetro G es A

$$BoundType(C, G) = A$$

Ejemplo Teniendo la clase genérica declarada de la forma:

$$A[G \rightarrow Comparable]$$

Si en las instanciaciones genéricas, de la clase genérica $A[G]$ de la forma $A[X]$ con $X \in SubstFormalPar(A, G)$, se encuentra un ancestro común subtipo de $Comparable$, se puede llevar a cabo la refactorización.

En nuestro código se puede tener: $A[Integer], A[Number]$, etc. Suponiendo que al analizar el código se tiene:

$$SubstFormalPar(A, G) = \{Integer, Number, Double\}$$

la refactorización nos llevaría a declarar:

$$A[G \rightarrow Number]$$

puesto que *Number* es ancestro común a todos los elementos del conjunto.

Specialize Bound F

Descripción La refactorización consiste en especializar el tipo de la acotación del parámetro formal en una declaración de clase genérica con acotación F, sustituyendo la declaración con acotación F por una acotación por subtipado, con un tipo completamente instanciado como acotación (Figura 9). Esto es sólo aplicable en aquellos lenguajes que permitan genericidad con acotación F.

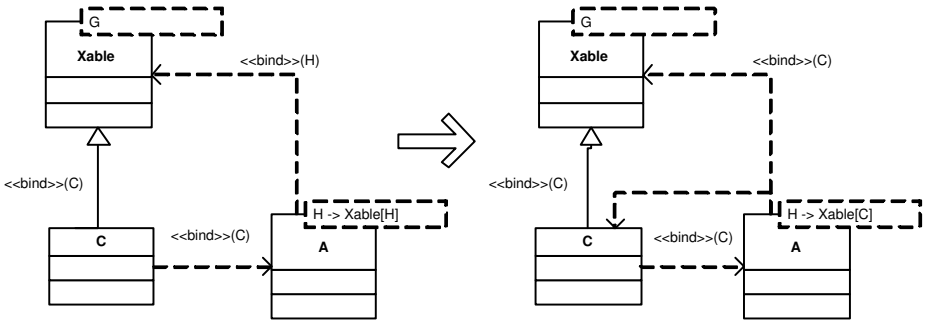

Figura9. Specialize Bound F

Motivación En genericidad con acotación F, se puede encontrar el caso de que todas las instanciaciones genéricas toman como sustitución real un único tipo. En este caso se puede sustituir la acotación F por subtipado, simplificando la complejidad de la acotación F pasando al caso particular de acotación por subtipado.

La refactorización puede ser especialmente útil en la transformación hacia un lenguaje que no soporte acotación F, pero sí soporte acotación por subtipado (e.g. EIFFEL).

Entradas Una clase C, un parámetro genérico formal G y un nuevo tipo como acotación A.

Precondiciones Para poder llevar a cabo la refactorización ***SpecializeBound-F(C,G,A)*** se deben cumplir:

1. El tipo G es un identificador genérico formal con acotación F (de la forma $B[G]$) en la clase C.

   $$G \in FormalParam(C) \wedge BoundType(C, G) = B[G]$$

   Análisis de la precondición: para poder iniciarse la refactorización es necesario que sobre el parámetro genérico exista acotación F con respecto a una clase genérica B.

2. Todas las instanciaciones genéricas de C tiene un único tipo común completamente instanciado como sustitución a G.

$$SubstFormalPar(C, G) = A$$

Análisis de la precondición: se asegura que las sustituciones al parámetro genérico son todas iguales, de tal forma que se podrá modificar la acotación en función del tipo A sin afectar al resto de instanciaciones genéricas.

3. La acotación del parámetro formal en los descendientes de las clase genérica C es un subtipo de $B[A]$.

$$\forall C' \in Desc(C)/BoundType(C', G) = T \wedge IsCompleted(T) \Rightarrow T \in Subtype(B[A])$$

Análisis de la precondición: se asegura que al cambiar la acotación, ésta sigue siendo correcta respecto a la declaración de acotación en los descendientes de la clase C si existen.

Acciones

1. $ReplaceBoundType(C, G, B[A])$

Postcondiciones

1. La acotación actual del parámetro G es $B[A]$:

$$BoundType(C, G) = B[A]$$

Ejemplo Teniendo la clase genérica declarada de la forma:

$$C[G \rightarrow Comparable[G]]$$

Si en las instanciaciones genéricas, de la clase genérica C de la forma $C[A]$ con $SubstFormalParameter(C, G) = A$, siendo A el único tipo que se utiliza como sustitución, se puede llevar a cabo la refactorización.

En nuestro código se puede tener que todas las sustituciones son del tipo $C[Integer]$ donde $C[Integer]$ hereda de $Comparable[Integer]$, la refactorización nos llevaría a declarar:

$$C[G \rightarrow Comparable[Integer]]$$

Replace Formal Parameter with Type

Descripción La refactorización consiste en eliminar un parámetro formal de la definición de una clase genérica, sustituyendo en el cuerpo de la clase el uso del parámetro formal como tipo por un tipo completamente instanciado (Figura 10).

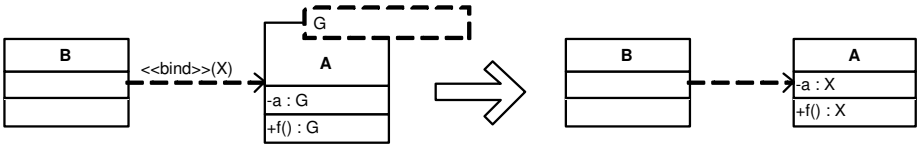

Figura10. Replace Formal Parameter with Type

Motivación La refactorización surge cuando dada una clase genérica con acotación por subtipado, la sustitución real para uno de sus parámetros formales es siempre el mismo tipo.

Como solución se plantea la eliminación del parámetro formal en la declaración de la clase, común en todas las instanciaciones, sustituyendo el tipo genérico con el tipo completamente instanciado en el cuerpo de la clase genérica.

Esta refactorización afecta a los clientes directos y descendientes directos de la clase puesto que la declaración de tipo que referencia a la clase, cambia al eliminarse un parámetro genérico formal. Esto permite ver la diferencia entre las refactorizaciones que afectan siempre a varias clases, frente a las primitivas de edición que afectan a la estructura de una única clase.

Entradas Una clase genérica C, un parámetro formal G y un nuevo tipo como sustitución A.

Precondiciones Para poder llevar a cabo la refactorización ***ReplaceFormalParameterWithType(C,G,A)*** se deben cumplir:

1. El tipo G es un identificador genérico formal en la clase C.

   $$G \in FormalParam(C)$$

   Análisis de la precondición: debe existir un parámetro genérico formal G en la clase C para poder iniciarse la refactorización.
2. Todas las instanciaciones genéricas de C tiene un único tipo común completamente instanciado como sustitución a G.

   $$SubstFormalPar(C, G) = A$$

   Análisis de la precondición: toda sustitución al parámetro es un único tipo. Esto posibilita que se pueda eliminar la genericidad en la clase, sustituyendo por el tipo, sin necesidad de utilizar ningún mecanismo de conversión (*cast*).

Acciones

1. $DeleteFormalParameter(C, G)$
2. $SubstituteFormalParameter(C, G, A)$
3. $\forall T \in \mathcal{T} / CD(T) = C \Rightarrow DeleteRealParameter(B, C, G)$

Postcondiciones

1. La clase C no tiene parámetro formal G:

$$G \notin FormalParam(C)$$

Ejemplo Teniendo la clase genérica declarada de la forma $C[G_1, ..., G_i, ..., G_n]$ donde todas las instanciaciones genéricas, de la clase genérica son de la forma:

$$C[X_1, ..., Integer, ..., X_n].$$

La refactorización nos llevaría a reescribir la nueva clase C sustituyendo las declaraciones de tipo G_i por el tipo *Integer* eliminando el parámetro formal en la declaración de la clase:

```
class C[G,H]

    attributes
        a:G
...
end
```

tras realizar la refactorización *ReplaceFormalParameterWithType(C,G,Integer)* se tiene:

```
class C[H]
    ...
    attributes
        a:Integer
...
end
```

Replace Formal Parameter with Type Bound

Descripción La refactorización consiste en eliminar un parámetro formal de la definición de una clase genérica, sustituyendo en todo uso del parámetro formal como declaración de tipo por el tipo acotación de dicho parámetro (Figura 11). Como caso particular se tiene el caso de genericidad no restringida, donde la acotación es el tipo universal ***Object*** y se obraría de igual manera.

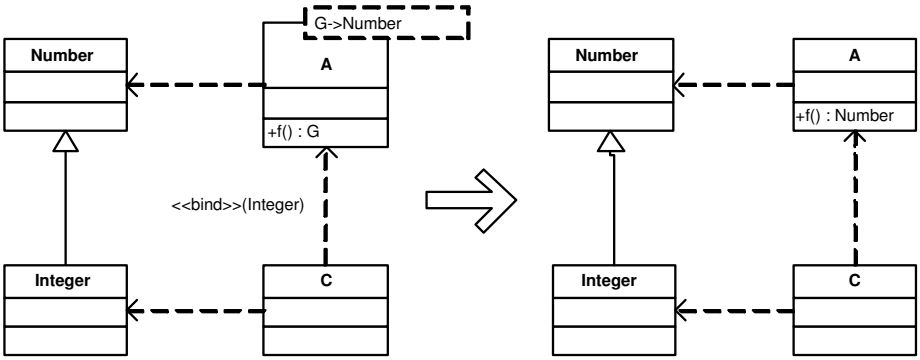

Figura11. Replace Formal Parameter with Type Bound

Motivación La refactorización surge cuando dada una clase genérica con acotación se plantea la eliminación del parámetro genérico por el tipo acotación. Se consigue la eliminación de genericidad en la clase y una solución al problema para facilitar la integración de código legado no genérico (e.g. *raw types* en la propuesta GJ [BOSW98]).

Esta refactorización afecta a los clientes directos y descendientes directos de la clase puesto que la declaración de tipo que referencia a la clase, cambia al eliminarse un parámetro genérico formal. Además en esta refactorización, los clientes estarán obligados al uso de conversión de tipos (*downcast*), con el objetivo de la simulación del polimorfismo paramétrico con polimorfismo de inclusión.

El problema que se plantea es que se elimina el polimorfismo parámetrico por el polimorfismo de inclusión. Como ya se señaló en trabajos anteriores [Cre00], esto conlleva el uso de mecanismos de conversión propios del lenguaje como el intento de asignación '?=' en EIFFEL o *type casts* en Java o C++. El tipo del objeto que se asigna es el tipo dinámico de la expresión en la parte derecha de la asignación, y no el tipo estático (asignación de la forma ':='). Por lo tanto se comprueba su corrección en ejecución. Para que la operación sea correcta se debe cumplir que el tipo dinámico que se intenta asignar sea un subtipo del tipo estático de la parte izquierda de la expresión, según la definición de subtipado en las variantes de MOON.

En el caso de EIFFEL, el intento fallido, por incompatibilidad de tipos, no provoca excepciones (asignación a valor nulo por defecto), mientras que en JAVA se produce el lanzamiento de excepciones.

Sin embargo en una primera propuesta del lenguaje minimal se eliminó de la sintaxis cualquier regla de producción con este fin. Para poder llevar a cabo este tipo de refactorización, es necesario introducir una modificación a la gramática de MOON para permitir el intento de asignación.

En primer lugar se modifica, ampliando, la regla 32, añadiendo el intento de asignación como instrucción:

32 INSTR $\triangleq$ COMPOUND_INSTR | CREATION_INSTR |
ASSIGNMENT_INSTR | CALL_INSTR |
INTENTION_ASSIGMENT_INSTR

En segundo lugar introducimos la sintaxis propia de la instrucción de intento de asignación:

52 INTENTION_ASSIGMENT_INSTR $\triangleq$ VAR_ID '?=' EXPR

El resultado de la operación resultante, cuando no se pueda realizar correctamente la operación, será propio en cada lenguaje. Por simplificación, al no existir el concepto de excepción en MOON, se entiende que si la asignación no se produce adecuadamente, se asigna el valor nulo o vacío.

Entradas Una clase genérica C y un parámetro genérico formal G acotado.

Precondiciones Para poder llevar a cabo la refactorización ***ReplaceFormalParameterWithTypeBound(C,G)*** se deben cumplir:

1. El tipo G es un identificador genérico formal en la clase C acotado explícitamente por un tipo X o bien por el tipo universal ***Object***, que pasa a sustituir a X.

$$G \in FormalParam(C) \wedge (BoundType(C,G) = X \vee BoundType(C,G) = \textbf{Object})$$

   Análisis de la precondición: debe existir un parámetro genérico formal G en la clase C con acotación para poder iniciarse la refactorización.

Acciones

1. $DeleteFormalParameter(C,G)$
2. $SubstituteFormalParameter(C,G,X)$
3. $\forall T \in \mathcal{T}/CD(T) = C \Rightarrow DeleteRealParameter(B,C,G)$

Postcondiciones

1. La clase C no tiene parámetro formal G:

$$G \notin FormalParam(C)$$

Ejemplo Teniendo la clase genérica declarada de la forma:
$C[G_1, ..., G_i \rightarrow Number, ..., G_n]$
donde todas las instanciaciones genéricas, de la clase genérica son de la forma:

$$C[X_1, ..., Number, ..., X_n]$$

o subtipos de $Number$ de acuerdo a la acotación.

La refactorización lleva a reescribir la nueva clase C sustituyendo las declaraciones de tipo G_i por el tipo *Number* eliminando el parámetro formal en la declaración de la clase.

```
class A[G->Number]
    attributes
        a:G
    methods
        f(x:G);
        g:G;
...
end

class Cliente
    methods
        main
            a:A[Integer];
            b:Integer;
        do
            create b;
            a.f(b);
            ...
            b:=a.g;
        end
    ...
end
```

Tras realizar la refactorización *ReplaceFormalParameterWithTypeBound(A,G)* se obtiene:

```
class A
    ...
    attributes
        a:Number
    methods
        f(x:Number);
        g:Number;
        ...
end

class Cliente
    methods
        main
```

```
            a:A;
            b:Integer;
        do
          create b;
          a.f(b);
          ...
          b ?= a.g; --mecanismo de intento de asignacion
        end
        ...
end
```

Add Where Clause

Descripción La refactorización consiste en añadir una nueva cláusula ***where*** como acotación a uno de los parámetros formales de la clase (Figura 12).

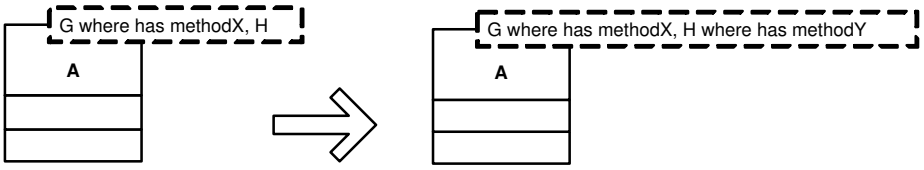

Figura12. Add Where Clause

Motivación Cuando todas las clases dadas como sustitución al parámetro formal tienen en común un conjunto de signaturas, se puede añadir como acotación de dicho parámetro una cláusula **where** que agrupe el conjunto de signaturas comunes. Esto limita el conjunto de sustituciones reales a dicho parámetro especializando la clase, permitiendo el uso futuro de las características añadidas y mejorando la comprensión de la solución dada.

Entradas Una clase genérica C, un parámetro formal G y un conjunto de signaturas que forman una acotación por cláusula **where** W.

Precondiciones Para poder llevar a cabo la refactorización ***AddWhereClause(C,G,W)*** se deben cumplir:

1. El tipo G es un identificador genérico formal en la clase C.

$$G \in FormalParam(C)$$

Análisis de la precondición: para poder iniciarse la refactorización el parámetro G debe estar definido en la clase.

2. El identificador genérico G en la clase C no está acotado.

$$BoundSigs(C, G) = \emptyset$$

Análisis de la precondición: para poder acotar el parámetro con una nueva cláusula **where** es necesario que el parámetro G no esté acotado explícitamente.

3. Para toda signatura s en la cláusula **where** W, existe un método en cada clase que es sustitución, tal que su signatura es igual a sustituir el parámetro genérico G por la clase en la signatura s.

$$\forall s \in W \Rightarrow \exists m \in Methods(C') \wedge C' \in SubstFormalPar(C, G)/SignatureWithSubstitution(s, G, C') = Signature(m)$$

Análisis de la precondición: en cada una de las sustituciones completas del parámetro G en las instanciaciones genéricas, se debe cumplir que la clase tiene un conjunto de métodos cuya signatura es equivalente a la sustitución del tipo determinante de la clase por el parámetro genérico. Si esto no se cumple, la clase no sería una sustitución correcta por no tener algún método cuya signatura se corresponda a la nueva acotación por cláusula **where** introducida.

Acciones

1. $AddWhereClause(C, G, W)$

Postcondiciones

1. El parámetro G tiene una lista de signaturas como acotación igual a W.

$$BoundSigs(C, G) = W$$

Ejemplo Si se tiene la declaración de una clase A genérica con parámetros genéricos formales G y H:

```
class A[G,H]
    where H has methodZ(b:H):H
    signatures
        attributes
            a:G;
            b:H;
        ...
... end
```

Suponiendo que las instanciaciones genéricas de A[G,H] son de la forma A[C_1,D_1] y A[C_2, D_2], donde:

- $\exists m_1 \in Methods(C_1)/Signature(m_1) = methodX : NONE \rightarrow C_1$
- $\exists m2 \in Methods(C_2)/Signature(m_2) = methodX : NONE \rightarrow C_2$
- $\exists n_1 \in Methods(C_1)/Signature(n_1) = methodY : C_1 \rightarrow NONE$
- $\exists n2 \in Methods(C_2)/Signature(n_2) = methodY : C_2 \rightarrow NONE$

Se puede acotar el parámetro genérico G con:

$W = \{methodX : NONE \rightarrow G; methodY : G \rightarrow NONE\}$

Al aplicar la refactorización de añadir la cláusula **where** W al parámetro genérico formal G (*AddWhereClause(A,G,W)*) se obtiene:

```
class A[G,H]
    where G has methodX:G; methodY(a:G),
          H has methodZ(b:H):H
    signatures
        attributes
            a:G;
            b:H;
        ...
end
```

Add Signature in Where Clause

Descripción La refactorización consiste en añadir a la acotación por cláusulas ***where*** de uno de los parámetros genéricos formales, una nueva signatura de método (Figura 13).

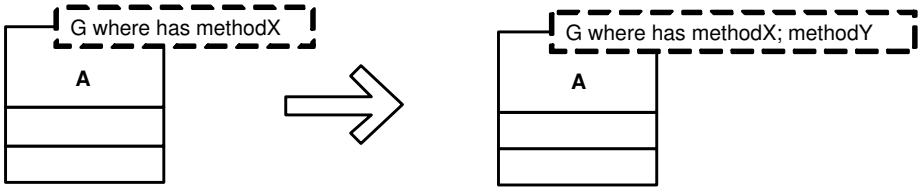

Figura13. Add Signature in Where Clause

Motivación Cuando todas las clases dadas como sustitución a un parámetro formal tienen en común un método, se puede añadir a la acotación de dicho parámetro en la cláusula **where** dicho método. Esto limita el conjunto de sustituciones reales a dicho parámetro permitiendo el uso futuro de esa propiedad a través de las entidades cuya declaración de tipo es el parámetro formal.

Entradas Una clase genérica C, un parámetro formal G y una signatura de método s.

Precondiciones Para poder llevar a cabo la refactorización ***AddSignatureIn-WhereClause(C,G,s)*** se debe cumplir:

1. El tipo G es un identificador genérico formal en la clase C.

$$G \in FormalParam(C)$$

   Análisis de la precondición: para poder iniciarse la refactorización el parámetro G debe estar definido en la clase.
2. Para la signatura s añadida en la cláusula **where** del parámetro G, existe un método en cada clase sustitución del parámetro formal tal que la signatura resultante de sustituir la clase a la que pertenece por el parámetro genérico, pertenece a la acotación.

$$\exists m \in Methods(C) \wedge C \in SubstFormalPar(C,G)/SignatureWithSubstitution(s,G,C) = Signature(m)$$

   Análisis de la precondición: en cada una de las sustituciones completas del parámetro G en las instanciaciones genéricas, se debe cumplir que la clase tiene un método cuya signatura es equivalente a la sustitución del parámetro genérico por el tipo determinante de la clase. Si esto no se cumple, la clase no sería una sustitución correcta por no tener ningún método cuya signatura se corresponda a la nueva signatura introducida en la acotación.

Acciones

1. *AddSignatureInWhereWhere(A,G,F)*

Postcondiciones

1. El tipo funcional F está en la lista de acotaciones del parámetro G en la clase C.

$$F \in BoundSigs(C,G)$$

Ejemplo Si se tiene la declaración de una clase A genérica con parámetros G y H:

```
class A[G,H]
    signatures
    where G has methodY(a:G),
          H has methodZ(b:H):H
        attributes
            a:G;
            b:H;
        ...
... end
```

Suponiendo que las instanciaciones genéricas de A[G,H] son de la forma A[C_1,D_1] y A[C_2, D_2], donde:

- $\exists m_1 \in Methods(C_1)/Signature(m_1) = methodX : NONE \rightarrow C_1$
- $\exists m2 \in Methods(C_2)/Signature(m_2) = methodX : NONE \rightarrow C_2$

Al aplicar la refactorización de añadir en la cláusula **where** el método methodX en el parámetro G de la clase A se obtiene:

```
class A[G,H]
    where G has methodX:G ; methodY(a:G),
          H has methodZ(b:H):H
    signatures
        attributes
            a:G;
            b:H;
        ...
...
end
```

4.4. Conclusiones a la Definición de Refactorizaciones en cuanto a Genericidad

Desde el punto de vista del soporte que proporciona MOON de cara a la definición de refactorizaciones sobre aquellos aspectos vinculados a los tipos paramétricos y clases genéricas se ha podido observar, en el desarrollo de esta sección, como el conjunto de predicados necesarios en la definición de refactorizaciones puede ser definido directamente sobre la información contenida en las clases escritas en lenguaje MOON.

Esto lleva a concluir que el catálogo de refactorizaciones puede ser definido por completo sobre MOON sin necesidad de dejar parte de la implementación posterior dependiente del lenguaje concreto utilizado.

La conclusíión es que el esfuerzo de definición de las refactorizaciones de especialización en cuanto a genericidad es aprovechado (suponiendo la existencia de un soporte) independientemente del lenguaje utilizado, siempre que éste incluya en su especificación tipos paramétricos como en el caso de Eiffel o C++, o bien se vaya a incluir en un futuro más o menos cercano, como en los casos de Java[9] y C#.

5. Conclusiones y Líneas de Trabajo Futuro

5.1. Conclusiones

Partiendo del estudio realizado en el **Sección 2. Trabajos Relacionados**, se ha intentado enfocar el desarrollo del trabajo en la línea de validar el soporte

[9] El anuncio de la inclusión de genericidad en Java 1.5, para finales del año 2003 es un hecho. Se puede obtener información en http://java.sun.com

de la definición de refactorizaciones de especialización sobre el lenguaje modelo MOON, para su reutilización en el contexto de lenguajes orientados a objetos estáticamente tipados.

Mediante la definición de refactorizaciones sobre dicho lenguaje modelo, se abre la posibilidad a la implementación de las refactorizaciones y reutilización del esfuerzo realizado, con independencia del lenguaje concreto con el que se trabaja.

De dicho estudio se ha verificado que se pueden definir refactorizaciones, dejando el cálculo diferido de ciertos predicados a la información concreta que se obtiene de cada lenguaje (e.g. niveles de acceso a propiedades).

Por otro lado, la imposibilidad de definir sobre el lenguaje refactorizaciones dependientes del flujo de control, motivada por la ausencia de dicha información en el lenguaje, impide la definición de refactorizaciones de este tipo con independencia del lenguaje.

Sin embargo, en la definición de refactorizaciones de especialización en cuanto a genericidad, se observa que la información necesaria en el conjunto de predicados que conforman las refactorizaciones, se encuentra recogida por completo en el lenguaje modelo. Las posibilidades de reutilización de la definición de refactorizaciones en este sentido animan a continuar en esta línea.

Concluyendo, en el desarrollo del presente trabajo se han llegado a los siguientes resultados:

- Definición de un conjunto de predicados sobre el lenguaje modelo MOON que recogen la información de los elementos estructurales y relaciones entre éstos.
- Definición de un conjunto de primitivas que operan directamente sobre el código en base a la gramática de MOON.
- Identificación de predicados cuya implementación concreta se debe diferir a una instanciación concreta sobre un lenguaje.
- Detección de refactorizaciones no definibles a partir de la información contenida en el lenguaje modelo.
- Propuesta de extensión y modificación a la gramática original del lenguaje modelo MOON para soportar conceptos como referencia paterna (*super*) y conversiones de tipos (*cast*).
- Inicio de definición de un catálogo de refactorizaciones de especialización en cuanto a genericidad.
- Validación del soporte de refactorizaciones de especialización sobre el lenguaje modelo MOON para su posterior soporte sobre frameworks.

5.2. Líneas de Trabajo Futuras

Dentro de las líneas de trabajo abiertas en el desarrollo del trabajo surgen varias líneas a continuar:

- Estudio de refactorizaciones, no sólo de especialización, sino en otras direcciones como la generalización, tomando otras refactorizaciones de catálogos como [Opd92] y [Fow00].

- Estudio de nuevas soluciones a la hora de la definición de refactorizaciones, y otras técnicas aplicadas en la reestructuración de programas.
- Profundizar en el análisis de las refactorizaciones propuestas, desde el punto de vista de:
  - Preservación del comportamiento
  - Corrección de tipos resultante
  - Consecuencias en los clientes
  - Consecuencias en los objetos
  - Método (bajo demanda o por inferencia)
  - Intervención humana frente ejecución automática
- Definición de nuevas refactorizaciones en cuanto a genericidad ampliando el conjunto inicial planteado.
- Integración de la definición de refactorizaciones dada sobre un soporte basado en frameworks.

```
1   MODULE              ≜ CLASS_DEF
2   CLASS_DEF           ≜ [deferred] class CLASS_NAME HEADER SIGNATURES CLASS_BODY end
3   CLASS_NAME          ≜ CLASS_ID [FORMAL_PARAMETERS]
4   HEADER              ≜ INHERITANCE_LIST
5   SIGNATURES          ≜ signatures SIG_LIST
6   CLASS_BODY          ≜ body { METHODS_IMPL ';' ... }
7   FORMAL_PARAMETERS   ≜ '[' { FORMAL_PAR ',' ... }+ ']' [ BOUND_W ]          variante w
8   FORMAL_PAR          ≜ FORMAL_GEN_ID [ BOUND_S ]                            variante s
9   INHERITANCE_LIST    ≜  { INHERITANCE_CLAUSE ';' ... }
10  INHERITANCE_CLAUSE  ≜ inherit CLASS_TYPE OPLUS                             hereda
11  OPLUS               ≜ { MODIFIER ';' ... }
12  MODIFIER            ≜ rename PROP_ID as PROP_ID | redefine PROP_ID | makedeferred PROP_ID |
                          makeeffective PROP_ID | select PROP_ID
13  SIG_LIST            ≜ ATTRIBUTE_DECS METHOD_DECS
14  ATTRIBUTE_DECS      ≜ attributes { ATT_DEC ';' ... }
15  METHOD_DECS         ≜ methods { METH_DEC ';' ... }
16  ATT_DEC             ≜ VAR_DEC
17  METH_DEC            ≜ ROUTINE_DEC | FUNCTION_DEC
18  ROUTINE_DEC         ≜ WITHOUT_RESULT
19  FUNCTION_DEC        ≜ WITHOUT_RESULT ':' TYPE                              es cliente
20  WITHOUT_RESULT      ≜ [deferred] METHOD_ID [FORMAL_ARGUMENTS]
21  FORMAL_ARGUMENTS    ≜ '(' { VAR_DEC ',' ... }+ ')'
22  VAR_DEC             ≜ VAR_ID ':' TYPE                                      es cliente
23  TYPE                ≜ FORMAL_GEN_ID | CLASS_TYPE
24  CLASS_TYPE          ≜ CLASS_ID [REAL_PARAMETERS]                           ¿de qué clases?
25  REAL_PARAMETERS     ≜ '[' { TYPE ',' ... }+ ']'                            ¿de qué clases?
26  METHOD_IMPL         ≜ NONE_DEFERRED_R | NONE_DEFERRED_F
27  NONE_DEFERRED_R     ≜ MSIG [LOCAL_DECS] METHOD_BODY
28  NONE_DEFERRED_F     ≜ MSIG ':' TYPE [LOCAL_DECS] METHOD_BODY
29  LOCAL_DECS          ≜ { VAR_DEC ';' ... }+
30  METHOD_BODY         ≜ do INSTR end
31  MSIG                ≜ METHOD_ID [FORMAL_ARGUMENTS]
32  INSTR               ≜ COMPOUND_INSTR | CREATION_INSTR |
                          ASSIGNMENT_INSTR | CALL_INSTR      no hay instrucciones de control
33  COMPOUND_INSTR      ≜ {INSTR ';' ... }
34  CREATION_INSTR      ≜ create VAR_ID
                          ^
```

Los símbolos BOUND_S y BOUND_W que aparecen en las reglas anteriores, dependen de qué variante de acotación de parámetros genéricos formales esté en consideración. Las reglas correspondientes a estos símbolos se definen a continuación de acuerdo a la variante de lenguaje a la que se dará lugar.

Cuando la forma de acotación de los parámetros genéricos formales que se defina para el lenguaje esté dada por acotación mediante subtipado o conformidad, las reglas que se utilizan son las marcadas con S (regla 48S y 49S). Cuando la forma de acotación de los parámetros genéricos formales que se defina para el lenguaje esté dada por acotación mediante cláusulas tal que, las reglas que se utilizan son las marcadas con W (regla 48W, 49W y 50W).

35	ASSIGNMENT_INSTR	≙ VAR_ID ':=' EXPR	
36	CALL_INSTR	≙ CALL_INSTR_LONG1 | CALL_INSTR_LONG2	*no hay llamadas en cascada*
37	CALL_INSTR_LONG1	≙ METHOD_ID [REAL_ARGUMENTS]	
38	CALL_INSTR_LONG2	≙ CALL_EXPR_LONG1'.'CALL_INSTR_LONG1	
39	EXPR	≙ MANIFEST_CONSTANT | CALL_EXPR	*se reducen las formas de expresiones*
40	MANIFEST_CONSTANT	≙ **nil** | REAL_CONSTANT | INTEGER_CONSTANT | BOOLEAN_CONSTANT | CHAR_CONSTANT | STRING_CONSTANT	
41	CALL_EXPR	≙ CALL_EXPR_LONG1 | CALL_EXPR_LONG2	*no hay llamadas en cascada*
42	CALL_EXPR_LONG1	≙ ENTITY [REAL_ARGUMENTS]	
43	CALL_EXPR_LONG2	≙ ENTITY'.'CALL_EXPR_LONG1	
44	REAL_ARGUMENTS	≙ '(' {EXPR_ATOM ','...}+ ')'	
45	EXPR_ATOM	≙ MANIFEST_CONSTANT | CALL_EXPR_LONG1	
46	ENTITY	≙ VAR_ID | **result** | **self**	
47	PROP_ID	≙ VAR_ID | METHOD_ID	

48S	BOUND_S	≙ '->'CLASS_TYPE	*es cliente*
48W	BOUND_S	≙ ϵ	
49S	BOUND_W	≙ ϵ	
49W	BOUND_W	≙ **where** { WHERE_CLAUSE ','...}+	
50W	WHERE_CLAUSE	≙ FORMAL_GEN_ID **has** SIG_LIST	

Referencias

[AGH01] Ken Arnold, James Gosling, and David Holmes. *El Lenguaje de Programación JAVA*. Serie Java. Pearson Educación - Addison Wesley, 3rd edition, 2001.

[AM02] Gabriela Arévalo and Tom Mens. Analysing object oriented application frameworks using concept analysis. In *OOIS 2002, MAnaging SPEcialization/Generalization HIerarchies (MASPEGHI) Workshop*, 2002.

[Bac98] David Francis Bacon. Fast and effective optimization of statically typed object-oriented languages. Technical Report CSD-98-1017, 5, 1998.

[BCK+01] Gilad Bracha, Norman Cohen, Christian Kemper, Steve Marx, Martin Odersky, Sven-Eric Panitz, David Stoutamire, Kresten Thorup, and Philip Wadler. Adding generics to the java programming language: Participant draft specification, 2001.

[BG96] David Binkey and Keith Brian Gallagher. *Advances in Computers*, volume 43, chapter Program Slicing. Academic Press San Diego, 1996.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Gj: Specification, 1998.

[Cas90] Eduardo Casais. Managing class evolution in object-oriented systems. Technical report, Centre Universitaire dt'Informatique, University of Geneve, 1990.

[Cas92] Eduardo Casais. *An Incremental Class Reorganization Approach*, volume LCNS 615, pages 114–132. 1992.

[Cas94] Eduardo Casais. *Automatic reorganization of object-oriented hierarchies: a case study*, pages 95–115. 1994.

[CN00] Mel Ó Cinnéide and PÑixon. Composite refactorings for java programs, proceedings of the workshop on formal techniques for java programs. In *Workshop on Formal Techniques for Java Programs, European Conference on Object-Oriented Programming*, 2000.

[Cre00] Yania Crespo. *Incremento del potencial de reutilización del software mediante refactorizaciones*. PhD thesis, 2000.

[DLT00] Stéphane Ducasse, Michele Lanza, and Sander Tichelaar. Moose: an extensible language-independent environment for reengineering object-oriented systems. In *Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000)*, 2000.

[DTS99] Serge Demeyer, Sander Tichelaar, and Patrick Steyaert. FAMIX 2.0 - the FAMOOS information exchange model. Technical report, 1999.

[Fow00] Martin Fowler. *Refactoring. Improving the Design of Existing Code*. Addison Wesley, 2000.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[KS01] Andrew Kennedy and Don Syme. Design and implementation of generics for the .net common language runtime. In ACM, editor, *ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*. Microsoft Research, ACM, 2001.

[Läm02] Ralf Lämmel. Towards Generic Refactoring. In *Proc. of Third ACM SIGPLAN Workshop on Rule-Based Programming RULE'02*, Pittsburgh, USA, October5 2002. ACM Press. 14 pages.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 2nd edition, 1997.

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, Urbana-Champaign, IL, USA, 1992.

[PCML00] Félix Prieto, Yania Crespo, José Manuel Marqués, and Miguel Ángel Laguna. Mecanos y análisis de conceptos formales como soporte para la construcción de frameworks. In *Actas JISBD'00, Valladolid, España*, November 2000.

[Rob99] Donald Bradley Roberts. *Practical Analysis for Refactoring*. PhD thesis, 1999.

[SRVRH99] Vijay Sundaresan, Chrislain Razafimahefa, Raja Vallée-Rai, and Laurie Hendren. Practical virtual method call resolution for java. Technical report, 1999.

[ST97] Gregor Snelting and Frank Tip. Reengineering class hierarchies using concept analysis. Technical Report RC 21164(94592)24APR97, IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA, 1997.

[TB99] Lance Tokuda and Don S. Batory. Evolving object-oriented designs with refactorings. In *Automated Software Engineering*, 1999.

[TCFR96] Frank Tip, Jong-Deok Choi, John Field, and G. Ramalingam. Slicing class hierarchies in c++. In *Conference on Object-oriented Programming Systems, Languages and Applications*, pages 179–197, 1996.

[Tic01] Sander Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, 2001.

[TLSS99] Frank Tip, Chris Laffra, Peter F. Sweeney, and David Streeter. Practical experience with an application extractor for Java. *ACM SIGPLAN Notices*, 34(10):292–305, 1999.