Trabajo de Grado

Evaluación de enfoques de implementación de lenguajes de dominio específico

Autor: Pablo Andrés Barrientos

Director: Dr. Pablo E. Martínez López

Agradecimientos

Quisiera mencionar en primer lugar a mi familia: Ruben, María Teresa, Anabelia, Guillermina y María Sabina, por el apoyo continuo y silencioso y por aguantarme cuando estaba demasiado metido "en mis cosas" durante toda mi carrera.

Agradezco a toda la gente que conforma el Laboratorio de Investigación y Formación en Informática Avanzada (LIFIA), por lo bien que me sentí trabajando ahí. Quisiera agradecer especialmente a Mariano y Gastón, excelentes compañeros del grupo de investigación, que estuvieron siempre a mi lado.

Sin esperanzas de que puedan entender el castellano, pero con la intención de que estén presentes aquí, quiero mencionar a Marjan Mernik, Tomaž Kosar y Damijan Rebernak, con quienes he estado investigando sobre los lenguajes de dominio específico.

Quisiera agradecer a Pablo E. Martínez López (Fidel), por sus consejos, tanto para esta tesis como a lo largo de mi licenciatura. También darle las gracias por escuchar siempre todo lo que le quise decir o hacerle conocer y por su sinceridad. En él encontré un excelente docente con verdadera vocación, que me motivó en las pequeñas tareas docentes que he emprendido.

Agradezco a quienes me inspiraron, estuvieron a mi lado y/o me dieron fuerzas (quizás sin saberlo) durante mi carrera: Hernán Badenes (H) y Esteban de la Canal (Steve), Andrés Fortier(@) y Juan Cappi, Mariano Montone y Gastón Fournier (de nuevo), Gabriel Videla, David Ducos, Valeria De Cristófolo y Cecilia Challiol y a todos los que también estuvieron en algunos momentos, cortos o largos y que sin embargo he sabido apreciar. Quiero reconocer especialmente a Hernán por haberse tomado además la ardua tarea de leer una versión avanzada de este trabajo y darme sugerencias muy valiosas.

Finalmente, el agradecimiento más grande: a Tere, por su infinito amor en todas las formas posibles, sin el que yo no sería nada. A ella, por quererme así y por ser mi "little darling".

Pablo A. Barrientos
La Plata, febrero de 2006

Este documento fue procesado usando el sistema de macros LaTeX 2ε.

Prefacio

Un lenguaje de dominio específico es un lenguaje de programación diseñado para ser útil para un conjunto específico de tareas, en contraste con los lenguajes de propósito general. Usar un lenguaje de dominio específico comprende riesgos y oportunidades. Se gana expresividad y se obtiene mayor productividad, pero por otro lado el costo de desarrollar un DSL suele ser alto.

El presente trabajo de grado es una ampliación y a su vez una síntesis de todo el trabajo realizado dentro de un proyecto bilateral de colaboración entre nuestro país y Eslovenia sobre lenguajes de dominio específico, del cual formé parte. El proyecto se llevó a cabo entre la Facultad de Informática de la Universidad Nacional de La Plata y la Facultad de Ingeniería Eléctrica y Ciencias de la Computación de la Universidad de Maribor, Eslovenia.

El objetivo del proyecto fue la investigación del proceso de desarrollo de lenguajes de dominio específico, principalmente las etapas de diseño e implementación. Se abordaron principalmente el estudio de (nuevos) patrones de diseño para DSLs y la evaluación de enfoques de implementación de DSLs con el fin de obtener pautas o guías generales para tal tarea.

Este trabajo de grado está dirigido a los desarrolladores de lenguajes interesados en brindar soluciones para dominios específicos que tienen poca experiencia en esta tarea, o que buscan guías para la implementación de los lenguajes de dominio específico.

Índice general

Capítulo 1

Introducción

"In my opinion, a domain-specific language
is the 'ultimate abstraction'. "
Paul Hudak [Hud96a]

Un lenguaje formal es un conjunto finito de cadenas de símbolos. Los lenguajes formales se usan para describir de forma precisa soluciones a problemas. Un lenguaje de programación es una técnica estandarizada para expresar computaciones. La definición de un lenguaje particular consiste en un conjunto de reglas sintácticas (que describen cómo se combinan los diferentes símbolos del lenguaje para generar construcciones del lenguaje) y un conjunto de reglas semánticas (que dan el significado de esas construcciones).

Cuando se habla de diseñar soluciones a problemas, el uso de lenguajes de programación hace más efectiva la tarea del programador. Los lenguajes de programación pueden incrementar inmensamente la productividad de los programadores, a través de la escritura de código fácilmente mantenible, legible, genérico y escalable. Se pueden usar lenguajes de propósito general (GPL, por sus siglas en inglés — General Purpose Language) o se puede usar un lenguaje especial cercano al dominio del problema (DSL, por sus siglas en inglés — Domain-Specific Language). La primera alternativa da flexibilidad y muchas formas de representar la solución, pero usualmente se requiere de experiencia para obtener el adecuado nivel de abstracción y a la vez obtener soluciones eficientes. Sin embargo, cuando un experto en el campo quiere diseñar una solución, no tiene esa experiencia, y por lo tanto necesita de un buen programador con el cual colaborar. La alternativa es usar un lenguaje de dominio específico y expresar la solución por sí mismo.

Un DSL es un lenguaje diseñado de forma tal que provee una notación hecha a medida al dominio de aplicación, y se basa solamente en los conceptos relevantes y las características del dominio. Como tal, un DSL es una forma

de describir y generar miembros de una familia de programas en el dominio dado, sin necesidad de tener conocimientos sobre programación general. La principal desventaja de los DSLs es el costo de su desarrollo y éste es el principal problema abordado en el presente trabajo. En particular se estudian los diferentes enfoques para implementarlos.

El desarrollo de un lenguaje es considerado una tarea difícil. Este problema dentro del campo de los DSL es una de las razones por las cuales su popularidad no ha alcanzado las expectativas de muchos investigadores; incluso no se han aceptado ampliamente los DSL con notación visual o lenguajes visuales de dominio específico (DSVL — Domain-Specific Visual Languages) [BB94]. Una de las razones es que no hay demasiado trabajo en el campo teórico sobre los DSLs — no hay una clasificación estándar de los DSLs, y menos aún una definición fija de lo que son; además hay una cantidad insuficiente de ejemplos de experiencias de implementación en la literatura relacionada. Esto también hace que los desarrolladores de software opten por soluciones más tradicionales y que creen que son mejores, como es por ejemplo el uso de bibliotecas específicas.

El desarrollo de un DSL implica un esfuerzo de ingeniería de software apreciable, requiriendo una inversión considerable de tiempo y otros recursos. En la práctica, debe buscarse un compromiso entre el nivel de soporte al DSL, su capacidad de uso, su eficiencia, su capacidad de evolución de la implementación, la cantidad de recursos disponibles, y otros criterios. Sin embargo, una vez creados, los DSLs incrementan la productividad, la fiabilidad y mantenimiento del software. Pero los beneficios de los DSLs no son gratuitos. Sin una metodología y herramientas adecuadas, la inversión inicial necesaria para soportar un DSL puede exceder lo que se ahorra por su utilización en el desarrollo de aplicaciones.

Dentro del proceso de desarrollo de un DSL se pueden identificar las etapas de decisión, análisis, diseño, implementación, y finalmente despliegue [MHS05]. En la práctica, el proceso de desarrollo no es secuencial y existen influencias entre las diferentes etapas. En la fase de decisión se intenta ver si es conveniente desarrollar el DSL, de acuerdo a la situación en que se plantea esta idea. En la fase de análisis, se identifica el dominio del problema y se recolecta el conocimiento del mismo. Luego se diseña el DSL para describir de manera concisa las aplicaciones del dominio. Esta tarea necesita de muchas iteraciones y evaluaciones para poder capturar todos los elementos del dominio. La etapa de despliegue es muy particular a cada caso y es imposible caracterizarla. En la etapa de implementación se puede elegir entre varios enfoques de implementación y, por supuesto, se debería elegir, entre ellos, el más adecuado al caso en que se aplica. Sin embargo, hay una enorme falta de pautas que ayuden a los desarrolladores de DSL a elegir la implementa-

ción más conveniente. Más aun, no hay pautas claras para decidir cuándo es conveniente crear DSLs y cuándo usar GPLs.

Un primer intento de clasificar estos enfoques o patrones se encuentra en el trabajo de Spinellis, donde hay una lista de patrones para DSLs [Spi01]. Recientemente, estos patrones han sido mejorados y extendidos en el trabajo de Mernik [MHS05]. Esta última clasificación está centrada en el proceso de desarrollo de los DSLs y es mucho más adecuada, abarcando incluso la etapa de decisión.

El objetivo principal de este trabajo es proveer una ayuda significativa en la toma de decisiones dentro del proceso de desarrollo de los lenguajes específicos de dominio, particularmente en la etapa de implementación. Se realiza una comparación de los diferentes enfoques de implementación tomando un DSL particular como caso de estudio.

Se toman las pautas dadas por Mernik sobre cómo proceder con la implementación de DSLs, extendiéndola y dando una detallada descripción de la experiencia práctica con cada una de ellas. Para ello se usan diferentes herramientas de software que ayudan a realizar los distintos enfoques: compiladores, intérpretes, lenguajes, generadores de parsers, herramientas de generación automática de compiladores basadas en definiciones ejecutables de lenguajes, etc. Con esto se intenta facilitar el desarrollo de lenguajes para expertos de un dominio que no son diseñadores de lenguajes.

La contribución de este trabajo es la experiencia adquirida usando los diferentes enfoques, que puede ser usada como parte de un conjunto de pautas para el desarrollo de DSLs. Para cada enfoque se proveeen detalles de implementación que pueden ser útiles para los profesionales cuando deseen aplicar el enfoque a su propio DSL, aliviando la carga de la tarea.

Este trabajo se estructura como sigue: en el Capítulo 2 se establece una definición de los DSLs y se presentan los diferentes enfoques de implementación. El Capítulo 3 presenta el DSL que fue seleccionado como caso de estudio y se describen algunas características del mismo. En el Capítulo 4 se describe una interfaz común de diseño y programación de FDL para todos los enfoques. En el Capítulo 5 se procede con la descripción de cada una de las implementaciones del lenguaje seleccionado con los diferentes enfoques. En el Capítulo 6 se realiza la comparación de los distintos enfoques de implementación en varias dimensiones. En el Capítulo 7 se presentan algunos trabajos relacionados y finalmente en el Capítulo 8 se presentan las conclusiones.

Capítulo 2

Lenguajes de dominio específico

Cuando se habla de lenguajes de dominio específico, se tiene una idea intuitiva de lo que son. Sin embargo no hay establecida una definición fija y estable en la literatura relacionada. Incluso no hay una clara separación entre lo que son los lenguajes de propósito general y los lenguajes de dominio específico. Sumado a esto, existen varios trabajos que hablan sobre patrones de diseño e implementación de DSLs que parecen ser poco reconciliables. Por eso es necesario establecer una definición que abarque los aspectos más característicos de los DSLs, que guíen el resto del trabajo.

La Sección 2.1 establece una definición de los que son los DSLs y establece una línea suficientemente clara de división entre los lenguajes que son de dominio específico y los que no lo son. En la Sección 2.2 se dan detalles de los enfoques de implementación que son el eje sobre el que gira este trabajo.

2.1 Definición de DSL

Es importante empezar explicando algunos detalles sobre los lenguajes de dominio específico. Como no hay una definición estándar de lo que constituye un DSL, es necesario acotar la "esfera de discurso".

Un lenguaje de programación es una notación para expresar algoritmos en forma tal que los humanos y las máquinas puedan leerlo [Lou03]. Existen lenguajes de bajo nivel (lenguaje assembly) que son cercanos al lenguaje de la computadora pero que son difíciles de entender para los humanos y en general poseen un bajo nivel de abstracción. Existen también lenguajes de alto nivel (como Java, Lisp y Pascal), que son más cercanos al entendimiento del programador y lejanos al código binario. Esto se debe al uso de estructuras y abstracciones de alto nivel (procedimientos, funciones, módulos, clases, etc.). Esto permite al programador abstraerse de los detalles de ejecución de

la computadora, para pensar en término de objetos, funciones o instrucciones complejas, que son las construcciones que provee el lenguaje.

Los lenguajes de dominio específico [Ben86a, vDKV00] pertenecen a la última clasificación porque son lenguajes concisos, más declarativos que imperativos, que proveen construcciones y notaciones para capturar la semántica de un dominio particular y acotado[1], de tal forma que es fácil entenderlo, escribir programas y razonar, para las personas expertas en ese dominio específico. Los DSLs son "el último nivel de abstracción" [Hud96a]. Además son denominados usualmente pequeños lenguajes (little languages) [Ben86b], mini lenguajes (minilanguages) [Ray04], lenguajes de aplicación (application languages) o lenguajes de muy alto nivel (very high level languages).

Un DSL resulta de la necesidad de una comunidad dada de expresar (a través de sintaxis y semántica bien definidas), las ideas dentro de un domino, sin necesidad de conocimiento previo sobre programación general. Para definir un DSL, todos los conceptos del dominio (ni más ni menos) deben ser definidos, minimizando la distancia entre el domino del problema y el programa, ocultando los detalles de implementación. Consecuentemente, los expertos del dominio o miembros de la comunidad pueden entender, validar, modificar y desarrollar programas en el DSL.

Al reducir la distancia conceptual entre el problema y el lenguaje usado para expresar el problema, la programación se vuelve más simple, fácil y fiable, pues las soluciones pueden ser expresadas en el idioma y nivel de abstracción del dominio del problema. La cantidad de código que se escribe es mucho menor, incrementando la productividad y decrementando el costo de mantenimiento. Por eso, la traducción de los conceptos del dominio a una definición semántica del DSL debe ser clara, precisa y completa.

Los DSLs se crean específicamente para resolver los problemas de su dominio y no por fuera de él. No son usualmente *Turing-complete*[2], lo cual puede ser una forma más de caracterizarlos.

Para dar una versión resumida de la definición establecida anteriormente, podemos decir que "un DSL es un lenguaje de programación que es mucho más expresivo — en el sentido que es más fácil de entender y es más conciso — en un dominio particular que un lenguaje de propósito general (GPL), debido al uso de notaciones y abstracciones cercanas a tal dominio. Como tal, un DSL puede generar una familia de sistemas diferentes en su dominio."

Pueden ser definidas abstracciones similares en bibliotecas, pero deben ser usadas en el contexto del GPL y están limitadas a sus mecanismos de

[1]Por dominio se entiende un área de conocimiento o actividad caracterizada por un conjunto de conceptos y terminología entendida por profesionales de ese área.

[2]Un lenguaje Turing-complete puede teóricamente ser usado para programación de propósito general y es equivalente a cualquier otro lenguaje Turing-complete.

abstracción y sintaxis particulares. Sin embargo, cuando se diseña una biblioteca el programador generalmente tiene en su mente de forma inconsciente un 'DSL subyacente'. En contraste, los DSL pueden dar una sintaxis más directa que refleja el idioma del dominio del problema. Por esa razón es importante intentar identificar el DSL subyacente y probar implementarlo con algún enfoque.

Los DSLs tienen una desventaja considerable: el costo de construcción. Su realización requiere usualmente muchas iteraciones de análisis, diseño, implementación y evaluación. Sin embargo, una vez que son creados, incrementan en general la productividad, fiabilidad y mantenimiento — considerar los lenguajes SQL [vdL88] o HTML [Aro94].

SQL, Yacc [MB90] y Tex [Knu84] son buenos ejemplos de lenguajes que se pueden ver como DSLs. Sus áreas de aplicación son: acceso a bases de datos, generación de compiladores y composición de textos, respectivamente. Estos lenguajes no son GPLs dado que no se pueden usar para resolver problemas generales. Pero la línea de división entre DSLs y GPLs algunas veces resulta poco clara y para muchos lenguajes es difícil determinar si son GPLs o DSLs. Por ejemplo, COBOL puede ser considerado como un lenguaje cuyo dominio es el comercio, sin embargo con él se pueden escribir programas que van más allá del dominio y por lo tanto puede ser considerado como un GPL [HM02, vDKV00]. En estas ocasiones, decidir si un lenguaje particular es o no GPL tiene que ver con la subjetividad y con el tipo de problemas para los cuales se crean soluciones con el lenguaje, más que con los conceptos que definen a unos y otros lenguajes.

2.2 Enfoques de implementación

La noción de patrón de diseño fue establecida por el arquitecto Christopher Alexander et al. [AIS+77]. Alexander resume cómo la relación entre un problema recurrente y su solución establece un patrón de la siguiente manera:

> "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"[3]

[3] "Cada patrón describe un problema que ocurre recurrentemente en nuestro entorno, y describe el núcleo de la solución a ese problema, de forma tal que pueda usar esa solución un millón de veces más, sin volver a resolver el problema dos veces de la misma manera nunca más".

Veinte años después Gamma aplicó estas ideas en el campo del diseño de software orientado a objetos reusable [GHJV95]. Los patrones de diseño capturan soluciones que han sido desarrolladas por mucho tiempo. Ofrecen una forma conveniente de capturar, documentar y transmitir el conocimiento existente de un área dada con un formato consistente y accesible. Los conceptos que describen no se pueden codificar y no son frameworks, ya que no describen la estructura de un sistema completo. El uso de patrones hace posible que la documentación del diseño sea simple y pueda ser reusado en diferentes aplicaciones.

En el campo de los DSLs, Spinellis [Spi01] clasificó cada patrón como *creacional* si comprende la creación de un DSL, *estructural* si comprende la estructura de un sistema que implica el uso de un DSL, y *de comportamiento* si describe interacciones de DSLs. Esta clasificación es muy parecida a la que realizó Gamma y parece no adecuarse correctamente a los DSLs.

Una segunda forma de clasificar fue dada por Mernik et al. [MHS05]. Ésta última clasificación parece ser más adecuada pues hace una división clara entre patrones de decisión, análisis, diseño e implementación, que se corresponden con cada una de las etapas de desarrollo de los DSLs.

A continuación se explican cada uno de los patrones de implementación basando la clasificación en el trabajo de Mernik. La clasificación dada por Spinellis sólo se toma como referencia.

2.2.1 Preprocessing

Las construcciones del DSL son traducidas a construcciones en el lenguaje base en una fase de preprocesamiento. El análisis estático del código fuente se limita al hecho por un preprocesador en el lenguaje base. Hay importantes subpatrones:

Macro processing
: En este enfoque, el significado de las nuevas expresiones se definen en términos de otras construcciones en el lenguaje mediante el uso de macros. Estas nuevas construcciones son reemplazadas en el código fuente por el preprocesador. Las macros agregan un grado útil de extensibilidad a un lenguaje de programación al proveer una forma conveniente de expresar abstracciones sintácticas.

Source-to-source transformation
: El código fuente del DSL se traduce a código fuente de un lenguaje existente. La traducción se realiza usualmente usando patrones dirigidos por sintaxis. Los chequeos y generación de código se producen

luego, usando las herramientas del lenguaje al que es traducido el código fuente del DSL.

Pipeline
Existe una "línea de preprocesadores", donde cada uno traduce un sublenguajes de un DSL y lo pasa como entrada al siguiente preprocesador.

Lexical processing
En este enfoque se requiere solamente exploración léxica simple, sin análisis sintáctico basado en estructuras de árboles.

Una ventaja de este enfoque es que la implementación de un DSL es muy fácil, dado que mucho del análisis semántico, si no todo, se pospone y se realiza por el procesador del lenguaje base. Sin embargo, la ausencia de análisis semántico es también causa de una de las principales desventajas, ya que no se pueden hacer chequeos estáticos y optimizaciones en ese nivel. Otra desventaja es que el reporte de errores se hace en términos de conceptos del lenguaje base.

2.2.2 Embedding

En este enfoque, los mecanismos existentes en el lenguaje base se usan para construir una biblioteca de operaciones de dominio específico, definiendo nuevos tipos de datos abstractos y operaciones. Se trata al DSL como un tipo abstracto de datos. Los mecanismos sintácticos del leguaje host se usan para expresar el idioma del dominio. El enfoque embebido hereda las construcciones genéricas del lenguaje host y agrega primitivas del dominio específico que son cercanas al usuario del DSL. No se usa preprocesamiento ni expansión de macros. El lenguaje resultante se denomina lenguaje embebido de dominio específico o DSEL (por sus siglas en inglés: Domain-Specific Embedded Language).

Una ventaja de este enfoque es que el compilador o intérprete del lenguaje host se reusa tal como está para el DSL, minimizando el esfuerzo de implementación y dando un lenguaje más poderoso. Además, si el usuario ya conoce el lenguaje host, el aprendizaje del DSL se hace mucho más fácil.

La principal desventaja es la limitación en la expresividad al usar los mecanismos sintácticos del lenguaje host (usualmente difíciles de extender). En muchos casos, la notación óptima se compromete seriamente. Como en el enfoque previo, el reporte de errores es también problemático por las mismas razones. Además las optimizaciones del dominio son difíciles de realizar, lo que afecta la eficiencia.

Los lenguajes funcionales son especialmente adecuados para embeber los DSL.

Un caso especial de este patrón, que bien puede ser considerado un subpatrón, se da con el uso de plantillas o *templates*. Las plantillas son (en su idea original) una herramienta que permite dar genericidad al código escrito mediante parametrización, para lograr un mayor reúso. El uso de las plantillas como técnica de *metaprogramming* estática[4] fue accidental. Erwin Unruh dio un primer ejemplo de metaprogramación usando templates para generar errores de compilación que listan una secuencia arbitraria de números primos [Unr94].

Así, un programa de un DSL tiene código estático, que se evalúa en tiempo de compilación, y código dinámico, que se compila y ejecuta luego.

Este (sub)enfoque de implementación de DSLs tiene las mismas ventajas y desventajas que el enfoque embebido. Además permite realizar transformaciones y optimizaciones en tiempo de compilación y generar el código adecuado para dar semántica al lenguaje, aunque muchas veces se hace sacrificando aún más la notación embebida respecto de la notación del DSL original.

2.2.3 Interpreter/Compiler

En este enfoque se usan las técnicas de compilación o interpretación para implementar un DSL. En el caso de los compiladores, las construcciones del DSL se traducen a construcciones del lenguaje base y llamados a bibliotecas de funciones. Se hace un análisis estático completo de la especificación/programa del DSL. En el caso de los intérpretes, se reconocen las construcciones del DSL y se interpretan usando un ciclo buscar-decodificar-ejecutar (*fetch-decode-execute*).

Una importante desventaja es el costo de construir un compilador o intérprete desde cero. Otra desventaja es la gran dificultad para extender un lenguaje existente. Además se deben desarrollar las herramientas de soporte (entorno de debugging y desarrollo). Estas desventajas se pueden minimizar con el enfoque que se describe luego de este.

La principal ventaja es que se consigue una sintaxis cercana a la notación usada por los expertos del dominio y además se genera un buen reporte de errores. El enfoque usando un intérprete es bueno cuando se trata de un lenguaje de carácter dinámico o si su eficiencia no es una cuestión importante. La

[4] La metaprogramación es una técnica de programación con la que se escriben programas que representan y manipulan otros programas (por ejemplo: compiladores, generadores de programas, intérpretes) o a sí mismos (*reflection*)

ventaja de la interpretación sobre la compilación es el mayor control sobre el entorno de ejecución. Además, el intérprete procesa el programa en presencia de los datos y produce la respuesta directamente. En cambio el compilador produce un 'nivel de indirección', pues el programa a ser procesado debe ser primero compilado para ser luego ejecutado.

2.2.4 Compiler generator

Este enfoque es similar al anterior, excepto que algunas fases del compilador del DSL (o generador de aplicaciones) se implementan usando sistemas de desarrollo de lenguajes o lo que se llaman herramientas de escritura de compiladores (*compiler-compilers*). Partes del compilador, como ser el parser, se generan automáticamente a partir de una descripción del lenguaje (para el caso del parser, se hace usualmente usando notación BNF o EBNF, más el código asociado a cada regla de la gramática, que se ejecuta cuando se aplica).

De esta manera, el costo de implementación se minimiza y por lo tanto se minimizan las desventajas del enfoque anterior. La sintaxis del DSL puede ser muy cercana a las notaciones de los expertos del dominio y se puede realizar un buen reporte de errores. Además se pueden hacer optimizaciones y verificaciones.

2.2.5 Extensible compiler/interpreter

El compilador/intérprete del GPL se extiende con construcciones, reglas de optimización y/o generación de código del dominio específico. Técnicas como reflección e introspección son muy útiles para este enfoque. Comparado a los dos últimos enfoques, el esfuerzo de implementación se minimiza debido al reuso de toda la infraestructura del compilador existente y sus herramientas asociadas. Por ejemplo, se puede reutilizar el compilador y la forma en que éste genera el reporte de errores para generar los chequeos de errores sintácticos y semánticos del lenguaje extendido.

Como desventaja, extender un compilador es difícil, pues se debe tener mucho cuidado para prevenir cualquier interferencia entre la notación del dominio específico con la que ya existe.

2.2.6 Commercial Off-The-Shelf (COTS)

Se usan herramientas o notaciones existentes para un dominio específico (por ejemplo, DSLs basados en XML). Este enfoque da alternativas fiables

para resolver problemas de un dominio particular (por ejemplo, XML da soluciones muy buenas en el procesamiento y consulta sobre documentos). En algunos casos el usuario final puede sentir que no está realmente programando, cuando en realidad las herramientas de soporte se adaptan para que el usuario realice las especificaciones de los programas bajo el dominio y éstas puedan ser procesadas como una especificación de un DSL.

En particular, XML tiende a ser difícil de leer y escribir por los humanos, pero usado con otras herramientas produce soluciones razonables con notación medianamente cercana al dominio. Entre las herramientas de XML útiles para implementar un DSL están: DTD y XML schema (validación), XSLT (transformación de documentos), CSS (presentación) y DOM y SAX (procesamiento).

Otros ejemplos de herramientas que pueden ser usadas para implementar un DSL en este enfoque lo constituyen las planillas de cálculo (en particular con el sistema de macros y funciones), las presentaciones visuales del estilo de PowerPoint, etc.

2.2.7 Hybrid

Es una combinación de dos o más de los enfoques anteriormente descriptos. Algunos ejemplos son: *interpreter + compiler*, *embedding + compiler*, *embedding + macro-processing*, *macro-processing + extensible compiler/interpreter*, etc.

Capítulo 3

Lenguaje estudiado: FDL

En este capítulo se presenta el lenguaje elegido para llevar a cabo la comparación de los enfoques de implementación. En la Sección 3.1 se presenta detalladamente el lenguaje elegido y en la Sección 3.2 se dan algunas ideas y comentarios interesantes sobre este lenguaje particular, que se podrían aplicar a cualquier otro lenguaje de dominio específico.

3.1 Descripción del lenguaje FDL

Para elegir el DSL de estudio se tuvo en cuenta que no debía ser muy simple pero tampoco muy complejo respecto a los tiempos en los que se tenía previsto desarrollar el trabajo. Además, el lenguaje debía ser representativo de las características de los DSLs, como se describió en la Sección 2.1.

Se estudiaron algunos DSLs como ser GraphViz [Nor92][1], para definir estructuras descriptas mediante grafos; FDL [DK02], para describir la composición de sistemas y Frob [PHE99], para el control de robots; entre otros.

El lenguaje seleccionado es FDL, por su siglas en inglés (Feature Description Language) [DK02]. El lenguaje es relativamente sencillo de implementar (más allá de cada enfoque) debido a una semántica clara y concisa.

FDL se usa para capturar las partes comunes y variables de dominios de aplicación y sus dependencias, que se obtiene mediante de un análisis de dominio. El análisis de dominio es el proceso de identificar, analizar y representar un modelo de dominio y arquitectura de software del estudio de un sistema existente, teoría subyacente, tecnología emergente e historia de desarrollo dentro del dominio de interés. El lenguaje FDL fue creado para dar una descripción textual para diagramas de características [CE00], que es una técnica que se usa en el análisis de dominios siguiendo la metodología

[1] Graphviz - Graph Visualization Software, disponible en `http://www.graphviz.org/`

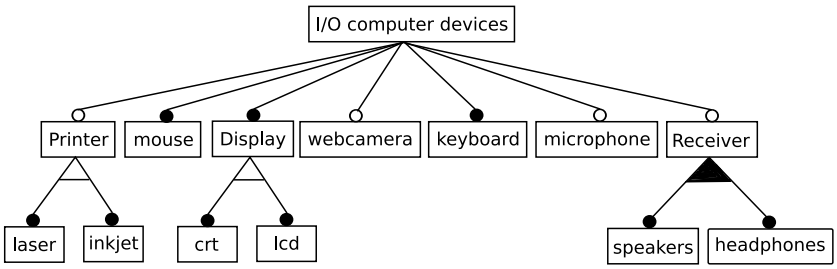


Figura 3.1: Diagrama para dispositivos de E/S de computadoras

```
IODevice  :  all (opt(Printer), mouse, Display, opt(webcamera), keyboard,
                 opt(microphone), opt(Receiver))
Printer   :  one-of (laser, inkjet)
Display   :  one-of (crt, lcd)
Receiver  :  more-of (speaker, headphones)
```

Figura 3.2: Descripción textual del diagrama de la Figura 3.1

FODA (Feature Oriented Domain Analysis [KCH+90]) y que intenta definir el modelo de dominio de una aplicación. Los diagramas de características proveen formas de expresar la composición de un sistema dado (describen de forma concisa todas las posibles configuraciones de un sistema). FDL captura esto de forma textual.

Para introducir los diagramas de características y el lenguaje simultáneamente, se da un ejemplo simple sobre la composición de hardware de una PC. El diagrama de características se puede ver en la Figura 3.1. La descripción textual correspondiente se puede ver en la Figura 3.2.

Básicamente, se expresa que un IODevice está compuesto por muchas partes (indicado por una estructura en árbol, cuya raíz es el nodo IODevice en el diagrama visual, y por la característica con nombre IODevice en el programa FDL), algunas partes son opcionales (indicado por un círculo vacío en el diagrama visual y por la operación opt en el programa FDL) y otras partes son obligatorias (indicadas por un círculo lleno en el diagrama visual y por el operador default o simplemente sin el opt en el programa FDL). Además, algunas características son *atómicas*, es decir, son unidades básicas de descripción (indicadas por hojas en el diagrama visual y por identificadores que comienzan en letra minúscula en el programa FDL) y otras características están formadas por subcaracterísticas (indicadas por una estructura en forma de árbol en el diagrama visual y por características con nombre — que

<FDLSpecification>	::=	*<FeatureDefinitions>* *<ConstraintDefinitions>*
<FeatureDefinitions>	::=	*<FeatureDefinition>* *<FeatureDefCont>*
<FeatureDefCont>	::=	λ | *<FeatureDefinitions>*
<FeatureDefinition>	::=	*#FeatureName* **:** *<FeatureExpression>*
<Features>	::=	*<FeatureExpression>* *<FeaturesCont>*
<FeaturesCont>	::=	ϵ | **,** *<Features>*
<FeatureExpression>	::=	*#AtomicFeature*
	|	*#FeatureName*
	|	**opt(***<FeatureExpression>***)**
	|	**all(***<Features>***)**
	|	**one-of(***<Features>***)**
	|	**more-of(***<Features>***)**
	|	**default =** *<FeatureExpression>*
<ConstraintDefinitions>	::=	*<Constraints>*
<Constraints>	::=	*<Constraint>* *<ConstraintsCont>*
<ConstraintsCont>	::=	ϵ | *<Constraints>*
<Constraint>	::=	*#AtomicFeature* **requires** *#AtomicFeature*
	|	*#AtomicFeature* **excludes** *#AtomicFeature*
	|	**include** *#AtomicFeature*
	|	**exclude** *#AtomicFeature*

Figura 3.3: Gramática BNF del lenguaje FDL

comienzan con letra mayúscula — en el programa FDL).

Respecto de las características compuestas, hay dos formas de seleccionar alternativas: la primera forma es tomando sólo una entre varias (identificado en un diagrama visual por un triángulo vacío y por la operación one-of en un programa FDL), y la segunda forma es tomar muchas opciones combinadas de todas las formas posibles (identificado en el diagrama visual por un triángulo lleno y por la operación more-of en un programa FDL).

En la Figura 3.3 se da la gramática completa para el lenguaje FDL [DK02], que incluye restricciones (constraints), que serán explicadas más adelante. Los identificadores aparecen con un '#' delante. El identificador *#FeatureName* representa un identificador que comienza con letra mayúscula y *#AtomicFeature* representa un identificador que comienza en letra minúscula.

El significado del diagrama de características es el conjunto de todas las posibles configuraciones del sistema. Para el ejemplo dado, las posibles configuraciones de dispositivos de entrada/salida son 96.

El lenguaje FDL introducido en [DK02] tiene muchos beneficios sobre las descripciones visuales. Por ejemplo, se puede computar la variabilidad

(es decir, el número de alternativas de configuración del sistema), se pueden expresar restricciones sobre los diagramas, se pueden transformar los diagramas a diagramas UML y luego se puede generar código fuente de forma automática.

Las restricciones sobre el diagrama fueron tienen el propósito de limitar la variabilidad del sistema. Una restricción puede tener una de las siguientes formas:

- f1 requires f2: si la característica f1 está presente, entonces la característica f2 debe estar presente también,
- f1 excludes f2: si la característica f1 está presente, entonces la característica f2 no debe estar presente,
- include f: la característica f debe estar presente,
- exclude f: la característica f no debe estar presente.

Las primeras dos restricciones se denominan restricciones de diagrama, pues se aplican cuando se quiere evitar la formación de variaciones inválidas, inherentes al diagrama. Las últimas dos restricciones se denominan restricciones del usuario y le brindan la posibilidad de evitar la aparición de ciertas configuraciones no deseadas.

Como ejemplo, si se introducen las restricciones webcamera requires microphone e include lcd al ejemplo dado, las 96 posibles configuraciones originales se reducen a sólo 36.

Para evaluar programas FDL, se necesitan aplicar los siguientes pasos de transformación (reducciones) secuencialmente.

Regularización: Las apariciones de características con nombre se sustituyen por las correspondientes definiciones. Para el ejemplo de IODevice, la forma no regularizada es la dada en la Figura 3.2 y la versión regular es:

all(opt(one-of(laser, inkjet)), mouse, one-of(crt, lcd), opt(webcamera), keyboard, opt(microphone), opt(more-of (speaker, headphones)))

Normalización: Se obtiene usando reglas que simplifican una expresión dada eliminando los duplicados y casos degenerados de los constructores (casos en que se tiene el constructor con una sola característica). En la figura 3.5 se detallan cada una de las reglas, en forma de ecuaciones. En ella y en las subsiguientes figuras se usan las convenciones sintácticas de la Figura 3.4.

Variable	*Tipo*
F	FeatureExpression
Ft	FeatureExpression {“,” FeatureExpression }*
Fs	∈ | Ft
A	AtomicFeature
C	Constraint
Cs	Constraint*

Figura 3.4: Convenciones sintácticas

Regla	*Ecuación*	
$[N1]$	Fs, F, Fs', opt(F), Fs''	= Fs, F, Fs', Fs''
$[N2]$	Fs, F, Fs', F, Fs''	= Fs, F, Fs', Fs''
$[N3]$	opt(opt(F))	= opt(F)
$[N4]$	all(F)	= F
$[N5]$	all(Fs, all(Ft), Fs')	= all(Fs, Ft, Fs')
$[N6]$	one-of(F)	= F
$[N7]$	one-of(Fs, one-of(Ft), Fs')	= one-of(Fs, Ft, Fs')
$[N8]$	one-of(Fs, opt(F), Fs')	= opt(one-of(Fs, F, Fs'))
$[N9]$	more-of(F)	= F
$[N10]$	more-of(Fs, more-of(Ft), Fs')	= more-of(Fs, Ft, Fs')
$[N11]$	more-of(Fs, opt(F), Fs')	= opt(more-of(Fs, F, Fs'))
$[N12]$	default = A	= A

Figura 3.5: Reglas de normalización

La regla $[N1]$ combina características con el mismo nombre que son obligatorias y opcionales al mismo tiempo en una lista, en una sola característica que es obligatoria. La regla $[N2]$ remueve las características duplicadas en una lista y deja sólo una. La regla $[N3]$ une opcionales duplicados y deja uno solo. Las reglas $[N4]$ y $[N5]$ normalizan casos degenerados y anidados de all respectivamente. Las reglas $[N6]$ y $[N7]$ normalizan casos degenerados y anidados de one-of respectivamente. La regla $[N8]$ transforma un one-of que contiene una característica opcional en un one-of opcional. Las reglas $[N9]$ y $[N10]$ normalizan casos degenerados y anidados de more-of respectivamente. La regla $[N11]$ transforma un more-of que contiene una característica opcional en un more-of opcional. Finalmente la regla $[N12]$ elimina la notación default =.

Regla	*Ecuación*
$[E1]$	$\mathsf{all(Fs,\ opt(F),\ Ft)} = \mathsf{one\text{-}of}\left(\begin{array}{l}\mathsf{all(Fs,\ F,\ Ft)}\\ \mathsf{all(Fs,\ Ft)}\end{array}\right)$
$[E2]$	$\mathsf{all(Ft,\ opt(F),\ Fs)} = \mathsf{one\text{-}of}\left(\begin{array}{l}\mathsf{all(Ft,\ F,\ F)}\\ \mathsf{all(Ft,\ Fs)}\end{array}\right)$
$[E3]$	$\mathsf{all(Fs,\ one\text{-}of(F,\ Ft),\ Fs')} = \mathsf{one\text{-}of}\left(\begin{array}{l}\mathsf{all(Fs,\ F,\ Fs')}\\ \mathsf{all(Fs,\ one\text{-}of(Ft),\ Fs')}\end{array}\right)$
$[E4]$	$\mathsf{all(Fs,\ more\text{-}of(F,Ft),Fs')} = \mathsf{one\text{-}of}\left(\begin{array}{l}\mathsf{all(Fs,\ F,\ Fs')}\\ \mathsf{all(Fs,\ F,\ more\text{-}of(Ft),Fs')}\\ \mathsf{all(Fs,\ more\text{-}of(Ft),\ Fs')}\end{array}\right)$

Figura 3.6: Reglas de expansión

Expansión: Las reglas de expansión se aplican repetidamente para transformar expresiones normalizadas a expresiones en *forma normal disyuntiva* (DNF) definida de la siguiente manera:

$$\mathsf{one\text{-}of}\left(\begin{array}{c}\mathsf{all}(A_{1_1},\ldots,A_{1_n})\\ \vdots\\ \mathsf{all}(A_{m_1},\ldots,A_{m_n})\end{array}\right)$$

En la figura 3.6 se detallan cada una de las reglas, en forma de ecuaciones. Notar que luego de cada aplicación de una regla de expansión, el resultado puede necesitar ser normalizado nuevamente.

Las reglas $[E1]$ y $[E2]$ transforman un all que contiene una característica condicional en dos casos de all: uno con la característica opcional y otro sin ella. La regla $[E3]$ transforma un all que contiene un one-of en dos casos: uno es un all con la primer característica y el otro es un one-of sin ella. Finalmente, la regla $[E4]$ transforma un all que tiene un more-of en tres casos: uno es un one-of similar al all original, pero sólo con la primer alternativa del more-of; el segundo caso es un all similar, con la primer alternativa también y el resto del more-of; y el tercer caso es similar al all original, con el resto del more-of, pero sin su primer característica.

Procesamiento de restricciones: En esta fase se testean las restricciones impuestas, removiendo aquellas configuraciones que no satisfacen alguna de ellas. Las reglas que definen los diferentes tipos de restricciones se definen en la figura 3.7. La FND del ejemplo dado, a la que se le aplicaron las restricciones se puede ver en la figura 3.8.

Regla	*Ecuación*
$[S1]$	is-element(A2, Fs) | is-element(A2, Fs') = true ========== sat(all(Fs, A1, Fs'), Cs (A1 excludes A2) Cs') = false
$[S2]$	is-element(A2, Fs) | is-element(A2, Fs') = false ========== sat(all(Fs, A1, Fs'), Cs (A1 excludes A2) Cs') = sat(all(Fs, A1, Fs'), Cs Cs')
$[S3]$	is-element(A2, Fs) | is-element(A2, Fs') = false ========== sat(all(Fs, A1, Fs'), Cs (A1 requires A2) Cs') = false
$[S4]$	is-element(A2, Fs) | is-element(A2, Fs') = true ========== sat(all(Fs, A1, Fs'), Cs (A1 requires A2) Cs') = sat(all(Fs, A1, Fs'), Cs Cs')
$[S5]$	is-element(A,Ft) = true ========== sat(all(Ft), Cs (include A) Cs') = sat(all(Ft), Cs Cs')
$[S6]$	is-element(A,Ft) = false ========== sat(all(Ft), Cs (include A) Cs') = false
$[S7]$	is-element(A,Ft) = true ========== sat(all(Ft), Cs (exclude A) Cs') = false
$[S8]$	is-element(A,Ft) = false ========== sat(all(Ft), Cs (exclude A) Cs') = sat(all(Ft), Cs Cs')
[default-$S9$]	sat(all(Ft), Cs) = true

Figura 3.7: Reglas de satisfacción

```
one-of(
       all( inkjet, mouse, lcd, webcamera, keyboard, microphone,          headphones),
       all( inkjet, mouse, lcd, webcamera, keyboard, microphone, speaker            ),
       all( inkjet, mouse, lcd, webcamera, keyboard, microphone, speaker, headphones),
       all( laser,  mouse, lcd, webcamera, keyboard, microphone,          headphones),
       all( laser,  mouse, lcd, webcamera, keyboard, microphone, speaker            ),
       all( laser,  mouse, lcd, webcamera, keyboard, microphone, speaker, headphones),
       all( inkjet, mouse, lcd, webcamera, keyboard, microphone                     ),
       all( laser,  mouse, lcd, webcamera, keyboard, microphone                     ),
       all( inkjet, mouse, lcd,            keyboard, microphone,          headphones),
       all( inkjet, mouse, lcd,            keyboard, microphone, speaker            ),
       all( inkjet, mouse, lcd,            keyboard, microphone, speaker, headphones),
       all( laser,  mouse, lcd,            keyboard, microphone,          headphones),
       all( laser,  mouse, lcd,            keyboard, microphone, speaker            ),
       all( laser,  mouse, lcd,            keyboard, microphone, speaker, headphones),
       all( inkjet, mouse, lcd,            keyboard, microphone                     ),
       all( laser,  mouse, lcd,            keyboard, microphone                     ),
       all( inkjet, mouse, lcd,            keyboard,                      headphones),
       all( inkjet, mouse, lcd,            keyboard,              speaker           ),
       all( inkjet, mouse, lcd,            keyboard,              speaker, headphones),
       all( laser,  mouse, lcd,            keyboard,                      headphones),
       all( laser,  mouse, lcd,            keyboard,              speaker           ),
       all( laser,  mouse, lcd,            keyboard,              speaker, headphones),
       all( inkjet, mouse, lcd,            keyboard                                 ),
       all( laser,  mouse, lcd,            keyboard                                 ),
       all(         mouse, lcd, webcamera, keyboard, microphone,          headphones),
       all(         mouse, lcd, webcamera, keyboard, microphone, speaker            ),
       all(         mouse, lcd, webcamera, keyboard, microphone, speaker, headphones),
       all(         mouse, lcd, webcamera, keyboard, microphone                     ),
       all(         mouse, lcd,            keyboard, microphone,          headphones),
       all(         mouse, lcd,            keyboard, microphone, speaker            ),
       all(         mouse, lcd,            keyboard, microphone, speaker, headphones),
       all(         mouse, lcd,            keyboard, microphone                     ),
       all(         mouse, lcd,            keyboard,                      headphones),
       all(         mouse, lcd,            keyboard,              speaker           ),
       all(         mouse, lcd,            keyboard,              speaker, headphones),
       all(         mouse, lcd,            keyboard                                 ),
      )
```

Figura 3.8: El significado de IODevice con restricciones

Regla	*Ecuación*	
$[V1]$	$var(\mathsf{A})$	$= 1$
$[V2]$	$var(\mathsf{opt(F)})$	$= var(\mathsf{F}) + 1$
$[V3]$	$var(\mathsf{all(F, Ft)})$	$= var(\mathsf{F}) * var(\mathsf{all(Ft)})$
$[V4]$	$var(\mathsf{all(F)})$	$= var(\mathsf{F})$
$[V5]$	$var(\mathsf{one\text{-}of(F, Ft)})$	$= var(\mathsf{F}) + var(\mathsf{one\text{-}of(Ft)})$
$[V6]$	$var(\mathsf{one\text{-}of(F)})$	$= var(\mathsf{F})$
$[V7]$	$var(\mathsf{more\text{-}of(F, Ft)})$	$= var(\mathsf{F}) + (var(\mathsf{F})+1) * var(\mathsf{more\text{-}of(Ft)})$
$[V8]$	$var(\mathsf{more\text{-}of(F)})$	$= var(\mathsf{F})$

Figura 3.9: Reglas para el cálculo de variabilidad

Por último, para cada diagrama FDL se puede calcular su variabilidad. La variabilidad es un número natural que indica la cantidad de configuraciones posibles del sistema y se calcula usando las reglas de la figura 3.9.

Es interesante notar que en ciertos casos la variabilidad de la versión original de la característica difiere de la de las versiones normalizada y expandida (más allá de que difiera de la versión con restricciones). Por ejemplo en el caso de la característica more-of (opt(a1), a1), la variabilidad original calculada con las reglas de la Figura 3.9 es 5[2], pero la variabilidad de la versión normalizada: a1, obtenida luego de aplicar las reglas de normalización $[N1]$ y $[N9]$ (ver figura 3.5), es 1. En el ejemplo de IODevice la variabilidad se mantiene entre las versiones original, normalizada y expandida. En cualquier caso, la variabilidad correcta de la expresión es la correspondiente a la versión normalizada (y/o expandida), dado que esta anomalía se presenta en las versiones no normalizadas.

3.2 Algunos comentarios sobre FDL

Al trabajar con FDL se notaron algunas sutilezas que se comentan en esta subsección.

3.2.1 Características atómicas con nombres repetidos

El primer problema que se encontró fue al trabajar con características atómicas con el mismo nombre pero que pertenecen a diferentes características compuestas. Para este tipo de configuraciones, no es posible encontrar la

[2] var (more-of (opt(a1), a1)) = var (opt(a1)) + (var(opt(a1)) + 1) * var(more-of(a1)) = (var (a1) + 1) + ((var (a1) + 1) + 1) * var(more-of(a1)) = (1 + 1) + ((1 + 1) + 1) * var(a1) = (1 + 1) + ((1 + 1) + 1) * 1 = 2 + ((1 + 1) + 1) * 1 = 5

forma normal correcta dado que no se distingue si las características tienen la intención o no de aparecer repetidas. El problema queda mucho más claro con un ejemplo concreto: si se usan dos características compuestas diferentes Monitor y LCD-Display en la misma expresión FDL, y ambas se definen como one-of(15inches, 17inches), entonces el resultado luego del proceso de normalización contendrá sólo una característica 15inches, incluso si el sistema tiene ambos (monitores de 15 pulgadas y pantallas LCD de 15 pulgadas) al mismo tiempo.

Para solucionar este problema, el usuario final puede "calificar" las características que poseen el mismo nombre, poniendo un prefijo o tag. Con esta solución, la expresión Monitor se define como one-of(Monitor.15inches, Monitor.17inches), y LCD-Display como: one-of(LCD-Display.15inches, LCD-Display.17inches) y al procesarlas quedan expresadas correctamente ambas partes del sistema descrito.

3.2.2 Orden de reducción

Otra clase de cuestión, que es mucho más problemática para quien desee implementar FDL, es que el orden en el cual las reglas de reducción se aplican es importante: si se utiliza otro orden diferente al dado en [DK02], se puede obtener un resultado totalmente diferente.

En particular, es necesario aplicar las reglas de normalización antes y luego de cada paso de reducción. Además, las reglas de normalización tienen que ser aplicadas en el orden específico dado.

Sería deseable que dos reducciones diferentes de la misma expresión llegaran a formas normales equivalente, independientemente del orden de aplicación. Esta noción se conoce como *confluencia* en sistemas de reescritura [Bar88].

La solución para esto es bastante difícil, e implica el diseño de una semántica denotacional para el lenguaje — que necesita algunos cambios en la sintaxis. Un trabajo donde se formaliza la semántica de FDL se puede ver en [MLI05] y se trata brevemente en la Sección 3.2.4.

3.2.3 Forma normal de las expresiones

Se encontraron además algunos problemas para encontrar las formas normales de algunas características.

El primer problema es que con la presente definición de FDL [DK02] existen expresiones que no pueden ser reducidas a DNF. Esto sucede cuando el operador más externo no es un all — por ejemplo: more-of (f1,f2) u opt(f).

Esta particularidad hace difícil el uso de una forma uniforme de determinar cuándo una característica está normalizada. Aunque este problema puede ser solucionado fácilmente con la redefinición de lo que constituye una forma normal, se pospone su discusión hasta la Subsección 3.2.4.

El segundo problema, mucho más sutil, está relacionado con la aplicación contextual de las reglas de normalización. La regla $[N2]$ establece que cuando dos expresiones idénticas aparecen en la misma lista, se puede remover una de ellas. Pero en el caso que estas expresiones aparezcan en un contexto mucho más grande, el orden de reducción dado puede dar diferentes resultados a los esperados. Por ejemplo, sea fe = all (one-of(f1, f2),one-of(f1, f2)); con el orden de reducción dado, fe reduce a one-of(f1, f2), aún cuando el resultado esperado es one-of(f1, f2, all(f1, f2)). Este problema aparece sólo en FDL — y no en los diagramas de características originales — por la representación textual y las reglas elegidas. La solución a esto es dar un contexto a las reglas de reducción, y aplicar los cambios para hacer el resultado independiente del orden de reducción.

3.2.4 Modelos algebraicos

Como fue presentado en [DK02], FDL posee sólo una semántica operacional. Se propuso recientemente una semántica denotacional basada en un modelo algebraico claro de conjunto de partes de conjunto de partes de cadenas de caracteres (strings) [MLI05]. Esta tarea implica cambios en la sintaxis, una redefinición de algunas operaciones (las operaciones all, one-of y more-of se convirtieron en binarias y asociativas, para luego recuperar su naturaleza n-aria) y la aparición de una característica especial denominada Null que soluciona algunos de los problemas de las formas normales no definidas comentados en la subsección anterior — por ejemplo, la forma normal de la expresión opt(a) queda definida por one-of(Null, a).

A pesar de este problema, las implementaciones discutidas en este trabajo se basan en la semántica operacional definida anteriormente para FDL.

Como trabajo futuro, se planea revisar la comparación hecha y ver cómo los modelos algebraicos podrían modificar los resultados de las comparaciones.

Capítulo 4

API de FDL

Una forma común de resolver un problema es definir una biblioteca de funciones usando un GPL particular. La persona que diseña esa biblioteca usualmente tiene un DSL en su cabeza, pero no necesariamente de forma explícita.

Antes de proceder con la descripción de los enfoques de implementación para FDL, se presenta una interfaz de programación de aplicación o más conocida como API (Application Programming Interface). La API comprende un conjunto de rutinas, protocolos y herramientas para construir aplicaciones de software, que ayudan a definir las bibliotecas de FDL. Se presentan las APIs para el paradigma orientado a objetos y funcional, que son los dos paradigmas que se usaron mayormente.

El presente capítulo se estructura como sigue. En la Sección 4.1 se describen los principales elementos que conforman la API, tanto en el paradigma orientado a objetos como en el funcional. En las secciones 4.2 y 4.3 se describen las APIs en sus versiones orientada a objetos y funcional respectivamente. Y finalmente en la Sección 4.4 se hacen algunas consideraciones respecto de las APIs contra los lenguajes de dominio específico.

4.1 La API

Un programa en FDL representa una descripción de un sistema mediante una característica especial con nombre. De aquí en adelante, se denominará a dicha descripción simplemente como la "característica" o la "instancia FDL" representada por el programa.

Lo primero que una API debe proveer es una forma de construir características. Para ello, se necesitan operaciones de construcción para características atómicas, opcionales y compuestas (all, one-of y more-of), y restric-

ciones.

En la versión orientada a objetos de las biblioteca, para representar características se usa un enfoque similar al patrón de diseño *Factory* [GHJV95] combinado con el patrón de diseño *Composite* (ver Figura 4.1). En la versión en el paradigma funcional, se representa por una clase Haskell [PJH99] o un módulo de ML [Pau91], que usa tipos de datos algebraicos para representar la característica. Ambas versiones se presentan en las secciones siguientes bajo el nombre de *Construcción de características*.

El segundo elemento que la API tiene que proveer es una forma de calcular las versiones regulares, normalizadas y expandidas de las características. Esto se logra a través de la redefinición de la implementación de los diferentes métodos de las subclases del composite en la versión orientada a objetos, y con instancias particulares para las clases o módulos en la versión funcional. Se presentan en las siguientes secciones bajo el nombre de *Representación de la semántica*.

Finalmente, necesitamos una forma de comunicar el resultado al usuario. Y para ello se deben definir operaciones para producir una representación en forma textual de la característica. En las bibliotecas orientadas a objetos, esto se logra con métodos que delegan en los componentes de la agregación su representación textual, y luego combinan los resultados para construir la representación final. En la versión funcional, las operaciones se logran recorriendo la estructura recursiva en forma de árbol de la característica. Se presenta bajo el nombre de *Impresión de características* en las secciones siguientes.

4.2 Bibliotecas orientadas a objetos

Construcción de características

En esta sección se describirá la API orientada a objetos basada en el lenguaje Java.

La primera parte de la API se relaciona con la construcción de características. Para poder elegir entre las versiones regulares o normalizadas de las características (ver la semántica de los programas FDL en el Capítulo 3), se usa un enfoque similar al patrón Factory. Dos clases, `RuleManager` y `FeatureDefinition`, proveen métodos para construir características. Los métodos de la clase `RuleManager` construirán la forma regular o normal de la característica en base a un parámetro.

Otro punto importante en la representación es el uso de características con nombre (comenzado en mayúscula). Como este tipo de características

es usado en el programa FDL y durante el proceso de regularización y por lo tanto no es necesario en la representación interna, se decidió no representar estas características explícitamente, sobrecargando temporalmente el significado de las características atómicas.

La clase `FeatureDefinition` provee una operación para reemplazar las características con nombre.

```
public static Feature addFeature(Feature rootFeature,
                Feature aFeature, AtomicFeature aNamedFeature)
```

que se invoca para cada definición del programa FDL — el primer parámetro (rootFeature) debe ser un objeto nulo para la característica principal, y para el resto de las características, la instancia FDL a ser construida. El uso del método addFeature se puede observar en la Figura 4.2.

Los métodos de construcción usan además el patrón Composite (mostrado en la figura Figura 4.1 usando un diagrama conceptual de clases [RBP+91]) para representar las versiones regulares de las características, o las reglas de normalización para representar la forma normal.

En el Composite, se usan listas de características para expresar los argumentos de características compuestas (Figura 4.1, parte (a)). Las restricciones se tratan de manera similar — ver Figura 4.3.

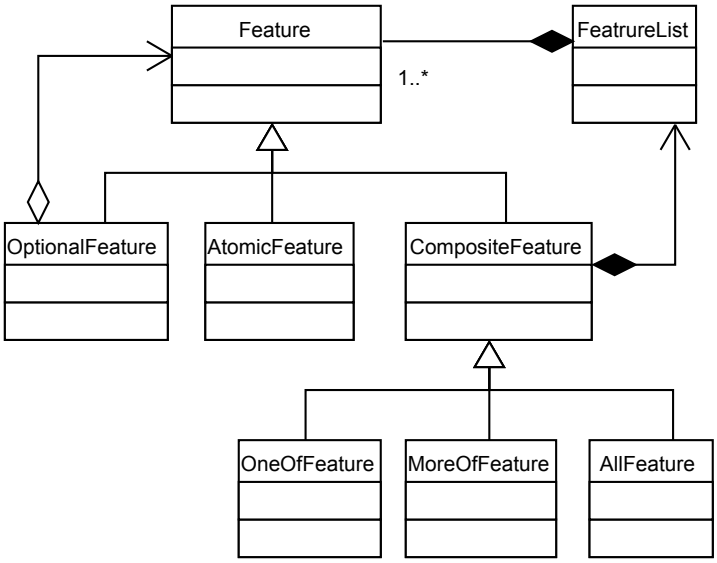
Representación de la semántica

La regularización consiste simplemente en el reemplazo de las características con nombre.

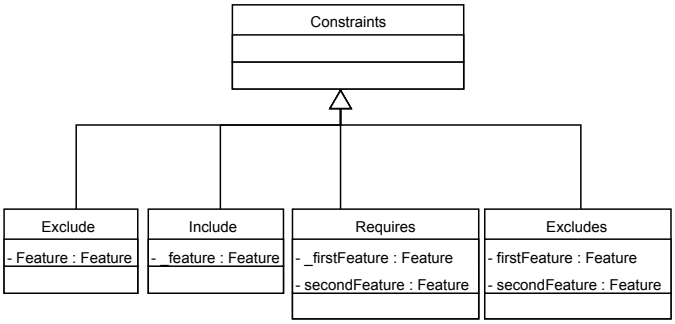
La normalización en cambio tiene que aplicar (doce) diferentes reglas para reducir características duplicadas, desanidar, etc. Como ejemplo de implementación para la normalización se eligió la quinta regla (presentada en la Capítulo 3). Esta regla normaliza características all, sacando ocurrencias anidadas de all en sus argumentos. La clase `RuleManager` posee el método `NormalizationRule5` que es invocado por el proceso que construye formas normales, cuando el usuario así lo pide. El método toma la lista de características que tiene los argumentos de un all, y verifica cada elemento, para ver si tiene una instancia de la clase all, en cuyo caso toma sus argumentos y los pone como argumentos del all que lo encierra — ver la Figura 4.4.

La clase `RuleManager` tiene además métodos para realizar la expansión. Por ejemplo, el método que implementa la segunda regla de expansión se puede ver en la Figura 4.5. Todos los métodos para la expansión asumen que la característica está normalizada y dan un resultado en forma normal.

El paso final en la representación de la semántica es la aplicación de restricciones. Para ello, la característica en forma normal disyuntiva se pasa



(a) Construcción de características



(b) Restricciones

Figura 4.1: Diagrama de clases para la construcción de programas FDL

```
boolean norm = true;
       // Esto se cambia para obtener una característica regular

//Primera definición ->
  // IODevice : all (opt(Printer), mouse, Display,
  // opt(webcamera), keyboard, opt(microphone), opt(Receiver))
FeatureList ioList = new FeatureList(norm);
  // opt(Printer)
Feature printerName = RuleManager.buildAtomicFeature("Printer");
Feature printerOpt  =
     RuleManager.buildOptionalFeature(printerName,norm);
ioList = RuleManager.buildFeatureList(ioList,printerOpt,norm);
  // mouse
Feature mouseAF = RuleManager.buildAtomicFeature("mouse");
ioList = RuleManager.buildFeatureList(ioList,mouseAF,norm);
  // Display
Feature displayName = RuleManager.buildAtomicFeature("Display");
ioList = RuleManager.buildFeatureList(ioList,displayName,norm);
  // opt(webcamera)
Feature webcamAF  = RuleManager.buildAtomicFeature("webcamera");
Feature webcamOpt =
      RuleManager.buildOptionalFeature(webcamAF,norm);
ioList = RuleManager.buildFeatureList(ioList,webcamOpt,norm);
  // keyboard
Feature keyboardName =
    RuleManager.buildAtomicFeature("keyboard");
ioList =
     RuleManager.buildFeatureList(ioList,keyboardName,norm);
  // opt(microphone)
Feature micAF  = RuleManager.buildAtomicFeature("microphone");
Feature micOpt = RuleManager.buildOptionalFeature(micAF,norm);
ioList = RuleManager.buildFeatureList(ioList,micOpt,norm);
  // opt(Receiver)
Feature recName = RuleManager.buildAtomicFeature("Receiver");
Feature recOpt  =
    RuleManager.buildOptionalFeature(recName,norm);
ioList = RuleManager.buildFeatureList(ioList,recOpt,norm);
  // IODevice : ...
Feature ioFeature    = null;
Feature ioAllFeature =
    RuleManager.buildAllFeature(ioList,norm);
```

Figura 4.2: Código Java del ejemplo IODevice

```
ioFeature  =
  FeatureDefinition.addFeature(ioFeature,ioAllFeature,
   (AtomicFeature)RuleManager.buildAtomicFeature("IODevice"));

<...>

Feature printerF =
   RuleManager.buildOptFeature(printerList,norm);
ioFeature        =
  FeatureDefinition.addFeature(ioFeature, printerF,
    (AtomicFeature)RuleManager.buildAtomicFeature("Printer"));

<...>

// Definición de restricciones
Vector cons = new Vector();
  // constraint ->webcamera requires microphone
Constraint requiresCons = new Requires(webcamAF, micAF);
cons.add(requiresCons);
  // constraint ->include lcd
Constraint includeCons = new Include(lcdF);
cons.add(includeCons);

<...>

System.out.println(FeatureSpec.printVersion(ioFeature,cons));
```

Figura 4.3: Código Java del ejemplo IODevice (continuación)

```
public static Feature NormalizationRule_5(FeatureList aFeature){
  FeatureList tmpList = new FeatureList();
  java.util.Enumeration tmpFeaturesEnumerator =
      aFeature.getFeatures().elements();
  while (tmpFeaturesEnumerator.hasMoreElements()) {
    Object obj = tmpFeaturesEnumerator.nextElement();
    if (obj instanceof AllFeature) {
      java.util.Enumeration tmpSubFeaturesEnumerator =
       ((FeatureList)
        ((AllFeature) obj).getFeatures()).getFeatures().elements();
      while (tmpSubFeaturesEnumerator.hasMoreElements())
        tmpList = buildFeatureList(tmpList, (Feature)
                tmpSubFeaturesEnumerator.nextElement(), true);
    }
    else
      tmpList = buildFeatureList(tmpList, (Feature) obj, true);
  }
  return new AllFeature(tmpList);
}
```

Figura 4.4: Implementación de la quinta regla de normalización

```
public static Feature ExpansionRule_2(AllFeature f) {
   FeatureList fs1 = f.getFeatures();
   OptionalFeature opt =
     (OptionalFeature)fs1.getOptionals().firstElement();
   f.getFeatures().remove(opt);
   FeatureList fs2 = fs1.deepCopy();
   f.getFeatures().add(opt.getFeature());
   Feature g = RuleManager.buildAllFeature(fs2, normalized);
   FeatureList fs3 = new FeatureList(normalized);
   fs3 = RuleManager.buildFeatureList(fs3,f, true);
   fs3 = RuleManager.buildFeatureList(fs3,g, true);
   return buildOneOfFeature(fs3, normalized);
}
```

Figura 4.5: Implementación de la segunda regla de expansión

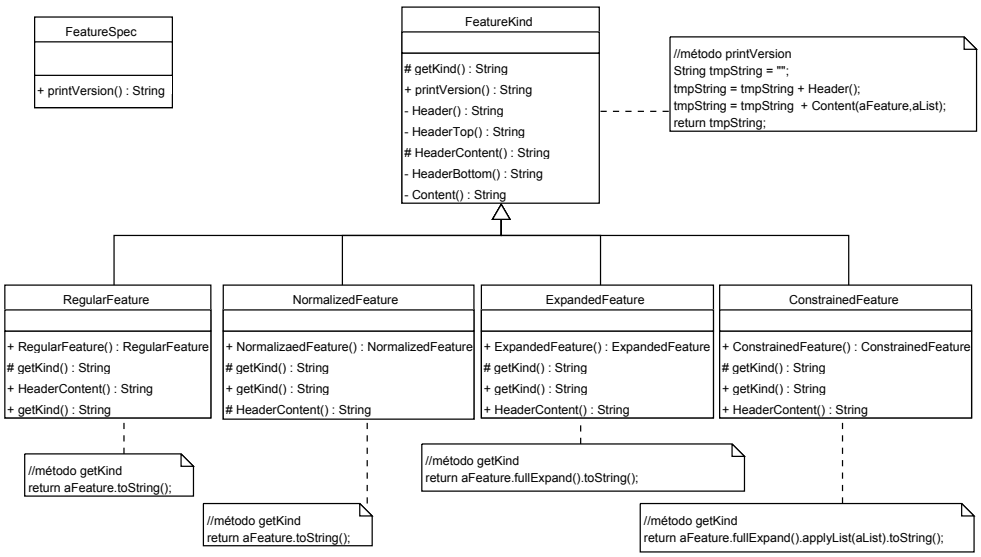


Figura 4.6: Diagrama de clases para la impresión de instancias FDL

sucesivamente entre el conjunto de restricciones en forma de pipeline. Los argumentos de la característica expandida se van modificando al pasar de restricción en restricción. La característica devuelta por la última restricción es el resultado final.

Impresión de características

La última funcionalidad de la API es obtener una representación textual de una instancia específica de FDL. Para construirla se usa el patrón *Template method* [GHJV95], por el cual se tiene una clase abstracta `FeatureKind` que define el método plantilla (template method) `printVersion()`, que usa otros métodos definidos en las subclases para construir la representación final (por ejemplo, `headerContent()` y `getKind()` — ver la Figura 4.6).

4.3 Bibliotecas funcionales

Construcción de características

Para crear las bibliotecas funcionales se usó el lenguaje de programación Haskell [PJH99] pero, por supuesto, son posibles otras alternativas y se pueden tomar decisiones de diseño similares.

Para proveer formas de construir características se define una clase en Haskell (similar a un módulo en ML), que tiene una operación extra para

obtener una representación textual de la característica. La definición es la siguiente:

```
type AtomicFeature = String
class Feature f where
 atomic     :: AtomicFeature -> f
 opt        :: f -> f
 all        :: [ f ] -> f
 one_of     :: [ f ] -> f
 more_of    :: [ f ] -> f

 ppFeature  :: f -> Doc
 isAtomic, isOpt, isOneOf, isMoreOf, isAll  :: f -> Bool
 unOpt :: f -> f
 unAll, unOneOf, unMoreOf  :: f -> f
```

Esta clase se instancia más adelante para representar características regulares, normalizadas y expandidas. La implementación básica (la que se usa para obtener características regulares) está basada en un tipo de datos algebraico, como sigue:

```
data BaseFeature = AF     AtomicFeature
                 | Opt    BaseFeature
                 | All    [ BaseFeature ]
                 | OneOf  [ BaseFeature ]
                 | MoreOf [ BaseFeature ]
                 deriving Eq

instance Feature BaseFeature where
  atomic     = AF
  opt        = Opt
  all        = All
  one_of     = OneOf
  more_of    = MoreOf

  ppFeature = ppBaseFeature

  isAtomic f  = case f of AF _ - > True;_ -> False
  isOpt f     = case f of Opt _ - > True;_ -> False
  isAll f     = case f of All _ - > True;_ -> False
```

```
isOneOf f   = case f of OneOf _ - > True;_ -> False
isMoreOf f  = case f of MoreOf _ - > True;_ -> False
unOpt f     = case f of Opt f' -> f';_ -> f
unAll f     = case f of All fs -> fs;_ -> [f]
unOneOf f   = case f of OneOf fs -> fs;_ -> [f]
unMoreOf f  = case f of MoreOf fs -> fs;_ -> [f]
```

La operación `ppBaseFeature` se discute en la próxima sección *Impresión de características*.

Para representar restricciones se usa un módulo Haskell separado, y se provee otro tipo de datos algebraico:

```
data Constraint = Requires AtomicFeature AtomicFeature
                | Excludes AtomicFeature AtomicFeature
                | Include  AtomicFeature
                | Exclude  AtomicFeature
```

Para contruir el ejemplo de IODevice usando estas operaciones, se escribe lo siguiente:

```
iodevice :: Feature f => f
iodevice = all [ opt (printer), atomic "mouse", display,
                 opt (atomic "webcamera"), atomic "keyboard",
                 opt (atomic "microphone"), opt (receiver) ]
printer  = one_of [ atomic "laser", atomic "inkjet" ]
display  = one_of [ atomic "crt", atomic "lcd" ]
receiver = more_of [ atomic "speaker", atomic "headphones" ]

c1 = Requires "webcamera" "microphone"
c2 = Include "lcd"
```

Observar que la expresión `iodevice` tiene un tipo general — no se limita a características regulares. Esto será útil en la siguiente sección.

Representación de la semántica

La semántica de las características — i.e. normalización y expansión — se representa con dos instancias de la clase definida, que incorporan las reglas en el proceso de construcción. Se definen tipos de wrapper para las dos instancias (`NormFeature` y `ExpFeature`), y luego se instancian con funciones que realizan la tarea requerida. Por ejemplo, la versión para el operador `all` en la instancia `NormFeature` es:

```
all = NF . buildAll . normAllArgs . normFeatureList . map unNF
```

Las operaciones `normFeatureList`, `normAllArgs`, y `buildAll` realizan diferentes tareas necesarias para la normalización (remoción de características duplicadas y opt redundantes — sexta y cuarta regla de normalización, respectivamente), mientras que las operaciones `NF` y `unNF` proveen el *casting* de `BaseFeature` a `NormFeature` y viceversa. El resto de los casos son similares. Cuando se consideran las mismas para la expansión, se tiene lo siguiente:

```
all = EF . normalize . runBF . All . map unEF
```

Nuevamente, las operaciones `EF` y `unEF` proveen el *casting* para y desde `BaseFeaure`, y la operación `runBF` realiza la expansión. Observar cómo la característica resultante se normaliza antes de ser retornada. Los otros casos de expansión son similares.

Un rasgo importante e interesante del sistema de clases de Haskell es que se pueden reusar las definiciones de la versión regular para calcular las versiones normalizadas o expandidas, tan sólo cambiando el tipo — esto es posible porque los constructores de características están sobrecargados.

```
normIOdev :: NormFeature
normIOdev = iodevice

expIOdev :: ExpFeature
expIOdev = iodevice
```

De esta forma, `normIOdev` es la forma normal de IODevice, y `expIOdev` es la forma expandida.

La aplicación de las restricciones se hace como una fase de postprocesamiento para versiones expandidas.

```
applyConstraint::(Eq f, Feature f) => Constraint -> [f]->Bool
applyConstraint (Requires f1 f2) xs = eqRequires f1 f2 xs
applyConstraint (Excludes f1 f2) xs = eqExcludes f1 f2 xs
applyConstraint (Include f) xs = eqInclude f xs
applyConstraint (Exclude f) xs = eqExclude f xs

applyConstraints::Feature f => [ Constraint ] -> f -> Maybe f
```

```
applyConstraints cs f =
   if (isOneOf f) then
        case (applyConstraintsL cs (unOneOf f)) of
             [] -> Nothing
             xs -> Just (one_of xs)
   else Nothing

applyConsAll::Feature f => [ Constraint ] -> f -> [f] -> [f]
applyConsAll cs f =
   if (isAll f) && (satisfiesList cs (unAll f)) then (f:)
   else id

applyConstraintsL::Feature f => [ Constraint ] -> [f] -> [f]
applyConstraintsL cs fs = foldr (applyConsAll cs) [] fs

satisfiesList cs f =
  foldr (\c b -> b && applyConstraint c f) True cs
```

Impresión de características

La construcción de representaciones textuales de características (dada por la operación de clase `ppFeature`), se implementó usando una biblioteca de *Pretty printing* [Hug95]. Esta biblioteca basada en combinadores permite mostrar datos estructurados de una forma adecuada de acuerdo a las características del medio donde se quiere mostrar. La operación se define simplemente recorriendo el tipo algebraico `BaseFeature` y usando los combinadores de pretty printing para construir el resultado deseado. Por ejemplo, el caso de `ppBaseFeature` para el constructor `all` es:

```
ppBaseFeature (All fs)  = text "all" <+> formatList fs

formatList fs = formList (map ppBaseFeature fs)
     where formList []     = text "[]"
           formList [d]    = char '(' <+> d <+> char ')'
           formList (d:ds) = cat (char '(' <+> d :
                                  map (\d -> char ',' <+> d) ds
                                  ++ [ nest (-1) (text " )") ])
```

4.4 Consideraciones sobre APIs y DSLs

Usando bibliotecas para capturar abstracciones, los GPLs se pueden usar para expresar el conocimiento del dominio. Sin embargo, los GPLs usualmente no contienen la estructura sintáctica adecuada para agregar notación específica del dominio. Por otro lado, los DSL tratan de proveer sintaxis concreta para dar simplicidad cuando se usa el conocimiento del dominio específico.

Como se dijo anteriormente en este trabajo, cuando un programador está diseñando una nueva biblioteca de fucniones para un GPL, tiene en general un DSL en mente (usualmente sin saberlo), y si este DSL fuera identificado e implementado adecuadamente podría obtener muchas ventajas. Pero esto no es tenido en cuenta por los desarrolladores de software, debido al gran costo de construcción y poca experiencia cuando construyen *lenguajes pequeños* [Ben86a]. De manera similar, en [BV04] se muestra cómo incluir una notación de DSL particular dentro de un GPL, por asimilación en ese lenguaje, traduciéndolo a alguna operación de la API.

En la literatura relacionada a los DSLs, las bibliotecas de aplicación son mencionadas frecuentemente como sus antecesores. Desde el punto de vista del diseñador hay un pequeño paso desde la construcción de la biblioteca de la aplicación a la construcción del DSL. Sin embargo, suele suceder que los diseñadores no tienen el conocimiento suficiente en la construcción del lenguaje de programación, y por lo tanto la mayor parte de las aplicaciones nunca van más allá de la etapa de creación de la biblioteca de aplicación.

Este trabajo intenta salvar esta distancia, proveyendo sugerencias y consejos sobre cómo conectar la construcción de las APIs con la realización de DSLs usando diferentes enfoques de implementación.

Capítulo 5

Descripción de los enfoques de implementación

En este capítulo se presentan detalles de las diferentes implementaciones del lenguaje FDL, usando los diferentes enfoques descriptos en la Sección 2.2. El orden en que se presentan las implementaciones es una continuación natural desde el "enfoque API" presentado en el Capítulo 4.

Se implementaron los siguientes enfoques, y algunos de ellos en más de un lenguaje, para dar una mayor dimensión de la comparación, teniendo en cuenta diferentes lenguajes de implementación:

- preprocessing
  - source-to-source (usando los lenguajes Java y Haskell)
  - macro processing (usando el lenguaje C++)
- embedded (usando el lenguaje Haskell y una extensión estándar que usa templates)
- interpreter/compiler (usando el lenguaje Java)
- compiler generator (usando las herramientas LISA 2.0 y SmaCC)
- extensible compiler/interpreter (usando Mono-C#)
- COTS (usando XML-XSLT)

para un total de diez implementaciones diferentes del mismo lenguaje, FDL. Los subpatrones de *lexical processing* y *pipeline* dentro del enfoque de *preprocessing* (ver Capítulo 2) fueron omitidos en la comparación, dado que estos enfoques no son adecuados para el lenguaje FDL.

Todas las implementaciones usan las mismas reglas para obtener el resultado final, dando una buena base para la comparación presentada en el Capítulo 6.

Para poner en claro las diferencias entre los enfoques, la Tabla 5.1 presenta una comparación visual entre ellos, conteniendo descripciones sobre la entrada, modelo de transformaciones y salidas.

Se incluyó la construcción de la API en la comparación, sólo para clarificar la diferencia entre GPLs y DSLs: escribir un programa usando la API estrictamente usa la notación del GPL para expresar las construcciones de FDL, mientras que en otros enfoques, el GPL se usa solamente como un lenguaje base o host, y la notación del lenguaje es de alguna manera asimilado en el GPL [BV04].

Todas las implementaciones usan la misma definición básica de FDL, aunque algunas de ellas necesitan hacer cambios menores para adaptarla al enfoque. Anteriormente, en la Figura 3.3, se presentó la gramática básica que se usó. Cuando se detalla cada enfoque, se describen sólo las posibles diferencias con la gramática allí definida.

5.1 Enfoque de Preprocessing

El patrón *preprocessing* puede ser dividido en muchos subpatrones. Se realizaron dos de ellos — *source-to-source transformation* y *macro-processing* — debido a la naturaleza del lenguaje FDL, y se han implementado usando diferentes lenguajes.

El enfoque de transformación fuente a fuente (*source-to-source transformation*) fue implementado de dos formas diferentes: con el lenguaje Java y con el lenguaje Haskell.

El enfoque *macro processing* fue implementado usando C++.

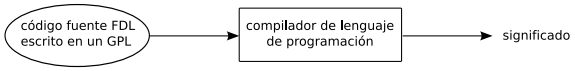
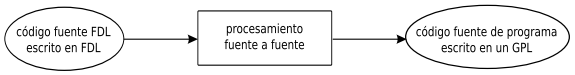
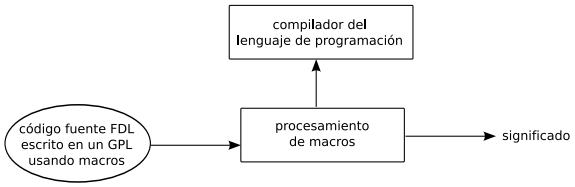
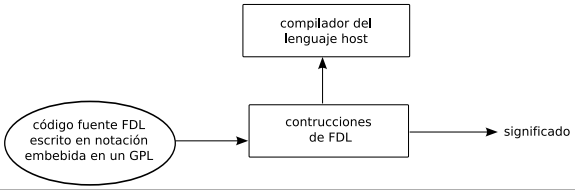
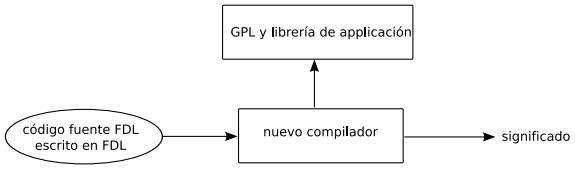
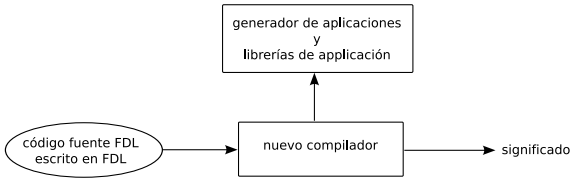
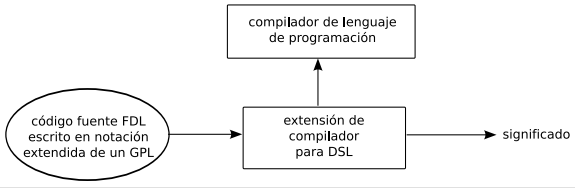
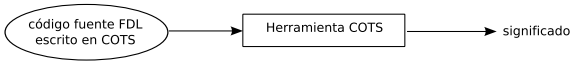
5.1.1 Source-to-source transformation

Este enfoque es el más fácil y natural de todos, pues lo único que se necesita es parsear la notación del DSL y producir un programa en el lenguaje base usando una API como la que fue presentada en el Capítulo 4.

El enfoque de preprocesador fuente a fuente (de [MHS05]) también ha sido llamado *asimilación* [BV04].

Transformación fuente a fuente usando Haskell

La transformación fuente a fuente en Haskell puede llevarse a cabo definiendo un parser particular para el lenguaje que dé el árbol sintáctico repre-

Enfoque	Representación visual
API	
Preprocessing source-to-source	
Preprocessing macro processing	
Embedded	
Interpreter/compiler	
Compiler generator	
Extensible compiler	
COTS	

Cuadro 5.1: Comparación de los enfoques implementados

sentado con un tipo de datos algebraico. En este caso, la entrada del parser es código FDL puro (como se describió en el capítulo Capítulo 3), y se produce un árbol de parsing que luego se recorre para producir código fuente Haskell que usa la API descripta en la sección Sección 4.3. La transformación fue escrita usando Haskell, que es el mismo lenguaje que el lenguaje destino. Por supuesto, esto no es ninguna restricción y se puede usar cualquier otro lenguaje.

Un enfoque muy popular para la construcción de parsers descendientes recursivos es modelarlos como funciones, y definir funciones de alto orden (también llamados *combinadores*) que implementan las construcciones de la gramática tales como selección, secuencia, repetición, etc. Estos parsers forman una instancia de una mónada, que es una estructura algebraica útil para abordar un gran número de problemas de la computación [HM96]. Los parser construidos de esta forma se denominan *parsers monádicos* [HM96, Wad93, HM98].

Siguiendo este enfoque, se definieron los siguientes tipos de datos algebraicos simples para representar el árbol de parsing que es recorrido para general el código de salida:

```
type FeatureName = String
type Name = String
data Diagram = Diagram [Definition] [Constraint]
data Definition = Definition Name Expression
data Expression = All [Expression] | OneOf [Expression]
          | MoreOf [Expression] | Optional Expression
           | Atomic FeatureName | FName FeatureName
data Constraint = Requires FeatureName FeatureName
           | Exclude FeatureName | Include FeatureName
           | Includes FeatureName FeatureName
```

Se realizó el parser monádico; en la Figura 5.1 se muestra parte de su definición, donde se puede observar que la transformación del código fuente es *syntax directed*. Esto quiere decir que para cada construcción del lenguaje FDL se define una única función de parsing, obteniéndose una transformación directa y simple.

El resultado generado por el proceso de parsing se procesa para producir código Haskell que se almacena en un archivo fuente, que cuando es ejecutado proporciona el resultado final deseado. Para poder compilar el archivo fuente generado y correrlo, simplemente se establece una convención de nombre de archivo generado y el mismo programa realiza una llamada al sistema para que ejecute el comando de compilación correspondiente y ejecute el programa. Una estrategia diferente se usó en la implementación en Java que se detalla a continuación.

```
pDiagram = do featureDefs       <-  pFeatureDefinitions
              featureContraints <-  pConstraints
              return (Diagram featureDefs featureContraints)

pFeatureDefinitions = many pFeatureDefinition
pFeatureDefinition = do { name       <- pName; token "$0";
                          expression <- pExpression;
                          return (Definition name expression)}
pExpression = pAtomic `orelse` pAll `orelse` pOneOf `orelse`
              pMoreOf `orelse` pFeatureName `orelse` pOpt
pExpressions = many pExpression
pAll = do {token "$5"; token "("; xs <- pExpressions;
           token ")"; return (All xs)}
pOneOf = do {token "$6"; token "(";xs <- pExpressions;
           token ")"; return  (OneOf xs)}
pMoreOf = do {token "$7"; token "("; xs <- pExpressions;
              token ")"; return (MoreOf xs)}
pOpt = do {token "$8"; token "("; f <- pExpression;
          token ")"; return(Optional f)}

pConstraints = many pConstraint
pConstraint = pInclude `orelse` pExcludes `orelse` pExclude
               `orelse` pRequires
pInclude = do {token "$3"; f <- pLowerCaseId;
               return (Include f)}
pExclude = do {token "$4"; f <- pLowerCaseId;
               return (Exclude f)}
pExcludes = do {f1 <- pLowerCaseId; token "$2";
               f2 <- pLowerCaseId;
               return (Includes f1 f2)}
pRequires = do {f1 <- pLowerCaseId; token "$1";
               f2 <- pLowerCaseId;
               return (Requires f1 f2)}
```

Figura 5.1: Definición del parser monádico en Haskell

Transformación fuente a fuente usando Java

En este caso la entrada también es código FDL, pero la salida es código Java puro. La salida representa características a través de llamadas a la API, construyendo la representación y realizando la regularización, normalización, expansión y la satisfacción de restricciones como se describió en la Sección 4.2.

El string de entrada que representa un programa FDL se transforma en un árbol de parsing usando un parser escrito "a mano". La principal diferencia de este parser con el que se definió en el enfoque *interpreter/compiler* (ver la Sección 5.3) es que la salida del primero es código Java, mientras que la salida del último es el significado del programa FDL.

En el análisis léxico, el flujo de caracteres que representa el programa fuente se lee de izquierda a derecha y se agrupa en *tokens*. El programa que realiza tal tarea se denomina *analizador léxico*, *scanner* o *lexer*. Las expresiones regulares son el método formal más usado para especificar los patrones que conforman los tokens. Los *autómatas de estado finito* son un método para especificar los patrones. El autómata (determinístico) comienza en un estado inicial y cambia de estado de acuerdo a la transición de estado marcada por el caracter actual que lee. El autómata acepta el string si el último caracter leído causa que esté en un estado final, caso contrario lo rechaza.

La forma más común de describir las transiciones de un autómata es mediante una *tabla de transición de estados*. Esta tabla tiene estados y caracteres de entrada como índices. Las entradas de la tabla son números de estado o flags de error.

Una parte del código Java correspondiente a la tabla de transición de estados se muestra en la Figura 5.2. Allí se definió el constructor de la clase Scanner. La clase Scanner tiene una tabla de dos dimensiones ('*automat*') que es inicializada (conteniendo 49 filas (estados) y 256 columnas, una por cada carácter). Además tiene los métodos initAutomat() que define las transiciones de estados (llena la tabla) y nextToken() que obtiene el siguiente token de la entrada.

En la siguiente fase de la implementación del compilador, los tokens se agrupan en frases gramaticales. El programa que realiza la tarea de determinar si las frases gramaticales satisfacen la sintaxis de FDL, se denomina *analizador sintáctico* o *parser*.

Existen dos tipos de enfoques para parsear código: *bottom-up*, que comienza con la entrada y trata de rescribirla hasta el símbolo inicial; y *top-down*, que comienza con el símbolo inicial y trata de rescribir la entrada comenzando desde el símbolo inicial. Los parsers bottom-up se denominan también parsers *shift-reduce*. En este caso, el token actual se pasa a una pila y se

```
public Scanner(String aFileName){
  ...
  automat = new int[48 + 1][256];
  finite = new int[48 + 1];

  ...
  initAutomat();
}
```

Figura 5.2: Definición del constructor de la clase Scanner en Java

lee el siguiente token. Los tokens en la pila se reducirán luego siguiendo alguna producción de la gramática. Los parsers LR pertenecen a este tipo de parsers. En los parsers top-down, se usan generalmente dos técnicas: backtracking y LL(1). Los parsers LL(1) se dividen a su vez en parsers recursivos descendientes, stack parser y table-driven parsers.

Para el análisis sintáctico de programas FDL se usó un *parser descendente recursivo* y esto significa que la siguiente producción a aplicarse debería ser la única determinada por el siguiente token en la entrada; esto se computa usando un conjunto de símbolos llamados *FIRST* y *FOLLOW* [ASU86]. La implementación del parser se obtiene por medio de un conjunto de procedimientos mutuamente recursivos, cada uno implementando una producción.

El árbol de parsing es una instancia de la clase FeatureExpression, correspondiente a la clase raíz de la jerarquía que se muestra en la figura Figura 5.3, que se basa en el patrón Composite [GHJV95].

El método compile de la jerarquía de clase descrito en la Figura 5.3 es el único responsable de producir el código Java final. Sin embargo, las llamadas a éstos métodos son realizadas por un objeto auxiliar, instancia de la clase JavaFDLCodeGenerator, que recorre el árbol de parsing, agregando además el código necesario para producir un programa Java bien formado.

La semántica para FDL se implementó usando la API presentada en la Sección 4.2.

Si se compila el programa FDL IODevice con el compilador source-to-source, se obtiene el código presentado en la Figura 5.4. Observar que el código generado es muy similar al código que un programador tiene que escribir cuando usa la API. La principal diferencia se podría encontrar en el nombre de los identificadores.

La implementación de este enfoque parece ser una tarea relativamente sencilla. Sin embargo, esta simplicidad no es gratis: el chequeo de errores se

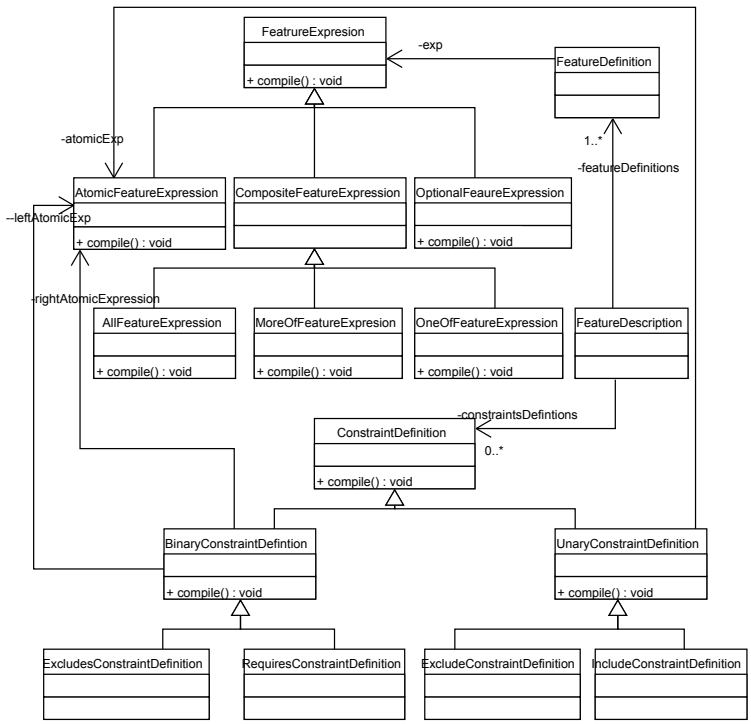


Figura 5.3: Diagrama UML del enfoque source-to-source en Java

deja al compilador Java (*javac*), por lo cual los mensajes no tendrán ninguna correspondencia con el código FDL original. Esto hace que el manejo de errores sea una tarea casi imposible para el experto del dominio.

5.1.2 Macro processing

Algunos compiladores de lenguajes comienzan el proceso de compilación haciendo correr un preprocesador en el código fuente. El preprocesador o *macro expander* es un programa simple que reemplaza patrones en el código fuente con texto específico que el programador definió (usando "directivas" del preprocesador) hasta encontrar una expresión del lenguaje base. Las directivas del preprocesador o *macros* se usan para realizar chequeo de tipos y para incrementar la claridad del código fuente. Por lo tanto, los GPLs que soportan preprocesamiento, como Scheme, C y en particular C++, son convenientes para embeber las construcciones de los DSLs. Las definiciones de macros se usan para definir especificaciones del dominio específico que luego son reemplazadas por código del lenguaje de propósito general que representan su significado.

La implementación usando expansión de macros es bastante compleja,

```
public static void main(String[] args) {
  boolean norm = true;
  Feature root = null;
  FeatureList var1 = new FeatureList(norm);
  Feature var3 = RuleManager.buildAtomicFeature("Printer");
  Feature var2 = RuleManager.buildOptionalFeature(var3,norm);
  var1 = RuleManager.buildFeatureList(var1,var2,norm);
  Feature var4 = RuleManager.buildAtomicFeature("mouse");
  var1 = RuleManager.buildFeatureList(var1,var4,norm);
  ...
  root = FeatureDefinition.addFeature(root,var0,
         (AtomicFeature)
            RuleManager.buildAtomicFeature("IODevice"));

  // Otras definiciones (Printer, Display, Receiver)
  ...

  Vector cons = new Vector();
  Constraint var25 = new Requires("webcamera","microphone");
  cons.add(var25);
  Constraint var26 = new Include("lcd");
  cons.add(var26);
  System.out.println(FeatureSpec.printVersion(root,cons));
}
```

Figura 5.4: Código obtenido por tranformación fuente a fuente para IODevice

porque para poder ser capaz de expresar una especificación DSL como código C++ se necesita forzar cambios en su sintaxis para que el código resulte más declarativo que imperativo.

Usando el preprocesador de C++ se obtiene una sintaxis amigable, que es expandida a código C++ — por ejemplo, se puede definir una macro `BeginSpecification(<spec-name>)` que toma el nombre de la especificación como parámetro y la expande en una definición de clase, donde el nombre de clase es la dada. La estructura de la especificación difiere sustancialmente del código FDL puro; necesita sentencias de apertura y cierre, con la estructura mostrada en el ejemplo de la Figura 5.5.

Una declaración de una característica se escribe como

```
feature <feature-name>= <feature-exp>;
```

donde `<feature-name>` es el nombre de la característica (se puede usar en

la siguiente `<feature-exp>`), y `<feature-exp>` tiene una de las siguiente formas::

- `atomic ("<feature-name>")`, ó `"<feature-name>"`
- `opt (<feature-exp>)`
- `one_of (<feature-exp-list>)`
- `more_of (<feature_exp-list>)`
- `all (<feature_exp-list>)`

donde `<feature-exp-list>` es una lista de `<feature-exp>` separadas por | (ver el ejemplo Figura 5.5). Las comillas dobles se necesitan para distinguir características atómicas de otros tokens, siendo así diferente a la definición del lenguaje FDL, donde la primera definición es la que define la característica que define el diagrama, acá es la última característica la que define el diagrama. La razón de esto es la falta de flexibilidad que proveen las macros — las operaciones usan objetos ya definidos y, por lo tanto, importa el orden en que se definen.

Una restricción se escribe de una de las siguientes formas (las comillas son importantes):

- `constraint "<feature-name>" requires "<feature-name>"`
- `constraint "<feature-name>" excludes "<feature-name>"`
- `constraint include "<feature-name>"`
- `constraint exclude "<feature-name>"`

Antes de escribir el código usando la sintaxis presentada, se debe incluir el archivo `feature_stx_begin.h`, que define las macros de la sintaxis FDL. Una vez que la sintaxis no se requiere más, se debe incluir el archivo `feature_stx_end.h` para descartar la sintaxis especial de FDL. De esta manera, cada especificación de un diagrama debe empezar con la sentencia `#include feature_stx_begin.h` y terminar con la sentencia `#include feature_stx_end.h`.

La semántica del enfoque de *macro processing* se presenta con una API similar a la presentada en el Capítulo 4 — para explicarla, se describen algunas de las macros definidas en el archivo `feature_stx_begin.h`.

La clase que representa programas FDL en la implementación de C++ se llama Specification, y es similar a la clase FeatureSpec definida en la API de JAVA de la Sección 4.2.

La macro para `BeginSpecification` se expande a una definición de la clase Specification, como sigue:

```
class Name : public Specification {
  private:
      list<Constraint*> _constraints;
  public:
      Name(void){
          _constraints = basic_constraints();
      }
      virtual ~Name(void){
          DELETE_ALL(Constraint,_constraints);
      };
      virtual list<Constraint*>& constraints(){
          return _constraints;
      }
      virtual const char* name(){
          return #Name;
      }
```

Observar que esta definición de clase está incompleta: no define los métodos basic_feature ni basic_constraints. Además, no tiene un `};` al final. Estos métodos se escriben entre las sentencias `Begin/EndFeature` y `Begin/EndConstraints`.

Las definiciones de características que aparecen luego de la sentencia `BeginFeature` son declaraciones de variables en C++ con inicialización. El nombre de la variable está dado por la sentencia `<feature-name>`.

Las declaraciones de restricciones son un poco más complicadas. Una declaración de restricción empieza siempre con la macro `constraint`, pero ésta puede ser seguida por un nombre de característica o la palabra clave `include` ó `exclude`, dependiendo de si la restricción es de diagrama o del usuario. El "truco" de implementación está en definir una máquina de estados con tres estados:

- `StInit`, el estado inicial, que implica que la máquina está esperando por un nombre de característica o una palabra clave. Si se da un nombre de característica, se lo guarda y cambia al estado `StCId`, para esperar por el operador binario. Si se da una palabra clave (`include` ó `exclude`), cambiará al estado `StName`.

- `StCId`, significa que la máquina está esperando una palabra clave (`requires` ó `excludes`), que representa el operador binario. Luego cambia al estado `StName` para esperar por el segundo operando.

- `StName` significa que la máquina está esperando un nombre de característica (el segundo nombre de la restricción binaria, o el único de una restricción unaria). Cuando se le da el nombre, la característica está lista para ser guardada, y la máquina vuelve a su estado inicial.

Esta máquina de estados se implementa con la clase ConstraintManag, que mantiene cada restricción definida en una lista. La macro `EndConstraints` recupera la lista y la retorna como resultado del método basic_constraints. Finalmente, la macro `EndSpecification` cierra la definición de la clase escribiendo `};`.

```
BeginSpecification (IODevice)
  BeginFeature
   feature Display  = one_of  ( "crt" | "lcd" )
   feature Receiver = more_of ( "speaker" | "headphones" )
   feature Printer  = one_of  ( "laser" | "inkjet" )
   feature IODevice = all     ( opt( Printer )
                              | "mouse"
                              | Display
                              | opt("webcamera")
                              | "keyboard"
                              | opt("microphone")
                              | opt(Receiver)  )
  EndFeature
  BeginConstraints
    constraint "webcamera" requires "microphone"
    constraint include "lcd"
  EndConstraints
EndSpecification
```

Figura 5.5: Código del ejemplo IODevice usando enfoque preprocessing

Una característica importante de esta implementación es el uso de memoria. Cuando se aplican las reglas de normalización y expansión, las expresiones FDL sufren varios cambios — se crean muchas construcciones FDL y otras se borran. Para simplificar esta tarea, se usan *smart pointers* [Str97]. Un *smart pointer* o puntero inteligente, no requiere borrado explícito; en vez de ello, controla las copias de si mismo para incrementar las referencias en el objeto apuntado, y es el objeto apuntado el que se borra a si mismo cuando ya no quedan referencias. Para controlar la cantidad de referencias al objeto creado, se hace uso del pattern Proxy [GHJV95]. Además se usan *smart*

lists y *smart vectors* que contienen smart pointers como elementos (es decir, agregan una referencia a cada elemento contenido). El código resultante de la clase RuleManager — ver Capítulo 4 — es mucho más simple debido al uso de ésta técnica.

Usando esta implementación, el ejemplo IODevice se puede escribir como se presenta en la Figura 5.5. Notar que el separador '|' entre los elementos de las listas es un operador binario, que toma una `FeatureList` a su derecha y una `Feature` a su izquierda. Además, hay una conversión automática de strings a características atómicas — haciendo la palabra `atomic` innecesaria (esta palabra se usa en algunas implementaciones que se verán más adelante). Después de reemplazar este patrón FDL de la Figura 5.5, el preprocesador obtiene el código fuente C++ presentado en la Figura 5.6.

Como comentario final, la dificultad encontrada en el proceso de debugging y reporte de errores cuando se realizó la tarea. La razón es que luego de cada fase de preprocesamiento, el compilador toma una entrada expandida (la salida del preprocesador), que es muy diferente del código fuente.

5.2 Enfoque Embedded

Un enfoque muy conocido y menos costoso para implementar DSLs es el de embeber las construcciones del dominio específico en un GPL. El DSL "extiende" el lenguaje host definiendo tipos de datos abstractos, clases, métodos y/u operadores [Hud98], de forma tal que los expertos del dominio pueden ser programadores sin aprender mucho sobre el lenguaje subyacente, expresando sus soluciones usando el lenguaje extendido de esta forma.

En este caso especial no se necesita ninguna traducción, lo que significa que el nuevo DSL es un subconjunto (a nivel sintáctico y semántico) de un lenguaje existente. Todo lo que se debe hacer es implementar el sistema de ejecución como una biblioteca de funciones en el lenguaje host.

Si bien cualquier GPL puede ser un lenguaje host, los promotores de los lenguajes funcionales a menudo argumentan que estos lenguajes son los más apropiados para embeber [Hud96a, Hug95]. En ese sentido, Haskell es el lenguaje más usado para implementar el enfoque embebido [PJH99]. Haskell es un lenguaje de programación funcional que usa evaluación *lazy* y un sistema de tipos polimórfico (una extensión del clásico algoritmo de inferencia de tipos de Hindley-Milner [Hin69, Mil78]). Además tiene funciones de alto orden y un sistema de clases que lo hace un lenguaje poderoso al permitir definiciones por recursión sobre los tipos.

```
class IODevice : public Specification {
  private:
    list<Constraint*> _constraints;
  public:
    IODevice(void){ _constraints = basic_constraints();}
    virtual ~IODevice(void){
      for(std::list<Constraint*>::iterator it=_constraints.begin();
          it!=_constraints.end(); ++it) {delete *it;}
    }
    virtual list<Constraint*>& constraints(){return _constraints;}
    virtual const char* name(){return "IODevice";}
  protected:
      virtual Feature* basic_feature(bool normalized) {
        _FEATURE Display  = RuleManager::buildOneOfFeature(
            &((*new FeatureList(normalized)) |
                        "crt" | "lcd"), normalized);
        _FEATURE Receiver = RuleManager::buildMoreOfFeature(
            &((*new FeatureList(normalized)) | "speaker"
                  | "headphones"), normalized);
        _FEATURE Printer  = RuleManager::buildOneOfFeature (
          &((*new FeatureList(normalized)) | "laser"
                  | "inkjet"), normalized);
        _FEATURE IODevice = RuleManager::buildAllFeature    (
          &((*new FeatureList(normalized))
           |RuleManager::buildOptionalFeature(Printer, normalized)
           | "mouse" | Display
           | RuleManager::buildOptionalFeature("webcamera",
                                        normalized) | "keyboard"
           | RuleManager::buildOptionalFeature("microphone",
                                        normalized)
           | RuleManager::buildOptionalFeature(Receiver,
                                        normalized)), normalized);
        return _FEATURE::_feature;
      }
```

Figura 5.6: Código del ejemplo IODevice usando el código del macro procesador C++

```
    private:
      virtual list<Constraint*> basic_constraints(){
        ConstraintManag cManag;
        cManag << "webcamera" << CIdReq << "microphone";
        cManag << CIdInc << "lcd";
        list<Constraint*> tmp = cManag.constraints;
        cManag.constraints.clear();
        return tmp;
      }
};
```

Figura 5.6: Código del ejemplo IODevice usando el código del macroprocesador C++ (cont)

5.2.1 Haskell como lenguaje host

Al comenzar la implementación embebida del lenguaje FDL en Haskell, la mayoría del trabajo estaba hecho ya por la API de la Sección 4.3. En esa sección se pueden encontrar un conjunto de estructuras de datos y funciones que representan la API de FDL en Haskell. Las construcciones de la API conforman una *biblioteca de combinadores* que se usa para expresar la sintaxis y la semántica del dominio.

Lo único que se necesita hacer es integrar las definiciones de características con restricciones en una estructura de datos particular. Para ello se define un nuevo tipo de datos para las especificaciones:

```
data Spec f = Spec (String,f) [(String, f)] [Constraint]
```

Esta especificación contiene la definición principal del programa (el primer par `(String,f)`, que es obligatorio), una lista de las definiciones (que puede estar vacía), y la lista de restricciones (que puede ser vacía también).

En la Figura 5.7 se presenta el programa completo del usuario final para el ejemplo de IODevice en la implementación embebida en Haskell. Para obtener los resultados, se debe evaluar la expresión `testIODevice` y la salida se escribe en una archivo llamado `IODevice.txt`.

Es necesario en este punto dar algunas explicaciones sobre la notación. Primero, en vez de usar características atómicas con un nombre, se usan strings (literales entre comillas dobles ") de la API, y la operación `atomic`, que transforma un string en una característica atómica. Por ello la expresión mouse se escribe `atomic "mouse"` en el programa. La segunda observación acerca de la notación es que, en Haskell, las funciones pueden ser convertidas a notación infija usando comillas (‘). Esto explica la definición de las

```
module Sample where
  import Feature
  import Spec
  import Prelude hiding (all)
  import Constraint

  ioDevice :: Feature f ⇒ f
  ioDevice = all[ printer, atomic "mouse", display,
                  opt(atomic "webcamera"), atomic "keyboard",
                  opt(atomic "microphone"), opt(receiver) ]

  printer, display, receiver :: Feature f ⇒ f
  printer  = one_of [atomic "laser", atomic "inkjet"]
  display  = one_of [atomic "crt", atomic "lcd"]
  receiver = more_of [atomic "speaker", atomic "headphones"]

  constraint1, constraint2 :: Constraint
  constraint1 = "webcamera" `requires` "microphones"
  constraint2 = include "lcd"

  spec :: Feature f ⇒ Spec f
  spec = Spec ("IODevice",  io_device)
              [("Printer", printer), ("Display", display)
              ,("Receiver", receiver)   ]
              [ constraint1,  constraint2   ]
  testIODevice :: IO ()
  testIODevice = do let p = "IODevice.txt"; sep = "\n\n"
                    writeFile  p ""
                    appendFile p (formatTitle
                                    "Regular version")
                    appendFile p
                        (show (io_device  :: RegFeature))
                    appendFile p sep
                    appendFile p (formatTitle
                                    "Normalized version")
                    appendFile p
                        (show (io_device  :: NormFeature))
                    appendFile p sep
```

Figura 5.7: Código del ejemplo IODevice usando el enfoque embebido

```
                    let ef = ioDevice::ExpFeature
                    appendFile p (show ef)
                    appendFile p sep
                    appendFile p
                        (formatTitle "Constrained  version")
                    let res = applyConstraints
                            [constraint1, constraint2 ] ef
                    appendFile p
                       (case res of
                          Just x -> show x
                          _ -> "*there are no features*")
formatTitle text =
 let line = replicate (length text + 10) '-'
 in line ++ "\n--- " ++ text ++ " ---\n" ++ line ++ "\n\n"
```

Figura 5.7: Código del ejemplo IODevice usando el enfoque embebido (cont.)

restricciones; en particular de `constraint1` en el ejemplo. Otro punto muy importante a notar es que, como se dijo en la Sección 4.3, la misma expresión que se usa para construir la expresión `io_device` se usa para obtener todas las versiones: regular, normalizada y expandida, tan solo cambiando el tipo con el que se instancian las funciones sobrecargadas. Como observación final, se hace hincapié en la simplicidad con la que el lenguaje fue embebido en Haskell, pero cuyo costo se paga con desventajas como las mencionadas en la Sección 2.2.

5.2.2 Metaprogramming en Haskell

La metaprogramación (metaprogramming) se refiere a una técnica para escribir programas que representan y manipulan otros programas — por ejemplo: compiladores, generadores de programas, intérpretes — o a ellos mismos (*reflection*).

Usando técnicas de metaprogramming se pueden realizar optimizaciones sobre el código fuente de un programa manipulando, en tiempo de compilación, el árbol sintáctico del código fuente y haciendo transformaciones. Así se produce un código final optimizado, que al ser compilado y ejecutado es mucho más eficiente respecto del código fuente original.

Template Haskell es una extensión estándar del lenguaje Haskell que soporta preprocesamiento en tiempo de compilación del código fuente de los programas [SJ02], permitiendo alterar su semántica. La extensión está so-

portada por el compilador GHC[1] en sus versiones 6 en adelante. Estas extensiones permiten que parte del programa sean generadas automáticamente o manipuladas usando la información estática dentro de una mónada llamada *Quotation*, que además permite generar variables frescas, operaciones de entrada/salida y la reificación de definiciones de tipos y funciones. Las características más relevantes de Template Haskell para este trabajo son los *splices*, la *quasi-notation* y la clase `Lift`, por lo que se recomienda leer el trabajo original sobre Template Haskell [SJ02] para conocer las demás características.

Template Haskell usa como lenguaje de manipulación del árbol sintáctico de los programas al mismo lenguaje de las expresiones manipuladas (Haskell), y las expresiones manipuladas son data types de Haskell que representan las expresiones del mismo lenguaje. Esto hace que los templates sean fáciles de entender y usar por un programador que ya conoce Haskell, algo que no pasa, por ejemplo en C++, donde el sublenguaje de templates conforma un sublenguaje distinto de C++, difícil de entender y de usar [Vel95b, Vel95a, CE00].

Una expresión `e` que debe ser evaluada durante la compilación se pone entre el *splice* $(`e`). La expresión `e` debe ser de tipo `ExpQ`. La expresión es en general la aplicación de alguna metafunción que se reemplaza por su resultado (en el caso de FDL son las reglas de normalización, extensión y la aplicación de las restricciones) dentro de la mónada [Lyn03].

Los templates se definen usando *quasi-quote notation*. Los *quasi-quote brackets* `[|...|]` se colocan alrededor de algún código Haskell que no se evalúa durante la compilación pero son insertados tal cual están en el resultado. El resultado de una *quasi-quotation* es de tipo `ExpQ`. El objetivo detrás de estos operadores es usarlos en expresiones de la forma:

```
e' = $( transformacion [| e |] )
```

Con esto, mientras se compila el código, se puede reificar cierta parte y transformarse para luego hacer el splice del resultado, o bien producir código totalmente nuevo en base a ciertos parámetros que se den.

Como ejemplo de lo antes dicho se presenta la definición de la función `fixpoint`, usada para aplicar las reglas de normalización y expansión repetidamente hasta alcanzar el mínimo punto fijo, es decir, hasta que las reglas no transformen la característica dada como parámetro.

```
fixpoint :: ( Feature -> Feature ) -> Feature -> ExpQ
fixpoint fun x = let x' = fun x
```

[1]GHC, the Glasgow Haskell Compiler, disponible en `http://www.haskell.org/ghc/`

```
                in if x == x' then [| x |]
                    else fixpoint fun x'
```

La normalización se define usando la función `normalize`, que aplica las reglas, de la siguiente manera:

```
normalized :: Feature -> ExpQ
normalized f = fixpoint normalize f
```

Finalmente si se quiere obtener la versión normalizada optimizada de una expresión `fdlProgram`, que representa una característica, se escribe usando *splicing*:

```
$(normalized fdlProgram)
```

Además se instanció la clase `Lift` (parte de la librería de templates), que permite llevar las expresiones de un tipo a la mónada *Quotation*, para ser manipulados. En el código de instanciación del tipo `Feature` para la clase `Lift` que sigue debajo, se puede observar el uso de los constructores de expresiones `ExpQ` que proveen las bibliotecas de templates.

```
instance Lift Feature where
  lift (AF f)      = appE (conE (mkName "AF"))
                          (litE (stringL f))
  lift (Opt f)     = appE (conE (mkName "Opt")) (lift f)
  lift (All fs)    = appE (conE (mkName "All"))
                          (listE (map lift fs))
  lift (OneOf fs)  = appE (conE (mkName "OneOf"))
                          (listE (map lift fs))
  lift (MoreOf fs) = appE (conE (mkName "MoreOf"))
                          (listE (map lift fs))
```

El código que el usuario final debe escribir es igual al del enfoque embebido (Haskell puro), salvo que el código debe almacenarse en un módulo denominado `FDL`, separado del módulo `Main` (este último lo importa). Esta decisión se tomó dado que Template Haskell (aún) no soporta la reificación de funciones de top-level, y por lo tanto se debieron escribir las expresiones normalizadas, expandidas y restringidas en módulos separados que se importan desde el módulo principal.

Finalmente, la traducción del código Haskell embebido (realizado antes del enfoque con templates) a código con los operadores de splices y quasi-

quotations no requirió mucho esfuerzo de implementación — se alteró mínimamente la API de la Sección 4.3 en la construcción de constraints. El principal problema resultó saber poner los operadores descriptos en el lugar adecuado.

5.3 Enfoque Interpreter/Compiler

El siguiente enfoque de implementación se denomina enfoque *handwritten interpreter/compiler* y fue realizado en Java. Usualmente se implementa de las siguientes formas:

- transformando el programa DSL en un código de máquina ejecutable,
- traduciendo las construcciones del DSL en construcciones y llamados a bibliotecas del lenguaje base.

La última alternativa es similar al enfoque de transformación fuente a fuente, pero la principal diferencia es que se realiza un análisis semántico completo en el nivel del DSL.

Las fases de análisis léxico y sintáctico son las mismas que en la implementación fuente a fuente de la Subsección 5.1.1 y no se repetirán nuevamente.

La semántica, por otra parte, es diferente al enfoque anterior. En este caso, las construcciones de FDL no son traducidas a programas en Java que contienen llamadas a la API, sino que son traducidas directamente a su significado. Para cada construcción FDL se define la semántica. Se usa gramática con atributos [Knu68, DJL88, DJ90, AM91, MP00] para especificar formalmente las construcciones de FDL. En las gramáticas con atributos, los atributos se adjuntan a los símbolos no terminales, y para cada producción se adjunta una regla semántica que define cómo se computan los atributos. Un ejemplo de este tipo de regla se muestra a continuación:

Semantic domain:

```
AtomicFeature = (string)
Feature = (string, FeatureList)
FeatureList = Array-of (Feature)
```

Functions:

```
buildAtomicFeature: string -->AtomicFeature
addFeature : Feature * Feature * AtomicFeature -->Feature
```

Description of attributes:

```
Inh: inFeature: Feature
```

```
Syn: outFeature: Feature
```

Production "FeatureDefinition" with attribute evaluation rules:

```
FeatureDefinition ::= featureName : featureExpression
{ FeatureDefinition.aFeature =
        buildAtomicFeature (featureName.lexval)
  FeatureDefinition.outFeature =
       addFeature (FeatureDefinition.inFeature,
       featureExpression.outFeature, aFeature)}
```

Dado que la gramática con atributos para FDL es una gramática que tiene atributos a derecha (*L-attributed*)[2] [DJL88]), los atributos se pueden computar mientras se parsean. Se puede extender un parser descendente recursivo fácilmente insertando las reglas semánticas en el lugar apropiado.

Los atributos heredados actúan como parámetros de entrada (implementados con *call-by-value*) mientras que los atributos sintetizados actúan como parámetros de salida (implementados con *call-by-reference*). En el caso que sólo se adjunten a la producción atributos sintetizados, el atributo puede ser retornado de manera funcional.

Este enfoque difiere de los previos en muchos aspectos. Una de las ventajas de este enfoque es la posibilidad de extender la implementación básica con fases adicionales como chequeo de tipos, optimizaciones y verificaciones específicas del dominio, rutinas de reporte de errores, etc. El mayor problema con este enfoque es que lenguajes no tan pequeños como FDL pueden llegar a tomar un gran tiempo de implementación/codificación. El enfoque que sigue soluciona en parte este problema.

5.4 Enfoque Compiler Generator

Una técnica relacionada al enfoque de *interpreter/compiler* es el enfoque de *compiler/interpreter generator*, que usa generadores de compiladores para automatizar (al menos parcialmente) el proceso de construcción del compilador.

Existen muchas herramientas tradicionales para ayudar en el desarrollo de un lenguaje de programación, que se pueden usar para construir además DSL — ejemplos de esto son los basados en LEX/YACC [ASU86, MB90]

[2]Una gramática de atributos es *L-attributed* si cada atributo heredado en el lado derecho de la producción depende sólo de los atributos de los símbolos previos de la producción de la derecha y los atributos heredados del no terminal a la izquierda de la misma regla de producción.

— u otros entornos de desarrollos de lenguajes modernos como ASF+SDF [BDH+01] y LISA [MKŽ95]. Muchos de estos generadores de compiladores están basados en diferentes métodos formales tales como: gramáticas con atributos, especificaciones algebraicas, semántica natural, etc.

En particular, las gramáticas con atributos [AM91, DJL88, DJ90] son muy útiles para la especificación de la semántica de los lenguajes de programación, la construcción automática de compiladores e intérpretes, la especificación y generación de entornos interactivos de programación y en muchas otras áreas [Paa95]. Las gramáticas de atributos son una generalización de las gramáticas libres de contexto, en las que cada símbolo tiene asociado un conjunto de atributos que conllevan la información semántica, y con cada producción se asocia un conjunto de reglas semánticas. La semántica se da de forma declarativa en vez de operacional.

Para este enfoque se usaron dos herramientas: LISA [MKŽ95] y Smacc[3]. Ambas herramientas tienen el mismo objetivo: facilitar la creación de (parte de) los compiladores de los lenguajes, aunque la primera ofrece mayores ventajas respecto de las herramientas disponibles en el proceso de desarrollo de lenguajes, y la segunda ofrece una adaptación más fácil para lenguajes visuales. En las siguientes subsecciones se dan detalles de cada una.

5.4.1 Entorno de desarrollo LISA

LISA es un sistema portable (escrito en Java) que soporta el desarrollo incremental y visual de los lenguajes, y que genera (además del compilador del lenguaje en Java) un conjunto de herramientas útiles en el desarrollo de los lenguajes. La herramienta LISA integra las siguientes características [MLAŽ00]:

- posibilidad de trabajar en un entorno visual o textual;
- una IDE donde los usuarios pueden especificar, generar, compilar y ejecutar programas programas en un lenguaje nuevo que definen;
- analizadores léxicos, sintácticos y semánticos, que pueden ser de diferentes tipos y operar *standalone*;
- representación visual de diferentes estructuras, tales como autómatas finitos del analizador léxico, diagrama BNF, árbol sintáctico y grafo de dependencias del lenguaje especificado;

[3]SmaCC, a parser generator for Smalltalk — disponible en `http://www.refactory.com/Software/SmaCC/`

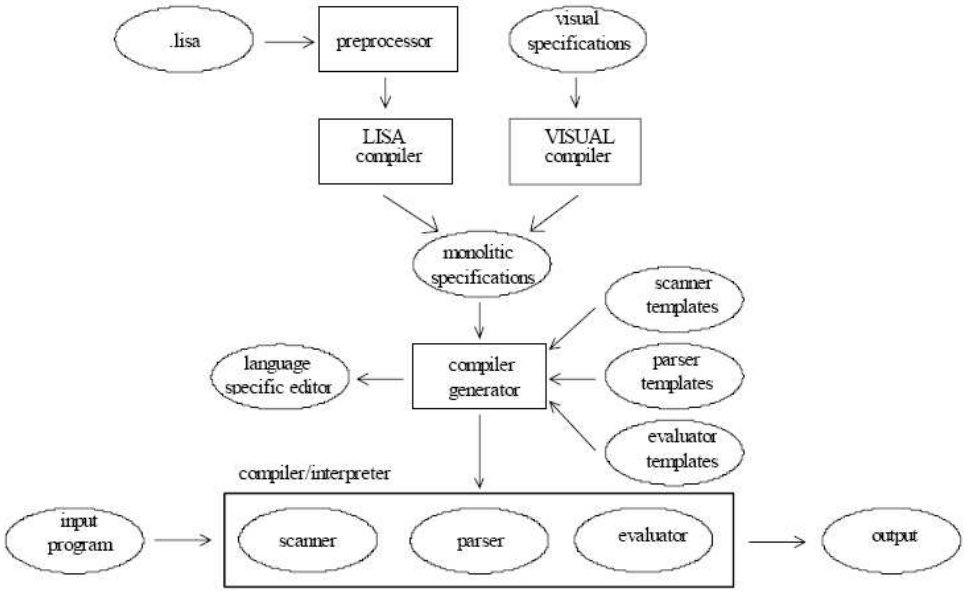


Figura 5.8: Arquitectura del sistema LISA

- especificación de lenguajes con soporte de herencia de gramáticas de atributos múltiples que permite diseñar un lenguaje de forma incremental o reusar algún componente de una especificación de otro lenguaje.

La arquitectura del sistema es la mostrada en la Figura 5.8. En la misma se observa que al ingresar la especificación del lenguaje, junto con las especificaciones visuales, el sistema genera un editor del lenguaje, el scanner, el parser y el evaluador basado en templates. Luego se puede usar el mismo entorno para ingresar un programa del lenguaje especificado, para obtener el resultado final.

Para usar el generador de compiladores LISA, se usó una especificación basada en gramática con atributos. Lo primero que se realizó fue la definición de tokens. Para ello se usaron expresiones regulares, como se presenta en la Figura 5.9. Las expresiones regulares son similares a las que se encuentran en los entornos Unix y no se necesitan mayores explicaciones.

La especificación de FDL usando gramática con atributos se describe en la Sección 5.3, basada en la gramática BNF dada en la Figura 3.3. Se presenta un fragmento de la traducción a código en LISA en la Figura 5.10.

La herramienta LISA deriva automáticamente el tipo de los atributos (heredados o sintetizados), por lo que no es necesario especificar esto en el código. Los nombres de los atributos se eligieron de modo tal que son fáciles de entender — el prefijo `in` para atributos heredados (por ejemplo, `inFeature`)

```
lexicon {
 FeatureName [A-Z][a-zA-Z0-9_]+
 AtomicFeature [a-z][a-zA-Z0-9_]+
 AttributeWord one\-of | more\-of | all | opt
 ConstraintWord include | requires | exclude
 separator \( | \)
 assign \:
 comma ,
 ignore [\ \0x0D\0x09\0x0A]+
}
```

Figura 5.9: Expresión regular para tokens en LISA

y `out` para el atributo sintetizado (por ejemplo, `outFeature`). Sin embargo se deben especificar el tipo de los atributos y los no terminales a quienes se adjuntan — puede ser cualquier tipo válido de Java.

Los atributos se definen en un bloque que empieza con la palabra clave `attributes`, y usualmente el mismo nombre se adjunta a muchos no terminales diferentes, por lo que se puede usar el caracter comodín `*`.

En el código de FDL, el resultado de evaluar definiciones de características se mantiene en los atributos `inFeature` y `outFeature`. Y la clase Vector de Java se usa para almacenar la información de las restricciones (los atributos `inCS` y `outCS`).

Cuando se definen atributos, el usuario puede definir cualquier función, tipo, etc. Esas definiciones se escriben en un bloque que empieza con la palabra clave `method`, y se pueden importar los paquetes definidos por el usuario, para reducir el tamaño del método en LISA e incrementar la legibilidad de la especificación.

Una característica adicional de LISA es la generación automática de varias herramientas para trabajar con el código [HPM+05], tales como un editor del lenguaje (que conoce las definiciones del vocabulario y por lo tanto puede proveer *syntax-highlighting*) y varios inspectores (por ejemplo un visualizador del autómata finito o una animación del árbol sintáctico y semántico) — que son útiles para definir el lenguaje para el dominio específico (ver la Figura 5.11).

5.4.2 Herramienta Smacc

Smalltalk[4] es un lenguaje de programación orientado a objetos puro, basado en conceptos muy simples y con una sintaxis directa. Su claridad y

[4]Smalltalk — disponible en `www.smalltalk.org`

```
language FDL {
  lexicon {
    ...
  }
  attributes
    Feature     *.inFeature, *.outFeature          ;
    FeatureList *.inList   , *.outList             ;
    Vector      *.inCS     , *.outCS               ;
    Constraint  *.cons                             ;
    String      *.inName   , FEATURE_DIAGRAM.result;
  rule Fdl {
    FEATURE_DIAGRAM ::=
        FEATURE_DEFINITIONS
            FEATURE_CONSTRAINTS  compute {
      FEATURE_DEFINITIONS.inFeature = null;
      FEATURE_CONSTRAINTS.inCS      = new Vector();
      FEATURE_DIAGRAM.result        =
           FeatureSpec.printVersion(
              FEATURE_DEFINITIONS.outFeature,
              FEATURE_CONSTRAINTS.outCS);
    };
  }
  ... // algunas reglas se omiten
  rule Fd {
    FEATURE_DEFINITION ::=
      #FeatureName \: FEATURE_EXPRESSION compute {
       FEATURE_DEFINITION.outFeature =
         FeatureDefinition.addFeature(
           FEATURE_DEFINITION.inFeature,
           FEATURE_EXPRESSION.outFeature,
        (AtomicFeature)
        RuleManager.buildAtomicFeature(#FeatureName.value()));
    };
  }
  ... // más definiciones
}
```

Figura 5.10: Código LISA para las producciones con atributos

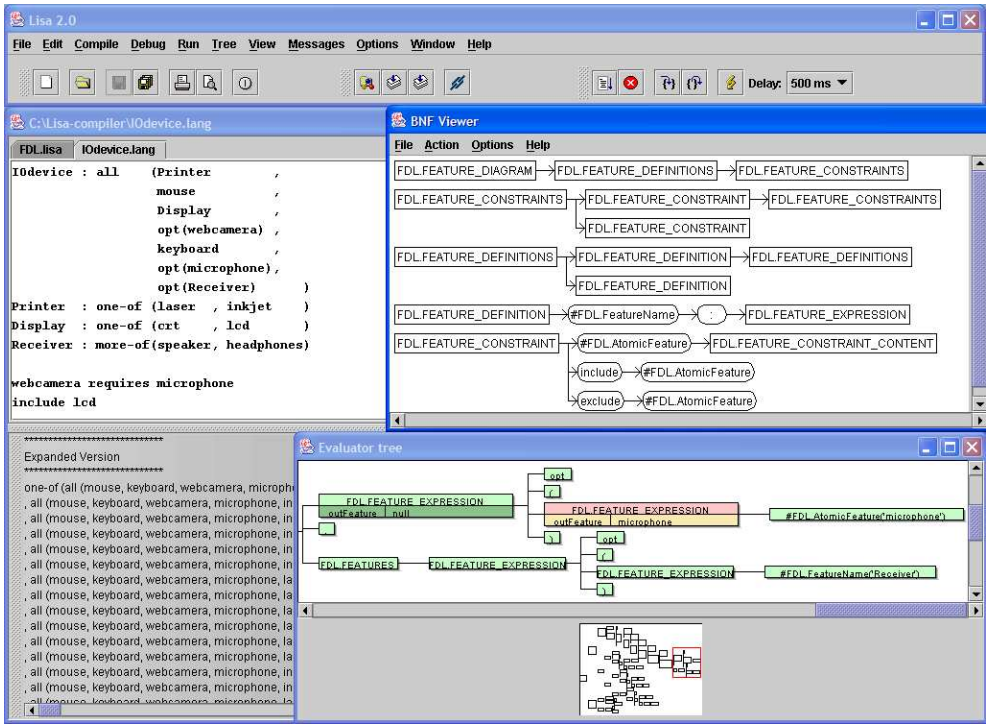


Figura 5.11: Entorno de desarrollo Lisa

simplicidad lo hacen, además, fácil de aprender. Sus defensores dicen que con Smalltalk se puede resolver un problema dos a tres veces más rápido que si se eligen otros lenguajes, y la implementación de FDL no parece ser una excepción. Sin embargo, como se discutirá en el capítulo Capítulo 8, esto es muy relativo.

La implementación de FDL en Smalltalk se hizo usando la versión no comercial de Smalltalk Visual Works 7.3.1, que es un entorno de desarrollo robusto que provee compatibilidad entre diferentes plataformas. Para la generación del compilador de FDL, se usó particularmente la herramienta SmaCC, que proporciona una implementación del conocido Yacc [ASU86, MB90] para Smalltalk. SmaCC (Smalltalk compiler-compiler) es un generador de parser que soporta parsing LALR y LR. El resto del trabajo fue simplemente definir una API similar a las que se desarrollaron anteriormente y una interfaz de usuario simple.

Se escribieron las definiciones del scanner y el parser para FDL dentro del entorno que provee Smacc. En la Figura 5.12 se pueden ver ambas definiciones. Para la especificación de la gramática se usa notación similar a Yacc, pero la semántica (que se encuentra entre corchetes — {}) se define con código Smalltalk al final de cada producción. Para más detalles sobre la

definición del scanner y el parser se puede ver el web site de Smacc.

El resto de la implementación incluye una interfaz visual que usa el framework MVC (Model-View-Controller). Este framework permite separar el modelo (en este caso dado por las clases que definen la semántica de FDL) de la forma en que se vé, dando flexibilidad a la implementación. Si se piensa tener diferentes vistas del mismo modelo (por ejemplo, para adaptar el lenguaje o la herramienta que se usa de acuerdo a los diferentes expertos del dominio), se puede tener en cuenta el framework MVC y los lenguajes que lo soportan.

En la Figura 5.13 se puede ver el workspace creado para trabajar con el lenguaje FDL, usando el framework MVC.

5.5 Enfoque Extensible Compiler/Interpreter

En el enfoque *extensible compiler/interpreter*, los *compiladores abiertos* permiten al programador extender el lenguaje. Los desarrolladores de DSLs necesitan acceder a la definición de la notación del lenguaje base para incorporar la definición sintáctica. Además, se necesita notación combinada para integrar la semántica del DSL con la del lenguaje base.

Mono[5] es una plataforma open-source de desarrollo bajo la licencia GNU (General Public License) [Fou91], patrocinado por Novell y compatible con Microsoft .NET, que encapsula las siguientes tecnologías:

- un compilador para los lenguajes C#, Visual Basic.Net y JScript,
- un entorno de ejecución virtual: un compilador JIT (Just-In-Time, esto es, que compila el código justo antes de ser ejecutado), un compilador AOT (ahead-of-time, esto es, que compila a código nativo un archivo y de esta forma no necesita la compilación JIT cada vez que se ejecute el programa), gestión automática de memoria, un intérprete (Mint) y un motor multiproceso,
- una máquina virtual (Common Language Runtime — CLR) para los bytecodes del Lenguaje Intermedio Común (CLI),
- una implementación de la biblioteca de clases de .NET: manipulación XML, entrada/salida, funciones matemáticas, criptografía, xslt, etc.,
- bibliotecas de clases multiplataforma para el acceso a bases de datos,
- biblioteca de clases UNIX (Mono.Posix) y GNOME (la familia Gtk#).

[5]Plataforma Mono — disponible en `http://www.mono-project.com/`

```
<whitespace> : \s+;
<eol> : \r | \n | \r\n ;
<line> : \% ;
<FeatureName> : [A-Z] \w*;
<AtomicFeature> : [a-z] \w*;
```

(a)Definición del scanner

```
%right "all";
%right "one_of" "more_of";

Diagram: Feature_Definitions 'dl' Feature_constraints 'cs'
                    {FeatureDiagram new: dl constraints:cs};
Feature_Definitions: {FeatureDefinitions new}
          | Feature_Definitions Feature_definition {'1' addDefinition: '2'};
Feature_definition: <FeatureName> 'name' ":" Feature_expression 'f'
                    {FeatureDefinition new: name value feature: f};
Feature_expression:
<AtomicFeature> 'name' {RuleManager buildAtomic: name value}
  | <FeatureName> 'name' {name value}
  | "opt" "(" Feature_expression 'f' ")" {RuleManager buildOptional: f}
  | "one-of" "(" FeatureList 'fs' ")" {RuleManager buildOneOf:fs}
  | "more-of" "(" FeatureList 'fs' ")" {RuleManager buildMoreOf:fs}
  | "all" "(" FeatureList 'fs' ")" {RuleManager buildAll:fs};

FeatureList:
     {OrderedCollection new}
   | FeatureList Feature_expression  {'1' add: '2' ;yourself}
   | FeatureList  Feature_expression "," {'1' add: '2' ;yourself};

Feature_constraints:
    {OrderedCollection new}
  | Feature_constraints Feature_constraint {'1' add: '2';yourself}
  | Feature_constraints Feature_constraint "," {'1' add: '3';yourself};

Feature_constraint:
     <AtomicFeature> 's1' "requires" <AtomicFeature> 's2'
                     {Requires new: s1 value requires: s2 value}
   | <AtomicFeature> 's1' "excludes" <AtomicFeature> 's2'
                     {Excludes new: s1 value excludes: s2 value}
   | "include" <AtomicFeature> 's1' {Include new: s1 value}
   | "exclude" <AtomicFeature> 's1' {Exclude new: s1 value};
```

(b) Definición del parser

Figura 5.12: Definiciones para FDL usando Smacc

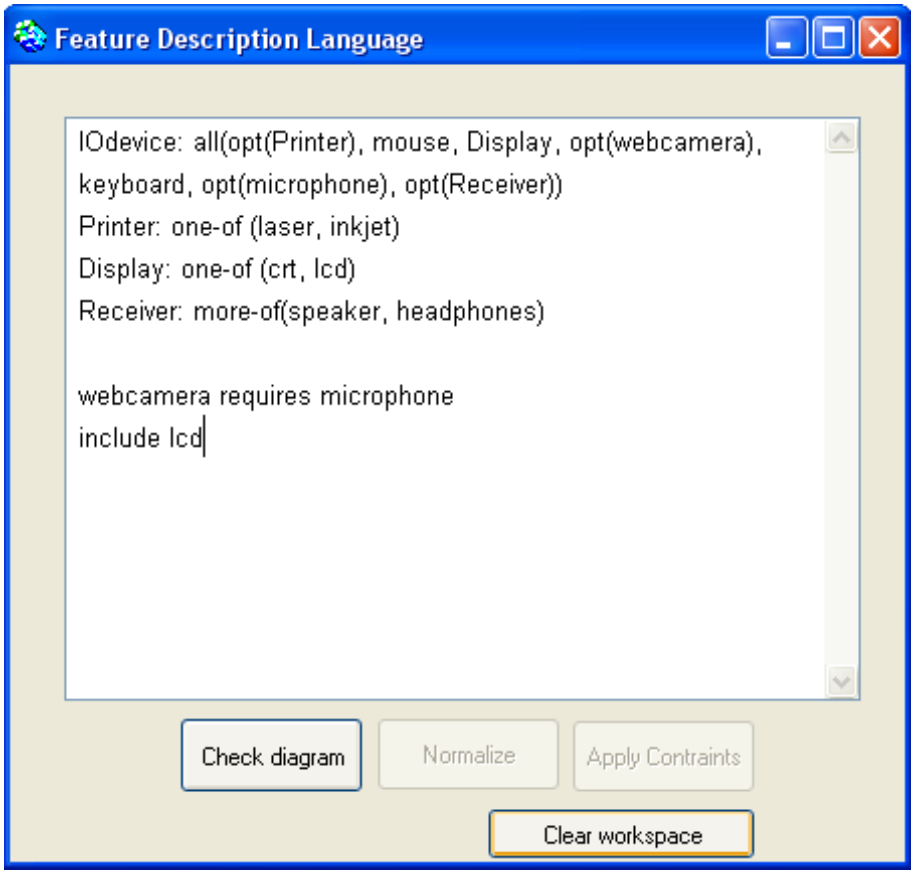


Figura 5.13: Workspace para FDL

Estas tecnologías son parte del framework .Net [TL03]. Por lo tanto, Mono incluye las herramientas de desarrollo y la infraestructura necesaria para correr las aplicaciones de forma independiente de la plataforma — Mono es capaz de correr los binarios de Windows$^{(R)}$ (compilados con Visual Studio .NET, por ejemplo) bajo el entorno de GNU/Linux y otras plataformas.

En la Figura 5.14 se muestra el entorno de desarrollo de Mono. Mono tiene un compilador de C# (MCS) que produce bytecodes de CIL. El bytecode se traduce, por el entorno de ejecución de Mono, a código nativo (soportado con enlaces a bibliotecas de clases). Además, Mono tiene una máquina virtual que es capaz de correr el bytecode directamente.

Comparado con otros lenguajes de programación, Mono provee las siguientes ventajas:

- independencia de plataforma — las aplicaciones de .NET se hacen y ejecutan bajo Linux y son compatibles con varias plataformas,

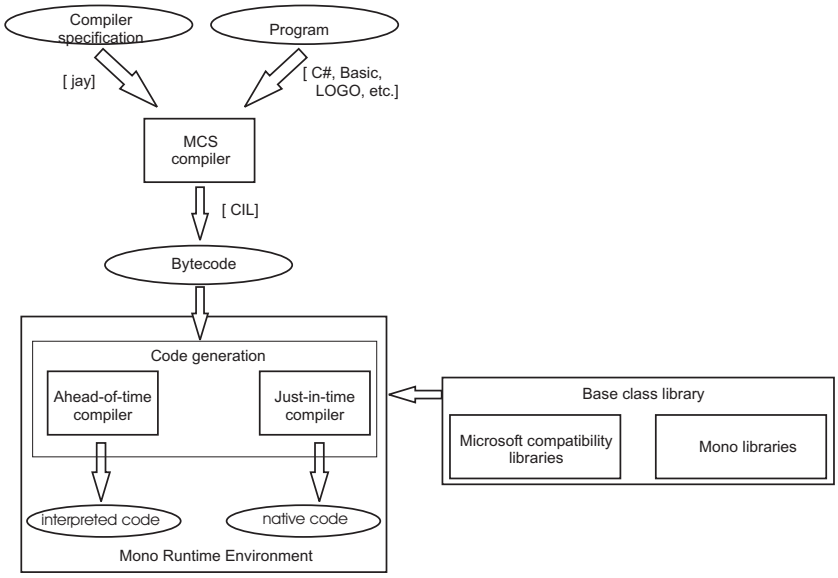


Figura 5.14: Plataforma Mono

- independencia del lenguaje — CIL es una representación en bytecodes que puede ser usada por muchos lenguajes diferentes,
- velocidad — CIL está optimizado[6] y por lo tanto su código nativo producido es mucho más rápido que el interpretado,
- conjunto de bibliotecas poderosas — extenso soporte de bases de datos, servicios web, soporte de xml, etc.,
- encapsulamiento de soluciones de lenguajes (por ejemplo, *garbage collection*),
- enlace a bibliotecas de clases de terceras partes (Gtk+, Gnome).

Se usó Mono 1.1.10 para integrar FDL en notación C#.

La especificación de Mono viene en un archivo `cs-tokenizer.cs` que contiene la especificación para las palabras claves de C#, palabras clases especiales, operadores de caracteres simples y múltiples, números, identificadores, etc. Para incorporar la notación FDL en la de C#, se necesita extender el analizador léxico añadiendo tokens en el método initToken() de la clase Tokenizer. En particular, se incluyen los tokens que son palabras a la tabla hash C# de palabras claves a través del método AddKeyword().

[6]El CLR de Mono implementa un motor JIT para la máquina virtual de CIL, JIT usa un enfoque de generador de generador de código para la compilación

```
AddKeyword ("begin_spec", Token.BEGIN_SPEC);
AddKeyword ("end_spec", Token.END_SPEC);
AddKeyword ("feature", Token.FEATURE);
AddKeyword ("all", Token.ALL);
...
```

Se decidió incluir algunas palabras claves que no están en la especificación original de FDL, pero que se usaron en la implementación del enfoque de macros — **begin_spec**, **end_spec**, etc. Estos tokens adicionales separan claramente la notación básica de C# de la que se agrega para FDL. El reúso de los componentes ya definidos es una fase importante de la construcción del compilador en este enfoque.

El parser de Mono se especifica usando *jay*, modificado para usar C# para las acciones semánticas. El archivo `cs-parser.jay` contiene la especificación del compilador Mono en notación YACC estándar. En FDL se usan inevitablemente algunos tokens que ya existían tales como `OPEN_PARENS`, `CLOSE_PARENS`, `IDENTIFIER`, etc. En particular, como `IDENTIFIER` machea con cualquier cadena de caracteres (sin diferenciar mayúsculas de minúsculas, que permiten diferencias características atómicas y con nombre especial) se decidió mantener la notación de la implementación del enfoque con macros, usando `LITERAL_STRING` para características atómicas (es decir que tienen dobles comillas).

Las producciones de FDL se añaden al archivo `cs-parser.jay`. En la Figura 5.15 se presentan las producciones añadidas y que difieren de la gramática dada en la Figura 3.3, pero sin la correspondiente semántica.

Para incorporar finalmente la sintaxis FDL en C#, se identificó un punto de extensión sintáctica, que es un punto en la gramática de C# donde el símbolo de comienzo <*FDLSpecification*> se puede incluir. Dicho punto es el no terminal `class_member_declaration`, permitiendo al programador agregar las expresiones FDL en cualquier clase C# (como si fuera un miembro).

```
class_member_declaration
 : constant_declaration
 | field_declaration
 ... // Más declaraciones de C#
 | type_declaration
 | FDLSpecification
 ;
```

Para la semántica FDL se usó una API escrita en C#, similar a la descripta en la Sección 4.2. En Mono, la semántica se agrega usando un

```
FDL_specification:
  BEGIN_SPEC OPEN_PARENS IDENTIFIER opt_spec_variable CLOSE_PARENS
  opt_features_decl opt_constraints_decl
  END_SPEC OPEN_PARENS CLOSE_PARENS opt_semicolon ;
opt_spec_variable:
  /* empty */ | COMMA IDENTIFIER ;
opt_features_decl:
  /* empty */ |
  BEGIN_FEATURES OPEN_PARENS CLOSE_PARENS features_decl
  END_FEATURES OPEN_PARENS CLOSE_PARENS opt_semicolon ;
features_decl:
  feature_decl | features_decl feature_decl ;
feature_decl:
  FEATURE IDENTIFIER ASSIGN feature ;
feature:
  all_feature | one_of_feature | more_of_feature|
  opt_feature | atomic_feature ;
all_feature:
  ALL OPEN_PARENS feature_list_elements CLOSE_PARENS ;
one_of_feature:
  ONE_OF OPEN_PARENS feature_list_elements CLOSE_PARENS ;
more_of_feature:
  MORE_OF OPEN_PARENS feature_list_elements CLOSE_PARENS ;
opt_feature:
  OPT OPEN_PARENS feature_list_element CLOSE_PARENS ;
atomic_feature:
  LITERAL_STRING ;
feature_list_elements:
  feature_list_element |
  feature_list_elements COMMA feature_list_element ;
feature_list_element:
  IDENTIFIER | feature ;
opt_constraints_decl:
  /* empty */ |
  BEGIN_CONSTRAINTS OPEN_PARENS CLOSE_PARENS opt_constraints_decl
  END_CONSTRAINTS OPEN_PARENS CLOSE_PARENS opt_semicolon ;
opt_constraints_decl:
  /* empty */ | constraints_decl ;
constraints_decl:
  constraint_decl | constraints_decl constraint_decl ;
constraint_decl:
  CONSTRAINT LITERAL_STRING REQUIRES LITERAL_STRING
| CONSTRAINT LITERAL_STRING EXCLUDES LITERAL_STRING
| CONSTRAINT INCLUDE LITERAL_STRING
| CONSTRAINT EXCLUDE LITERAL_STRING ;
```

Figura 5.15: Producciones de FDL agregadas a la gramática de Mono

bloque {} en cualquier lugar de una producción dada — este bloque contiene el código (asignaciones C#) que define los atributos del no terminal correspondiente.

Compilar el compilador mono de C# es un proceso de dos pasos. Primero, se compilan las especificaciones del parser usando *jay*, obteniendo código C# para el compilador (archivo `cs-parser.cs`). El segundo paso consiste en compilar el proyecto Mono entero (incluyendo las bibliotecas de FDL, etc.) con una versión existente de MSC, para obtener la nueva versión del compilador (incluyendo las extensiones deseadas).

Una vez que se tiene el compilador Mono, se puede escribir el ejemplo IODevice como se presenta en la Figura 5.16.

```
using System;
class hello_IO_Device
  begin_spec(IODevice, strResult)
    begin_features()
      feature IODevice = all ( opt(Printer), "mouse", Display,
                               opt("webcamera"), "keyboard",
                               opt("microphone"), opt(Receiver))
      feature Printer = one_of ( "laser" , "inkjet" )
      feature Display = one_of ( "crt" , "lcd" )
      feature Receiver = more_of ( "speaker" , "headphones" )
    end_features()
    begin_constraints()
      constraint "webcamera" requires "microphone"
      constraint include "lcd"
    end_constraints()
  end_spec()
  static void Main() {
    Console.WriteLine (strResult);
  }
}
```

Figura 5.16: Código de IODevice usando el enfoque extensible compiler en C#

Extender un compilador existente puede requerir astucia del diseñador del DSL. Los tokens de C# pueden ser reusados cuando se extiende la notación DSL (por ejemplo, palabras claves como `begin`) y a pesar de que esta superposición se desea realizar frecuentemente, el diseñador debe estar percatado del significado actual, para evitar confusiones. Esta ambigüedad se puede evitar dando nombre no terminales únicos para la extensión.

Para finalizar, es importante decir que cuando se extiende un lenguaje de propósito general, la información de depuración (debugging) provista al usuario final del DSL es muy importante. Se debe extender la tarea de reportar los errores semánticos (los errores sintácticos se detectan automáticamente). En mono, esto se lleva a cabo mediante la función

```
static void Report.Error(int code, string msg, Location location)
```

que imprime el mensaje de error y termina el proceso de compilación. En el parser Jay, se accede a la variable que contiene la posición actual de parsing (`lexer.Location`), y si se quiere incluir una definición de característica en vez de un nombre de característica (cuando se realiza la normalización) pero la referencia no existe, el error se puede reportar con un mensaje apropiado que indica la posición del error.

5.6 Enfoque COTS

La última implementación considerada en este trabajo es la del enfoque COTS (Commercial Off-The-Shelf) [MHS05, Wil01]. Los productos contenidos en COTS, como Microsoft Access, presentaciones Power Point[7], etc., simplifican el trabajo cuando el análisis y la representación gráfica se puede representar de forma tabular.

Un ejemplo importante de este enfoque se da con el uso del lenguaje XML (*e**X**tensible **M**arkup **L**anguage* o lenguaje de marcado extensible), que fue desarrollado por el World Wide Web Consortium[8] como una versión más simple de SGML. Si bien se lo denomina lenguaje, XML no es exactamente un lenguaje como se ha definido en el Capítulo 1, sino que es una especificación para crear lenguajes de marcado[9]. XML es un protocolo para manejar y contener información que se ayuda con herramientas[10] que capturan el formato de la infraestructura, filtración de datos de documentos, etc. [Ray01].

Actualmente XML es el formato de representación de datos más usada en el mundo, que sirve como estándar para almacenamiento de datos, transporte de datos, intercambio de datos entre diversas aplicaciones, etc.

Las características más importantes de XML para poder crear un DSL son:

[7]Microsoft Access y Microsoft PowerPoint — disponibles en `http://www.microsoft.com/office/`

[8]World Wide Web Consortium — disponible en `http://www.w3c.org/`

[9]Cada marca es información del documento que da el significado de los datos y define además la correlación entre ellos. Usualmente los datos se encierran entre símbolos de marcado denominados *tags*.

[10]XMLSoftware — `http://www.xmlsoftware.com`

- estándar abierto sin dependencia particular en ningún software o hardware,
- representación y almacenamiento de cualquier clase de datos orientados a un dominio específico,
- posibilidades para definir la estructura de los datos (reglas sintácticas usando DTDs (Document Type Definitions), XML Schemas[11], etc.)
- estructura de datos clara y no ambigua que puede ser leída por humanos y por parsers (usando DOM[12] y SAX[13], por ejemplo),
- diversas herramientas como CSS[14] y XSLT[15], que permiten definir diferentes representaciones y formas de organizar los documentos XML,
- definición de una estructura sintáctica y múltiples formas de mostrar los documentos,

Hay dos tipos de documentos XML: libres y estructurados. Los documentos libres tiene un número mínimo de reglas sobre cómo y dónde usar los tags, se puede usa cualquier nombre para los tags y pueden aparecer en cualquier orden y cualquier número de veces. El tipo de documento estructurado involucra un modelo de documento, que es una especificación escrita para capturar la sintaxis específica. Teniendo el modelo definido, el documento XML (llamado instancia) se valida sobre éste. Un documento inválido se puede deber a errores léxicos, orden incorrecto de los tags o algún dato que no se puso. Está claro que para definir la sintaxis de los DSLs usando XML se debe usar un documento estructurado.

La sintaxis y la semántica de un DSL se definen de forma separada en XML. Por ejemplo, la solución más genérica para especificar la notación es por medio de DTDs, donde el modelo de los documentos se define con un conjunto de reglas que capturan el vocabulario del lenguaje (conjunto de elementos permitidos), el orden y el número de ocurrencias de cada elemento y la lista de atributos de los elementos. Un DTD se puede ver como una extensión de BNF. Se pueden encontrar más detalles sobre los DTDs en [Ray01].

El DTD para el lenguaje FDL es el siguiente:

```
<?xml version=" 1.0" encoding="UTF-8"?>
```

[11] XMLSchema — `http://www.w3.org/XML/Schema`
[12] Document object model — `http://www.w3.org/DOM/`
[13] Simple API for XML —`http://www.saxproject.org/`
[14] Cascade Style Sheets — `http://www.w3.org/Style/CSS/`
[15] XSL Transformations — `http://www.w3.org/TR/xslt`

```
<!ELEMENT featureDiagram (featureDefinition*, featureConstraint*)>
<!ELEMENT featureDefinition (featureName, featureExpression)>
<!ELEMENT featureExpression (featureName | atomicFeature | all |
                             one-of | more-of | option)>
<!ELEMENT featureList (featureExpression)*>
<!ELEMENT all (featureList)>
<!ELEMENT one-of (featureList)>
<!ELEMENT more-of (featureList)>
<!ELEMENT option (featureExpression)>
<!ELEMENT featureName (#PCDATA)>
<!ELEMENT atomicFeature (#PCDATA)>

<!ELEMENT featureConstraint (diagramConstraint | userConstraint)>
<!ELEMENT diagramConstraint (requires | excludes)>
<!ELEMENT userConstraint (include | exclude)>
<!ELEMENT requires (atomicFeature, atomicFeature)>
<!ELEMENT excludes (atomicFeature, atomicFeature)>
<!ELEMENT include (atomicFeature)>
<!ELEMENT exclude (atomicFeature)>
```

El primer elemento en una declaración es el string `<!ELEMENT`, seguido del nombre de tag (etiqueta) (por ejemplo `featureDiagram`) y su modelo de contenido se encierra entre paréntesis; la definición termina con el delimitador `>`. Un modelo de contenido puede describir alternativas usando el operador (`|`) — ver la definición del elemento `featureExpression`). Cuando se permiten datos caracter, se usa la palabra clave `#PCDATA`, sin embargo, la restricción de que una `featureName` debe empezar con mayúsculas no se puede expresar en el DTD. Esta es una gran desventaja de los DTDs respecto de las gramáticas libres de contexto.

Otra forma de especificar la estructura de un documento XML es usando XML Schema, que es un estándar creado luego de los DTDs. Esta herramienta usa plantillas (templates) para definir el estilo de los elementos del documento. Contrariamente a los DTD, los XML Schema son archivos XML en sí mismos, que usan un sistema de tipos de datos más potente (tipos base: integer, string, etc.) que pueden restringirse para soportar las necesidades específicas del usuario. Los elementos pueden ser de tipos simples o de tipos complejos; estos últimos pueden contener solamente datos textuales, mientras que los primeros pueden contener atributos y otros elementos. Los XML Schema parecen ser más adecuados definir la sintaxis del lenguaje de dominio específico.

A continuación se presenta una parte del XML Schema para FDL para los elementos `featureName` y `atomicFeature`, donde se puede observar el uso de patrones de restricción en la definición del elemento `featureName`:

```
<xs:element name="featureName">
    <xs:simpleType>
        <xs:restriction base="xs:string">
            <xs:pattern value="[A-Z][a-z_]*"/>
        </xs:restriction>
    </xs:simpleType>
</xs:element>
<xs:element name="atomicFeature">
    <xs:simpleType>
        <xs:restriction base="xs:string">
            <xs:pattern value="[a-z][a-z_]*"/>
        </xs:restriction>
    </xs:simpleType>
</xs:element>
```

Una característica importante de XML es la existencia de herramientas para transformar documentos de manera fácil: diferentes partes de un documento dado se pueden reordenar y presentar de distintas formas. Por ejemplo, la serialización de los objetos, la representación en HTML o en otros formatos (como VoiceXML [Luc00]) se pueden obtener por transformación. Dar semántica para un DSL es otra posible aplicación de las transformaciones.

En el caso del lenguaje FDL, las transformaciones XML se usan para presentar las características en forma regular, normalizada y expandida, y para producir características con restricciones. La salida se hace en formato HTML, pero los resultados parciales se almacenan en formato XML.

La forma más común de especificar transformaciones es mediante XSLT (Extensible Style Language for Transformation (XSLT). Una instrucción de XSLT toma un documento, reorganiza el contenido y genera un resultado usando un motor de transformación.

Un archivo de XSLT es en sí mismo un archivo XML (observar el comienzo del código en la Figura 5.17). Luego se define un elemento del documento del tipo *<xsl:stylesheet ... >* que contiene el resto de los elementos XSLT con que se realiza la transformación.

XSLT se basa en el principio de dividir el documento en partes más chicas que son más manejables, a partir de *templates* o plantillas. Cada reglas de template se enfoca en un nivel del documento XML sin ocuparse del resto del contenido. En el código que sigue se puede encontrar una regla básica de template para FDL. Esta regla contiene referencias a otras reglas (*call-template*), que realizan el procesamiento del documento en otro nivel. En este mismo ejemplo se pueden observar varias llamadas a otros templates (tales como *featureRegular*, *featureNormalized* y *specHTML*). El primer template recibe el elemento raíz del documento XML (*select='/'*) como valor de entrada del parámetro *'feature'*; los demás reciben el resultado de las transformaciones

previas. Los resultados de las llamadas a templates se almacenan en variables usando la regla *xsl:variable*. La segunda parte del template (luego del comentario) genera el texto en el archivo de salida (<*html*>, <*body*>) y llama a la regla *specHTML*, que imprime el resultado final.

La reorganización del documento con XSLT se acopla con el lenguaje XPath, usado para encontrar y unir partes de un documento.

La instancia de documento XML de la Figura 5.18 que corresponde al ejemplo IODevice, produce el resultado de la Figura 5.19 cuando se procesa con XSLT.

```
<?xml version ="1.0" encoding ="UTF-8"?>
<xsl:stylesheet version = "2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html" version="1.0" encoding="UTF-8" indent="yes"/>
<xsl:strip-space elements="*"/>

<xsl:template match="/">
    <xsl:variable name="tmpRegular">
        <xsl:call-template name="featureRegular">
            <xsl:with-param name="feature" select="/featureDiagram[1]"/>
        </xsl:call-template>
    </xsl:variable>
    <xsl:variable name="tmpNormalized">
        <xsl:call-template name="featureNormalize">
            <xsl:with-param name="feature" select="$tmpRegular"/>
        </xsl:call-template>
    </xsl:variable>
    <xsl:variable name="tmpExpanded">
        <xsl:call-template name="featureExpand">
            <xsl:with-param name="feature" select="$tmpNormalized"/>
        </xsl:call-template>
    </xsl:variable>
    <xsl:variable name="tmpConstrained">
        <xsl:choose>
            <xsl:when test="count(featureDiagram/featureConstraint) = 0">
                <xsl:copy-of select="$tmpExpanded"/>
            </xsl:when>
            <xsl:otherwise>
                <xsl:call-template name="applyConstraints">
                    <xsl:with-param name="featurePath" select="$tmpExpanded"/>
                    <xsl:with-param name="constraintsPath"
                        select="featureDiagram/featureConstraint/*"/>
                </xsl:call-template>
            </xsl:otherwise>
        </xsl:choose>
    </xsl:variable>
    <html> <body>
        <xsl:call-template name="specHTML">
            <xsl:with-param name="featureReg" select="$tmpRegular"/>
            <xsl:with-param name="featureNorm" select="$tmpNormalized"/>
            <xsl:with-param name="featureExp" select="$tmpExpanded"/>
            <xsl:with-param name="featureConst" select="$tmpConstrained"/>
        </xsl:call-template>
    </body> </html>
</xsl:template>
```

Figura 5.17: Inicio del código XSLT para la transformación de FDL

```
<?xml version=‘‘1.0’’ encoding=‘‘UTF-8’’?>
<!DOCTYPE featureDiagram SYSTEM ‘‘fdl.dtd’’>
<?xml-stylesheet type=‘‘text/xsl’’ href=‘‘fdl_transform.xsl’’?>
<featureDiagram>
    <featureDefinition>
        <featureName>IODevice</featureName>
        <featureExpression>
            <all>
                <featureList>
                    <featureExpression>
                        <option>
                            <featureExpression>
                                <featureName>Printer</featureName>
                            </featureExpression>
                        </option>
                    </featureExpression>
                    <featureExpression>
                        <atomicFeature>mouse</atomicFeature>
                    </featureExpression>
                    <featureExpression>
                        <featureName>Display</featureName>
                    </featureExpression>
                    ...
                </featureList>
            </all>
        </featureExpression>
    </featureDefinition>
    <featureDefinition>
    ...
    <featureConstraint>
        <diagramConstraint>
            <requires>
                <atomicFeature>webcamera</atomicFeature>
                <atomicFeature>microphone</atomicFeature>
            </requires>
        </diagramConstraint>
    </featureConstraint>
...
</featureDiagram>
```

Figura 5.18: Parte del código XML para el ejemplo de IODevice

```
<html>
<body>
<h2> Regular version </h2>
<p> all(carbody,one-of(automatic,manual) <br>
,more-of(electric,gasoline) <br>
,one-of(lowPower,mediumPower,highPower) <br>
,opt (pullsTrailer)) <br>
</p>
<h2>Normalized version </h2>
<p> all(carbody,one-of(automatic,manual) <br>
,more-of(electric,gasoline) <br>
,one-of(lowPower,mediumPower,highPower) <br>
,opt (pullsTrailer)) <br>
</p>
<h2> Expanded version </h2>
<p> one-of(all(electric,lowPower,automatic,pullsTrailer,carbody) <br>
,all(gasoline,electric,lowPower,automatic,pullsTrailer,carbody) <br>
,all(gasoline,lowPower,automatic,pullsTrailer,carbody) <br>
,all(electric,mediumPower,automatic,pullsTrailer,carbody) <br>
,all(gasoline,electric,mediumPower,automatic,pullsTrailer,carbody) <br>
,all(electric,highPower,automatic,pullsTrailer,carbody) <br>
,all(gasoline,electric,highPower,automatic,pullsTrailer,carbody) <br>
,all(gasoline,highPower,automatic,pullsTrailer,carbody) <br>
,all(electric,lowPower,manual,pullsTrailer,carbody) <br>
,all(gasoline,electric,lowPower,manual,pullsTrailer,carbody) <br>
,all(gasoline,lowPower,manual,pullsTrailer,carbody) <br>
,all(electric,mediumPower,manual,pullsTrailer,carbody) <br>
,all(gasoline,electric,mediumPower,manual,pullsTrailer,carbody) <br>
,all(gasoline,mediumPower,manual,pullsTrailer,carbody) <br>
...
</p>
<h2> Constrained version </h2>
<p> one-of(all(electric,highPower,automatic,pullsTrailer,carbody) <br>
,all(gasoline,electric,highPower,automatic,pullsTrailer,carbody) <br>
,all(gasoline,highPower,automatic,pullsTrailer,carbody) <br>
,all(electric,highPower,manual,pullsTrailer,carbody) <br>
,all(gasoline,electric,highPower,manual,pullsTrailer,carbody)<br>
,all(gasoline,highPower,manual,pullsTrailer,carbody) <br>
) <br>
</p>
</body>
</html>
```

Figura 5.19: Representación HTML de la salida XSLT

Capítulo 6

Comparación de los enfoques

En este capítulo se comparan los enfoques desarrollados en los capítulos anteriores. La comparación se divide en dos áreas principales:

- esfuerzo de implementación del DSL, y
- esfuerzo del usuario final usando el DSL.

La primera área da una idea del esfuerzo que el desarrollador del lenguaje necesita para implementar un DSL (se comparan entre sí diferentes enfoques de implementación y con el "enfoque" API), y la otra área compara el esfuerzo de los usuarios finales para escribir un programa específico del DSL en la variante del lenguaje resultante de aplicar el enfoque.

La primera condición para ser capaz de comparar los enfoques es proveer la misma funcionalidad para todos los enfoques. En los capítulos anteriores se mostró que en todos los enfoques se usó un modelo unificado del lenguaje FDL para obtener la misma funcionalidad. Sin embargo, proveer esto es difícil, dado que algunos enfoques traen consigo herramientas de soporte sin costo alguno (por ejemplo, soporte para reporte de errores).

En este trabajo se siguieron las siguientes reglas:

- respetar el diseño unificado y los detalles de implementación presentados en el Capítulo 5;
- incluir la mínima funcionalidad para presentar el dominio;
- preparar la notación DSL de forma tan óptima como sea posible para los usuarios finales (se considera la notación presentada en el Capítulo 3 como óptima);
- preparar ejemplos para ser compilados y corridos desde una consola.

Enfoque	Lenguaje	Sintaxis	Semántica	LDCE	Ranking
source-to-source	Java	631	1283	1914	10
	Haskell	98	480	578	3
macro processing	C++		1482	1482	8
embedded	Haskell		294	294	2
embedded - templates	Haskell		270	270	1
interpreter/compiler	Java	512	1003	1515	9
compiler generator	LISA	48	911	959	6
	SmaCC	29	794	823	5
extensible compiler	C#	103	515	618	4
COTS	XSLT		1380	1380	7

Cuadro 6.1: Comparación del tamaño del código

6.1 Comparación del esfuerzo de implementación

La medición del esfuerzo de implementación de un DSL es una tarea difícil. Se usaron lenguajes muy diferentes — desde orientados a objetos y funcionales hasta imperativos e híbridos — y por otra parte diferentes enfoques contienen diferentes secciones de código. Por ejemplo, mientras que el enfoque *interpreter/compiler* puede tener cientos de líneas de código para el análisis sintáctico, el enfoque *embedded* no tendrá ninguna. Por eso podría no ser razonable comparar simplemente las líneas de código o el tiempo de codificación para cada una de las implementaciones como una medida del esfuerzo total. Por lo tanto, se usaron diferentes métricas para comparar las implementaciones.

Realizar una comparación en base a medidas o apreciaciones subjetivas no puede llevar a obtener resultados generales y objetivos. Es por eso que la primera comparación describe el esfuerzo necesario para implementar un DSL con una medida (burda) en términos de las líneas de código (LDC). La métrica de LDC es una métrica orientada al tamaño del software que no toma en cuenta la complejidad o funcionalidad del software y depende además de las técnicas de programación y de la forma de diseñar. Sin embargo, las LDC son la base sobre la que se apoyan muchas otras métricas y ha sido usada ampliamente en los proyectos de software. Para mejorar la precisión de las LDC, se midió el tamaño del código con el número de líneas efectivas de código (*LDCE*), que comprenden las líneas sin comentarios, las que no están en blanco, con corchetes o paréntesis solamente.

El esfuerzo de desarrollo del DSL en términos de LDCE se presenta en

la Tabla 6.1. La mayor parte de los enfoques presentados en este trabajo requieren algún tipo de análisis sintáctico y semántico, por lo que el procesamiento se dividió en dos partes: sintáctica y semántica, representadas en las columnas *Sintaxis* y *Semántica*. En la columna *Sintaxis* se presenta la cantidad de LDCE de las producciones del tokenizer y el parser. Las ecuaciones semánticas se incluyen en la columna *Semántica*. En aquellos casos donde no hay necesidad de análisis sintáctico — como es el caso del enfoque *embedded* — la celda correspondiente de la tabla se deja vacía. La última columna de la Tabla 6.1 presenta el ranking. Es importante aclarar que en el caso del enfoque de *macro processing* se dejó la celda vacía pues no se escribió específicamente un lexer ni un parser. Sin embargo se escribieron 130 LDCE para hacer el procesamiento basado en macros.

Entre aquellos enfoques en que se requiere escribir código para la sintaxis FDL, se puede apreciar que las LDCE de los que enfoques con generadores de compiladores son inferiores a los demás. Además se obtienen las ventajas propias de estos generadores (herramientas de desarrollo y soporte asociadas que pueden ser generadas automáticamente, como en el caso de LISA). Para el enfoque *extensible compiler*, se usó el generador de compiladores *jay* (Berkley Yacc parser generator[1]), que explica el número de LDCE similar a los anteriores. La cantidad de código escrito usando parsers monádicos es poca (enfoque *source to source* en Haskell) y cercana a la cantidad de LDCE de los enfoques que toman menos esfuerzo. Claro que en este caso no se tiene ninguna herramienta de adicional generada de forma automática y el reporte de errores suele ser peor (ver Subsección 6.2.3). Finalmente las LDCE en el enfoque de *macro processing* son cercanas a las de los enfoques de menor esfuerzo, sin embargo, como se verá en la Subsección 6.2.1, la sintaxis puede no ser óptima.

Desde el punto de vista de las LDCE totales toma menos esfuerzo implementar DSLs usando los enfoques *embedded* (ambos) o *source-to-source* (usando Haskell), *extensible compiler* y *compiler generator* (SmaCC, Lisa). Se necesita mucho más esfuerzo cuando se usan los enfoques *macro processing*, *interpreter/compiler* o *source-to-source* (usando Java). Si se observa el número total de líneas de código de cada enfoque, se puede observar que los enfoques usando Haskell requieren mucho menos cantidad de código que el resto. La razón para esto es que por un lado Haskell es un lenguaje muy expresivo, y por otro lado este DSL en particular tiene una muy buena estructura subyacente [MLI05], que lo hace adecuado para una implementación funcional.

Sin embargo, lograr dominar un lenguaje como Haskell requiere mucho

[1]Berkley Yacc — disponible en `http://dickey.his.com/byacc/byacc.html`

Enfoque	Leng.	PF	FL	LDCEE	LDCE	Desv.	Ranking
source-to-source	Java	35	53	1431	1914	3.18 %	8
	Haskell	35	38	1026	578	-56.54 %	4
macro processing	C++	27	53	1007	1482	3.56 %	9
embedded	Haskell	27	38	722	294	-71.34 %	2
embedded - templates	Haskell	27	38	722	270	-73.68 %	1
interpreter/compiler	Java	35	53	1431	1515	-18.32 %	6
compiler generator	LISA	35	53	1431	959	-48.30 %	5
	SmaCC	25	27	729	823	-12.91 %	7
extensible compiler	C#	35	53	1431	618	-66.68 %	3
COTS	XSLT	27			1383		

Cuadro 6.2: Comparación del código luego de normalización con PF

esfuerzo (no es simple para programadores novatos obtener programas eficientes) y de esta manera, la aparente ventaja se debe balancear con el resto de las métricas y con otras consideraciones inherentes al enfoque en sí.

Las líneas de código (efectivas) pueden no ser una buena métrica debido a la falta de estándares entre paradigmas o incluso entre lenguajes del mismo paradigma, la forma de diseñar y programar el software. Las métricas de software orientadas a la función estiman el esfuerzo basados en la funcionalidad provista, que luego se usa para normalizar el esfuerzo de la construcción del DSL para cada implementación de forma separada [FSS02].

Albrecht y Gaffney [AG83] han propuesto la métrica de *function points* o puntos de función (PF), que está orientada a la función, lo cual implica que es una medida indirecta del software y del proceso por el cual se desarrolla. La meta es estimar el tamaño del código fuente del proyecto en términos de las LDCE y a partir de un mínimo conocimiento de las funcionalidades y entidades que intervienen. El enfoque se basa en contar los puntos de función a partir de seis componentes: número de entradas de usuario (formularios, entradas desde pantalla, diálogos, etc.), salidas de usuario (reportes, mensajes, etc.), peticiones al usuario, archivos externos e interfaces externas (con otros sistemas). Contempla el sistema como un todo que se divide en determinadas funciones y es independiente de la metodología que vaya a ser utilizada y de la experiencia y del estilo de programación.

En la columna *PF* de la Tabla 6.2, se pueden ver los valores de los puntos de función para cada una de las implementaciones. Para calcular el valor de PF, se asocian tres valores de complejidad (subjetivos) a cada componente. Se usó la fórmula estándar definida por Pressman [Pre92] y se usaron los pesos estándares definidos para cada una de los componentes y complejidades

(complejo, medio, simple). Para entender cómo se obtiene el valor de PF se toma, por ejemplo el cálculo del enfoque *interpreter/compiler* que tiene: dos entradas (la definición del sistema y las restricciones) y se tienen tres salidas (el reporte de FDL, los errores del analizador léxico y los errores del parser). El resto de las categorías quedan vacías. La fórmula queda como sigue:

$$\begin{aligned}\text{entradas} &= \underbrace{0}_{cantidad} * \underbrace{3}_{peso\ para\ simple} + \underbrace{0}_{cantidad} * \underbrace{4}_{peso\ para\ medio} + \\ &\quad \underbrace{2}_{cantidad} * \underbrace{6}_{peso\ para\ complejo} = 12\end{aligned}$$

$$\begin{aligned}\text{salidas} &= 2*4+0*5+1*7=15\\ \text{peticiones} &= 0*3+0*4+0*6=0\\ \text{archivos} &= 2*7+0*10+0*15=0\\ \text{interfaces} &= 0*5+0*7+0*10=0\\ \text{PF} &= \text{entradas} + \text{salidas} + \text{peticiones} + \text{archivos} + \text{interfaces}\\ &= 12+15+0+0+0=27\end{aligned}$$

Algunos enfoques (ver la Tabla 6.2) tienen menos puntos de función que el ejemplo del enfoque *interpreter/compiler*. Por ejemplo, el enfoque *embedded* difiere en las salidas dado que no reporta errores del analizador léxico ni del parser.

El valor calculado de PF se denomina uFP (*unadjusted FP*) o valor de PF no ajustado, que es la forma más usada de PF. Pero el valor se puede ajustar calculando el valor aFP (*adjusted FP*) o valor de PF ajustado. Para esto se deben responder quince preguntas (relacionadas con la comunicación de datos, prestaciones, funciones distribuidas, velocidad de las transacciones, facilidad de instalación, etc.), asignando un valor de 0 a 5 de acuerdo a su influencia. Sin embargo, dos personas que quisieran calificar la aplicación en quince preguntas no lo harán (seguramente) de la misma forma. Además la precisión de los valores no mejoran significativamente para proyectos pequeños y pensados para un procesador, dado que el resultado final de aPF resulta del 65 % de uFP más los valores acumulados de las calificaciones dadas a las preguntas divididos por cien. Por lo tanto se usó la fórmula uFP en la presente comparación respecto de la funcionalidad.

El cálculo de las líneas de código efectivas estimadas (LDCEE) para PF se relaciona directamente con el lenguaje elegido. En el libro de Jones [Jon97] se pueden encontrar los valores promedio de sentencias por puntos de función definidos para cada lenguaje — un *factor del lenguaje* (ver la columna FL en la Tabla 6.2). Algunos de los lenguajes usados en este trabajo (por ejemplo LISA o Smacc) no están presentes en ese trabajo y por lo tanto se tomaron

Enfoque	Lenguaje	API	DSL	Desviación	Ranking
source-to-source	Java	885	910	102.82 %	10
	Haskell	294	284	96.60 %	9
macro processing	C++	1369	113	8.25 %	4
embedded	Haskell	294	0	0 %	1
embedded - template	Haskell	294	0	0 %	1
interpreter/compiler	Java	885	630	71.19 %	8
compiler generator	LISA	885	74	8.36 %	5
	SmaCC	794	29	3.65 %	3
extensible compiler	C#	482	136	28.22 %	6
COTS	XSLT	1383	0	0 %	1

Cuadro 6.3: Comparación de las APIs con los DSLs

los lenguajes más cercanos disponibles. Por ejemplo, para Smacc se usó el factor de Smalltalk, dado que la semántica está dada en Smalltalk. En el caso de XML, el factor del lenguaje es desconocido y esa celda se dejó en blanco.

La comparación de las LDCE con las LDCEE se presenta como el *factor de desviación* en la Tabla 6.2. Se obtuvo menor esfuerzo que el estimado cuando se implementó el DSL usando Haskell como lenguaje host/base y con los enfoques *extensible interpreter/compiler* y *compiler generator* (LISA). Pero se necesitó mucho más esfuerzo que el estimado con el enfoque de *macro processing*.

¿Qué se puede concluir de la Tabla 6.1 y la Tabla 6.2? A pesar de haber usado los puntos de función para normalizar la métrica de LDC, el enfoque más eficiente para implementar el DSL es todavía el *embedded*. Por otro lado, este enfoque tiene una importante desventaja: la notación; por lo que este enfoque debería ser usado si la notación puede no cumplirse estrictamente respecto de la notación original (ver [MHS05]).

Es interesante investigar cuánto esfuerzo se necesita para pasar de la API al DSL. Para llevar a cabo esto, se divide la implementación en dos partes (Tabla 6.3): la API y el resto del trabajo (columna *DSL*). La columna *desviación* muestra la proporción entre ambas partes. Por ejemplo, en el enfoque *handwritten interpreter/compiler* se necesitó 71,19 % de código adicional para obtener el DSL. El costo de implementación puede ser modesto usando generadores de compiladores (por ejemplo, usando LISA sólo se necesitó 8,36 % de trabajo adicional).

Enfoque	Leng.	IODevice	Car	Bicicleta	Test	Desv.	Rank.
FDL puro		6	6	8	14		
API	Java	58	53	69	153	9.79	10
source-to-source	Java	6	6	8	14	1	1
	Haskell	6	6	8	14	1	1
macro processing	C++	13	13	14	20	1.76	6
embedded	Haskell	39	37	39	45	4.7	9
embedded - templates	Haskell	16	16	18	24	2.17	7
interpreter/compiler	Java	6	6	8	14	1	1
compiler generator	LISA	6	6	8	14	1	1
	SmaCC	6	6	8	14	1	1
extensible compiler	C#	19	16	17	23	2.20	8
COTS	XSLT	103	95	110	285	17.44	11

Cuadro 6.4: Comparación de los tamaños de los programas DSL (LDCE)

6.2 Esfuerzo del usuario final

6.2.1 Comparación de programas de usuarios finales

Durante el desarrollo de este trabajo se han mostrado desventajas de algunos enfoques. Algunos de ellos son incapaces de expresar la notación pura de FDL (como se ha dicho en el Capítulo 5 y se puede observar en la Figura 6.1) debido al reúso de las facilidades de los lenguajes usados o simplemente por seguir sus reglas. Sin embargo las implementaciones del DSL elegido (excepto con los enfoques *embedded* y COTS) proveen notaciones suficientemente cercanas (ver la Tabla 6.4) a la notación original [DK02]. Para obtener una idea más precisa de cuán diferentes pueden ser los códigos resultantes, se midieron las LDCE para el programa estudiado de la Figura 3.1 y otros programas FDL adicionales que se presentan en la Tabla 6.4. Todos los programas de ejemplo se pueden ver en el Apéndice A, en notación FDL pura. El *factor de desviación* en la Tabla 6.4 denota las proporciones promedio entre el DSL obtenido y la notación FDL pura. Se puede observar que los programas DSL en algunos enfoques (COTS, *embedded*, *extensible compiler* y *macro processing*) necesitan muchas más líneas (efectivas) de código para escribir programas FDL. Por ejemplo, en el enfoque *embedded* puro se necesitan casi cuatro veces más líneas de código para expresar los programas FDL. Además, la Tabla 6.4 confirma las ventajas de los DSLs respecto a la solución basada en APIs, dado que los programas DSL fueron ocho veces más cortos (comparar los programas escritos con la API y aquellos escritos con

Source to source (Java, LISA, Haskell) - Interpreter/compiler (Java) - Compiler generator (LISA, Smalltalk/Smacc)

```
IODevice: all (opt(Printer), mouse, Display, opt(webcamera), keyboard, opt(microphone), opt(Receiver))
Printer : one-of (laser, inkjet)
Display : one-of (crt, lcd)
Receiver: more-of(speaker, headphones)

webcamera requires microphone
include   lcd
```

Macro processing (C++)

```
BeginSpecification(IODeviceSpec)
BeginFeature
    feature Display  = one_of  ( "crt" | "lcd" )
    feature Receiver = more_of ( "speaker" | "headphones" )
    feature Printer  = one_of  ( "laser" | "inkjet" )
    feature IODevice = all ( opt( Printer ) | "mouse" | Display | opt("webcamera") | "keyboard" |
                             opt("microphone") | opt(Receiver))
EndFeature
BeginConstraints
    constraint "webcamera" requires "microphone"
    constraint include "lcd"
EndConstraints
EndSpecification
```

Embedded (Haskell, Template Haskell)

```
ioDevice,printer, display, receiver :: Feature f => f
ioDevice = all [opt (printer), atomic "mouse", display, opt (atomic "webcamera"), atomic "keyboard",
                opt (atomic "microphone") , opt (receiver)]
printer   = one_of [ atomic "laser", atomic "inkjet"]
display   = one_of [ atomic "crt", atomic "lcd"]
receiver  = one_of [ atomic "speakers", atomic "headphones"]

constraint1, constraint2 :: Constraint
constraint1 = "webcamera" `requires` "microphone"
constraint2 = include "lcd"

spec :: Feature f => Spec f
spec = Spec ("IODevice", ioDevice) [("Printer", printer),("Display", display),("Receiver", receiver)]
            [constraint1, constraint2]
```

Extensible Compiler (C#)

```
using System;
class hello_IODevice{
   begin_spec(IODevice, strResults)
    begin_features()
    feature IODevice = all (opt(Printer), "mouse", Display, opt("webcamera"), "keyboard", opt ("microphone"),
                            opt(Receiver))
    feature Printer = one_of ("laser", "inkjet")
    feature Display = one_of ("crt","lcd")
    feature Receiver = more_of ("speaker", "headphones")
    end_features()
    begin_constraints()
    constraint "webcamera" requires "microphone"
    constraint include "lcd"
    end_constraints()
   end_spec()

   static void Main(){ Console.WriteLine(strResults); }
}
```

COTS (XML)

```
<featureDiagram>
<featureDefinition>
    <featureName>IODevice</featureName>
      <featureExpression>
          <all>
           <featureList>
             <featureExpression> <featureName>Printer</featureName></featureExpression>
             <featureExpression> <atomicFeature>mouse</atomicFeature></featureExpression>
             <featureExpression><featureName>Display</featureName></featureExpression>
  ...
```

Figura 6.1: Comparación del código de IODevice usando los diferentes enfoques

las especificaciones FDL en la Tabla 6.4).

Sin embargo, las LDCE pueden no dar información precisa sobre las diferencias sintácticas entre los diferentes programas. Por eso se realizó una comparación en términos de la cantidad de palabras, usando el comando UNIX, *wc* [Ray04]. Los resultados se pueden observar en la Tabla 6.5. En cada celda se observa la cantidad de palabras usadas en cada implementación para los ejemplos de programas de la tabla anterior, junto con la diferencia (encerrada entre paréntesis) con respecto a la notación FDL original. Es importante remarcar que se contaron como palabras los separadores, delimitadores, etc., y en el caso especial de la implementación COTS, se tomaron como palabras cada uno de los tags junto con los delimitadores($<$ y $>$) y no por separado. En la columna $\sum$ se observa la suma de esas diferencias y luego se presenta una columna de ranking.

Al observar la Tabla 6.5, se puede deducir que una palabra de la notación original puede demandar mucha cantidad de palabras en algunos enfoques (como *embedded* o COTS) para poder expresar lo mismo. En el caso más evidente, la API necesita hasta casi diez veces la cantidad de palabras que se necesitan en el DSL. Esto es una evidencia suficiente para convencerse del beneficio del uso de los DSLs.

Es importante observar además que las diferencias entre las dos implementaciones en Haskell no debe ser considerada importante, pues en realidad la notación de la implementación pura puede ser llevada fácilmente a la notación con templates. Como última observación importante, en la suma en la implementación API se consideró que cada mensaje enviado era una nueva palabra (además del punto que separa al objeto del mensaje), por lo que se explica en parte la gran diferencia con el resto de las implementaciones.

Las líneas de código y la cantidad de palabras pueden no brindar aun suficiente información sobre diferencias a nivel sintáctico y léxico. Por ejemplo, se usó el operador '|' como separador en las listas en la implementación del enfoque *macro processing* (en vez de la coma de la notación FDL original)), etc. Por lo tanto, las soluciones difieren en mucho más de lo que las LDC y la cantidad de palabras pueden decir.

Para capturar dichas diferencias se pueden usar programas que comparan similaridad, como por ejemplo MOSS [SWA03], que son muy útiles para detectar copias cuando se usan programas bajo licencia GPL [Fou91]. En este caso se usó un programa bajo dicha licencia de detección de plagios llamado Copyfind[2] que permite contar los matchings existentes entre los diferentes códigos de programa FDL. Este programa examina un conjunto de docu-

[2]Copyfind - Software to detect plagiarism — disponible en `http://plagiarism.phys.virginia.edu/software.html`

Enfoque	Leng.	IODevice	Car	Bicicleta	Test	$\sum$	Rank.
	FDL	**30**	**26**	**31**	**78**		
API	**Java**	**308 (278)**	**279 (253)**	**352 (321)**	**839 (761)**	**1613**	**11**
source-to-source	**Java**	**30 (0)**	**26 (0)**	**31 (0)**	**78 (0)**	**0**	**1**
	Haskell	**30 (0)**	**26 (0)**	**31 (0)**	**78 (0)**	**0**	**1**
macro processing	**C++**	**43 (13)**	**39 (13)**	**48 (17)**	**99 (21)**	**64**	**6**
embedded	**Haskell**	**205 (175)**	**200 (174)**	**214 (183)**	**304 (226)**	**758**	**10**
embed. - templates	**Haskell**	**86 (56)**	**84 (58)**	**95 (64)**	**186 (108)**	**286**	**8**
interp./compiler	**Java**	**30 (0)**	**26 (0)**	**31 (0)**	**78 (0)**	**0**	**1**
compiler generator	**LISA**	**30 (0)**	**26 (0)**	**31 (0)**	**78 (0)**	**0**	**1**
	SmaCC	**30 (0)**	**26 (0)**	**31 (0)**	**78 (0)**	**0**	**1**
extensible compiler	**C#**	**58 (28)**	**54 (28)**	**62 (31)**	**122 (44)**	**131**	**7**
COTS	**XSLT**	**183 (153)**	**151 (125)**	**173 (142)**	**410 (332)**	**752**	**9**

Cuadro 6.5: Comparación de cantidad de palabras de los programas FDL

mentos de los que extrae porciones para observar palabras de una longitud determinada mínima que se corresponden. El programa genera reportes que contienen los textos comparados, los fragmentos donde se corresponden y un porcentaje de correspondencia.

Se compararon los códigos de los cuatro programas de la Tabla 6.4 con respecto al programa FDL en notación pura (no se tuvo en cuenta en este caso el resultado de la API). Se consideró además que había un matching sólo en el caso que las palabras tuviesen los mismos caracteres (respetando minúsculas y mayúsculas) que componen la palabra, para tener buena precisión.

Los resultados de la comparación respecto de los matching se muestran en la Tabla 6.6. En cada celda se indica el porcentaje de matching respecto de la cantidad total y en la penúltima columna se presenta un promedio de esos porcentajes para cada enfoque. En la última columna se muestra la posición en el ranking.

De la tabla Tabla 6.6 se puede observar cómo difiere la notación específica de la notación FDL pura. Se confirma que los enfoques COTS y *embedded* (incluido el enfoque con templates) alteran dráticamente la sintaxis original. En los enfoques *source-to-source*, *interpreter/compiler* y *compiler generator* se alcanzó 100 % de correspondencia y en los enfoques *macro processing* y *extensible compiler* la similaridad no es tan buena (alrededor o menor del 50 %).

Enfoque	Leng.	IODevice	Car	Bicicleta	Test	Prom.	Rank.
source-to-source	Java	100 %	100 %	100 %	100 %	100 %	1
	Haskell	100 %	100 %	100 %	100 %	100 %	1
macro processing	C++	42 %	23 %	20 %	48 %	33.25 %	7
embedded	Haskell	9 %	5 %	3 %	17 %	8.5 %	10
embedded - templates	Haskell	19 %	12 %	7 %	25 %	15.75 %	8
interpreter/compiler	Java	100 %	100 %	100 %	100 %	100 %	1
compiler generator	LISA	100 %	100 %	100 %	100 %	100 %	1
	SmaCC	100 %	100 %	100 %	100 %	100 %	1
extensible compiler	C#	46 %	30 %	42 %	58 %	44 %	6
COTS	XSLT	11 %	10 %	14 %	13 %	12 %	9

Cuadro 6.6: Comparación de correspondencia de los programas DSL

6.2.2 Performance

La performance es un factor que no debería ser desestimado o subestimado en algunos dominios de problemas. Con ese propósito, se creó un ejemplo figurativo (Test de la Tabla 6.4), que se puede ver en el Apéndice A, y cuyo objetivo es evaluar la performance. El programa consta de 15 LDC y tiene una variabilidad original de 5940. Por otra parte, se midieron los demás programas y dado que los resultados fueron similares, fueron excluidos de la Tabla 6.7.

Todos los enfoques fueron testeados en la misma computadora[3], corridos en consola y por cada uno se tomó el promedio de cinco ejecuciones (la varianza de los tiempos tomados fue despreciable en todos los casos). El programa FDL se lee desde un archivo (sin interacción intermedia por parte del usuario) y el resultado se almacena en un archivo de texto.

En el caso del enfoque COTS, la salida es un archivo HTML, que se puede leer en cualquier navegador. En el caso de la implementación con Smalltalk/Smacc, la versión que se usó de VisualWorks no es comercial y no se pueden generar archivos ejecutables que puedan correrse desde consola. Por ello se preparó una imagen *headless* del entorno de desarrollo Smalltalk, es decir, una imagen que no necesita interacción con el usuario y no hace uso de pantallas. La imagen ejecuta el código Smalltalk que se encuentra en el archivo *headless-startup.st*.

Para tomar los tiempos de ejecución se usó CYGWIN[4], un entorno de

[3]Características: AMD 64bits Athlon 3000+ box, 512 Mb RAM, sistema operativo Microsoft$^{(R)}$ Windows XP

[4]Cygwin - Linux-like environment for Windows — disponible en

Enfoque	Lenguaje	Tiempo	Ranking
source-to-source	Java	53s 313ms	8
	Haskell	15s 795ms	3
macro processing	C++	22s 109ms	4
embedded	Haskell	15s 074ms	2
embedded - templates	Haskell	9m 3s 337ms	9
interpreter/compiler	Java	41s 244ms	6
compiler generator	LISA	47s 364ms	7
	SmaCC	27s 244ms	5
extensible compiler	C#	2s 387ms	1
COTS	XSLT	Sin resultados	10

Cuadro 6.7: Comparación de performance temporal de los programas DSL

ejecución con licencia GPL [Fou91] para los sistema operativos Windows que consiste en:

- una DLL (cygwin1.dll) que actúa como un emulador de la API de Linux, en términos de llamadas al systema Win32, y
- una colección de herramientas de Linux

Entre las herramientas disponibles se encuentra el intérprete de comandos *GNU Bourne Again Shell* o *Bash* que se usó como consola, el comando *time* para medir los tiempos de ejecución de los programas y el compilador *g++*.

En la Tabla 6.7, la columna Tiempo presenta los tiempos de compilación y ejecución del programa FDL. En el enfoque *source to source* el tiempo para la transformación al código del lenguaje base se incluye en el resultado.

Los enfoques que más rápido realizan el trabajo son el *extensible compiler*, seguido de los traductores escritos en Haskell (a excepción del que usa templates), *macro processing*, el enfoque *compiler generator* con Smacc y finalmente los traductores escritos en Java. Por otro lado, el enfoque COTS no retornó los resultados finales de la transformación y el procesador XSLT indicó que había demasiadas expresiones anidadas para procesar.

En el caso del enfoque con templates en Haskell, se han tenido dificultades para compilar ejemplos extensos o de variabilidad superior a 200, debido a que no era suficiente el tamaño de la heap del compilador. Para superar estas dificultades se parametrizó el compilador *GHC* para modificar el tamaño de la heap. El tiempo que tarda en dar los resultados este enfoque (mayor a 9

`http://www.cygwin.com/`

minutos) se debe a la conversión que se hace del código Haskell a su representación monádica, que se va generando hasta llegar a la versión definitiva. En C++, por ejemplo, las sentencias de loop deben ser escritas con templates recursivos, ya que no se pueden usar las sentencias de loop del lenguaje pues se compilan directamente sin transformarse [CE00]. Esto lleva a una gran ineficiencia temporal para generar los programas con este enfoque. Sin embargo una vez generados los programas, se ejecutan eficientemente — en el caso de estudio se ejecuta en menos de 400ms cada vez.

Se decidió compilar cada código fuente de cada implementación usando los compiladores estándares de cada lenguaje usado — por ejemplo: GHC para Haskell, *g++* para C++, *javac* para Java, etc. Esta decisión se tomó en base a la idea que la mayoría de los implementadores usarán esos compiladores y herramientas. Además, se compiló cada código fuente sin parámetros adicionales o innecesarios (a excepción del enfoque con templates en Haskell), simplemente para evitar optimizaciones u otros cambios que pueden dar ventajas no deseadas para algunos enfoques.

El enfoque *embedded* no especifica nada sobre si el código fuente embebido se compila o interpreta. Esto le da tanto al implementador como al usuario final la posibilidad de compilar o interpretar el código. En general, si no siempre, el código compilado es más eficiente que el interpretado y el implementador debe tenerlo en cuenta si la eficiencia es una cuestión importante.

6.2.3 Comprensión

Con el objetivo de tener una idea de cuanto le toma a un usuario final no experimentado escribir un programa en el DSL, se seleccionaron dos personas no familiarizadas con la programación (pero sí con el manejo de una computadora), a dos estudiantes de la Licenciatura en informática de la facultad sin conocimiento previo sobre DSLs y a dos miembros de un laboratorio de la Facultad experimentados con los DSLs. Luego de explicarles las nociones básicas sobre FDL, se les pidió que escribiesen dos ejemplos en cada uno de los enfoques. Los programas FDL correspondientes se encuentran en el Apéndice A con los nombres de *BandaDeRock* y *Periodico*.

A ellos se les preguntó luego sobre los siguientes puntos:

- la inteligibilidad de la notación particular para el enfoque,
- la facilidad de uso de las herramientas provistas,
- las facilidades de debugging, y

- la inteligibilidad del reporte de errores.

Para cada uno de los puntos se usó una escala de cinco grados, de 'muy malo' a 'muy bueno'. En las Tablas 6.8 y 6.9, por cada ítem analizado hay tres columnas para cada enfoque, que representan las calificaciones puestas por cada uno de los dos usuarios.

Comparando los resultados de los usuarios experimentados y no experimentados, son similares en términos de la inteligibilidad de la notación del DSL, con pequeñas variaciones en el enfoque XML. Pero es importante remarcar que los usuarios no programadores prefirieron la notación visual a la notación textual brindada por FDL, y es por eso que la calificación en aquellos enfoques en los que se respetó en un 100 % la sintaxis del lenguaje original, argumentaron que la inteligibilidad era 'buena' (y no 'muy buena', como puede esperarse). Argumentaron de esa forma no sólo porque no había que entender las palabras en inglés, sino porque además los dibujos eran intuitivos para usar[5]. Podría ser interesante estudiar las posibilidades de crear un lenguaje visual antes que uno textual. La representación icónica y gráfica pueden llevar a un rápido e intuitivo aprendizaje del lenguaje y un uso universal [MY93]. Como esta discusión va más allá de la implementación en sí de los lenguajes (está mucho más relacionada al diseño de lenguajes), queda fuera del alcance de este trabajo.

Se notaron desviaciones en las herramientas de soporte, como son interfaces, reporte de errores y las facilidades de debugging. En el caso de las interfaces, durante las pruebas con los enfoques COTS y *embedded*, se le dijo a los usuarios no experimentados que debían escribir los programas con un editor estándard sin soporte para XML ni Haskell. Luego se les mostró la herramienta *Peter's XML editor*[6] que posibilita ver el documento XML en forma de árbol y se basa en la definición del DTD en el caso de XML; y un editor con *syntax highlighting* en el caso de Haskell. Con estas herramientas, la productividad del usuario mejoró notablemente. En la Figura 6.2, se puede ver una captura de pantalla de la herramienta para editar documentos XML, en la que se puede observar el documento mostrado en una estructura en forma de árbol y que resulta más fácil de manipular que su versión textual. Sin embargo el resto de los usuarios no expresó mayor satisfacción con esta herramienta ya que argumentaban estar acostumbrados a manipular este tipo de expresiones.

Respecto de las facilidades de debugging o depuración, en general no se

[5]Se han desarrollado varios entornos visuales para escribir diagramas de características (como ser el desarrollado por Czarnecki [AC04]).

[6]Peter's XML editor — disponible como freeware en
`http://www.iol.ie/~pxe/index.html`

Enfoque	Lenguaje	Inteligibilidad		
Source to source	Java	B/B	MB/MB	MB/MB
	Haskell	MB/B	MB/MB	MB/MB
macro processing	C++	B/B	B/B	B/B
embedded	Haskell	R/B	B/MB	B/B
embedded - templates	Haskell	R/B	MB/B	B/B
interpreter/compiler	Java	B/B	MB/MB	MB/MB
compiler generator	Lisa	B/B	MB/MB	MB/MB
	Smacc	B/B	MB/MB	MB/MB
extensible compiler	C#	B/B	B/B	B/B
COTS	XML	R/MM	M/M	M/R

Enfoque	Lenguaje	Interface		
Source to source	Java	R/R	B/B	B/R
	Haskell	R/R	R/B	B/R
macro processing	C++	B/B	R/B	B/R
embedded	Haskell	B/R	R/B	B/R
embedded - templates	Haskell	B/R	R/B	B/R
interpreter/compiler	Java	R/R	B/B	B/B
compiler generator	Lisa	B/B	B/B	MB/B
	Smacc	B/B	MB/MB	MB/MB
extensible compiler	C#	R/R	R/R	B/R
COTS	XML	R/R	M/M	B/M

MB = muy bueno
B = bueno
R = regular
M = malo
MM = muy malo

Cuadro 6.8: Esfuerzo del usuario final en usar el DSL. Parte I.

Enfoque	Lenguaje	Debugging		
Source to source	Java	M/M	M/M	M/M
	Haskell	M/M	M/M	M/M
macro processing	C++	M/M	M/M	MM/MM
embedded	Haskell	B/R	R/R	R/R
embedded - templates	Haskell	B/R	R/R	R/R
interpreter/compiler	Java	R/R	R/R	R/R
compiler generator	Lisa	M/M	R/R	R/M
	Smacc	R/M	R/R	R/B
extensible compiler	C#	R/R	R/R	B/B
COTS	XML	MM/MM	MM/MM	M/MM

Enfoque	Lenguaje	Reporte de errores		
Source to source	Java	M/M	M/M	M/M
	Haskell	M/M	MM/MM	MM/MM
macro processing	C++	R/R	M/M	MM/MM
embedded	Haskell	M/M	M/M	R/R
embedded - templates	Haskell	M/M	M/M	R/R
interpreter/compiler	Java	R/R	R/R	M/M
compiler generator	Lisa	B/B	R/R	R/R
	Smacc	B/R	B/B	R/B
extensible compiler	C#	R/R	R/R	MB/M
COTS	XML	R/R	R/R	B/B

Cuadro 6.9: Esfuerzo del usuario final en usar el DSL. Parte II.

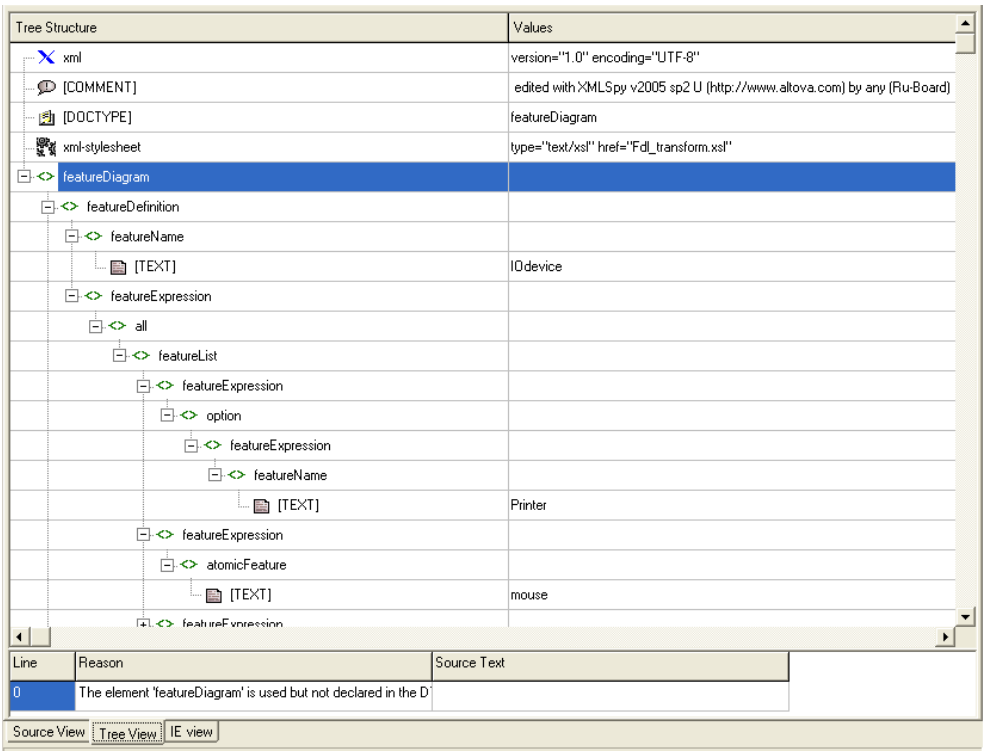


Figura 6.2: Editor *Peter's XML*

obtuvieron buenas calificaciones y las variaciones observadas no sin significativas. Los usuarios no programadores con su falta de experiencia tendieron a calificar muy bajo a cada enfoque y esto se debe a la falta de experiencia en programación.

En el caso de los reportes de errores se observan variaciones importantes en algunos casos. Por ejemplo, en el enfoque de *macro processing*, los usuarios experimentados con DSLs produjeron intencionalmente errores cuyo origen era imposible de descifrar y por ello han calificado este punto de 'muy malo'. El poco conocimiento de cómo trabajan las macros puede ser la causa de las calificaciones no tan bajas con los demás usuarios.

En el enfoque *extensible compiler*, uno de los usuarios experimentados con DSLs argumentó que la calificación era 'muy bueno' pues señalaba los posibles tokens que se podían poner en el lugar del error. El otro usuario experimentado sin embargo calificó el reporte de errores como 'malo' pues prefería errores menos técnicos y similares a los que se mostraban en otros enfoques.

En la herramienta LISA, si bien los reportes de errores eran de 'regulares' a 'buenos' (principalmente porque se basan en el uso de colores y posicio-

namiento automático sobre el código) se observó que varios de ellos eran en esloveno (el lenguaje de los desarrolladores de esta herramienta), lo que originó dificultades al momento de entenderlos. Es importante que los errores, advertencias u otra interacción que se produzca con el usuario resulten amigables y que estén de acuerdo a los patrones culturales de ellos. Incluso podría ser interesante estudiar qué tipo de herramientas están acostumbrados a usar habitualmente, así el aprendizaje del lenguaje y el uso de las herramientas asociadas resultan más naturales y rápidos.

6.3 Comentarios

La tarea de comparar enfoques de implementación no es sencilla. Para conseguir una comparación justa entre ellos se usaron métricas estándares (LDCE y PF). Estas métricas muestran que el enfoque más eficiente en términos de LDC para implementar es el *embedded*.

Llevar una API a un DSL no es muy difícil en términos del esfuerzo adicional si se elige un adecuado enfoque (*embedded*, *compiler generator*).

La comparación en LDCE es importante para el implementador de DSLs; sin embargo el punto de vista del usuario final no se debe desestimar. La notación original fue difícil de mantener en algunas implementaciones y esto generó consecuencias en la productividad del usuario final. Un adecuado soporte con herramientas que mejoraran la visión de esa notación particular puede resultar en una mejor aceptación del usuario final. La inteligibilidad de la notación, la interface de programación, debugging (depuración) y reporte de errores son insatisfactorias en algunos enfoques.

¿Cuál enfoque de implementación es la mejor solución? Para responder a esta pregunta se pueden usar los rankings de las tablas que se presentaron en este capítulo como orientaciones o pautas. Pero la elección final dependerá no sólo del dominio del problema sino también sobre qué ejes girará el desarrollo del DSL.

Se pueden establecer guías generales de acuerdo a circunstancias dadas:

Si una comunidad necesita un DSL, el reporte de errores no es tan importante y la notación se puede alterar o no cumplir estrictamente, entonces se recomienda el enfoque *embedded*, más aun si ya existe una API escrita que permita fácilmente embeber el lenguaje.

Si en cambio se tiene una API desarrollada y se quiere crear un DSL, con reporte de errores específicos y algunas herramientas de soporte y desarrollo, los generadores de compiladores son la mejor opción (enfoque *interpreter/compiler generator*).

En casos en que los usuarios finales están acostumbrados a usar determi-

nadas herramientas comerciales, se podría implementar el DSL usando esas herramientas (enfoque COTS). Todo lo relacionado con el soporte será fácil de manejar para ellos (ya familiarizados con esa tecnología) y se obtendrá una buena productividad y fácil aprendizaje.

Los demás enfoques no parecen tener características que los haga mejores ante alguna circunstancia, aunque no pueden ser descartados. La decisión debe ser tomada en cada caso por cada desarrollador teniendo en cuenta todos los factores aplicables a su caso.

Capítulo 7

Trabajo relacionado

Se han escrito muchos artículos sobre el desarrollo de DSLs usando enfoques particulares, pero existe poca literatura relacionada a las metodologías de desarrollo [CM98, Hud98, Wil01]. Además esos artículos describen y defienden sólo una metodología de desarrollo particular y en general usan apreciaciones subjetivas y tendenciosas.

Un tratamiento sobre el desarrollo completo de DSLs fue hecho por Mernik et. al [MHS05], donde se identifican y definen los diferentes patrones que ayudan en las fases de desarrollo de DSLs (decisión, análisis, diseño e implementación). El presente trabajo se puede considerar una extensión de ese trabajo en la parte de implementación, pues se dan más detalles sobre los enfoques de implementación, se hace una comparación más completa y profunda sobre ellos y se establecen algunas pautas que muestran cuándo aplicar cada uno de los enfoques. En este capítulo se describen los trabajos más importantes de las implementaciones de los enfoques detallados en [MHS05].

7.1 Enfoque preprocessing

7.1.1 Enfoque macro processing

Los compiladores con macros se han usado ampliamente como una forma de extender los lenguajes de programación por mucho tiempo. Por ejemplo, Kernighan y Ritchie describieron cómo extender el lenguaje C para tener una definición polimórfica de la función `max` [KR88]. Más allá que las macros agregan un grado muy útil de extensibilidad sintáctica a los lenguajes y proveen una forma conveniente de abreviar el idioma, sufren de problemas tales como:

- se permite que el cuerpo de una definición de una macro sea cualquier secuencia arbitraria de caracteres,

- se permite que el cuerpo de una macro se expanda en una secuencia arbitraria de tokens,

- la interacción entre las referencias del identificador introducido y las ligaduras con identificadores existentes o entre las ligaduras introducidas y las referencias existentes (problema de captura de variables libres),

- uso de alcance dinámico en vez del estático o léxico,

- imposibilidad de localizar el código fuente a través del proceso de expansión de las macros.

Estos problemas se pueden resolver (y de hecho se resuelven) en la mayoría de los sistemas de macros. Los primeros dos existen en sistemas de macros que operan en el nivel léxico (caracteres o tokens). Las macros léxicas permiten a los tokens ser sustituidos por una secuencia arbitraria de caracteres o tokens. El procesamiento de las macros léxicas precede al parsing y por lo tanto ignora la sintaxis del lenguaje base. La solución es usar macros sintácticas (por ejemplo, los templates de C++ [Vel95b, Vel95a], Scheme [DHB93], JTS [BLS98]) que operan en los árboles de parsing, lo que requiere conocimiento de la gramática del lenguaje base.

El problema de la captura de variables libres fue tratado por Kohlbecker et. al [KFFD86] donde se presentó un algoritmo de expansión de macros "higiénico" para el λ-cálculo y que fue posteriormente implementado en diversos lenguajes. Los identificadores introducidos por la definición de macros se renombran (usando α-conversión y marcas de tiempo), por lo que sólo se introducen identificadores "frescos".

El uso de alcance dinámico en vez del estático o léxico se resuelve por las llamadas macros de transparencia referencial local, donde las variables libres que ocurren en el cuerpo de una macro se resuelven en el entorno léxico de la macro, en vez de resolverse en el entorno de uso de la macro.

El último problema se resuelve por correlación entre el código fuente y el código de la macro expandida. Por lo tanto, el compilador, el sistema de run-time y el depurador son capaces de comunicarse con el programador en términos del código fuente original. El preprocesador de C, por ejemplo, emite una directiva *#line* cuando realiza una expansión de más de una línea o incluye un archivo. Luego el compilador interpreta estas directivas y ajusta el número de línea cuando hace el reporte de errores.

El trabajo de resolución de los problemas se hace evidente en el potente sistema de macros en Scheme [DHB93], que soporta macros higiénicas, macros de transparencia referencial local y correlación de códigos. El sistema de macros de Scheme inspiró otros. Weise et al. [WC93] describen las macros sintácticas que operan en el árbol de sintaxis abstracta (AST - abstract syntax tree), donde las macros declaran los tipos sintácticos que ejecutan y, por lo tanto, garantizan transformaciones sintácticas válidas. De manera similar Brabrand y Schwartzbach [BS02] introducen las macros de sintaxis metamórfica que aceptan colecciones de reglas gramaticales que extienden la sintaxis del lenguaje de programación subyacente. La nueva sintaxis se traduce directamente en un árbol de parsing en un lenguaje base. La sintaxis definida por los no-terminales definidos por el usuario se transforma a la sintaxis base, por lo que los AST definidos por el usuario no son soportados.

Los templates o plantillas del lenguaje C++ se pueden ver también como macros sintácticas [Vel95b, Vel95a] con algunas limitaciones como, por ejemplo, no usan expansión higiénica.

El enfoque *macro processing* está soportado por algunas herramientas de desarrollo de lenguajes y frameworks, tales como JTS [BLS98], JSE [BP01] y Maya [BH02]. Un buen resumen de lenguajes con macros se da en el trabajo de Brabrand [BS02].

7.1.2 Enfoque source to source transformation

Al principio, las transformaciones fuente a fuente fueron primeramente usadas para resolver un dilema frecuente en el diseño de lenguajes: elegir entre un lenguaje amigable para el usuario con un conjunto rico en notaciones y un pequeño, pero conceptualmente simple, núcleo de lenguaje. La transformación de los programas escritos en los programas ricos a los lenguajes pequeños se define por patrones dirigidos por sintaxis.

Posteriormente, el enfoque fue usado exitosamente para la implementación de DSLs y algunas herramientas de desarrollo de lenguajes, tales como DMS [BPM04], TXL [CHP91] y Stratego [Vis01].

7.1.3 Enfoque de lexical processing

Los DSLs simples se pueden implementar con un enfoque de procesamiento léxico cuando se requiere sólo análisis léxico sin un análisis complicado basado en el árbol sintáctico. El enfoque fue defendido por Bentley [Ben86a]. Este tipo de DSLs se pueden implementar con herramientas como awk [AKW88], Perl [L. 96], y Python [Mer03].

7.2 Enfoque Embedding

La idea esencial del enfoque popularizado por Hudak [Hud98], es aumentar un lenguaje host con una biblioteca de dominio específico de forma tal que se tiene la sensación de tratar con un nuevo lenguaje más que con una biblioteca o con una API. Este enfoque ha probado ser útil en muchos dominios, tales como: generación de parsers [Fok95], ingeniería financiera [JES00], manipulación de imágenes [SF97], animación 3D [Ell99], diseño de hardware [BCSS98], composición musical [Hud96b], GUIs [HC95], pretty-printing [Hug95], seguridad en redes [San99], etc. Los lenguajes declarativos, funcionales [Hud98] y lógicos [GP99] son especialmente adecuados para embeber DSLs y de estos paradigmas se han TOMADO los ejemplos citados anteriormente.

Respecto del uso de técnicas de metaprogramming, se ha hecho una comparación interesante usando diferentes lenguajes de implementación: Haskell, MetaOCaml y C++ [COST]. Los resultados ayudan a tomar decisiones en cuanto a cuál lenguaje se podría usar para implementar DSLs, usando meta-programación. Otro trabajo relacionado compara el sistema de clases de Haskell y los templates de C++ [KS05].

7.3 Enfoque Interpreter/compiler

Spinellis [Spi01] argumentó que el desarrollo de los DSLs es radicalmente diferente del desarrollo de los GPLs, dado que los primeros son usualmente una pequeña parte de un proyecto mucho mayor y, por lo tanto, el costo de su desarrollo debe ser modesto. Sin embargo este punto de vista no es del todo aceptable pues un DSL que será usado a gran escala por muchos usuarios, que además solicitan que las notaciones sean cumplidas estrictamente, debería ser implementada usando un enfoque *interpreter/compiler*. Este enfoque demanda el mayor esfuerzo de implementación, mientras que se pueden alcanzar la mayoría de los beneficios de los DSLs. En muchos libros y artículos se puede encontrar cómo implementar un DSL usando este enfoque; entre ellos el libro de Aho, Ullman y Sethi [ASU86] es quizás uno de los más significativos.

7.4 Enfoque Compiler generator

La función principal de las herramientas de software y los entornos fue siempre mejorar la productividad del programador, de forma directa o in-

directa. Ejemplos de esto son las herramientas Yacc y Lex [MB90] y las herramientas similares, basadas en ellas.

El desarrollo de los primeros compiladores en las años cincuenta sin herramientas adecuadas resultó una tarea muy complicada. Por ejemplo, la implementación del lenguaje FORTRAN tomó cerca de 18 años. Luego se desarrollaron métodos formales que hicieron que esta tarea fuera mucho más fácil. Y finalmente los métodos formales contribuyeron a la creación de herramientas para generación automática de lenguajes. Así, el alto esfuerzo de implementar un DSL siguiendo el enfoque anterior se puede reducir enormemente con sistemas de desarrollo de lenguajes (los denominados *compiler compilers*). Para la parte léxica del DSL, generalmente se usan expresiones regulares. Para definir la parte sintáctica se usa en general notación BNF [BBG+60] o alguna variante como EBNF. Sin embargo no se ha llegado a ningún acuerdo sobre cómo describir la semántica de los lenguajes.

Las gramáticas con atributos fueron definidas en 1968 por Knuth [Knu68] para especificar e implementar los aspectos semántica estáticos de los lenguajes de programación. Desde entonces las gramáticas con atributos has sido estudiadas intensamente desde puntos de vista conceptuales y prácticos. En sus primeros años fueron usadas para generar compiladores, pero luego también se le dio uso en la generación de debuggers, editores, intérpretes, etc. El resultado de las investigaciones posteriores es una gran cantidad de subclases de gramáticas con atributos con algoritmos avanzados de implementación y diferentes dialectos. Un repaso de las investigaciones y resultados sobre gramáticas con atributos se puede ver en un resumen hecho por el mismo Knuth [Knu90].

Finalmente, una comparación de sistemas de desarrollo de lenguajes se puede ver en el trabajo de Mernik et. al [MHS05].

7.5 Enfoque Extensible compiler

Este enfoque requiere que la implementación del compilador o intérprete sea abierta de tal forma de que se pueda modificar por parte del programador. Esto se puede hacer con *reflection*, que es la habilidad de un programa de examinar y posiblemente modificar su estructura de alto nivel. Existen muchos niveles de *reflection* [KdRB91]:

- introspección: investigación de algunas partes del sistema,
- invocación explícita: permitir a los programas invocar operaciones de forma explícita, evitando la sintaxis en algunos casos, y

- intercesión: una especialización de meta-nivel que agrega o modifica las características existentes.

Sin embargo, sólo el último nivel soporta directamente la implementación de DSLs usando el enfoque *extensible compiler*. Recientemente se implementaron algunos lenguajes orientados a aspectos [Bol99, PB99, Sei99] usando el enfoque *extensible compiler/interpreter*.

7.6 Enfoque COTS

El enfoque COTS se ha popularizado a partir de trabajo de Wile [Wil01], de donde sale su nombre. El primer ejemplo de este enfoque son los DSLs basados en XML. Con las herramientas de XML se pueden crear varios de forma rápida. Germon [Ger01] utilizó XML y las herramientas de XML para implementar un compilador de un GPL.

Existen algunos trabajos de ampliación de herramientas como Excel, Access y PowerPoint que tratan de acercar las posibilidades de uso de estas herramientas para dominios específicos, creando funciones de dominio específico [JBB03], formas de visualización adaptadas al usuario [BBN03], etc.

7.7 Enfoque Hybrid

En un enfoque híbrido se mezclan dos o más enfoque de los que se mencionaron anteriormente. Se pueden reconocer varios enfoques híbridos. Consel y Marlet [CM98] integraron el enfoque *interpreter* y *compiler generator*. Se propuso una metodología para diseñar e implementar DSLs basada en los métodos formales existentes, con los siguientes pasos:

- especificación de la semántica del DSL con semántica denotacional,
- derivación del intérprete del DSL desde la semántica denotacional a través de una máquina abstracta,
- aplicar evaluación parcial en el intérprete para producir el compilador.

Los generadores de compiladores aceptan como entrada especificaciones de alto nivel del lenguaje que contienen sus partes léxicas, sintácticas y semánticas. En este caso particular, un evaluador parcial, que actúa como generador de compiladores, acepta como entrada un intérprete y genera un compilador. Elliot et al. [EFM00] integran los enfoques *embedding* y *compiler generator*, con Haskell como lenguaje host (Haskell) que hace el papel de generador de programas. El enfoque consiste de los siguiente pasos:

- se especifica e implementa un nuevo DSL definiendo los tipos claves del dominio y las operaciones en términos de los tipos primitivos provistos por el framework y el lenguaje host,
- el usuario ejecuta el programa Haskell para producir un programa optimizado en un lenguaje de primer order simple y principalmente funcional,
- los programas generados se proporcionan luego a un compilador simple (también escrito en Haskell) para la generación del código del back-end.

Clements et al. [CFF+04] integraron los enfoques *embedding* y *macro processing* desarrollando el DSL S-XML en Scheme usando un enfoque *embedded* mientras que además explota el procesador de macros de Scheme. Chiba [Chi95] integró los enfoques *preprocessing* y *extensible compiler/interpreter*. Dado que el protocolo MOP (Meta Object Protocol) afecta la performance de ejecución, el autor usa metaobjetos para controlar la compilación de programas. OpenC++ MOP controla la transformación fuente a fuente (*source-to-source transformation*) desde OpenC++ a C++. El programa OpenC++ se parsea y divide en definiciones de alto nivel para clases y funciones miembro. Luego se construye un metaobjeto para cada definición que traduce las definiciones en código C++.

Capítulo 8

Conclusiones y trabajo futuro

Este trabajo presentó una comparación de enfoques de implementación usando un mismo DSL: Feature Description Language (FDL). Las distintas implementaciones, realizadas con varios lenguajes (Java, Haskell, Smalltalk, C++, C#, XML-XSLT, etc.) para asegurar diversidad, han resultado en una gran experiencia y resultados muy interesantes. Todo el trabajo de investigación relacionado a DSLs se ha podido verificar en cada uno de los enfoques de implementación, así como también las ventajas que tiene el uso de DSLs contra el uso de bibliotecas de dominio específico. También se mostró cuánto esfuerzo adicional se necesita para llevar una API a un DSL.

Cuando se comenzaron las implementaciones de los enfoques, se decidió implícitamente escribirlos en lenguajes en que se tenían cierto dominio o experiencias previas. Ésta es una observación importante: en este punto, la subjetividad del programador/desarrollador importa enormemente. Algunos lenguajes son más adecuados para un enfoque de implementación en especial que otros, y la historia del programador y el conocimiento en diferentes paradigmas de programación influencian a la hora de tomar la decisión sobre qué enfoque elegir. El programador debe decidir cuál enfoque de implementación usar teniendo en cuenta las características básicas de cada uno descriptas en la Sección 2.2 y las detalladas durante el desarrollo de esta tesis. Además debe considerar las capacidades que posee para cada enfoque el o los lenguajes en que posee cierto conocimiento, para obtener una buena solución y resultados mejores en términos de eficiencia, esfuerzo, tiempo de desarrollo o cualquier otra dimensión donde necesite poner atención.

Si bien se ha realizado mucho trabajo de investigación por parte de la comunidad de DSLs, los profesionales aún no los usan comúnmente para expresar los dominios donde programan. En vez de esto, siguen escribiendo sus programas usando bibliotecas de dominio específico, descuidando las cuestiones que caracterizan a los DSLs. Una de las razones por las que los DSLs

no alcanzan su debida atención es la creencia general de que son lenguajes hechos para usuarios finales y que no forman parte de una comunidad de programadores. Sin embargo, se ha trabajado y se continúa asimilando a los DSLs en nuevos lenguajes como Java y C#, donde la notación se simplifica — por ejemplo el acceso a datos XML en C#. Estos son buenos ejemplos de uso de DSLs dentro de la comunidad de programadores, donde los profesionales pueden ver las ventajas de simplificar la notación del lenguaje. En necesario entonces darles a los programadores pautas para usar sus propios DSLs dentro de sus proyectos, para mejorar la eficiencia.

A lo largo de este trabajo se han descrito los diferentes enfoques de implementación, se ha elegido un lenguaje de estudio y se ha profundizado en las consecuencias de implementarlo para cada uno de los enfoques. Estas consecuencias, así como las tablas y comentarios dados en el Capítulo 6 son una guía general o conjunto de pautas para implementar DSLs que sirve a los desarrolladores que desean encarar la creación de un DSL.

Para implementar el caso de estudio (FDL) se hicieron diferentes análisis del dominio y se encontraron revelaciones importantes [MLI05] que muestran que el análisis del dominio del problema es una parte importante del desarrollo de los DSLs, como se describe en [MHS05]. Como trabajo futuro se planea revisar las comparaciones usando la semántica definida en [MLI05] para observar cómo impacta esto en el esfuerzo de desarrollo de los DSLs. Los resultados podrían aplicarse también a los GPLs.

Se hizo la comparación con un solo ejemplo, y puede resultar importante comparar los diferentes enfoques nuevamente usando dos o más ejemplo, para poder obtener resultados más precisos. Además el enfoque *interpreter/compiler* se hizo creando sólo el compilador. Esto se debe a que el caso de estudio no es apropiado para la interpretación. Como trabajo futuro, incluir varios DSLs adecuados tanto para compilación como para interpretación pueden dar resultados más profundos dentro de este enfoque, separando la construcción del intérprete y el compilador. La misma idea se puede aplicar en el enfoque *preprocessing*, donde los (sub)enfoques *pipeline* y *lexical preprocessing* no se realizaron por la misma causa. El enfoque híbrido [MHS05] no es adecuado para una comparación con los otros enfoques, simplemente porque hay muchas formas de obtener un enfoque híbrido (muchas formas de combinar los demás enfoques básicos). La implementación de un DSL con este enfoque se da bajo ciertas condiciones — por ejemplo, usando una combinación de los enfoques *preprocessing* y *embedded* — que no cubre todas las posibles formas de tener un enfoque híbrido.

Otro trabajo futuro interesante es estudiar los lenguajes de dominio específico visuales. Este tipo de DSLs podría aportar más datos sobre cuál enfoque de implementación es más adecuado.

Anexo A

Programas FDL

- IODevice

```
IODevice : all ( opt(Printer), mouse, Display,
               opt(webcamera), keyboard, opt(microphone),
               opt(Receiver) )
Printer  : one-of ( laser, inkjet )
Display  : one-of ( crt, lcd )
Receiver : more-of ( speaker, headphones )

webcamera requires microphone
include   lcd
```

- Car

```
Car : all ( carbody, Transmition, Engine, Horsepower,
          opt(pullsTrailer) )
Transmition : one-of ( automatic, manual )
Engine : more-of ( electric, gasoline )
Horsepower : one-of ( lowPower, mediumPower, highPower )

include pullsTrailer
pullsTrailer requires highPower
```

- Test

```
F1  : all ( a1, F2 , opt(F3), a2, F4, F5, a3, F6, opt(F7))
F2  : one-of ( F8, opt(a4), opt(F9) )
F3  : all ( opt(a5), a6 )
F4  : more-of( F10, opt(a7), a8 )
F5  : one-of ( opt(a9) , opt(a10), F11 )
```

```
F6  : all ( a11, opt(a12), a13, F12 )
F7  : one-of ( a14, a15 )
F8  : all ( a16, a17 )
F9  : all ( a18, a19, a20 )
F10 : all ( a21, a22 )
F11 : all ( opt(a23) , a24, a25 )
F12 : all ( a26, a27, a28 )

a1 requires a6
include a4
```

- Bicicleta

```
Bicicleta : all ( Tamanio, Ruedas, Tipo, Frenos )
Tamanio   : one-of ( pequenia, grande )
Ruedas    : more-of ( izquierda, derecha )
Tipo      : one-of ( paseo, montania )
Frenos    : one-of ( comunes, pedal )

paseo requires pedal
izquierda requires pequenia
derecha requires pequenia
```

- BandaDeRock

```
BandaDeRock : all ( Voces, Cuerdas, Percusion,
                                    opt(teclado) )
Voces : all ( cantante, opt (coro) )
Cuerdas : more-of ( guitarra, bajo )
Percusion : all ( bateria, opt (tambor) )

bajo requires bateria
include guitarra
include bateria
```

- Periodico

```
Periodico: all ( Portada, opt(editorial), Secciones,
                opt(necrologicas), opt(sorteos),
                opt(chistes), pronostico )
Portada: all ( nombre, fechaDeEdicion, titularPrincipal,
```

```
                opt(precio), noticiasDestacadas )
    Secciones: more-of ( economica, politica, deportiva,
                         Espectaculos, policial,
                         internacional, cultural,
                         avisosClasificados )
    Espectaculos: more-of ( cine, television, radio, teatro,
                            conciertos )

    include chistes
    exclude avisosClasificados
    politica requires economica
```

Bibliografía

[AC04] Michal Antkiewicz and Krzysztof Czarnecki. Featureplugin: feature modeling plug-in for eclipse. In *Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*, pages 67–72. ACM Press, october 2004.

[AG83] A. J. Albrecht and J. E. Gaffney, Jr. Software function, source lines of code, and development effort prediction: A software science validation. *IEEE Transactions on Software Engineering*, 9(6):639–648, November 1983.

[AIS+77] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language*. Oxford University Press, 1977.

[AKW88] A. V. Aho, B. W. Kernighan, and P. J. Weinberger. *The AWK Programming Language*. Addison-Wesley, 1988.

[AM91] Henk Alblas and Borivoj Melichar, editors. *Attribute Grammars, Applications and Systems, International Summer School SAGA, Prague, Czechoslovakia, June 4-13, 1991, Proceedings*, volume 545 of *Lecture Notes in Computer Science*. Springer, 1991.

[Aro94] L. Aronson. *HTML manual of style*. Ziff-Davis Press, Emeryville, CA, USA, 1994.

[ASU86] A. V. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, USA, 1986.

[Bar88] H. P. Barendregt. Introduction to Lambda calculus. In *Aspenæs Workshop on Implementation of Functional Languages, Göteborg*. Programming Methodology Group, University of Göteborg and Chalmers University of Technology, 1988.

[BB94] M. M. Burnett and M. J. Baker. A classification system for visual programming languages. *Journal of Visual Languages and Computing*, 5(3):287–300, 1994.

[BBG+60] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Report on the algorithmic language ALGOL 60. *Commun. ACM*, 3(5):299–314, 1960.

[BBN03] Daniel Ballinger, Robert Biddle, and James Noble. Spreadsheet visualisation to improve end-user understanding. In *CRPITS '24: Proceedings of the Australian symposium on Information visualisation*, pages 99–109, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

[BCSS98] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: hardware design in Haskell. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 174–184. ACM Press, 1998.

[BDH+01] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Oliver, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: A component-based language development environment. In R. Wilhelm, editor, *Compiler Construction (CC'01)*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer-Verlag, 2001.

[Ben86a] J. Bentley. Little languages. *Communications of the ACM*, 29(8):711–721, 1986.

[Ben86b] Jon Bentley. Programming pearls: little languages. *Commun. ACM*, 29(8):711–721, 1986.

[BH02] J. Baker and W. C. Hsieh. Maya: Multiple-Dispatch Syntax Extension in Java. In *Conference on Programming Language Design and Implementation*, pages 270–281. ACM Press, 2002.

[BLS98] D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: Tools for implementing domain-specific languages. In *Proceedings of the 5th International Conference on Software Reuse*, pages 143–153. IEEE Computer Society, 1998.

[Bol99] K. Bollert. On weaving aspects. In *Proceedings of the Aspect-Oriented Programming Workshop*, pages 141–147. ACM Press, 1999.

[BP01] J. Bachrach and K. Playford. The Java syntactic extender JSE. In *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications OOPSLA'01*, pages 31–42. ACM Press, 2001.

[BPM04] I. Baxter, C. Pidgeon, and M. Mehlich. DMS: Program transformation for practical scalable software evolution. In *Proceedings of the International Conference on Software Engineering ICSE*, pages 625–634, May 2004.

[BS02] C. Brabrand and M. I. Schwartzbach. Growing languages with metamorphic syntax macros. *ACM SIGPLAN Notices*, 37(3):31–40, march 2002.

[BV04] M. Bravenboer and E. Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'04)*, pages 365–383, Vancouver, Canada, October 2004. ACM Press.

[CE00] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, 2000.

[CFF+04] J. Clements, M. Felleisen, R. Findler, M. Flatt, and S. Krishnamurthi. Fostering little languages. *Dr. Dobb's Journal*, 29(3):16–24, March 2004.

[Chi95] S. Chiba. A metaobject protocol for C++. In *Proceedings of the 10th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, pages 285–299, October 1995.

[CHP91] J. R. Cordy, C. D. H., and E. Promislow. TXL: a rapid prototyping system for programming language dialects. *Computer Languages*, 16(1):97–107, 1991.

[CM98] C. Consel and R. Marlet. Architecturing software using a methodology for language development. In *Proceedings of the 10th*

International Symposium on Programming Language Implementation and Logic Programming, volume 1490, pages 170–194, September 1998.

[COST] Krzysztof Czarnecki, John O'Donnell, Jörg Striegnitz, and Walid Taha. Dsl implementation in MetaOCaml, Template Haskell, and C++.

[DHB93] R. K. Dybvig, R. Heib, and C. Bruggeman. Syntactic abstraction in scheme. *Lisp and Symbolic Computations*, 5(4):295–326, 1993.

[DJ90] Pierre Deransart and Martin Jourdan, editors. *Attribute Grammars and their Applications*, volume 461 of *Lecture Notes in Computer Science*. Springer, 1990.

[DJL88] P. Deransart, M. Jourdan, and B. Lorho. Attribute grammars: Definitions, systems and bibliography. *Lecture Notes in Computer Science*, 323, 1988.

[DK02] A. van Deursen and P. Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–17, 2002.

[EFM00] C. Elliott, S. Finne, and O. De Moor. Compiling embedded languages. In *Proceedings of the Workshop on Semantics, Applications, and Implementation of Program Generation (SAIG'00)*, pages 9–27. Springer-Verlag, September 2000.

[Ell99] Conal Elliott. An embedded modeling language approach to interactive 3D and multimedia animation. *IEEE Trans. Softw. Eng.*, 25(3):291–308, 1999.

[Fok95] Jeroen Fokker. Functional parsers. In *Advanced Functional Programming, AFP'95*, volume 925 of *Lecture Notes in Computer Science*, pages 1–23. Springer-Verlag, Båstad, Sweden, May 1995.

[Fou91] Free Software Foundation. GNU General Public Licence. available at `http://www.gnu.org/copyleft/gpl.html`, June 1991.

[FSS02] R. T. Futrell, D. F. Shafer, and L. I. Shafer. *Quality Software Project Managment*. Prentice Hall, 2002.

[Ger01] R. Germon. Using XML as an intermediate form for compiler development. In *XML 2001 Conference Proceedings*, 2001.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GP99] G. Gupta and E. Pontelli. A Horn logic denotational framework for specification, implementation, and verification of domain specific languages. Technical report, NMSU, 1999.

[HC95] T. Hallgren and M. Carlsson. Programming with Fudgets. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming: 1st International Spring School on Advanced Functional Programming Techniques*, volume 925 of *Lecture Notes in Computer Science*, pages 137–182. Springer, Berlin, Heidelberg, 1995.

[Hin69] J. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.

[HM96] G. Hutton and E. Meijer. Monadic parser combinators. Technical Report Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996.

[HM98] G. Hutton and E. Meijer. Monadic parsing in Haskell. *J. Funct. Program.*, 8(4):437–444, 1998.

[HM02] Jan Heering and Marjan Mernik. Minitrack introduction. In *HICSS*, page 279, 2002.

[HPM+05] P. R. Henriques, M. J. Varanda Pereira, M. Mernik, M. Lenič, J. Gray, and H. Wu. Automatic generation of language-based tools using the LISA system. *IEE Software*, 152(2):54–69, 2005.

[Hud96a] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4es):196–196, 1996.

[Hud96b] Paul Hudak. Haskore music tutorial. In *Advanced Functional Programming, Second International School-Tutorial Text*, pages 38–67. Springer-Verlag, 1996.

[Hud98] P. Hudak. Modular domain specific languages and tools. In *Proceedings of the 5th International Conference on Software Reuse (JCSR '98)*, pages 134–142. IEEE Computer Society, 1998.

[Hug95] John Hughes. The Design of a Pretty-printing Library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming: 1st International Spring School on Advanced Functional Programming Techniques*, volume 925 of *Lecture Notes in Computer Science*, pages 53–96. Springer Verlag, 1995.

[JBB03] Simon Peyton Jones, Alan Blackwell, and Margaret Burnett. A user-centred approach to functions in Excel. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 165–176, New York, NY, USA, 2003. ACM Press.

[JES00] Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: an adventure in financial engineering (functional pearl). *SIGPLAN Not.*, 35(9):280–292, 2000.

[Jon97] C. Jones. *Applied Software Measurement: Assuring Productivity and Quality*. McGraw-Hill, second edition, 1997.

[KCH+90] K. Kang, S. Cohen, J. Hess, W.Ñovak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, November 1990.

[KdRB91] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, 1991.

[KFFD86] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygenic macro expansion. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 151–161. ACM, August 1986.

[Knu68] D. E. Knuth. Semantics of contex-free languages. *Mathematical Systems Theory*, 2(2):127 – 145, 1968.

[Knu84] Donald E. Knuth. *The TEXbook*. Addison-Wesley, Reading, MA, USA, 1984.

[Knu90] Donald E. Knuth. The genesis of attribute grammars. In *WAGA: Proceedings of the international conference on Attribute grammars and their applications*, pages 1–12, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language Second Edition*. Prentice-Hall, Inc., 1988.

[KS05] S. Kothari and M. Sulzmann. C++ templates/traits versus Haskell type classes. Technical Report TRB2/05, The National University of Singapore, 2005.

[L. 96] L. Wall, R. L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., 1996.

[Lou03] K. C. Louden. *Programming Languages-Principles and Practice*. Thomson-Course Technology, 3rd edition edition, 2003.

[Luc00] Bruce Lucas. VoiceXML for Web-based distributed conversational applications. *j-CACM*, 43(9):53–57, September 2000.

[Lyn03] Ian Lynagh. Unrolling and simplifying expressions with Template Haskell. Unpublished, May 2003.

[MB90] Tony Mason and Doug Brown. *Lex & Yacc*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1990.

[Mer03] David Mertz. *Text Processing in Python*. Addison-Wesley, 2003.

[MHS05] M. Mernik, J. Heering, and A. Sloane. When and how to develop domain-specific languages. *To apear in ACM Computing Surveys, 37(4)*, December 2005.

[Mil78] Robin Milner. A theory of type polymorphism in programming. In *Journal of Computer and System Sciences*, volume 17 of *3*, 1978.

[MKŽ95] M. Mernik, N. Korbar, and V. Žumer. LISA: A tool for automatic language implementation. *ACM SIGPLAN Notices*, 30(4):71–79, April 1995.

[MLAŽ00] Marjan Mernik, Mitja Lenic, Enis Avdicausevic, and Viljem Žumer. Compiler/interpreter generator system LISA. In *HICSS '00: Proceedings of the 33rd Hawaii International Conference on System Sciences-Volume 8*, page 8059, Washington, DC, USA, 2000. IEEE Computer Society.

[MLI05] Pablo E. Martínez López and Jerónimo Irazábal. An algebra for describing features. In *Proceedings of the XXXI Conferencia Latinamericana de Informática (CLEI'05)*, October 2005.

[MP00] M. Mernik and D. Parigot. Attribute grammars and their applications. *Informatica*, 24(3), September 2000.

[MY93] Stuart Mealing and Masoud Yazdani. A computer-based iconic language. In Masoud Yazdani, editor, *Mutiligual multimedia: bridging the language barrier with intelligent systems*, Intellect books, 1993.

[Nor92] Stephen C. North. NEATO user's guide. Technical Report 59113-921014-14TM, AT&T Bell Laboratories, Murray Hill, NJ, USA, October 1992. This report, and the program, is included in the `graphviz` package, available for non-commercial use at `http://www.research.att.com/sw/tools/graphviz/`.

[Paa95] J. Paakki. Attribute grammar paradigms - A high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, 1995.

[Pau91] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991. 2nd edition.

[PB99] J. Pryor and N. Bastan. A reflective architecture for the support of aspect-oriented programming in Smalltalk. In *Proceedings of the ECOOP Workshop on Aspect-Oriented Programming*, pages 300–301, 1999.

[PHE99] John Peterson, Paul Hudak, and Conal Elliott. Lambda in motion: Controlling robots with Haskell. In *First International Workshop on Practical Aspects of‘ Declarative Languages (PADL)*, volume 1551 of *Lecture Notes in Computer Science*, pages 91–105, 1999.

[PJH99] S. P. Jones and J. Hughes (editors). Haskell 98: A non-strict, purely functional language, February 1999. Available at `http://www.haskell.org/onlinereport/`.

[Pre92] R. S. Pressman. *Software Engineering - A Practioner's Approach*. McGraw-Hill, 1992.

[Ray01] E. T. Ray. *Learning XML*. A Nutshell Handbook. O'Reilly & Associates, Inc., 2001.

[Ray04] Eric Steven Raymond. *The Art of UNIX Programming*. Pearson Education, 2004.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-oriented modeling and design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[San99] Andre Santos. Embedding a firewall programming language into Haskell. In *SBLP'99 - III Simposio Brasileiro de Linguagens de Programacao*, May 1999.

[Sei99] L. Seinturier. JST: An object synchronization aspect for Java. In *Proceedings of the ECOOP Workshop on Aspect-Oriented Programming*, pages 292–293, 1999.

[SF97] Daniel E. Stevenson and Margaret M. Fleck. Programming language support for digitized images or, the monsters in the closet. In *Proceedings of the Conference on Domain-Specific Languages*, pages 271–284, 1997.

[SJ02] Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16, New York, NY, USA, October 2002. ACM Press.

[Spi01] D. Spinellis. Notable design patterns for domain-specific languages. *The Journal of Systems and Software*, 56(1):91–99, 2001.

[Str97] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition edition, 1997.

[SWA03] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In ACM, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 76–85. ACM Press, 2003.

[TL03] T. Thai and H. Q. Lam. *.NET Framework Essentials*. A Nutshell Handbook. O'Reilly & Associates, Inc., third edition, 2003.

[Unr94] Erwin Unruh. Prime number computation. Technical Report X3J16-94-0075/ISO WG21-462, ANSI, 1994.

[vDKV00] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000.

[vdL88] R. F. van der Lans. *Introduction to SQL*. Addison-Wesley, Reading, MA, USA, 1988.

[Vel95a] T. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995.

[Vel95b] T. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995.

[Vis01] E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, 2001.

[Wad93] Philip Wadler. Monads for functional programming. In M. Broy, editor, *Program Design Calculi: Proceedings of the 1992 Marktoberdorf International Summer School*. Springer-Verlag, 1993.

[WC93] D. Weise and R. F. Crew. Programmable syntax macros. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 156–165. ACM Press, 1993.

[Wil01] D. S. Wile. Supporting the DSL spectrum. *Journal for Computing and Information Technology*, 9(4):263–287, 2001.