UNIVERSIDAD POLITECNICA DE VALENCIA

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

TESIS DE MASTER

Optimización de Programas Datalog Basada en Estrategias

ALUMNO:
Fernando Tarín Morales

DIRECTORES:
Christophe Joubert
María Alpuente

Año Académico 2010/2011

Agradecimientos

En primer lugar querría mostrar mi agradecimiento a mis directores, Christophe Joubert y María Alpuente. Sin ellos este trabajo no habría sido posible. En especial a Christophe, sus constantes ideas y sugerencias me han ayudado enormemente durante la realización del proyecto.

También me gustaría agradecer el recibimiento que me brindó el grupo ELP, siendo uno más desde el primer día. En especial a mis compañeros de laboratorio, con los cuales comparto el día a día en el ámbito universitario. También desde aquí aprovechar para mandar un saludo a mis compañeros de carrera.

Para acabar me gustaría dedicarle el trabajo a mi familia, en especial a mis padres y mis abuelos, que tanto me han ayudado a lo largo de mi vida.

Índice general

Índice de figuras

Índice de cuadros

CAPÍTULO 1

Introducción

1.1. Motivación

Actualmente estamos participando en un cambio de paradigma, una transformación de los criterios básicos con los que percibimos la realidad y como el individuo interactúa con la sociedad.

Este cambio de paradigma conlleva que los sistemas de computación se vuelvan cada vez más complejos, manejando una mayor cantidad de información y estableciendo nuevos modelos de interacción. Esta creciente complejidad acarrea la inestabilidad e inseguridad de los sistemas desarrollados y la incapacidad para manejar la información de una manera correcta y eficiente. Otra desventaja importante, derivada de lo anterior, es la disminución de la capacidad de analizar y garantizar el comportamiento fiable de estos sistemas. En consecuencia, la fiabilidad se está convirtiendo en un valor añadido cada vez más importante que permite ofrecer una serie de garantías dentro de una sociedad cada vez más exigente y competitiva.

Para garantizar la fiabilidad y el correcto uso de la información se deben desarrollar técnicas, métodos y herramientas que soporten el proceso de desarrollo. Dentro del gran número de posibilidades existentes, este trabajo se centra en el uso "ágil" (*lightweight*) de los métodos formales, y en particular los métodos basados en la lógica y en la ingeniería del software. La aproximación ágil se basa en la aplicación selectiva y focalizada de los métodos formales, que resulta más efectiva y rentable en la práctica.

Una de las técnicas que hoy en día se está utilizando para poder trabajar con modelos de problemas a un mayor nivel de abstracción es el lenguaje Datalog, desarrollado originalmente para dar soporte al desarrollo de bases de datos deductivas. Este lenguaje permite manejar la creciente complejidad generada, tanto por la necesidad de analizar y garantizar el comportamiento fiable de los sistemas actuales, como por otros problemas de interés que conllevan el manejo de grandes cantidades de información.

Datalog. El subconjunto de Prolog, llamado Datalog, se usó por primera vez a finales de los años 70 [GM78], y posteriormente fue popularizado por Ullman [Ull89] y Abiteboul [AHV95] a finales de los años 80. En el momento de su definición, Datalog era un lenguaje simple aunque con un poder expresivo mayor que otros lenguajes relacionales, como SQL, que no ha soportado la recursividad hasta su cuarta revisión. Desde entonces se han añadido algunas extensiones al lenguaje Datalog inicial, como pueden ser el uso de la negación, disyunciones en la cabeza de las reglas (DLV), restricciones, o la creación de nuevos lenguajes basados en Datalog como Axlog [ABM09], Elog [BFG01], y SociaLite [SSN+10].

En la última década se ha observado un resurgimiento de Datalog en diferentes áreas de la computación. Datalog se ha usado para resolver un número de problemas no triviales, referenciados en [LS09], como son el análisis de punteros, la simplificación de gramáticas regulares basadas en restricciones, consultas sobre caminos, model checking, *frameworks* de seguridad, consultas sobre datos semiestructurados y redes sociales. Datalog también es popular en el mundo empresarial en el que algunas empresas venden soluciones basadas en Datalog, como LogicBlox[1], Semmle Ltd. [2], Lixto[3] y Exeura[4].

Queda así fundamentado el creciente interés que suscita Datalog, siendo una herramienta potente, a la vez que sencilla, que permite lidiar con la creciente complejidad de los problemas computacionales actuales, y que además, esta sustentada tanto dentro del mundo académico como del empresarial.

Finalidad de este proyecto. El objetivo principal de esta tesis de máster es desarrollar un conjunto de técnicas y modelos para resolver de una manera eficiente un programa escrito en el lenguaje de especificación Datalog. Para ello nos basaremos en un formalismo propio basado en la lógica de primer orden; para especificar la estrategia de resolución dentro de un contexto formal. También nos basaremos en una estructura de datos altamente especializada que permita soportar el conjunto de operaciones requeridas por la especificación de la estrategia de resolución de una manera eficiente.

En la literatura podemos encontrar una de las aproximaciones más recientes definida en el trabajo de Liu y Stoller en su artículo presentado a la conferencia internacional PPDP'2003 y detallado en su artículo de la revista TOPLAS [LS09]. El trabajo se basa en el desarrollo de una estructura especializada que garantiza tiempo de acceso constante para el conjunto de operaciones incrementales generadas por la estrategia de resolución de programas Datalog desarrollada por [BB79] así como el desarrollo de funciones de coste tanto de tiempo como de memoria.

En este proyecto pretendemos seguir una aproximación similar a la anterior. Siendo la eficiencia uno de los principales factores para la construcción de nuestro resolutor, el trabajo define nuevas técnicas, metodologías y estructuras de datos cuando las aproximaciones actuales no satisfacen los requerimientos necesarios para alcanzar los objetivos de este proyecto. En las siguientes secciones detallaremos más en profundidad estas técnicas y mejoras.

1.2. Aportaciones originales

Las principales contribuciones de este trabajo son las siguientes. Describe la estrategia de resolución en base a un formalismo original fundado en la lógica de primer orden que se especializa en un sistema de ecuaciones. Como consecuencia se ha desarrollado un motor de evaluación especializado para el sistema de ecuaciones utilizado en la especificación.

[1] http://www.logicblox.com
[2] http://www.semmle.com
[3] http://www.lixto.com
[4] http://www.exeura.com

Los resultados obtenidos experimentalmente son mejores a los encontrados en la literatura tanto en tiempo como en memoria. Por otro lado, el uso de la especificación basada en la lógica de primer orden permite fundamentar de una manera formal el trabajo realizado.

Se ha desarrollado un capítulo sobre técnicas para manejar la complejidad de problemas lógicos que complementa algunos de los conceptos presentados durante la especificación de la estrategia de resolución. En este capítulo se trabaja a un nivel de abstracción mayor para tratar de obtener la menor complejidad a la hora de resolver un problema lógico.

Otra aportación importante es la creación de una estructura de datos especializada y optimizada para soportar las operaciones generadas por la estrategia desarrollada. Esta estructura está basada en árboles de prefijos, también llamados tries, mapas de bits y opcionalmente en un gestor de conjuntos. Además resulta enormemente versátil, pudiendo modificarse según los requerimientos.

Para finalizar, se ha desarrollado una estrategia de evaluación utilizando un grafo de dependencias generado a partir de las reglas del problema que permite simplificar algunas de las operaciones utilizadas en la estrategia de resolución y establecer un orden eficiente para la evaluación de programas lógicos **negados** [5].

En última instancia se ha creado una herramienta en la que están plasmadas algunas de estas ideas y se ha utilizado un *benchmark* estándar para el problema de interés desarrollado que nos ha permitido comparar el prototipo con otras herramientas que se pueden encontrar en el estado del arte obteniendo resultados mejores tanto en tiempo de ejecución como en consumo de memoria.

Parte de los resultados de este trabajo fueron aceptados y presentados en el congreso nacional PROLE (Programación y Lenguajes) 2010 [FJT10a] y en el congreso internacional de AVOCS (International Workshop on Automated Verification of Critical Systems) 2010 [FJT10b]. Posteriormente se publico una versión extendida [6] en la revista electrónica *Electronic Communications of the EASST* [FJT10c]. Además se recibió una invitación para presentar un trabajo extendido [7] en el *Special Issue on Automated Verification of Critical Systems* de la revista *Science of Computer Programming* (Q2 JCR) de la editorial Elsevier. También se presentaron algunos de los resultados de este trabajo en la conferencia *Datalog 2.0* [8]

1.3. Sinopsis

El documento presenta paulatinamente los conceptos necesarios para la comprensión del trabajo realizado dentro del proyecto.

La presente introducción da una explicación sobre cuáles han sido las motivaciones que han llevado a su consecución así como la visión global sobre en qué consiste dicho proyecto. El segundo capítulo expone los fundamentos del lenguaje Datalog. En el tercer capítulo, se aborda con mayor profundidad el problema de interés escogido para demostrar

[5] Se da soporte a la evaluación de programas lógicos que se basan en la semántica de la negación estratificada.

[6] Se encuentra en el apéndice A.

[7] Desarrollado en esta tesis.

[8] `http://datalog20.org/`

la viabilidad de nuestra aproximación. En el cuarto se plantea la estrategia de resolución desarrollada. En el quinto capítulo se presentan un conjunto de técnicas para tratar de reducir la complejidad de los problemas lógicos. En el sexto capítulo se trata en profundidad la estructura especializada que se ha desarrollado para la resolución del problema y su comparación tanto con la mostrada en el trabajo [LS09] como con otras existentes en la literatura así como posibles extensiones y mejoras. En el séptimo capítulos se describe el prototipo desarrollado y los resultados obtenidos a partir de éste. Para acabar, se exponen las conclusiones obtenidas así como de posibles mejoras y trabajos futuros.

CAPÍTULO 2

Fundamentos sobre Datalog

En este capítulo se van a introducir los conceptos básicos sobre Datalog, lenguaje que utilizaremos durante el resto del manuscrito y para el cual vamos a crear un resolutor eficiente. Datalog es un lenguaje de consultas para bases de datos deductivas, basado hechos lógicos y reglas que resulta extremadamente sencillo de describir pero a la vez posee un gran poder expresivo.

2.1. Sintaxis y semántica

Datalog es un lenguaje que usa una notación parecida a Prolog, pero cuya semántica es mucho más simple. Para empezar, los componentes básicos de un programa Datalog son *átomos* de la forma $p(t_1, t_2, ..., t_n)$, donde:

1. p es un símbolo de *predicado* —un símbolo que, por ejemplo en el caso de $entrada(B, D)$, podría representar un tipo de afirmación como "una definición D llega al comienzo de un bloque B".

2. $t_1, t_2, ..., t_n$ son términos que pueden ser variables o constantes.

Un átomo básico (del inglés *ground atom*) es un predicado con sólo constantes como argumentos. Todo átomo básico afirma un hecho particular y su valor es verdadero o falso. A menudo es conveniente representar un predicado mediante una *relación*, o tabla de sus átomos básicos verdaderos. Cada átomo básico está representado por una única fila, o *tupla*, de la relación. Las columnas de la relación se llaman atributos, y cada tupla tiene un componente para cada atributo. Los atributos corresponden a los componentes de los átomos básicos representados por la relación. Todo átomo básico en la relación es verdadero y los átomos básicos que no estén en la relación son falsos.

También se permitirán átomos con la forma de comparaciones entre variables y constantes. Un ejemplo podría ser $X! = Y$ o $X = 10$. En estos ejemplos, el símbolo de predicado es realmente el operador de comparación. Esto es, podemos pensar en $X = 10$ (una comparación) como si estuviera escrito en la forma de predicado: *equals(X,10)*. De todas formas, hay una diferencia importante entre los predicados de comparación y el resto. Un predicado de comparación tiene su interpretación estándar, mientras que un predicado como *entrada* se interpreta sólo en función de su definición en el programa Datalog.

Un *literal* es o un átomo o la negación de un átomo. Se indica la negación con la palabra NOT delante del átomo. Así, $NOT\ entrada(B, D)$ es una afirmación que indica que la definición D no alcanza el comienzo del bloque B.

2.1.1. Reglas en Datalog

Las reglas son una forma de expresar inferencias lógicas. En Datalog, las reglas también sirven para sugerir cómo debería llevarse a cabo una computación de los hechos verdaderos. La forma de una regla es:

$$H :- B_1, B_2, ..., B_n. \tag{2.1.1}$$

donde:

- H y B_1, B_2, ..., B_n son literales —átomos o negaciones de átomos.
- H es la *cabeza* y B_1, B_2, ..., B_n forman el *cuerpo* de la regla.
- Cada uno de los B_is a veces es llamado *subobjetivo* de la regla.

El símbolo $:-$ ha de ser leído como "si". El sentido de una regla es "la cabeza es verdadera si el cuerpo es verdadero". Más precisamente, aplicamos una regla a un conjunto de átomos básicos como sigue: se consideran todas las posibles sustituciones de constantes por variables de la regla. Si una sustitución hace que todo subobjetivo del cuerpo sea verdadero (asumiendo que solamente los átomos básicos dados son verdaderos), entonces podemos inferir que la cabeza, instanciada con esta sustitución de constantes por variables, es un hecho verdadero. Las sustituciones que no hacen todos los subobjetivos verdaderos no nos dan información; el átomo de la cabeza podría ser verdadero o no[1].

Un programa Datalog es una colección de reglas. Este programa se aplica a "datos", esto es, a un conjunto de átomos básicos. El resultado del programa es el conjunto de átomos básicos inferidos mediante la aplicación sucesiva de las reglas hasta que no se puedan hacer más inferencias.

2.1.2. Convenciones en Datalog

Los programas Datalog utilizan normalmente una serie de convenciones léxicas para facilitar su interpretación:

1. Todas las variables comienzan con letra mayúscula.
2. El resto de elementos comienzan con letra minúscula u otros símbolos como los dígitos. Estos elementos incluyen predicados y constantes.

Otra convención en los programas Datalog es distinguir los predicados entre:

1. EDB, del inglés *Extensional Database*, o *Base de Datos Extensional*, son predicados definidos a priori. Esto es, los hechos verdaderos de estos predicados se dan en una relación o tabla, o se dan mediante el significado del predicado (como sería el caso para un predicado de comparación).

[1] El hecho de que una cabeza no se haga verdadera al aplicar una regla no implica que ésta sea falsa. Podría haber otra regla (o varias) que, al ser aplicadas, la hicieran verdadera. Por lo tanto, una cabeza sólo se hará falsa si no es verdadera según ninguna regla.

2. IDB, del inglés *Intensional Database*, o *Base de Datos Intensional*, predicados que están definidos mediante reglas.

Un predicado debe ser IDB o EDB, y puede ser solamente uno de éstos. En consecuencia, cualquier predicado que aparezca en la cabeza de una o más reglas debe ser un predicado IDB. Los predicados que aparecen en el cuerpo pueden ser IDB o EDB.

Al usar programas Datalog para expresar análisis de flujo de datos, los predicados EDB se computan a partir del grafo de flujo mismo. Los predicados IDB se definen mediante las reglas, y el problema de flujo de datos se resuelve infiriendo todos los posibles hechos IDB a partir de las reglas y los hechos EDB dados.

2.2. Ejecución de programas Datalog

Todo conjunto de reglas Datalog define relaciones para sus predicados IDB como una función de las relaciones dadas por sus predicados EDB. Se comienza con la suposición de que las relaciones IDB están vacías, esto es, que los predicados IDB son falsos para todos sus posibles argumentos. Entonces, se aplican las reglas repetidamente, infiriendo nuevos hechos cuando las reglas los requieran. Cuando el proceso converge, se finaliza, y las relaciones IDB resultantes forman la salida del programa. Este proceso se formaliza en el Algoritmo 1:

Algoritmo 1 Evaluación simple de programas Datalog.

ENTRADA: Un programa Datalog y un conjunto de hechos para cada predicado EDB.

SALIDA: Un conjunto de hechos para cada predicado IDB.

MÉTODO: Para cada predicado p en el programa, sea R_p la relación de hechos que son verdaderos para ese predicado. Si p es un predicado EDB, entonces R_p es el conjunto de hechos dados para ese predicado. Si p es un predicado IDB, debemos computar R_p ejecutando el Algoritmo 2.

Algoritmo 2 Cómputo de hechos IDB.

Para todo (predicado $p \in IDB$) **hacer**
$R_p := \emptyset$;
Mientras (Ocurran cambios a cualquier R_p) **hacer**
Considera todas las posibles sustituciones de constantes por variables en todas las reglas.
Determina, para cada sustitución, si todos los subobjetivo del cuerpo son verdaderos usando los R_ps actuales para determinar la verdad de los predicados EDB e IDB.
Si (Una sustitución hace el cuerpo de una regla verdadero) **Entonces**
Añadir la cabeza a R_q si q es la cabeza del predicado.

2.2.1. Evaluación incremental de programas Datalog

Hay una mejora de eficiencia posible para el algoritmo anterior. Nótese que un hecho IDB nuevo puede ser descubierto en la iteración i-ésima si se cumple que es el resultado de una sustitución de constantes en una regla tal que al menos uno de los subobjetivos ha sido descubierto en la iteración $i-1$. La prueba para esta observación es que, si todos los subobjetivos fueran conocidos en la iteración $i-2$, entonces los hechos "nuevos" se habrían descubierto al hacerse la sustitución de constantes en la iteración $i-1$.

Para explotar esta observación, para cada predicado IDB p se introduce un predicado $nuevoP$ que contendrá solamente los hechos de p descubiertos en la iteración anterior. Cada regla que contenga entre sus subobjetivos uno o más predicados IDB se reemplazará por una colección de reglas. Cada regla de la colección se forma reemplazando exactamente una ocurrencia de algún predicado IDB q en el cuerpo, por un predicado $nuevoQ$. Finalmente, para todas las reglas, se reemplazan los predicados h de la cabeza por $nuevoH$. Las reglas resultantes se dice que están en *forma incremental*.

Las relaciones para cada predicado IDB p acumulan todos los hechos-p. En una iteración:

1. Se aplican las reglas para evaluar los predicados $nuevoP$.
2. Entonces, se substrae p de $nuevoP$ para asegurar que los hechos en $nuevoP$ son realmente nuevos.
3. Se añaden los hechos de $nuevoP$ a p.
4. Se inicializan todas las relaciones $nuevoX$ a $\emptyset$ para la siguiente iteración.

Estas ideas se formalizan en el Algoritmo 3.

Algoritmo 3 Evaluación incremental de programas.

ENTRADA Un programa Datalog y un conjunto de hechos para cada predicado EDB.

SALIDA Un conjunto de hechos para cada predicado IDB.

MÉTODO Para cada predicado p en el programa, sea R_p la relación de hechos que son verdaderos para ese predicado. Si p es un predicado EDB, entonces R_p es el conjunto de hechos dados para ese predicado. Si p es un predicado IDB, debemos computar R_p. Además, para cada predicado IDB p, sea R_{nuevoP} la relación de hechos "nuevos" para el predicado p.

1. Modificar las reglas a su forma incremental según las explicaciones dadas anteriormente.
2. Ejecutar el Algoritmo 4.

Algoritmo 4 Cómputo incremental de hechos IDB.

Para todo (predicado $p \in IDB$) **hacer**
$R_p := \emptyset$
$R_{nuevoP} := \emptyset$
Repite
Considera todas las posibles sustituciones de constantes por variables en todas las reglas.
Determina, para cada sustitución, si todos los subobjetivos del cuerpo son verdaderos, usando los R_ps y R_{nuevoP}s actuales para determinar la verdad de los predicados EDB
e IDB.
Si (Una sustitución hace el cuerpo de una regla verdadero) **Entonces**
Añadir la cabeza a R_{nuevoH} si h es la cabeza del predicado.
Para todo (predicado p) **hacer**
$R_{nuevoP} = R_{nuevoP} - R_p$
$R_p = R_p \cup R_{nuevoP}$
Hasta ($\forall R_{nuevoP} = \emptyset$)

2.2.2. Reglas Datalog problemáticas

Hay ciertas reglas Datalog que no tienen sentido técnicamente y, por lo tanto, no deberían usarse. Los riesgos importantes son los siguientes:

1. *Reglas inseguras*: aquéllas que tienen una variable en la cabeza que no aparece en el cuerpo. En este caso, la variable en cuestión no está restringida y puede tomar cualquier valor de su dominio.

2. *Programas no estratificados*: conjuntos de reglas que tienen recursión en la que interviene una negación.

Para evitar las reglas inseguras, cualquier variable que aparezca en la cabeza de una regla debe también aparecer en el cuerpo. Además, la aparición no puede ser únicamente en un átomo negado, en un operador de comparación, o en los dos. La razón para esta política es evitar reglas que nos permitan inferir un número infinito de hechos.

Para que un programa esté estratificado, la recursión y la negación han de separarse. El requisito formal es el siguiente. Debe ser posible dividir los predicados IDB en *estratos*, tal que si hay una regla con un predicado p en la cabeza y un subobjetivo de la forma $NOT\ q(\cdots)$, entonces q es, o bien un predicado EDB, o es un predicado IDB en un estrato más bajo que el de p. Mientras se satisfaga esta regla, se podrá evaluar el estrato más bajo mediante un algoritmo convencional y, entonces, tratar las relaciones para los predicados IDB de ese estrato como si fueran EDB para la computación de un estrato más alto. No obstante, si violamos esa regla, entonces el algoritmo iterativo podría no converger.

2.3. Semántica de la negación

Como se ha comentado anteriormente, hemos extendido la técnica de evaluación del resolutor para permitir la evaluación de programas que incluyan la negación, concretamente restringida de la negación conocida como negación estratificada. De no ser así se podrían crear programas que no pudiesen representarse mediante ningún punto fijo o que existiesen varios puntos fijos minimales derivando en varios significados para un mismo programa. A continuación, se proporciona una definición para dicho modelo.

Comenzaremos presentando el caso en el que la negación se aplica únicamente a predicados extensionales. La semántica de la negación es directa en este caso. A continuación a partir de este simple caso, se dará la semántica estratificada de una forma extremadamente natural.

Datalog¬ Semipositivo Consideramos ahora los programas Datalog¬ Semipositivos, que solo aplican la negación a predicados extensionales. Por ejemplo la diferencia entre R y R' puede definirse mediante el programa:

$$Diff(x) \ :- \ R(x), \ \neg R'(x). \quad (1)$$

Para dar semántica al literal $\neg R'(x)$, simplemente nos basamos en la asunción del mundo cerrado [Rei87].

Definición 1 *Un programa P es semipositivo si, cada vez que un literal negativo $\neg R(x)$ aparece en el cuerpo de una regla de P, $R \in edp(P)$.*

Se podría eliminar la negación de estos programas añadiendo un nuevo predicado extensional que contuviese el complemento de R (con respecto al dominio activo) y reemplazando $\neg R(x)$ por $R(x)$. De esta manera, los programas Datalog¬ Semipositivos tienen un modelo mínimo único y un punto fijo minimal también único.

Consideremos ahora una extensión natural de los programas semipositivos, en las que, el uso de la negación no esté restringido a los predicados extensionales. Supongamos que tenemos definidas algunas relaciones. Una vez una relación ha sido definida por un programa, otro programa puede posteriormente tratarla como una relación extensional y aplicarle la negación. Esta simple idea conduce a una importante extensión de los programas semipositivos, llamados programas estratificados. Cada relación intensional es definida por una o más reglas de P. Si somos capaces de establecer una "lectura" del programa de tal manera que, para cada relación intensional R, la parte de P que define R está antes de que se use la negación de R, entonces podemos simplemente computar R antes de que su negación sea utilizada, y eso es todo. Tomemos por ejemplo el siguiente programa:

$$\begin{array}{ll} T(X,Y) \ :- \ G(X,Y). & (1) \\ T(X,Y) \ :- \ G(X,Z), \ T(Z,Y). & (2) \\ CT(X,Y) \ :- \ \neg T(X,Y). & (3) \end{array}$$

Claramente, existe la intención de que T este definido por las dos primeras reglas antes de que su negación se utilice en la regla que define el predicado CT. De esta forma, las

primeras dos reglas se evalúan antes que la tercera. Esta forma de "lectura" del programa P se llama una estratificación de P y a continuación vamos a dar una idea intuitiva de su definición.

Intuitivamente, una estratificación de un programa P permite encontrar una forma de analizar el programa P como una secuencia de subprogramas $P_1, \ldots, P_n$ en el que en cada uno se definen varias relaciones intensionales. Si una relación R' se utiliza de forma positiva en la definición de R, entonces R' debe ser definida anterior o simultáneamente con R (esto permite la recursión). Si la negación de R' se utiliza en la definición de R, entonces la definición de R' debe estar estrictamente antes que la de R. Desafortunadamente, no todos los programas Datalog¬ poseen una estratificación. A los programas que poseen una estratificación se les llama estratificables. A continuación se presenta la definición sobre lo que es un grafo de dependencias de predicados y una proposición que utiliza dicho grafo para dar un condición suficiente y necesaria para que un programa sea estratificable.

Definición 2 (Grafo de dependencias de predicados) *Un grafo de dependencias de predicados es un grafo dirigido que describe la relación de dependencia que existe entre los diferentes predicados de un programa* Datalog*. Se construye de la siguiente manera:*

1. *El conjunto de nodos o vértices se crea a partir de los diferentes símbolos de predicado. Por cada uno, independientemente de si aparece en la cabeza o en el cuerpo de la regla, se construye un nodo etiquetado con el nombre del predicado. Además para dar soporte a la negación si un predicado aparece negado en el cuerpo de alguna regla se debe añadir una marca a dicho nodo*

2. *El conjunto de arcos se crea a partir de las diferentes reglas del programa* Datalog*. Dos nodos se unen mediante un arco orientado si existe una regla en el programa en la que el nodo origen pertenece al cuerpo y el nodo destino pertenece a la cabeza.*

Se debe notar que la orientación de los arcos es inversa a otras aproximaciones que se pueden encontrar en la literatura. Las razones para optar por esta aproximación son por un lado, el establecimiento de la estratificación y por otro, para representar el cálculo ascendente del método de evaluación [2]. A continuación se da una descripción formal del grafo de dependencias de predicados caracterizándolo como una estructura de Kripke.

$$
\begin{aligned}
&S = \{S | S = \{nombre,\ marca\}\} \\
&R = \{\forall (H :- B_1, \ldots, B_n) \longrightarrow \{(m_{B_1}, m_H), \ldots, (m_{B_n}, m_H)\}\} \\
&L(S) = \{nombre = P\ si\ \exists P(X_1, \ldots, X_n)\} \cup \\
&\qquad \{marca = \neg\ si\ \exists (H :- \neg B_1.\ \vee\ H :- \neg B_1, B_2.\ \vee \\
&\qquad\qquad H :- B_1, \neg B_2.) \wedge H = P\}
\end{aligned}
$$

Proposición 1 *Un programa* Datalog¬ *es estratificable si su grafo de dependencias no contiene ningún ciclo con algún vértice negativo.*

[2] La estrategia de evaluación se detalla en el Capítulo 4

Dicha proposición se puede representar mediante la no satisfacción de la siguiente fórmula de lógica modal:

$$\forall X \in Predicados \Box((nombre = X \wedge marca = \neg) \longrightarrow \bigcirc\Box\neg(nombre = X \wedge marca = \neg))$$

En definitiva, la estratificación provee una manera simple y elegante de definir la semántica de los programas Datalog$\neg$. Sin embargo, tiene algunas limitaciones prácticas por imponerse restricciones a los programas evaluables.

CAPÍTULO 3

Problema de interés: análisis de programas

Como ya se ha comentado en la introducción, el lenguaje Datalog se está utilizando en la actualidad para resolver multitud de problemas. Para demostrar la viabilidad de nuestra aproximación y compararla con otras soluciones existentes, en este proyecto vamos a profundizar en su uso para el análisis de programas. A continuación se detalla en qué consiste este problema.

El análisis de programas es un subcampo de las ciencias de la computación que trata con la obtención de aproximaciones estáticas[1] lo más precisas posibles a la ejecución de programas.

3.1. Conceptos básicos

3.1.1. Análisis de flujo de datos

El análisis de flujo de datos (del inglés *data-flow analysis*) se refiere a un conjunto de técnicas que extraen información sobre el flujo de los datos a lo largo de los caminos de ejecución de un programa.

La ejecución de un programa puede ser vista como una serie de transformaciones del estado del programa, que consiste en los valores de todas las variables del mismo, incluyendo aquéllas presentes en los registros de activación de la pila de ejecución. La ejecución de una instrucción transforma un estado de entrada en un estado de salida. El estado de entrada se asocia con el punto del programa *anterior* a la instrucción y el estado de salida se asocia con el punto del programa *posterior* a la instrucción.

Al analizar el comportamiento de un programa, debemos considerar todas las posibles secuencias de puntos de programa, que son llamadas *caminos*, a través del grafo de flujo que puede seguir la ejecución del programa. A partir de los caminos se extrae, de todos los posibles estados del programa, la información necesaria para resolver el problema particular de análisis de flujo de datos.

3.1.2. Grafo de llamadas

Un grafo de llamadas (del inglés *call-graph*) de un programa es un conjunto de nodos y arcos tales que:

[1] En tiempo de compilación.

1. Hay un nodo por cada procedimiento en el programa.
2. Hay un nodo por cada punto de llamada (del inglés *call site*), esto es, un punto del programa donde se invoca un procedimiento.
3. Si el punto de llamada c puede invocar al procedimiento p, entonces existe una arista desde el nodo asociado a c al nodo asociado a p.

Si la invocación a procedimientos es directa, el destino de la llamada puede conocerse estáticamente. En tal caso, cada punto de llamada tendría solamente un arco hacia un procedimiento en el grafo de llamadas.

Sin embargo, la norma en los lenguajes orientados a objetos con enlace dinámico es la invocación indirecta a procedimiento —también se puede encontrar en lenguajes como C mediante el uso de los punteros a función, o en FORTRAN mediante el uso de parámetros que permiten recibir referencias a procedimiento. Específicamente, cuando se sobrescribe un método m en una subclase B de una clase A, una llamada a m sobre una variable polimórfica de tipo A puede referirse a distintos métodos, dependiendo del objeto receptor de la misma. El uso de esas invocaciones *virtuales* a método implica que se debe conocer el tipo del receptor antes de que podamos determinar el método que es invocado.

En general, la invocación indirecta nos obliga a realizar una aproximación estática de las condiciones en las que se realizan las llamadas a procedimientos —condiciones que pueden ser valores de punteros a función, referencias a procedimientos o tipos de objeto receptores, dependiendo del contexto. Dicha aproximación estática no es sino un caso particular de análisis de programas.

3.1.3. Análisis intraprocedimental vs. interprocedimental

Una de las formas de clasificar el análisis de programas es según el alcance de sus técnicas en un programa. Siguiendo esta clasificación, se distinguen los análisis intraprocedimentales e interprocedimentales.

Las técnicas de análisis intraprocedimental son una familia de técnicas de análisis de programas cuyo alcance son los procedimientos a nivel local. Esto es, intentan analizar un procedimiento independientemente de las relaciones de éste con el resto del programa. Dado lo restringido de su ámbito, existen muchas técnicas eficientes para realizarlos. No obstante, la información que extraen de los programas es mucho menos precisa que la que permiten extraer técnicas cuyo ámbito sea mayor, esto es, técnicas que actúen sobre el programa de forma global. Estas técnicas son calificadas como “interprocedimentales”, y son aquellas que son de mayor interés dadas su gran complejidad y los valiosos resultados entregados por ellas.

3.2. Análisis interprocedimental

El análisis interprocedimental es, como previamente se introdujo, un tipo de análisis cuyo alcance es todo el programa. Por alcance total se entiende que el análisis tiene en cuenta todas las divisiones del programa (procedimientos) a la hora de extraer información.

El análisis interprocedimental es complicado porque el comportamiento de cada procedimiento es dependiente del contexto en el cual es llamado.

Una aproximación simple al análisis interprocedimental pero a la vez imprecisa es el llamado *análisis insensible al contexto*. En él, cada instrucción de llamada y retorno de procedimiento se consideran instrucciones *goto* (salto incondicional), creando un *super grafo de flujo de control* con arcos de flujo de control adicionales que unen:

1. cada punto de llamada con el comienzo del procedimiento al que llama, y

2. cada instrucción de retorno con la siguiente instrucción a la del punto de llamada.

Además, se añaden instrucciones de asignación para copiar cada parámetro actual en su correspondiente parámetro formal y cada valor retornado en la variable que recibe el resultado. Entonces, se aplica un análisis estándar intraprocedimental sobre el *super grafo de flujo de control* para obtener resultados interprocedimentales insensibles al contexto. Debido a su simplicidad, este modelo elimina la relación entre los valores de entrada y salida en las invocaciones a procedimiento. Esto se debe a que no se establece ninguna correspondencia entre cada arco de entrada y salida a procedimiento, tratando todas las entradas y salidas al mismo como un *todo*, lo que supone una fuente de imprecisión.

3.2.1. Cadenas de llamadas

Un contexto de llamada queda determinado por el contenido de la pila de llamadas en el momento de realizar la invocación. Se define una *cadena de llamadas* (del inglés *call string*) como la secuencia de puntos de llamada que hay en la pila.

Al diseñar un análisis sensible al contexto se puede elegir la precisión del mismo basándose en la forma de distinguir los contextos. Si se distinguen los contextos exclusivamente por los k puntos de llamada más inmediatos tendríamos un análisis de contexto k-*limitado*. Por otra parte, se podrían distinguir completamente todas las cadenas de llamadas acíclicas (cadenas sin ciclos recursivos) para acotar el número de contextos distintos a analizar. Las cadenas de llamadas con recursión podrían simplificarse reduciendo cada secuencia recursiva de puntos de llamada a un punto de llamada único. No obstante, aún para programas sin recursión, el número de contextos de llamada puede ser exponencial con el número de procedimientos en un programa.

3.2.2. Técnicas de análisis sensibles al contexto

Existen distintas aproximaciones para realizar análisis sensibles al contexto, entre las que destacamos dos: la que se basa en la *clonación* y la que se basa en el *resumen*.

Una análisis sensible al contexto basado en clonación consiste en clonar *conceptualmente* cada procedimiento para cada contexto de interés. De esta forma, cada contexto tendrá un clon exclusivo del procedimiento y, por lo tanto, podremos aplicar un análisis insensible al contexto sin que haya ambigüedad (ya que cada invocación a procedimiento siempre se hará desde un único contexto).

Un análisis sensible al contexto basado en resumen consiste en la representación de cada procedimiento mediante una concisa descripción, el "resumen"[2], que describe parte del comportamiento observable del mismo, y el uso de éstas para calcular el efecto de todas las llamadas ocurridas en el programa.

3.3. ¿Para qué es necesario el análisis interprocedimental?

Dada la dificultad del análisis interprocedimental, se deben remarcar las razones por las que este tipo de análisis es útil. Por ejemplo, resulta muy eficaz para la optimización de invocaciones a métodos virtuales, el análisis de punteros, la paralelización de programas, y la detección de errores y vulnerabilidades.

3.3.1. Invocaciones a métodos virtuales

Los programas escritos en lenguajes orientados a objetos suelen estar compuestos de muchos métodos de tamaño reducido. Si solamente se intentaran optimizar los métodos por separado (análisis intraprocedimental), las oportunidades para optimizar serían escasas, dado su reducido tamaño. Resolver las invocaciones a métodos permite aumentar las oportunidades de optimización.

Si el código fuente de un programa está disponible, es posible realizar un análisis interprocedimental para determinar los posibles tipos de objeto a los que podría apuntar una variable x en toda llamada a método $x.m()$. Si el tipo para una variable x fuera único, la invocación al método podría resolverse estáticamente y el método podría desplegarse, evitando que se realice una llamada a función al ejecutarlo.

Varios lenguajes, como Java, cargan dinámicamente sus clases, por lo que en tiempo de compilación no sabemos qué método m será invocado para la llamada $x.m()$. Una optimización común en compiladores *JIT* (del inglés *Just-In-Time*) es analizar la ejecución y así determinar los tipos más comunes. Los métodos de los tipos más comunes se despliegan y se inserta código que compruebe dinámicamente los tipos para asegurar una ejecución correcta.

3.3.2. Análisis de punteros

El análisis de punteros es un tipo de análisis interprocedimental que permite saber a qué puede apuntar un puntero en el programa. Los resultados de este análisis permiten mejorar la precisión de otros análisis tanto interprocedimentales como intraprocedimentales.

3.3.3. Paralelización

La forma más efectiva de paralelizar una aplicación es encontrar la granularidad más grande de paralelismo. Para ello, el análisis interprocedimental es esencial debido a su

[2]El fin del *resumen* es evitar analizar el cuerpo del procedimiento para cada punto de llamada que invoque al mismo.

actuación en partes más grandes del programa (granularidad más gruesa) y a su mayor precisión, lo que permitirá perder menos oportunidades de optimización.

3.3.4. Detección de errores en el software y vulnerabilidades

El análisis interprocedimental no sólo es útil para optimizar código. Sus técnicas pueden usarse para analizar muchos tipos de errores de codificación comunes. Estos errores, que hacen al software menos fiable, muchas veces no son detectables localmente a un procedimiento, sino que su búsqueda requiere una exploración interprocedimental.

Si los errores son vulnerabilidades de la seguridad se hace incluso más importante su total detección.

3.4. Un algoritmo de análisis de punteros sencillo

En esta sección se introducirá un algoritmo muy sencillo de análisis de punteros insensible al flujo de datos asumiendo que no hay llamadas a procedimiento. Más adelante se extenderá para que tenga en cuenta llamadas a procedimiento de forma sensible e insensible al contexto. La sensibilidad al flujo añade mucha complejidad y es menos importante que la sensibilidad al contexto para lenguajes como Java donde los métodos tienden a ser pequeños.

La pregunta fundamental del análisis de punteros es si, dados un par de punteros, éstos pueden ser aliados. Una forma de responder a esta pregunta es computar para cada puntero, la respuesta a esta otra pregunta: "¿a qué objetos puede apuntar este puntero?". Si dos punteros pueden apuntar al mismo objeto, entonces los punteros podrían estar aliados.

3.4.1. ¿Por qué el análisis de punteros es difícil?

El análisis de punteros para programas en C es particularmente difícil porque los programas C pueden realizar computaciones arbitrarias sobre los punteros. De hecho, se puede leer un entero y asignarlo a un puntero, lo que convertiría este puntero en un potencial *alias* para todas las otras variables puntero del programa. Los punteros en Java, conocidos como referencias, son mucho más simples. No se permite realizar aritmética con ellos y sólo pueden apuntar al comienzo de un objeto.

El análisis de punteros debe ser interprocedimental. Sin análisis interprocedimental, uno debe asumir que cualquier método que sea llamado puede cambiar el contenido de todas las variables puntero accesibles, haciendo inefectivo cualquier análisis intraprocedimental.

Los lenguajes que permiten llamadas a función indirectas representan un reto adicional para el análisis de punteros. En C, se puede invocar una función indirectamente llamando a un puntero a función. Es necesario conocer a qué puede apuntar el puntero a función antes de que se pueda analizar la función invocada. Además, después de analizar dicha función, pueden descubrirse más funciones a las que el puntero a función podía apuntar, y, como consecuencia de esto, el proceso necesita ser iterativo.

Mientras que la mayor parte de las funciones se llaman directamente en C, los métodos virtuales en Java causan que muchas invocaciones sean indirectas. Dada una invocación $x.m()$ en un programa Java, puede haber muchas clases a las que el objeto x podría pertenecer y que tienen un método llamado m. Cuanto más preciso sea nuestro conocimiento del tipo real de x, más preciso será nuestro grafo de llamadas. Idealmente, podríamos determinar en tiempo de compilación la clase exacta de x y, así, saber exactamente a qué método se refiere m.

Es posible aplicar aproximaciones que reduzcan el número de objetivos posibles de una llamada. Por ejemplo, se puede determinar estáticamente cuáles son todos los tipos de objetos creados, y podemos limitar el análisis a esos tipos. Pero podemos ser más precisos si descubrimos el grafo de llamadas *on-the-fly*[3], basándonos en el análisis de punteros obtenido de forma simultánea. El aumento de la precisión del grafo de llamadas lleva no sólo a resultados más precisos, sino a reducir también el tiempo de análisis.

Como ya hemos dicho, el análisis de punteros es complicado. A medida que se descubren nuevos valores posibles para un puntero, todas las instrucciones que asignan los contenidos de ese puntero a otro puntero deben analizarse otra vez.

3.4.2. Un modelo para punteros y referencias

Supongamos que nuestro lenguaje tiene las siguientes formas de representar y manipular referencias:

1. Algunas variables de programa son del tipo "puntero a T" o "referencia a T," donde T es un tipo. Estas variables son, o bien estáticas o bien viven en la pila de ejecución, también llamado *heap*. Las llamaremos simplemente variables.

2. Hay un *heap* de objetos. Todas las variables apuntan a objetos del *heap* y no a otras variables. Estos objetos serán llamados "objetos del *heap*".

3. Un objeto del *heap* puede tener *campos*, y el valor de un campo puede ser una referencia a un objeto del *heap* (pero no a una variable).

Java se puede modelar adecuadamente con esta estructura. El leguaje C se modela peor ya que las variables puntero pueden apuntar a otras variables puntero y, en principio, cualquier valor puede ser visto como un puntero.

Dado que estamos realizando un análisis insensible al flujo, solamente tenemos que afirmar que una variable v puede apuntar a un objeto h del *heap* dado. Así pues, no tenemos que preocuparnos del problema de en qué lugar del programa v puede apuntar a h, o en qué contextos v puede apuntar a h. Las variables serán identificadas mediante sus nombres completos —que podría incluir el nombre del paquete, clase, método y bloque en el que está definida—, permitiendo así distinguir entre variables con el mismo identificador.

Los objetos del *heap* no tienen nombre. Una convención para referirse a estos objetos es mediante la instrucción en la que fueron creados. Como una instrucción puede ejecutarse varias veces creando múltiples objetos, una afirmación como "v puede apuntar a h" en

[3] Se puede traducir como "al vuelo" o "bajo demanda".

realidad quiere decir "v puede apuntar a uno o más de los múltiples objetos creados en la instrucción etiquetada con h".

El objetivo del análisis es determinar a qué puede apuntar cada variable y cada campo de cada objeto del *heap*. Nos referimos a esto como "análisis de punteros" (en inglés *points-to analysis*); dos punteros están aliados si la intersección de sus conjuntos *points-to*[4] es no vacía. El análisis que se describirá será *inclusion-based* (basado en la inclusión); esto es, una instrucción como $v = w$ causa que una variable v pueda apuntar a todos los objetos a los que puede apuntar w pero no al revés. Aunque esta aproximación parezca evidente, existen otras alternativas como el análisis *equivalence-based* (basado en la equivalencia) en el que la instrucción $v = w$ convertiría a v y w en una clase de equivalencia, permitiendo así que cada una de ellas apunte a todos los objetos a los que puedan apuntar las dos. A pesar de que esta última formulación no aproxima bien las alianzas de punteros, provee una rápida, y a menudo buena, respuesta a la pregunta de qué variables apuntan al mismo tipo de objetos.

3.4.3. Insensibilidad al flujo

Un análisis sensible al flujo se guía por el flujo de control del programa, añadiendo la información de cada instrucción requerida para el análisis pero, teniendo en cuenta, a la vez, los efectos que la nueva información tiene sobre la que se tenía antes.

Un análisis insensible al flujo ignora el flujo de control, lo que, en esencia, asume que toda instrucción del programa podría ser ejecutada en cualquier orden. Computa un mapa *points-to* indicando a qué puede apuntar cada variable en cualquier punto de ejecución del programa. Si una variable puede apuntar a dos objetos diferentes en un programa, registramos que puede apuntar a ambos objetos. En otras palabras, en un análisis insensible al flujo, una asignación no "cancela" ninguna relación *points-to*, sino que "genera" más relaciones *points-to*. Para computar los resultados insensibles al flujo, se añaden repetidamente los efectos *points-to* de cada instrucción sobre la relación *points-to* hasta que no se encuentren nuevas relaciones. Claramente, la falta de sensibilidad al flujo hace que los resultados del análisis sean más pobres pero tiende a reducir el tamaño de la representación de los resultados y hace que el algoritmo converja más rápidamente.

3.4.4. La formulación en Datalog

Ahora se propondrá la formalización en Datalog del análisis de punteros insensible al flujo discutido anteriormente. De momento se ignorarán las llamadas a procedimiento y nos concentraremos en los cuatro tipos de instrucción que pueden afectar a los punteros:

1. *Creación de un objeto* "$h : T\ v\ =\ new\ T()$;": Esta instrucción crea un nuevo objeto del *heap*, y la variable v puede apuntar a él. Se formulará en Datalog como vP0(V, H).

[4]Por comodidad, usaremos la denominación en inglés, conjunto *points-to*, para referirnos al conjunto de objetos del *heap* al que puede apuntar un puntero.

2. *Instrucción de copia* "$v = w$;": Aquí, v y w son variables. La instrucción hace a v apuntar a todo objeto del *heap* al que w pueda apuntar en ese momento. Se formulará en Datalog como A(V, W).

3. *Almacenamiento en un campo* " $v.f = w$;": El tipo de objeto al que apunta v debe tener un campo f, y este campo debe ser de algún tipo referencia. Si v es un puntero a un objeto del *heap* h, y w apunta a g, esta instrucción hace que el campo f de h ahora apunte a g. Nótese que la variable v se deja inalterada. Se formulará en Datalog como S(W, F, V).

4. *Cargar de un campo* "$v = w.f$;": Aquí, w es una variable apuntando a algún objeto del *heap* que tiene un campo f, y f apunta a algún objeto del *heap* h. La instrucción hace que la variable v apunte a h. Se formulará en Datalog como L(W, F, V).

Nótese que accesos compuestos a campos en el código fuente, como $v = w.f.g$, se pueden desglosar en varios accesos más primitivos como:

$$v1 = w.f;$$
$$v = v1.g;$$

Sólo queda expresar formalmente el análisis en reglas de Datalog. Primero, definiremos dos tipos de predicados IDB:

1. *vP*(V, H) quiere decir que la variable V puede apuntar al objeto del *heap* H.

2. *hP*(H, F, G) quiere decir que el campo F del objeto del *heap* H puede apuntar al objeto del *heap* G.

Las relaciones EDB se construyen a partir del programa mismo. Dado que la localización de las instrucciones en un programa es irrelevante cuando el análisis es insensible al flujo, sólo debe incluirse en la base de datos EDB la existencia de instrucciones que se ajusten a cierto patrón. En lo que sigue se hará una simplificación. En lugar de definir las relaciones EDB para que guarden la información extraída del programa, usaremos una instrucción entre comillas para sugerir la relación o relaciones EDB que representen la existencia de tal instrucción. Por ejemplo, el hecho "$H : T\ V = new\ T$" afirma que en la instrucción que ocupa la posición H, hay una asignación que hace que la variable V apunte a un nuevo objeto de tipo T. Asumiremos que, en la práctica, habrá una relación EDB correspondiente a éste que contendrá átomos básicos, uno para cada instrucción de esta forma que ocurre en el programa.

Con esta convención, todo lo que necesitamos para escribir el programa Datalog es una regla para cada uno de los cuatro tipos de instrucción:

1) $vP(V, H)$ $:-$ vP0(V, H).

2) $vP(V, H)$ $:-$ A(V, W),
$vP(W, H)$.

3) $hP(H, F, G)$ $:-$ S(W, F, V),
$vP(W, G)$,
$vP(V, H)$.

4) $hP(V, H)$ $:-$ L(W, F, V),
$vP(W, G)$,
$hP(G, F, H)$.

Figura 3.1: Programa Datalog para el análisis de punteros insensible al flujo.

En la Figura 3.1 la regla (1) dice que la variable V puede apuntar al objeto del *heap* H si la instrucción H es una asignación del nuevo objeto a V. La regla (2) dice que, si hay una instrucción de copia $V = W$, y W puede apuntar a H, entonces V puede apuntar a H. La regla (3) dice que, si hay una instrucción de la forma $V.F = W$, W puede apuntar a G, y V puede apuntar a H, entonces el campo F de H puede apuntar a G. Finalmente, la regla (4) dice que, si hay una instrucción de la forma $V = W.F$, W puede apuntar a G, y el campo F de G puede apuntar a H, entonces V puede apuntar a H. Nótese que vP y hP son mutuamente recursivos pero este programa Datalog puede ser evaluado por cualquiera de los algoritmos iterativos expuestos previamente ya que cumple las condiciones para su aplicación.

CAPÍTULO 4

Estrategia de resolución

En este capítulo se presenta la estrategia de resolución para ejecutar los programas especificados en Datalog. Dicha estrategia es completamente automatizable y puede verse como una compilación en la que, a partir de un programa Datalog, se obtiene otro programa altamente optimizado que obtiene las mismas respuestas que el programa Datalog original. Gracias a la transformación y la posterior estrategia de resolución, el programa generado ofrece garantías tanto en tiempo de ejecución como en memoria. Además, el tiempo de ejecución y consumo de memoria se pueden considerar óptimos en el sentido que la estrategia asegura que sólo se van a computar las combinaciones de hechos necesarias para obtener todas las soluciones y que dichas combinaciones sólo van a ser evaluadas una vez.

Una vez presentada la estrategia de resolución, definimos un algoritmo que propone un orden de evaluación sobre átomos y reglas a partir del cuál se puede determinar el conjunto mínimo de operaciones necesarias para obtener todas las soluciones. Además hemos extendido este algoritmo para soportar la evaluación de programas en los que existan reglas que incluyan la negación. Por motivos obvios, dicha negación debe ser una versión limitada, concretamente hemos dado soporte a la negación estratificada.

En el capítulo se incluye también una sección introductoria al tema de la complejidad de problemas lógicos y se presenta un procedimiento semi-automatizable para caracterizar la complejidad una especificación Datalog cuyo resultado es una cota más informada que la aproximación clásica [1].

4.1. Descripción

La estrategia que se va a describir a continuación tiene como objetivo calcular el conjunto mínimo de soluciones que se pueden obtener a partir de un programa Datalog. Dicho conjunto esta compuesto por el subconjunto de hechos extensionales o conocidos a priori, y el subconjunto de hechos intensionales. El subconjunto de hechos intensionales se obtiene por inferencia a partir de la especificación dada por el programa Datalog y el subconjunto de hechos extensional. La solución de interés a un programa Datalog es precisamente el subconjunto de hechos intensionales ya que es lo que calcula propiamente la estrategia de evaluación. Es importante resaltar las diferencias que existen entre ambos subconjuntos y cómo se relacionan entre ellos, ya que posteriormente la estrategia de evaluación hará uso de dicha información. Además es importante para establecer la complejidad de un programa Datalog.

[1] Dicha cota es m^k siendo m el tamaño de la base de hechos y k la aridad del predicado con mayor número de argumentos.

Basándonos en la definición ya dada de Datalog en el Capítulo 2 comenzaremos introduciendo un par de nociones que nos permitan caracterizar ciertos tipos de reglas y variables. Partiendo de una regla de un programa Datalog, una variable que ocurre múltiples veces en una hipótesis es una *equal card*. Intuitivamente obliga a que un hecho que unifique con la hipótesis tenga el mismo valor en las posiciones indicadas por esos argumentos. A una variable que ocurre una vez, y sólo una, en una hipótesis y no en la cabeza de la regla se le llama *wild card*. Una variable de este tipo no afecta al significado de la regla, pero es contraproducente ya que no acota los valores que se pueden tomar dentro del dominio representado y, por lo tanto, afecta negativamente a la eficiencia de la evaluación.

Para facilitar la exposición, se proporciona seguidamente una derivación formal de los algoritmos de resolución y complejidades asociadas para reglas que contienen como máximo dos hipótesis, donde las *equal cards* y *wild cards* sólo aparecen en las reglas con una hipótesis, y donde los argumentos de las relaciones, que como ya se ha comentado anteriormente en Datalog solo pueden ser variables, aparecerán agrupados y posiblemente reordenados. En principio esto plantea un problema, ya que los programas Datalog, según la definición dada, no tienen por qué estar restringidos de esta manera. Sin embargo como se verá posteriormente, las reglas con más de dos hipótesis pueden ser reducidas fácilmente a reglas equivalentes con dos hipótesis como máximo.

Llegados a este punto conviene resaltar el origen de dicha estrategia y por qué se ha elegido como estrategia base para el resolutor desarrollado. El funcionamiento de esta estrategia es bastante directo y ha sido utilizada en numerosas implementaciones que se pueden encontrar en la literatura. Por ejemplo, en la compilación de restricciones para el manejo de reglas [SSD+05] o en el artículo sobre compilación de reglas Datalog en programas eficientes con garantías de tiempo y espacio [LS09]. Quizás una de las primeras referencias a la técnica se puede encontrar en [BB79] que data de la década de 1970. La razón por la que se ha escogido en favor de otras, como por ejemplo la resolución SLD, es tratar de solventar un grave problema de eficiencia que dichas estrategias no pueden manejar de una manera adecuada, al estar ligadas a otros lenguajes lógicos más expresivos. Dicho problema está relacionado con la complejidad real de resolución de un problema lógico y se comentará en profundidad en la Sección 4.4.

Aunque nos adherimos a la definición original de la estrategia, conviene resaltar que con el método de evaluación aquí propuesto se pueden evitar algunas de las restricciones antes mencionadas. Por ejemplo, se puede obviar el caso de que en una regla con dos hipótesis existan *equal cards*.

La derivación formal considera reglas y hechos de la siguiente forma:

```
forma 2: Q2(X1s,X2s,Y's,C3s) :- H1(X1s,Ys,C1s), H2(X2s,Ys,C2s).
forma 1: Q1(Z's,Bs) :- H(Zs,As).
forma 0: Q(Cs).
```

Figura 4.1: Formas aceptadas por la estrategia.

Cada una de las formas soportadas por la estrategia representa a cada uno de los tipos de posibles reglas que se pueden encontrar. Por un lado, los predicados se nombran para

diferenciar si dicho predicado forma parte de la cabeza de una regla (`Q`, $\texttt{Q}_1$ y $\texttt{Q}_2$) e indicar el tipo de regla, o de su cuerpo y su posición en el cuerpo (`H`, $\texttt{H}_1$ y $\texttt{H}_2$). Por otro lado, los argumentos de cada predicado representan conjuntos. A continuación se detalla qué representa cada conjunto. Cada variable `X1s`, `X2s`, `Ys`, `Y's`, `Zs`, y `Z's` representa un conjunto de variables; cada variable `C1s`, `C2s`, `C3s`, `As`, `Bs`, y `Cs` representa un conjunto de constantes. Las variables existentes en los conjuntos `Y's` y `Z's` son subconjuntos de las variables existentes en los conjuntos `Ys` y `Zs` respectivamente. En la `forma 2`, las variables en `Ys` son exactamente aquéllas compartidas entre los dos predicados. Cada variable o constante en un grupo puede aparecer varias veces en el grupo, excepto para `X1s`, `X2s` y `Ys` en los predicados de la `forma 2`. Con esta restricción se asegura que no existen *equal cards* en las reglas con dos predicados. También, que no existen *wild cards* en las reglas con dos predicados, ya que cada variable ocurre en ambos predicados o en un predicado y la cabeza de la regla correspondiente. Queremos resaltar nuevamente que ésta es la definición original que se puede encontrar en la literatura, pero el método de evaluación aquí propuesto puede funcionar sin necesidad de tener en cuenta todas las restricciones. Por ejemplo, puede funcionar sin tener que respetar la restricción sobre las *equal cards*.

4.1.1. La aproximación formal

Se va a utilizar un lenguaje basado en conjuntos para el análisis y la aproximación formal de la resolución. El lenguaje está basado en SETL [SDSD86], un lenguaje de alto nivel basado en la teoría matemática de conjuntos, extendido con una operación de punto fijo [CP89], conjuntos de elementos heterogéneos y la operación de *"pattern matching"* (ajuste de patrones).

Los tipos de datos primitivos incluyen conjuntos, tuplas y mapas, es decir contienen relaciones binarias, representadas como conjuntos de pares (2-tuplas). Su sintaxis y operaciones están resumidas en la Tabla 4.1

$\{\}$	conjunto vacío
$\{X_1 \ldots X_n\}$	un conjunto con los elementos X_1,...,X_n
$[X_1 \ldots X_n]$	una tupla con los elementos $X_1, \ldots, X_n$ en orden
$\{[X_1Y_1] \ldots [X_nY_n]\}$	una aplicación que proyecta X_1 a Y_1,..., X_n a Y_n (mapa)
exists X **in** S	si S es un conjunto no vacío, vincula la variable X a cualquier elemento del conjunto S
S + T, S - T	unión y diferencia, respectivamente, de los conjuntos S y T
S **with** X, S **less** X	S + $\{X\}$ y S - $\{X\}$, respectivamente
$S \subseteq T$	comprueba si S es un subconjunto de T
X **in** S, X **not in** S	comprueba si X es un elemento de S o no
#S	número de elementos de S
dom(M)	dominio del mapa M, *e.g.*, $\{X : [XY\ ]$ **in** $M\ \}$
$M\{X\}$	imagen del conjunto X bajo el mapa M, *e.g.*, $\{Y : [XY\ ]$ **in** $M\ \}$

Cuadro 4.1: Conjuntos, tuplas, mapas, y operaciones sobre ellos

A continuación se detalla la operación de *pattern matching* (o ajuste de patrones).

X **of** $[Y_1 \ldots Y_n]$ encaja o ajusta X en el patrón tupla $[Y_1 \ldots Y_n]$

Devuelve falso si:

- X no es una tupla de longitud n.
- si siendo Y_i una constante no es la misma que X_i.
- si Y_i y Y_j son la misma variable pero X_i y X_j no son la misma constante.

En caso contrario, asocia cada variable Y_i, a la componente i-ésima de X y devuelve true.

A continuación se detalla la operación de ajuste de conjuntos.

$\{X\text{: } Y_1$ **in** S_1, ..., Y_n **in** $S_n \mid Z\}$ ajuste de conjuntos

Cada variable Y_i enumera los elementos del conjunto S_i. Para cada combinación de valores de Y_1, ..., Y_n, si el valor de la expresión Z es cierto, entonces el valor de la expresión X forma un elemento del conjunto resultante. Cada Y_i también puede ser un patrón de tupla, en cuyo caso cada elemento enumerado del conjunto S_i se ajusta primero al patrón antes de evaluar las expresiones Z y X. Si se omite Z, se asume implícitamente el valor de cierto.

LFP(S_0, F) denota el punto fijo mínimo del operador F que incluye S_0. En otras palabras, es el conjunto más pequeño de S que satisface la condición $S_0 \subseteq S$ y $F(S) = S$.

A parte de lo ya comentado, se utilizan las estructuras de control habituales **while**, **for**, e **if** y se abrevian las asignaciones del tipo $X := X$ **op** Y como X **op** $:= Y$

4.1.2. Especificación del punto fijo

Utilizando el lenguaje descrito, un hecho de la forma `Q(Cs)` se representa como `[Q Cs]`, es decir una tupla cuya primera componente es la constante `Q` y el resto de componentes son las constantes halladas en `Cs`. Una hipótesis de la forma `P(Xs,Cs)` se representa como `[P Xs Cs]`, donde `P` y los elementos de `Cs` son constantes y los elementos encontrados en `Xs` son variables.

Dado que todas las reglas de la misma forma se procesan de la misma manera, se describirá el método únicamente para una regla de la `forma 1` y una regla de la `forma 2`. Sea `e0` el conjunto de todos los hechos dados y sea `R` cualquier conjunto de hechos. Para la regla de la `forma 1`; sea `e1(R)` el conjunto de hechos `Q1(Z's, Bs)` de tal manera que `P(Zs,As)` existe en `R`; *e.g.*, `e1(R)` es el conjunto de hechos que pueden inferirse a partir de `R` utilizando la regla de la `forma 1` una vez. Para la regla de la `forma 2`; sea `e2(R)` el conjunto de hechos `Q2(X1s, X2s, Y's, C3s)` de tal manera que `P1(X1s, Ys,C1s)` y `P2(X2s, Ys,C2s)` existen en `R`; *e.g.*, `e2(R)` es el conjunto de hechos que pueden inferirse a partir de `R` utilizando la regla de la `forma 2` una vez. Esto es:

```
e0     = {[Q Cs] : Q(Cs) in hechosDados}
e1(R)  = {[Q1 Z's Bs] : [P Zs As] in R}
e2(R)  = {[Q2 X1s X2s Y's C3s] :
                [P1 X1s Ys C1s] in R y
                [P2 X2s Ys C2s] in R}
```

El significado del conjunto de reglas y hechos dados es:

LFP`(e0), donde F(R) = R + e1(R) + e2(R).`

4.2. Transformaciones sobre programas Datalog

Vamos a empezar recordando un ejemplo de especificación Datalog que servirá para ilustrar las transformaciones que proponemos. En su tesis [And94], Andersen dio una descripción para un análisis de punteros basados en inclusión, insensible tanto al flujo como al contexto, para programas escritos en lenguaje C. Dicho análisis tiene una complejidad cúbica en el peor caso, aunque en la práctica se suele tomar como referencia un coste cuadrático [SF09]. En el trabajo de [LWL05] podemos observar una especificación Datalog de tan sólo 4 reglas, equivalente a la expuesta por Andersen en su tesis.

Dicha especificación está basada en los cuatro tipos de sentencias de asignación en los que pueden intervenir los punteros en el lenguaje C. En la Figura 4.2 están representadas las sentencias declarativas de la especificación Datalog. $vP0$, A, S, L son predicados extensionales, mientras que vP y hP son los predicados intensionales que contendrán las soluciones de interés para dicho análisis. El significado de cada regla se puede consultar en el Capítulo 3.

$$vP(X,Y) \ :- \ vP0(X,Y). \quad (1)$$
$$vP(X,Y) \ :- \ A(X,Z),\ vP(Z,Y). \quad (2)$$
$$hP(Y,S,T) \ :- \ S(X,S,Z),\ vP(X,Y),\ vP(Z,T). \quad (3)$$
$$vP(Z,T) \ :- \ L(X,S,Z),\ vP(X,Y),\ hP(Y,S,T). \quad (4)$$

Figura 4.2: Reglas para el análisis de punteros de Andersen.

Como se puede observar, las reglas (3) y (4) de la Figura 4.2 no están en ninguna de las formas especificadas por la Figura 4.1 ya que, en este caso, existen reglas que contienen más de dos predicados. Para poder ejecutar el programa Datalog anterior, se debe transformar en un programa equivalente en el que todas sus reglas se acoplen a alguna de las formas especificadas en la estrategia. Es decir, que cada regla contenga como máximo dos predicados.

A continuación vamos a exponer en qué consiste dicha transformación. A partir de ahora nos referiremos a ella como *descomposición*, ya que consiste en descomponer cada regla compleja en varias reglas más sencillas. Aunque en este ejemplo se tratan reglas con tres predicados, el método de descomposición se puede aplicar a reglas con cualquier número de predicados, por lo que cualquier programa Datalog puede ser evaluado mediante esta estrategia.

La descomposición simplemente introduce relaciones auxiliares con los argumentos necesarios para mantener los resultados de combinar dos predicados a la vez. Exactamente, se aplican repetidamente las siguientes transformaciones a cada regla con más de dos predicados hasta que el programa únicamente contenga reglas con dos predicados:

1. Reemplazar dos átomos cualesquiera, digamos $P_i(X_{i1},\ ...,\ X_{ia_i})$ y $P_j(X_{j1},\ ...,\ X_{ja_j})$, de la regla con un nuevo átomo, $Q(X_1,\ ...,\ Xa)$, donde Q es una nueva relación, y los X_k son variables en los argumentos de P_i o P_j que también ocurren en los argumentos de otros átomos o en la cabeza de esa regla.

2. Añadir una nueva regla $Q(X_1,\ ...,\ Xa)\ :-\ P_i(X_{i1},\ ...,\ X_{ia_i}), P_j(X_{j1},\ ...,\ X_{ja_j})$.

Para una regla con `h` predicados, existen $(2h-3)!!$ formas posibles de descomponerla en reglas con dos predicados, siendo:

$$n!! = \begin{cases} 1, & \text{if } n = 0 \text{ o } n = 1 \\ n \times (n-2)!!, & \text{if } n \geq 2 \end{cases}$$

Típicamente, h suele ser una constante pequeña. Cada posible descomposición conduce a una cierta complejidad espacial y temporal. Estas complejidades se pueden calcular fácilmente con el método expuesto en la Sección 4.4. Las complejidades resultantes de cada descomposición se pueden comparar para tratar de determinar cuál es la mejor en términos de tiempo, espacio o posiblemente ambos. Puede existir un compromiso entre las complejidades temporales y espaciales si ninguna descomposición conduce, a la vez, a una solución óptima tanto para tiempo como espacio.

A continuación se van a desarrollar un par de posibles descomposiciones que se pueden obtener a partir de la especificación mostrada en la Figura 4.2.

$$\begin{array}{ll} vP(X,Y)\ :-\ vP0(X,Y). & (1) \\ vP(X,Y)\ :-\ A(X,Z),\ vP(Z,Y). & (2) \\ temp1(X,S,T)\ :-\ S(X,S,Z),\ vP(Z,T). & (3) \\ hP(Y,S,T)\ :-\ temp1(X,S,T),\ vP(X,Y). & (4) \\ temp2(Y,S,Z)\ :-\ L(X,S,Z),\ vP(X,Y). & (5) \\ vP(Z,T)\ :-\ temp2(Y,S,Z),\ hP(Y,S,T). & (6) \end{array}$$

Figura 4.3: Posible descomposición para el análisis de punteros de Andersen.

En la descomposición mostrada en la Figura 4.3 se han modificado dos reglas y se han añadido dos más para que el programa resultante pueda ser evaluado utilizando la estrategia descrita. Conviene recordar una vez más que este programa genera las mismas soluciones de interés que el análisis original expuesto en la Figura 4.2. Vamos a detallar a continuación cómo se han obtenido las reglas generadas:

- La regla (3) se ha obtenido combinando los predicados $S(X,S,Z)$ y $vP(Z,T)$ de la regla (3) de la especificación original, generando la nueva regla cuya cabeza es $temp1(X,S,T)$.

- La regla (5) se ha obtenido combinando los predicados $L(X,S,Z)$ y $vP(X,Y)$ de la regla (4) de la especificación original, generando la nueva regla cuya cabeza es $temp2(Y,S,Z)$.

En la figura 4.4 tenemos la descomposición que vamos a utilizar de ahora en adelante para resolver el análisis de punteros de Andersen.

$$
\begin{array}{ll}
vP(X,Y) \ :- \ vP0(X,Y). & (1)\\
vP(X,Y) \ :- \ A(X,Z),\ vP(Z,Y). & (2)\\
temp1(Z,Y,S) \ :- \ S(X,S,Z),\ vP(X,Y). & (3)\\
hP(Y,S,T) \ :- \ temp1(Z,Y,S),\ vP(Z,T). & (4)\\
temp2(Y,S,Z) \ :- \ L(X,S,Z),\ vP(X,Y). & (5)\\
vP(Z,T) \ :- \ temp2(Y,S,Z),\ hP(Y,S,T). & (6)
\end{array}
$$

Figura 4.4: Descomposición con mejor complejidad para el análisis de Andersen.

En esta descomposición también se han modificado dos reglas y se han añadido dos más. Vamos a detallar a continuación como se han obtenido las reglas generadas:

- La regla (3) se ha obtenido combinando los predicados $S(X,S,Z)$ y $vP(X,Y)$ de la regla (3) de la especificación original, generando la nueva regla cuya cabeza es $temp1(Z,Y,S)$.
- La regla (5) se ha obtenido combinando los predicados $L(X,S,Z)$ y $vP(X,Y)$ de la regla (4) de la especificación original, generando la nueva regla cuya cabeza es $temp2(Y,S,Z)$ de la misma forma que en el caso anterior.

En en la Sección 4.4 se va a realizar un estudio sobre la complejidad de la estrategia en base al tamaño de los dominios que intervienen en la resolución de los programas Datalog. Para justificar la elección de la descomposición, en esta sección se asume que, para el problema del análisis de punteros estudiado, los diferentes dominios poseen el mismo tamaño.

En concreto, la idoneidad de la segunda descomposición viene determinada por el conjunto Ys, dependiendo de si el conjunto de variables compartidas en las reglas de la `forma 2` es mayor o igual para todas las reglas de la descomposición. Asumiendo que los dominios de los diferentes conjuntos son del mismo tamaño, esto implica que se necesitará iterar sobre menos conjuntos para obtener nuevas soluciones. Conviene recordar que la estrategia descrita resolverá la descomposición elegida de una manera óptima en el sentido ya descrito anteriormente. La elección de la descomposición es parte de la tarea que consiste en encontrar un algoritmo óptimo a un nivel superior, que está ligado a la naturaleza de los problemas descritos por los programas Datalog.

4.3. Formalismo

Un programa Datalog puede verse como la especificación de un hipergrafo. Un hipergrafo es una generalización de un grafo, cuyas aristas se llaman hiperaristas y pueden relacionar a cualquier cantidad de vértices, en lugar de sólo un máximo de dos como en el caso particular de los grafos. En concreto la resolución de un programa Datalog es un grafo que se genera a partir de dicho hipergrafo y una base de datos extensional cualquiera.

En esta sección se va a presentar una descripción formal y general sobre cómo utilizar las formas discutidas en la Figura 4.1 para pasar del hipergrafo de la especificación al grafo que será la solución. Posteriormente se detallará el funcionamiento de dicho modelo para el programa de Andersen ya descompuesto y descrito en la Figura 4.4.

Como ya se ha comentado, vamos a discutir de manera formal cómo utilizar la estrategia presentada para pasar de cualquier programa Datalog, visto como un hipergrafo, a un grafo que será la solución de dicho programa. Conviene recordar que para la creación del grafo que será la solución es necesario trabajar con una base de datos extensional. El hipergrafo se va a presentar de manera simbólica a partir de la especificación del programa Datalog. La representación del grafo solución se realizará utilizando una notación ecuacional. En dicha representación, para todo vértice tendremos una ecuación en la que en la parte izquierda estará representado dicho vértice y en la parte derecha todos sus sucesores alcanzables mediante un paso. Si el vértice no tiene ningún sucesor se representará mediante la constante true. A continuación se detalla el formalismo en mayor profundidad.

4.3.1. Descripción

Como en [LS09], en el presente trabajo tratamos de imponer una evaluación ascendente (*bottom-up*), en la cual únicamente se van a computar las combinaciones de hechos necesarias para obtener todas las soluciones de la especificación Datalog. En nuestra derivación, cada solución o vértice se representa como una variable $X_{P(Cs)}$ identificada por el símbolo de predicado P y por el conjunto Cs, que representa los argumentos constantes que acompañan a dicho predicado.

Como ya se ha comentado anteriormente, para representar el grafo de solución vamos a utilizar una notación ecuacional. En la parte izquierda de las ecuaciones tendremos las variables que representan cada vértice, mientras que en la parte derecha tendremos la conjunción de todas las variables que se podrán generar aplicando un paso de los descritos en la Figura 4.5. Si una variable no tiene sucesores se representará mediante la constante true. De esta manera propagaremos el valor de verdad a través de las ecuaciones obteniendo así la satisfacibilidad de los vértices generados.

El método que representa la resolución ascendente de las formas aceptadas por la estrategia indicada en la Figura 4.1 se puede describir mediante las fórmulas lógicas descritas en la Figura 4.5.

Forma 0:

(1) $\forall\, \mathtt{Q(Cs)} \in hechosDados,\ X_{Q(CS)}$

Forma 1:

(2) $\forall\, X_{P(CS)} \,/\, \exists \mathtt{Q_1(Z's, Bs)} \ :-\ \mathtt{H(Zs, As)}. \wedge \{P/H\} \rightarrow \exists \sigma = \{\mathtt{Cs/Zs}\},\ X_{\mathtt{Q_1(Z's}\cdot\sigma\mathtt{,Bs)}}$

Forma 2:

(3) $\forall\, X_{P(CS)} \,/\, \exists \mathtt{Q_2(X1s, X2s, Y's, C3s)} \ :-\ \mathtt{H_1(X1s, Ys, C1s), H_2(X2s, Ys, C2s)}. \wedge \{P/H_1\}$
$\rightarrow \exists \sigma = \{\mathtt{Cs/(X1s, Ys)}\},\ \forall\, \mathtt{X2s} \mid (\mathtt{Ys}\cdot\sigma, \mathtt{X2s, C2s}) \in \mathtt{H_2},\ X_{\mathtt{Q_2(X1s}\cdot\sigma\mathtt{,X2s,Y's}\cdot\sigma\mathtt{,C3s)}}$

(4) $\forall\, X_{P(CS)} \,/\, \exists \mathtt{Q_2(X1s, X2s, Y's, C3s)} \ :-\ \mathtt{H_1(X1s, Ys, C1s), H_2(X2s, Ys, C2s)}. \wedge \{P/H_2\}$
$\rightarrow \exists \sigma = \{\mathtt{Cs/(X2s, Ys)}\},\ \forall\, \mathtt{X1s} \mid (\mathtt{Ys}\cdot\sigma, \mathtt{X1s, C1s}) \in \mathtt{H_1},\ X_{\mathtt{Q_2(X1s,X2s}\cdot\sigma\mathtt{,Y's}\cdot\sigma\mathtt{,C3s)}}$

Figura 4.5: Fórmulas para generación de nuevos vértices

Cada una de las fórmulas representa la manera en la que obtener nuevos vértices para el grafo de la solución. Todas las variables vienen identificadas por el símbolo de predicado P y el conjunto de constantes Cs. Cada nueva variable añadida es una cabeza *ground* inferida a partir de las reglas. En esta descripción falta la variable que representa al conjunto alcanzable a partir de los hechos dados. Para ello vamos a crear una variable simbólica llamada X_0. Dicha variable tiene como única función representar al conjunto alcanzable a partir de los hechos dados y al final de la computación debe tener el valor true. A continuación se detalla el comportamiento de cada una de las fórmulas especificadas en la Figura 4.5 y que representa a cada uno de los tipos de hipótesis que nos podemos encontrar en un programa Datalog:

(1) En las reglas de la `forma 0`, *e.g.*, $Q(Cs)$., se añade una nueva variable $X_{Q(Cs)}$ por cada hecho existente en la base de datos extensional.

(2) En las reglas de la `forma 1`, *e.g.*, $Q_1(Z's, Bs) \ :-\ H(Zs, As)$, por cada variable $X_{P(Cs)}$ cuyo predicado P unifique con el predicado H, se infiere un nuevo átomo *ground* $Q_1(Z's, Bs)$ aplicando la sustitución $\sigma = \{\mathtt{Cs/Zs}\}$. Para cada nuevo átomo *ground*, se genera una nueva variable $X_{1,Q_1(Z's\cdot\sigma,Bs)}$.

(3) En las reglas de la `forma 2`, *e.g.*, $Q_2(X1s, X2s, Y's, C3s) \ :-\ H_1(X1s, Ys, C1s)$, $H_2(X2s, Ys, C2s)$, por cada variable $X_{P(CS)}$ cuyo predicado P unifique con el predicado H_1, el primero de la parte derecha de la regla, se inferirá un nuevo átomo *ground* $Q_2(X1s, X2s, Y's, C3s)$ a partir de $P(Cs)$ aplicando la sustitución $\sigma = \{\mathtt{Cs/(X1s, Ys)}\}$, para cada instanciación de $X2s$, de tal forma que $H_2(X2s, Ys, C2s)$ sea un átomo *ground* computado previamente. Para cada nuevo átomo *ground*, se genera una nueva variable $X_{Q_2(X1s\cdot\sigma,X2s,Y's\cdot\sigma,C3s)}$.

(4) En las reglas de la `forma 2`, *e.g.*, $Q_2(X1s, X2s, Y's, C3s) \ :-H_1(X1s, Ys, C1s)$, $H_2(X2s, Ys, C2s)$, por cada variable $X_{P(CS)}$ cuyo predicado P unifique con el predicado H_2, el segundo de la parte derecha de la regla, se aplica la siguiente sustitución similar a la del caso anterior $\sigma = \{\mathtt{Cs/(X2s, Ys)}\}$. Análogamente se genera una nueva variable $X_{Q_2(X1s,X2s\cdot\sigma,Y's\cdot\sigma,C3s)}$.

La computación de los átomos *ground* inferidos se codifica como la satisfacción de X_0. La satisfacción de X_0 se obtiene siempre, incluso si no se puede inferir ningún átomo *ground* a partir de los hechos. En ese caso, el resultado de la evaluación ascendente del programa Datalog será el conjunto de los hechos. En el caso de que existan átomos *ground* deducibles, la terminación de la computación se obtiene cuando no se puede generar como sucesora ninguna variable nueva. Este es el caso cuando una variable $X_{P(Cs)}$ no tiene más sucesor que true (la conjunción vacía). Realmente, este formalismo es una tautología en la que todas las variables, incluyendo X_0, son ciertas al final de la computación. El objetivo es forzar la computación de todos los átomos *ground*.

Este sistema de ecuaciones produce un número de variables igual al número de átomos *ground* inferidos en el programa Datalog original. Por tanto, la complejidad tanto espacial como temporal de nuestro sistema debería ser lineal con respecto al número de soluciones. En el Capítulo 6 se presenta una estructura de datos especialmente diseñada para trabajar con conjuntos. Esta estructura hace posible la generación de nuevas variables en tiempo constante, manteniendo, por tanto, la complejidad lineal del sistema.

La capacidad para poder generar nuevas variables en tiempo constante se basa en la proyección de las vistas de datos necesarias. Una vista no es más que un orden nuevo sobre los argumentos en el que los elementos que pertenecen al conjunto $Y's$, es decir, el conjunto que contiene las variables comunes, se colocan primero. Puede existir más de una vista por hipótesis. Debe tenerse en cuenta que es necesario actualizar todas las posibles vistas que existan para un determinado predicado al mismo tiempo.

4.3.2. Aplicación al análisis de punteros de Andersen

Vamos ahora a aplicar el formalismo descrito a la resolución del análisis de punteros de Andersen. Para ello partimos de la especificación dada en la Figura 4.4. Partiendo de dicha descomposición, podemos aplicar ciertas optimizaciones al conjunto de fórmulas de la Figura 4.5. De esta forma obtenemos un conjunto de fórmulas especializadas para dicho análisis. Dichas optimizaciones se basan en precalcular ciertas operaciones teniendo en cuenta la naturaleza de los programas Datalog. En concreto las operaciones de las reglas que se pueden precalcular son las *unificaciones* y la obtención del *conjunto de sustituciones* a aplicar para generar nuevas variables. Esto resulta en una descripción del grafo de solución que evalúa el programa del análisis de punteros con garantías de tiempo y memoria. Esta descripción detallada está definida en la Figura 4.6. Como se puede observar en la figura, en cada caso se han aplicado todas las posibles unificaciones generando tantas ecuaciones como ha sido necesarias, además cuando dos ecuaciones tienen la misma parte izquierda las partes derechas se han agrupado mediante operadores de conjunción (esta operación se puede llevar a cabo ya que sabemos de antemano que todas la variables generadas van a resultar ciertas al final de la computación). Por ejemplo para el caso del predicado vP se generan 4 ecuaciones ya que *unifica con* la parte derecha de 4 reglas[2] se puede comprobar como en la Figura 4.6 salen agrupadas mediante conjunciones en la última ecuación. Un caso especial son los predicados pertenecientes a la base de hechos

[2] Como ya se ha comentado la especificación puede encontrarse en la Figura 4.4

$$
\begin{array}{lcl}
X_0 & = & \bigwedge\ \forall\, \mathbf{vP0(x,y)}\ \in\ \mathbf{hechosDados}.\ \ X_{vP0(x,y)} \\
 & & \wedge\ \bigwedge\ \forall\, \mathbf{A(x,z)}\ \ \in\ \mathbf{hechosDados}.\ \ X_{A(x,z)} \\
 & & \wedge\ \bigwedge\ \forall\, \mathbf{S(x,s,z)}\ \in\ \mathbf{hechosDados}.\ \ X_{S(x,s,z)} \\
 & & \wedge\ \bigwedge\ \forall\, \mathbf{L(x,s,z)}\ \in\ \mathbf{hechosDados}.\ \ X_{L(x,s,z)} \\
X_{vP0(c_1,c_2)} & = & \mathbf{vP(x,y)}\ :-\ \mathbf{vP0(x,y)}.\ \ X_{vP(c_1,c_2)} \\
X_{A(c_1,c_2)} & = & \mathbf{vP(x,y)}\ :-\ \mathbf{A(x,z)},\ \mathbf{vP(z,y)}.\ \ X_{vP(c_1,y)} \\
 & & \forall\, y \mid (c_2,y)\ \in\ vP_{xy} \\
X_{S(c_1,c_2,c_3)} & = & \mathbf{temp1(z,y,s)}\ :-\ \mathbf{S(x,s,z)},\ \mathbf{vP(x,y)}.\ \ X_{temp1(c_3,y,c_2)} \\
 & & \forall\, y \mid (c_1,y) \in\ vP_{xy} \\
X_{L(c_1,c_2,c_3)} & = & \mathbf{temp2(y,s,z)}\ :-\ \mathbf{L(x,s,z)},\ \mathbf{vP(x,y)}.\ \ X_{temp2(y,c_2,c_3)} \\
 & & \forall\, y \mid (c_1,y) \in\ vP_{xy} \\
X_{temp1(c_1,c_2,c_3)} & = & \mathbf{hP(y,s,t)}\ :-\ \mathbf{temp1(z,y,s)},\ \mathbf{vP(z,t)}.\ \ X_{hP(c_2,c_3,t)} \\
 & & \forall\, t \mid (c_1,t) \in\ vP_{xy} \\
X_{temp2(c_1,c_2,c_3)} & = & \mathbf{vP(z,t)}\ :-\ \mathbf{temp2(y,s,z)},\ \mathbf{hP(y,s,t)}.\ \ X_{vP(c_3,t)} \\
 & & \forall\, t \mid (c_1,c_2,t) \in\ hP_{yst} \\
X_{hP(c_1,c_2,c_3)} & = & \mathbf{vP(z,t)}\ :-\ \mathbf{temp2(y,s,z)},\ \mathbf{hP(y,s,t)}.\ \ X_{vP(z,c_3)} \\
 & & \forall\, z \mid (c_1,c_2,z) \in\ temp2_{ysz} \\
X_{vP(c_1,c_2)} & = & \mathbf{vP(x,y)}\ :-\ \mathbf{A(x,z)},\ \mathbf{vP(z,y)}.\ \ X_{vP(x,c_2)} \\
 & & \forall\, x \mid (c_1,x) \in\ A_{zx} \\
 & & \wedge\ \ \mathbf{temp1(z,y,s)}\ :-\ \mathbf{S(x,s,z)},\ \mathbf{vP(x,y)}.\ \ X_{temp1(z,c_2,s)} \\
 & & \quad \forall\, s,z \mid (c_1,s,z) \in\ S_{xsz} \\
 & & \wedge\ \ \mathbf{hP(y,s,t)}\ :-\ \mathbf{temp1(z,y,s)},\ \mathbf{vP(z,t)}.\ \ X_{hP(y,s,c_2)} \\
 & & \quad \forall\, y,s \mid (c_1,y,s) \in\ temp1_{zys} \\
 & & \wedge\ \ \mathbf{temp2(y,s,z)}\ :-\ \mathbf{L(x,s,z)},\ \mathbf{vP(x,y)}.\ \ X_{temp2(c_2,s,z)} \\
 & & \quad \forall\, s,z \mid (c_1,s,z) \in\ L_{xsz}
\end{array}
$$

Figura 4.6: Descripción detallada del grafo de solución para el análisis de punteros de Andersen

extensional, en dicho caso además de agruparse los diferentes predicados en la ecuación definida para X_0 también se agrupan las diferentes instancias que existan para cada predicado en la base de hechos. También para cada ecuación se han aplicado las *sustituciones* necesarias para la generación de nuevas variables, evitando de esta manera tener que calcularlas cada vez. Se puede comprobar como para cada una de las ecuaciones de la Figura 4.6 las variables generadas en la parte derecha ya incorporan las *sustituciones* pertinentes.

Consideremos ahora un pequeño programa en Java del que queremos obtener el análisis de punteros de Andersen. Dicho programa puede encontrarse en la Figura 4.7. En la columna de la izquierda tenemos el programa Java y en la derecha su equivalencia en predicados extensionales para el análisis de punteros de Andersen.

Example a = new Example();	vP0(v_a, h_1).
Example b = new Example();	vP0(v_b, h_2).
b = a;	A(v_b, v_a).
a.x = b;	S(v_a, x, v_b).
c = a.x;	L(v_a, x, v_c).

Figura 4.7: Programa en Java y su correspondencia con el análisis de Andersen

A partir del conjunto de hechos de la Figura 4.7 se genera el siguiente conjunto de hechos transformando por enteros los elementos de cada uno de los diferentes dominios (v_a=1, v_b=2, v_c=3; h_1=0, h_2=1; x=0)

$$vP0(1,0).\ vP0(2,1).\ A(2,1).\ S(1,0,2).\ L(1,0,3).$$

La resolución, bajo demanda, de la descripción especificada en la Figura 4.6 comienza con la con la variable simbólica X_0 y computa sus sucesores usando la ecuación que define dicha variable. Para este caso concreto, y teniendo en cuenta el conjunto de hechos dados, se generan las siguientes cinco variables $X_{1,vP_0(1,0)}$, $X_{1,vP_0(2,1)}$, $X_{1,A(2,1)}$, $X_{1,S(1,0,2)}$ y $X_{1,L(1,0,3)}$. Posteriormente el algoritmo computa las variables especificadas en la Figura 4.8 (utilizando una estrategia de búsqueda en amplitud).

$$\begin{array}{lcl}
X_0 & = & X_{vP_0(1,0)} \ \wedge \ X_{vP_0(2,1)} \ \wedge \ X_{A(2,1)} \wedge X_{S(1,0,2)} \wedge X_{L(1,0,3)} \\
X_{vP_0(1,0)} & = & X_{vP(1,0)} \\
X_{vP_0(2,1)} & = & X_{vP(2,1)} \\
X_{A(2,1)} & = & \mathsf{true} \\
X_{S(1,0,2)} & = & \mathsf{true} \\
X_{L(1,0,3)} & = & \mathsf{true} \\
X_{vP(1,0)} & = & X_{vP(2,0)} \wedge X_{temp1(2,0,0)} \wedge X_{temp2(0,0,3)} \\
X_{vP(2,0)} & = & \mathsf{true} \\
X_{temp1(2,0,0)} & = & X_{hP(0,0,0)} \wedge X_{hP(0,0,1)} \\
X_{temp2(0,0,3)} & = & \mathsf{true} \\
X_{hP(0,0,0)} & = & X_{vP(3,0)} \\
X_{hP(0,0,1)} & = & X_{vP(3,1)} \\
X_{vP(3,0)} & = & \mathsf{true} \\
X_{vP(3,1)} & = & \mathsf{true}
\end{array}$$

Figura 4.8: Ejemplo de resolución para el análisis de punteros de Andersen

El conjunto mínimo de todos los átomos *ground* que pueden ser inferidos utilizando la especificación Datalog para el análisis de punteros es el conjunto de todas las variables computadas, excepto la variable simbólica X_0. Por ejemplo, la variable $X_{1,A(2,1)}$ es igual a true porque cuando el algoritmo la evalúa, no se ha computado ningún átomo *ground* para vP. Por lo tanto, la ecuación que define $X_{A(2,1)}$ se reduce a $\wedge\ \emptyset$, que es true por definición. Incluso si no se genera ninguna variable a partir de $X_{A(2,1)}$, la estructura de datos [3] se actualizará con el átomo *ground* $A_{zx}(1,2)$. En el caso del análisis de punteros, estamos interesados en la base de datos intensional, es decir, los átomos *ground* inferidos para vP y hP. Estos átomos vienen dados por las variables computadas para $X_{vP(...)}$ y $X_{hP(...)}$, de las cuales obtenemos los átomos *ground* $vP(1,0)$, $vP(2,0)$, $vP(3,0)$, $vP(2,1)$, $vP(3,1)$, $hP(0,0,0)$, $hP(0,0,1)$ y deshaciendo las transformaciones para el predicado vP tenemos

[3] La estructura de datos se desarrolla en el Capítulo 6.

los siguientes resultados que indican las posiciones a las que puede apuntar cada variable $vP(v_a, h_1)$, $vP(v_b, h_1)$, $vP(v_c, h_1)$, $vP(v_b, h_2)$, $vP(v_c, h_2)$.

El ejemplo resuelto, por brevedad y claridad esta basado en una pequeña base de datos extensional. En la práctica los ejemplos reales pueden llegar a contener miles de hechos y generar millones como solución a un programa. En ese caso el grafo generado puede llegar a contener millones de nodos. Esto supone un problema ya que nos enfrentamos ante una posible explosión de estados. Una de las técnicas mas importantes para lidiar con este problema son los arboles de decisión binarios (BDD's de sus siglas en inglés). Aunque las técnicas descritas en el presente trabajo son altamente eficientes y mejoran el estado del arte existente, sería interesante estudiar la aplicación de los BDD's dentro de este contexto siguiendo los trabajos realizados por [WACL05] para tratar de obtener una escalabilidad y eficiencia todavía mayores.

4.4. Complejidad

En esta sección vamos a dar una pequeña introducción a los problemas a los que hay que hacer frente para calcular la complejidad de un programa lógico, en especial para Datalog. Una vez realizada dicha presentación se expondrán las razones por las que de entre varias opciones se ha escogido la estrategia presentada en este trabajo.

También se van a presentar un conjunto de fórmulas que permitan analizar con precisión el número de hecho procesados.

4.4.1. Introducción

En general, calcular la manera óptima de resolver un programa lógico no es posible. Esto se debe a que en un programa lógico se establecen relaciones entre diferentes dominios, pero estos dominios, en concreto los que pertenecen a la base de datos extensional y, en consecuencia, los que pertenecen a la base de datos intensional pueden variar de un problema a otro. El problema es similar al que se presenta al resolver una consulta SQL en una base de datos relacional.

Como en las bases de datos relacionales, el problema radica en establecer cual es la partición óptima, en este caso entre pares de predicados, para obtener el número mínimo de elementos en los conjuntos generados para poder obtener todas las soluciones. Veamos a continuación un pequeño ejemplo que nos permita establecer claramente esta afirmación. Para ello vamos a utilizar el programa que define la obtención de caminos de longitud par en la clausura de un grafo definido en la Figura 4.9

$$path2(U, V) \ :- \ edge(U, W),\ edge(W, V). \quad (1)$$
$$path2(U, V) \ :- \ edge(U, W),\ edge(W, X),\ path2(X, V). \quad (2)$$

Figura 4.9: Análisis para obtener los caminos de longitud par en la clausura de un grafo

En concreto para la regla (2) y tal y como se ha comentado en la Sección 4.2 existen 3 posibles particiones entre pares de predicados. Las tres posibles particiones que se pueden

dar entre los predicados son unir el primer predicado con el segundo o con el tercero y/o el segundo predicado con el tercero. El problema y lo que hace que no sea posible determinar la partición óptima en todos los casos viene dado por el hecho de que la base de datos extensional puede variar de un problema a otro sin tener que por ello variar la especificación. Es fácil generar pequeñas bases de datos extensionales y comprobar que no siempre una de las particiones es la mejor para el análisis propuesto.

Entre las técnicas para elegir o rechazar las posibles particiones podríamos considerar:

- Maximizar el conjunto de variables comunes al descomponer una regla.
- *Machine learning*.
- Técnicas heurísticas.

Teniendo únicamente como fuente de información la especificación de un análisis, la mejor opción que se puede escoger es maximizar el conjunto de variables comunes ya que minimiza el conjunto de operaciones necesarias para resolver el problema. Éste es el criterio que se ha escogido en la descomposición de punteros de Andersen. Las técnicas de *Machine learning* tratan de explotar la entropía subyacente a los datos en un determinado problema, mientras las técnicas heurísticas tratan de utilizar otro tipo aproximación y obtienen buenos resultados en algunos casos.

Llegados a este punto podemos ya comentar cuales han sido las razones que nos han llevado a elegir la estrategia descrita. Por un lado, genera programas Datalog en su forma más sencilla. Esto nos permite aplicar ciertas optimizaciones al proceso de resolución que de otra manera resultaría mucho más complejo. Además el hecho de tratar las particiones temporales como predicados obliga a aplicar técnicas de memorización lo que permite, en ciertos casos, no tener que volver a calcular particiones ya computadas previamente. La razón más importante es que no impone una decisión concreta sobre cuales van a ser dichas particiones. Otras técnicas de resolución de programas lógicos toman esta decisión sin tener en cuenta ningún tipo de información. Por ejemplo, la resolución SLD genera las particiones tomando pares de predicados de izquierda a derecha, mientras que en el caso de la estrategia descrita no obliga a ningún tipo de compromiso en la elección de dichas particiones. Esto permite tomar decisiones informadas sobre como realizar las particiones, teniendo en cuenta por ejemplo la naturaleza del problema que está siendo modelado o simplemente restringiendo la entrada a ciertos conjuntos de datos que sean de interés. Gracias a esto podemos aplicar otro conjunto de técnicas para la toma de la decisión, por ejemplo técnicas basadas en *machine learning*. La contrapartida de esta técnica es que solo permite realizar análisis ascendentes (*bottom-up*). Esto es un inconveniente si tenemos una consulta ya que perdemos la información de propagación de variables que nos proporcionan las técnicas descendentes (*top-down*). Para mitigar este problema se han desarrollado técnicas como los *magic-sets* que tratan de utilizar esa información en técnicas de evaluación ascendente. No obstante, dichas técnicas no incorporan mecanismos de decisión para la elección de particiones y tampoco garantizan que al utilizarlas se vaya a encontrar la solución óptima para un determinado problema. Sin embargo sería interesante tratar de estudiar como se puede crear un método de evaluación que tratase de combinar los conceptos de elección de particiones con la información propagación de variables de dichas técnicas.

4.4.2. Cálculo de la complejidad

Vamos ahora a presentar una manera de calcular la complejidad temporal partiendo de las reglas y expresándola en términos de caracterización de los hechos a partir de la especificación dada por Liu y Stoller [LS09]. La idea es analizar con precisión el número de hechos procesados, evitando aproximaciones que utilizan únicamente los tamaños de los dominios individuales de cada argumento.

Definiciones Se utiliza $P.i$ para denotar la proyección de P en su i-ésimo argumento. Se utiliza $P.I$, donde $I = \{i_1, i_2, ..., i_k\}$, para denotar la proyección de P en su posición $i_1, i_2, ..., i_k$.

El análisis utiliza los siguientes tamaños para caracterizar el conjunto de hechos dados, llamado tamaño de la relación, tamaño del dominio, número de argumentos, y tamaño del argumento relativo, respectivamente:

- #P: el número de hechos que se mantienen actualmente para la relación P
- #$D(P.i)$: el tamaño del dominio del cuál $P.i$ obtiene su valor.
- #$P.i$: el número de diferentes valores que $P.i$ puede tomar.
- #$P.I$: el número de diferentes combinaciones de valores que los elementos de $P.I$ pueden tomar. Para I = $\emptyset$, se toma I = 1
- #$P.i/j$: el máximo número de diferentes valores que $P.i$ puede tomar para cada posible valor de $P.j$, donde i $\neq$ j.
- #$P.I/J$: el máximo número de diferentes combinaciones de valores que los elementos de $P.I$ pueden tomar para cada posible combinación de valores de elementos de $P.J$, donde $I \cap J = \emptyset$. Para $I = \emptyset$, se toma #$P.I/J$ = 1. Para $J = \emptyset$, se toma #$P.I/J$ = $P.I$
  Si j o un elemento de J pueden tomar únicamente un valor constante, digamos c, se especifica siguiendo a j o al elemento de J, respectivamente, con =c

Es fácil observar que se mantienen las siguientes restricciones básicas:

$\#P = \#P.\{1, ..., a\}$ para una relación P de a argumentos
$\#P.i \leq \#D(P.i)$
$\#P.I \leq \#P.J$ para $I \subseteq J$
$\#P.(I \cup J) \leq \#P.I \times \#P.J/I$ y $\#P.J/I \leq \#P.J$ para $I \cap J = \emptyset$

Esto implica algunas restricciones comúnmente usadas, incluyendo en particular:

$$\#P.i \leq \#D(P,1) \times \ ... \ \times \#D(P.a)$$

En nuestro algoritmo, cada hecho se añade como variable booleana una sola vez. Cada hecho que hace el predicado de una regla de la `forma 1` cierto y cada combinación de hechos que hace simultáneamente ciertos ambos predicados de una regla de la `forma 2`

sólo se consideran una vez. A estas acciones se las llama disparo de la regla correspondiente. Cada combinación de hechos que hace ambos predicados ($P1$ y $P2$) de una regla simultáneamente ciertos se considera una sola vez. Para comprobar esto, únicamente para las reglas de la `forma 2` en la que ambos predicados sean intensionales, el concepto de ordenamiento expuesto hace esta demostración trivial para el caso en el que uno o ambos predicados sean extensionales. Volviendo al caso en el que ambos predicados sean intensionales, debe considerarse que los mapas auxiliares sobre los conjuntos que intervienen en las reglas de la `forma 2`, que fueron creados para optimizar la búsqueda para un hecho f de $P1$ o $P2$, se crean después de haber obtenido dicho hecho f a partir de la variable booleana correspondiente y se utilizan con posterioridad. Es decir, los mapas auxiliares sólo se actualizan tras haber evaluado la variable booleana correspondiente. Por lo tanto, un hecho $f1$ de $P1$ se combina una única vez con cada hecho de $P2$ obtenido antes que $f1$, y cada hecho de $P2$ obtenido después de $f1$ se combina una única vez con $f1$. Es por lo tanto fácil observar que la complejidad temporal es el número total de disparos de todas las reglas, como se analiza abajo, dado que cada disparo puede implicar un nuevo hecho como instancia de la conclusión por lo que, debe, en general, ser considerado al menos una vez. En este sentido, el tiempo de ejecución del algoritmo derivado es óptimo.

Para cada regla r, sea $r.\#numeroDisparos$ el número total de veces que r es disparado. Se usa IXs para denotar el conjunto de índices de los argumentos Xs en un predicado. Para una regla de la `forma 1`, tenemos

$$r.\#numeroDisparos \ = \ \#P$$

Para una regla de la `forma 2`, tenemos

$$r.\#numeroDisparos \ \leq min(\#P1 \times P2.IX2s/IYsC2s, \\ \#P2 \times P1.IX1s/IYsC1s)$$

4.5. Evaluación

El orden de evaluación de las reglas tiene un impacto directo en el rendimiento de la obtención de resultados, aunque sin afectar a la complejidad temporal. El tiempo de ejecución puede variar bastante según se aplique un ordenamiento de las reglas u otro, además la inclusión de la negación descrita en la Sección 2.3 también obliga a imponer un determinado orden. En esta sección estudiamos en profundidad la estrategia de ordenamiento desarrollada para establecer una evaluación de reglas eficiente y compatible con el modelo de negación descrito. A partir del concepto de grafo de dependencias de predicados (GDP) construiremos un algoritmo que nos permitirá establecer un orden eficiente para la evaluación de las reglas y compatible con el modelo de negación descrito.

4.5.1. Descripción del algoritmo de ordenamiento

En esta sección se va a detallar en profundidad el funcionamiento del algoritmo de ordenamiento. Para ello vamos a comenzar presentando varias definiciones.

Para mostrar el proceso completo de funcionamiento del algoritmo, vamos a trabajar sobre un ejemplo en el que modificamos ligeramente el programa descompuesto del análisis de punteros de Andersen con dos reglas adicionales. Definimos un átomo $G(X, Y)$ a partir de los siguientes átomos nuevos $M(X, Z)$, $H(Z, Y)$ y $T(X, Y)$. A continuación se muestran las reglas de dicho programa:

$$vP(X,Y) \ :- \ vP0(X,Y). \quad (1)$$
$$vP(X,Y) \ :- \ A(X,Z),\ vP(Z,Y). \quad (2)$$
$$T1(Z,Y,S) \ :- \ S(X,S,Z),\ vP(X,Y). \quad (3)$$
$$hP(Y,S,T) \ :- \ T1(Z,Y,S),\ vP(Z,T). \quad (4)$$
$$T2(Y,S,Z) \ :- \ L(X,S,Z),\ G(X,Y). \quad (5)$$
$$vP(Z,T) \ :- \ T2(Y,S,Z),\ hP(Y,S,T). \quad (6)$$
$$G(X,Y) \ :- \ M(X,Z),\ H(Z,Y). \quad (7)$$
$$G(X,Y) \ :- \ T(X,Y). \quad (8)$$

Figura 4.10: Programa de ejemplo para la evaluación de reglas

En la Figura 4.11 ilustramos el grafo de dependencias de predicados para el programa utilizado.

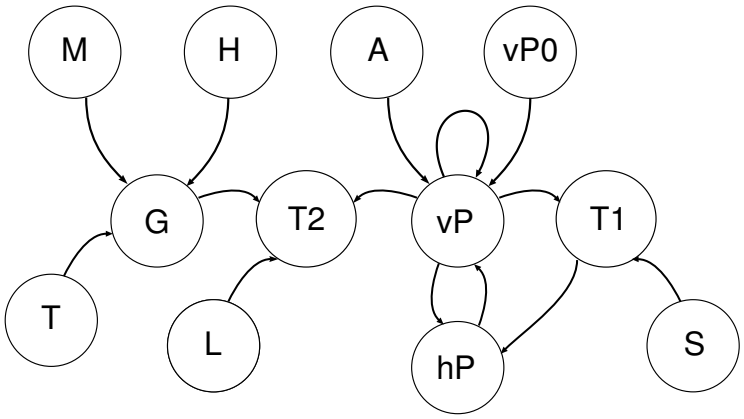

Figura 4.11: Grafo de dependencias de predicados

Definición 3 (Algoritmo de ordenamiento de reglas) *Una vez se ha construido el grafo de dependencias de predicados para un programa* Datalog*, se deben realizar los siguientes pasos para obtener el orden de ejecución de las reglas:*

1. *Se etiquetan los nodos que aparecen negados en alguna regla.*
2. *Se calcula el grado de incidencia de cada nodo. Teniendo en cuenta que es un grafo dirigido, el grado es el número de aristas que inciden en dicho nodo.*
3. *Se marcan los nodos cuyo grado de incidencia sea 0 o cuyo grado, siendo mayor, dependa única y exclusivamente de nodos ya marcados. Este proceso se realiza hasta que ya no se puedan marcar más nodos.*

4. *Se separan los nodos por bloques de la siguiente manera:*

   a) *Los nodos marcados con grado de incidencia 0 se colocan en el primer bloque.*

   b) *Los nodos marcados cuyo grado de incidencia no sea 0 se colocan en el segundo bloque.*

   c) *El resto de nodos se colocan en el tercer bloque.*

5. *Una vez obtenidos los bloques, se ordenan los nodos dentro de cada bloque de la siguiente manera:*

   a) *Los nodos del tercer bloque se colocan por orden decreciente usando el grado de incidencia de cada nodo, teniendo únicamente en cuenta los nodos incluidos en ese bloque. En caso de empate, se deben incluir los nodos del bloque anterior.*

   b) *Los nodos del segundo bloque se colocan teniendo en cuenta sus antecesores. Si existe un nodo con un antecesor dentro de este mismo bloque, debe colocarse después que su antecesor del mismo bloque.*

   c) *Los nodos del primer bloque deben colocarse teniendo en cuenta el orden del segundo bloque. Primero deben colocarse los nodos antecesores, en el primer nodo, de los nodos del segundo bloque. Después el del segundo y así sucesivamente. El orden del resto de nodos es indiferente.*

6. *Si en el tercer bloque existen predicados etiquetados, se genera una nueva sección de bloques volviendo a repetir desde el paso 1, incluyendo únicamente las reglas en las que aparecen dichos nodos marcados y todas las reglas que tengan predicados que dependan de éstas.*

En la Tabla 4.2 se muestra como quedaría cada bloque tras la aplicación del algoritmo anterior al grafo de la Figura 4.11.

$1^{\underline{er}}$ bloque	$2^{\underline{o}}$ bloque	$3^{\underline{er}}$ bloque
M	G	vP
H	T2	hP
T		T1
L		
A		
vP0		
S		

Cuadro 4.2: Nodos separados por bloques y ordenados.

La estrategia de resolución descrita en esta sección realiza dos operaciones para resolver el problema: actualizar la estructura de datos añadiendo una nueva solución y navegar a través de algún conjunto para obtener nuevas soluciones. Estas operaciones se realizan

en cada una de las ecuaciones del formalismo que sirven para modelar reglas de la `forma 2`. No siempre es necesario realizar tales operaciones para obtener el conjunto de soluciones ya que, utilizando el procedimiento descrito, se establece un orden que optimiza la navegación a través de los conjuntos y la propagación de información a través de las variables booleanas. A continuación se describen las condiciones necesarias para realizar cada operación:

– Si el nodo participa en una regla de la `forma 2` que genera un nodo que está en un bloque posterior:

  - Si el otro nodo que participa en la regla es de un nivel anterior, o es del mismo nivel y ya ha sido procesado, es necesario navegar a través de algún conjunto para comprobar si existen nuevas soluciones.
  - Si el otro nodo que participa en la regla es de un nivel posterior, o es del mismo nivel y no ha sido procesado, es necesario actualizar la estructura de datos añadiendo una nueva solución.

4.5.2. Aplicación al análisis de alcanzabilidad de definiciones

Vamos a presentar ahora una definición simple para el análisis de alcanzabilidad de definiciones. Hemos elegido dicho análisis ya que es sencillo, se encuentra dentro del marco del análisis estático de código y permite mostrar el funcionamiento del algoritmo de ordenamiento cuando existen predicados negados. Dicho análisis puede encontrarse en la Figura 4.12.

$$
\begin{array}{ll}
kill(I,D) \ :- \ defines(I,X),\ defines(D,X). & (1)\\
out(I,I) \ :- \ defines(I,X). & (2)\\
out(I,D) \ :- \ in(I,D),\ \neg kill(I,D). & (3)\\
in(I,D) \ :- \ out(J,D),\ pred(J,I). & (4)
\end{array}
$$

Figura 4.12: Análisis para la alcanzabilidad de definiciones

Para este análisis vamos a asumir que cada sentencia es un bloque y que cada sentencia define exactamente una variable. El predicado extensional $pred(I, J)$ significa que la sentencia I precede a la sentencia J. El predicado extensional $defines(I, X)$ significa que la variable definida por la sentencia I es X. Vamos a utilizar los predicados intensionales $in(I, D)$ y $out(I, D)$ para significar que la definición D alcanza el comienzo o el final de la sentencia I respectivamente. Una definición es el número de una sentencia.

A continuación se muestra el grafo de dependencias generado para dicho análisis. Como se puede observar en la figura el predicado $kill$ lleva la marca de negación.

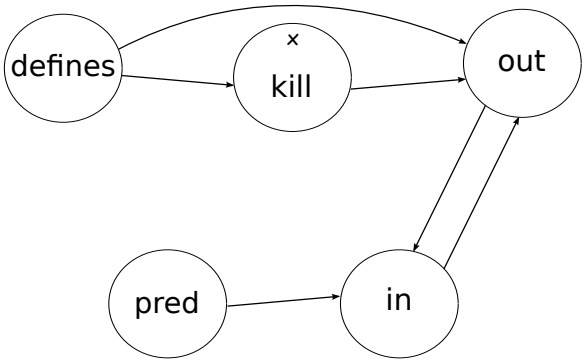

Figura 4.13: Grafo de dependencias de predicados

En la Tabla 4.3 se muestra como quedaría cada bloque tras la aplicación del algoritmo anterior al grafo de la Figura 4.13.

$1^{\underline{er}}$ bloque	$2^{\underline{o}}$ bloque	$3^{\underline{er}}$ bloque
defines	kill	in
pred		out

Cuadro 4.3: Nodos separados por bloques y ordenados.

En definitiva, el presente algoritmo mejora la estrategia de resolución determinando un conjunto reducido de operaciones a realizar para obtener todas las soluciones de un programa Datalog cualquiera. Además dicho orden permite ejecutar programas que incluyan predicados negados, siempre y cuando dichos programas sean estratificables. Por otra parte, aunque no ha sido descrito en el texto, el grafo de dependencias de predicados puede utilizarse como base para otros tipos de análisis. Por ejemplo:

- Eliminación de reglas inútiles a partir de las componentes conexas existentes en el grafo.
- Evitar añadir nuevas soluciones si en cada componente conexa del grafo únicamente existe un predicado intensional.

Conclusión En el presente capítulo se ha desarrollado un formalismo basado en la lógico que permite dar soporte a la estrategia de evaluación descrita en la Sección 4.1. Se ha presentado una pequeña introducción a la complejidad de los problemas lógicos que servirá como base para desarrollar en el Capítulo 5 técnicas que traten de obtener la menor complejidad a la hora de resolver un problema lógico. Además se ha desarrollado un algoritmo que aporta dos grandes beneficios: por un lado, permite reducir el número de operaciones necesarias para obtener las soluciones de un problema y por otro lado y más importante permite la evaluación de programas lógicos negados [4].

[4] Da soporte a la evaluación de programas estratificados.

CAPÍTULO 5

Mejorando la complejidad

En el presente capítulo se van presentar diversas estrategias y metodologías de transformación para tratar de reducir la complejidad de los programasa lógicos. Dichas técnicas abarcan un amplio conjunto de áreas de trabajo que comprenden desde la elección de la descomposición óptima para un determinado programa basándose en técnicas de *machine learning* o análisis estadísticos, hasta técnicas que tienen en cuenta la naturaleza y las relaciones de los dominios que conforman dichos programas. También se ha incluido una técnica ampliamente difundida en la literatura que permite mejorar la complejidad en algunos casos; dicha técnica es conocida como *magic-sets* [BMSU86a]. La técnica de los *magic-sets* se ha incluido por constituir un referente dentro de las técnicas que tratan de lidiar específicamente con la complejidad de programas lógicos.

En el gráfico de la Figura 5.1 se ilustran las diferentes fuentes de información que pueden utilizarse para establecer el análisis de un programa lógico y las relaciones que existen entre ellas. A partir de dichas fuentes, mediante un pequeño ejemplo se presentan más adelante las técnicas desarrolladas para tratar de reducir la complejidad de los programas lógicos.

Para tratar de establecer un análisis óptimo, las técnicas, pueden por un lado, utilizar el propio **programa** que especifica, ¿qué queremos conocer? y determina las relaciones existentes entre los diferentes dominios. Por otro lado, contamos con la **base de hechos**, que para un programa establece ¿qué es lo que ya se conoce de una instancia concreta del problema?. También podemos utilizar el propio mecanismo de **evaluación** que se encarga determinar como se obtienen las soluciones.

Cabe destacar que en el presente texto solo se expondrán brevemente las nociones básicas de cada técnica. No se va a entrar en gran profundidad ya que en algunos casos supondría dedicarles capítulos enteros y eso va mas allá de las pretensiones de este trabajo. Para las técnicas que no se desarrollen en profundidad por las razones ya mencionadas se incluirán las nociones básicas sobre las que poder seguir profundizando. Dicho ejercicio queda como trabajo futuro.

5.1. Estableciendo la descomposición óptima

En esta sección se van a presentar algunas técnicas para elegir la descomposición para un problema lógico. El concepto de la descomposición se ha desarrollado ampliamente en la Sección 4.2. Se debe tener en cuenta, como se ha comentado en la Sección 4.4.1 dedicada a la complejidad de los programas lógicos, que el problema no se puede resolver de manera óptima. Por ello se deben realizar aproximaciones basándose en la aplicación de técnicas

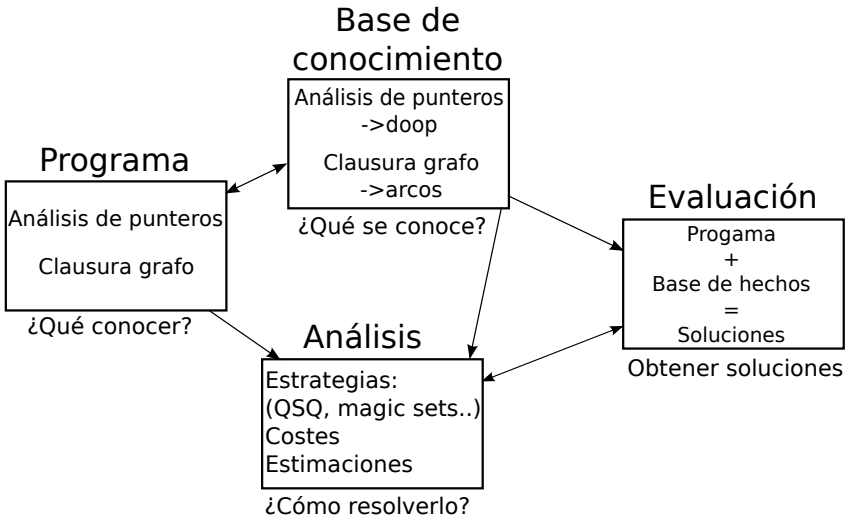

Figura 5.1: Fuentes para establecer el análisis de un programa lógico.

que proporcionen buenos resultados. Dichas técnicas tratan de explotar en la medida de lo posible las fuentes de información recogidas en la Figura 5.1.

Vamos a comenzar presentado una sencilla especificación que nos va a servir de base para exponer las técnicas desarrolladas durante el resto de la sección. Dicha especificación permite obtener las líneas genealógicas colaterales ordinarias masculinas ascendentes. A pesar de lo complejo del nombre la especificación señala los hombres de una familia que pueden desempeñar el papel de tío, independientemente de la generación a la que pertenezcan. A continuación se muestra dicha especificación:

$$ancestor(X,Y) \ :- \ parent(X,Y). \quad (1)$$
$$ancestor(X,Y) \ :- \ ancestor(X,Z),\ parent(Z,Y). \quad (2)$$
$$uncle(X,Y) \ :- \ man(X),\ ancestor(Z,Y),\ brother(Z,X). \quad (3)$$

Centrémonos en la regla número (3) que es la única que podemos descomponer. Se pueden obtener tres descomposiciones diferentes, dependiendo de las particiones seleccionadas. Tomando la partición con los predicados `man` y `ancestor` se obtiene la siguiente descomposición:

$$temp1(X,Z,Y) \ :- \ man(X),\ ancestor(Z,Y). \quad (1)$$
$$uncle(X,Y) \ :- \ temp1(X,Z,Y),\ brother(Z,X). \quad (2)$$

Tomando la partición con los predicados `man` y `brother` obtenemos la siguiente descomposición:

$$temp1(X,Z) \ :- \ man(X),\ brother(Z,X). \quad (1)$$
$$uncle(X,Y) \ :- \ temp1(X,Z),\ ancestor(Z,Y). \quad (2)$$

Para finalizar, tomando los predicados `ancestor` y `brother` obtenemos la siguiente descomposición:

$$temp1(X,Y) \ :- \ ancestor(Z,Y),\ brother(Z,X). \quad (1)$$
$$uncle(X,Y) \ :- \ man(X),\ temp1(X,Y). \quad (2)$$

Para mostrar la problemática existente a la hora de elegir una descomposición u otra basta con definir una pequeña base de hechos y comprobar el número de hechos que se obtienen para cada uno de los predicados temporales generados en cada descomposición. A continuación se muestra dicha pequeña base de hechos:

$man(juan).$	$man(jose)$	$man(luis)$
$man(miguel).$	$brother(juan, lola)$	$brother(lola, juan)$
$brother(jose, luis).$	$brother(luis, jose)$	$parent(juan, jose)$
$parent(juan, luis).$	$parent(lola, maria)$	$parent(jose, ana)$
$parent(jose, miguel)$		

La base de hechos presenta la descripción de una pequeña familia de dos generaciones, incluye a cuatro personas que son hermanos y a cuatro que son hombres. Utilizando dicha base de hechos para la primera descomposición se generan un total de **27** hechos temporales, para la segunda un total de **3** hechos temporales y para la tercera un total de **8** hechos temporales. Es fácil comprobar que entre la mejor descomposición (la segunda) y la peor (la primera) existe una diferencia en el coste temporal de un orden de magnitud (**27** frente a **3**). Por tanto, queda demostrada la necesidad de establecer algún mecanismo que permita elegir entre las diferentes descomposiciones de manera adecuada, máxime si tenemos en cuenta que, por regla general, las bases de hechos que se manejan habitualmente suelen tener como mínimo miles de hechos.

5.1.1. Maximizar el conjunto de variables comunes

La técnica de maximizar el conjunto de variables comunes trata de aprovechar la relación implícita que existe entre los diferentes dominios que forman parte de la especificación de un problema lógico. Es decir, la técnica trata de explotar la manera en la que se relacionan los diferentes dominios que forman parte de la especificación. Su funcionamiento se ha comentado de manera breve en el Capítulo 4 para justificar, en dicho capítulo, la elección de la descomposición para el análisis de punteros de Andersen. La técnica consiste en elegir, de entre todas las posibles descomposiciones para una regla, aquéllas que maximicen el conjunto de variables comunes.

Vamos a comentar ahora con mayor profundidad los beneficios y problemas que surgen de utilizar dicha técnica.
Beneficios:

- La técnica nos permite establecer ciertas relaciones semánticas a partir de la estructura de la especificación. Utilizando el ejemplo de las líneas genealógicas, la técnica permite separar las descomposiciones segunda y tercera de la primera. Vamos a comentar el significado semántico asociado a las reglas generadas por cada descomposición. La segunda genera una nueva regla temporal que busca primero todos los hombres que tienen un hermano. La tercera genera una nueva regla temporal que busca todos los ancestros que tienen un hermano. La primera obtiene el producto cartesiano de todos los hombres y todos los ancestros.

– Garantiza que nunca se va a escoger la descomposición con un peor coste [1]. Esto es así ya que al maximizar el conjunto de variables comunes hace que se reduzcan el resto de conjuntos [2].

– A la hora de resolver el programa trata de minimizar la navegación entre conjuntos [3].

Problemas:

– En ciertos casos la técnica es incapaz de diferenciar entre diferentes descomposiciones. En el caso del ejemplo anterior, es incapaz de distinguir entre la segunda y la tercera descomposición ya que ambas tienen las mismas variables comunes.

– No tiene en cuenta las bases de hechos. Esto no siempre es algo negativo, ya que en algunos casos no se puede acceder a la base de hechos antes de tener que tomar la decisión y en otros las bases de hechos disponibles no permiten aplicar otras técnicas.

5.1.2. Aprendizaje automático ("*Machine Learning*")

Se van a presentar ahora métodos que utilizan técnicas de Aprendizaje automático ("*machine learning*") para tratar de obtener la descomposición óptima. Estos métodos utilizan tanto propiedades estructurales del programa como diversas bases de hechos para tomar una decisión. Teniendo en cuenta estas fuentes de información, se extraen ciertas características para crear un modelo que nos permita predecir qué descomposición va a ser la mejor. Es importante destacar que para el buen funcionamiento de estos métodos es importante que exista una cierta entropía subyacente en las bases de hechos que se manejen ya ya que normalmente las técnicas de "*machine learning*" tratarán de explotar dicha entropía.

A continuación vamos a investigar un método de desarrollo en base al ejemplo de las líneas genealógicas. Se han generado 10 bases de hechos en las que se han impuesto ciertas condiciones:

– Cada padre puede tener un máximo de 3 hijos.

– Pueden existir 3 generaciones como máximo.

– Al menos el 50 % deben ser hombres.

Las características que se han utilizado para establecer el modelo son las siguientes:

– Caracterización del tamaño de la base de hechos:

  - Tamaño de los diferentes predicados (parent, man, ...).
  - Tamaño total de la base de hechos.

[1] El coste es m^k siendo m el tamaño de la base de hechos y k la aridad del predicado con mayor número de argumentos.

[2] La definición de los conjuntos se puede consultar en el apartado dedicado a la descripción de la estrategia en el Capítulo 4.

[3] Se desarrolla en el Capítulo 6.

- La frecuencia de los valores en los diferentes dominios; por ejemplo, cuántas personas tienen más de un hermano. Una aproximación sencilla sería contar cuántos valores repetidos existen por dominio.
- Ciertas propiedades estructurales del programa:
  - Tamaño del conjunto de variables compartidas por regla.
  - Aridad del predicado más grande.
  - Identificación de cada partición (valor nominal).

En base a estas características se ha desarrollado primero un modelo de regresión que trata de predecir el tamaño del predicado `temp1` para cada descomposición. Para generar el modelo se han obtenido 30 muestras (tres por cada base de hechos generada ya que existen tres posibles descomposiciones). Se han eliminado del modelo los datos de las características que no aportaban información para este problema. Para aprender el modelo se ha utilizado el software *weka* [4] aplicando un algoritmo basado en reglas de aprendizaje (tablas de decisión) y utilizando particiones de 5 elementos para validar el modelo. Se han obtenido los siguientes resultados:

- Coeficiente de correlación: $0{,}8006$
- Error medio: $5{,}4447$
- Error cuadrático medio: $9{,}5576$
- Error absoluto relativo: $48{,}4115\,\%$

Una vez obtenido el modelo de regresión se clasifican los datos generados por cada base de hechos y se comprueba si se han ordenado de forma correcta. Para este segundo modelo se han obtenido los siguientes resultados:

- Tasa de acierto: 75 % (Determinando la mejor descomposición).

Es importante resaltar que ese 75 % de acierto está asociado a la mejor descomposición para cada una de las diferentes bases de hechos. Otro dato importante es que nunca se ha escogido como mejor descomposición la peor. Entre los beneficios de la técnica encontramos:

- Aprovecha diferentes bases de hechos para tratar de tomar una decisión más informada.
- Los resultados para el ejemplo descrito son buenos y en ningún caso ha escogido la peor opción.

Problemas:

- Se requiere un número variable de bases de hechos para aprender el modelo. Además las bases de hechos deben contener un cierto grado de entropía.

[4] www.cs.waikato.ac.nz/ml/weka/

– No se puede aplicar en todos los casos ya que no siempre se pueden obtener un conjunto de bases de hechos representativas para un determinado problema.

Entre las posibles mejoras para obtener una precisión mayor podemos:

– Tener en cuenta la semántica del problema. Por ejemplo, para el problema anterior podemos añadir la información sobre cuántos hombres tienen hermanos.

– Crear un modelo por descomposición.

– Añadir información a partir de modelos estadísticos.

5.1.3. Modelos estadísticos

Los modelos estadísticos son de gran ayuda a la hora de tomar decisiones y se pueden utilizar por si mismos o en combinación con otras técnicas (por ejemplo, en combinación con las técnicas de "*machine learning*"). Para establecer un determinado modelo estadístico partimos otra vez de las bases de hechos y el propio programa. La finalidad es establecer un modelo o distribución de probabilidad con el cuál podamos tomar decisiones sobre qué descomposición es mejor para un conjunto de datos dado. Una distribución de probabilidad de una variable aleatoria es una función que asigna, a cada suceso definido sobre la variable aleatoria, la probabilidad de que dicho suceso ocurra.

El problema que nos encontramos a la hora de aplicar estas técnicas es la de encontrar una distribución ya conocida que permita modelar el comportamiento de las diferentes descomposiciones. Para solventar dicho problema podemos hacer uso del "Teorema Central del Límite" que establece que si tenemos un grupo numeroso de variables independientes y todas ellas siguen el mismo modelo de distribución (cualquiera que éste sea), la suma de ellas se distribuye según una distribución normal.

Otra distribución a tener en cuenta es la t de student. La distribución t de student es una distribución de probabilidad que surge del problema de estimar la media de una población normalmente distribuida cuando el tamaño de la muestra es pequeño. Con estas distribuciones podríamos estimar las medias poblacionales para cada descomposición y tomar una decisión en consecuencia.

5.2. *Magic-Sets*

A diferencia de las técnicas de evaluación descendentes, las técnicas de evaluación ascendentes no soportan el planteamiento de consultas, simplemente calculan todas las soluciones que existen para un problema. Sin embargo, en muchas ocasiones estamos interesados únicamente en las soluciones de una determinada consulta. Cuando se da esta situación se recurre a las técnicas de transformación de programas. Una de estas técnicas son los *Magic-Sets*, que son un conjunto de técnicas de transformación de programas diseñadas para trabajar junto a técnicas de evaluación ascendentes y cuyo rendimiento rivaliza con la eficiencia de las técnicas descendentes. Dicho conjunto de técnicas simula las

decisiones de propagación que ocurren en las aproximaciones descendentes. Esta simulación se consigue ya que su funcionamiento está fuertemente conectado con el algoritmo de evaluación descendente **QSQ** (*Query Subquery*).

El algoritmo **QSQ** generaliza la técnica de la resolución **SLD**, en cuyos principios semánticos está basado, pero aplicándose por una parte a conjuntos en lugar de a tuplas individuales, y diferenciándose también en el uso de constantes para seleccionar únicamente las tuplas relevantes tan pronto como sea posible. En particular, si un átomo de una relación intensional aparece en el cuerpo de una regla con una constante para algún atributo, esta constante puede ser propagada a las reglas que producen dicha relación intensional. De forma similar, se utiliza la "Transmisión de información lateral" para pasar información sobre el enlace de las constantes entre diferentes átomos en el cuerpo de una regla. Dichos enlaces se expresan utilizando patrones de enlace o adornos [5] en los átomos de las reglas, para indicar qué atributos están ligados [6] a alguna constante y cuáles están libres [7]. La técnica de los *magic-sets* presentada en esta sección trata de simular el algoritmo **QSQ** utilizando un método de evaluación ascendente.

La técnica de los *magic-sets* utiliza como fuentes de información un programa, una base de hechos y una consulta sobre dicha base de hechos. Vamos a desarrollar un ejemplo partiendo del programa de la sección anterior sobre lineas ascendentes laterales. En lugar de querer obtener el conjunto de todas las relaciones que existen para el predicado `uncle` para la base de hechos dada, podríamos estar interesados únicamente en las personas cuyo tío es Luís. Para ello, utilizando una técnica de evaluación descendente, bastaría con lanzar el siguiente objetivo:

$$uncle(luis, Y)?$$

Para poder trabajar con una técnica de evaluación ascendente como es nuestro caso debemos transformar el programa. Para ello vamos a utilizar la técnica de los *magic-sets* sobre dicho programa. Comenzamos utilizando la consulta como semilla, y teniendo en cuenta que para la consulta dada la primera componente está ligada y la segunda libre, se utilizarán dichas decoraciones para propagarlas al resto del programa. Los predicados extensionales no se decoran. De este modo obtenemos los siguientes predicados decorados:

$$ancestor^{bf}(X, Y); uncle^{bf}(X, Y)$$

El siguiente paso es obtener las relaciones de entrada para los predicados adornados y las relaciones suplementarias para cada regla. Por cada predicado adornado se genera otro que le servirá de entrada a dicho predicado y cuyas variables son las variables ligadas del predicado adornado original. Por ejemplo, para el predicado $uncle^{bf}(X, Y)$ se generará el predicado $input\_uncle^{bf}(X)$. Las relaciones suplementarias se obtienen de la siguiente manera:

– Para la primera relación suplementaria, el conjunto de atributos es el conjunto X_0 que contiene las variables ligadas del predicado de la cabeza; y para la última relación

[5] Un adorno es una cadena de letras que se añade a un predicado.
[6] El adorno se representa con la letra "b" del inglés "*bound*".
[7] El adorno se representa con la letra "f" del inglés "*free*".

suplementaria, el conjunto de atributos es el conjunto X_n que contiene las variables del predicado presente en la cabeza de la regla.

– Para $i \in [1, n-1]$, el conjunto de atributos de la i-ésima relación es el conjunto X_i que contiene las variables del predicado en cabeza que ya hayan aparecido y el conjunto de variables libres [8] que ocurran "antes" de X_i (*i.e.*, ocurren en H, $B_1, \ldots,$ B_i).

Las relaciones suplementarias se pueden observar en la Figura 5.2.

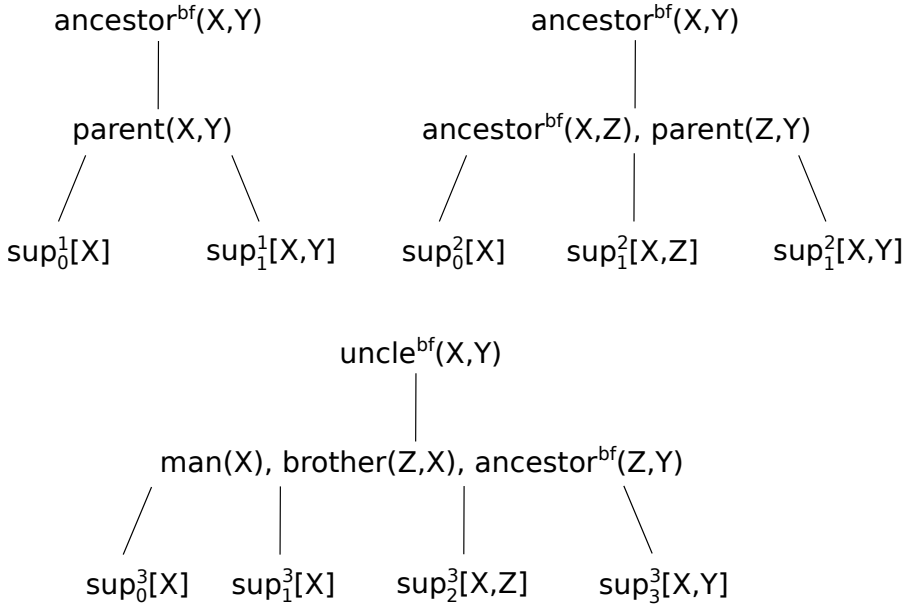

Figura 5.2: Relaciones suplementarias para el programa dado.

Una vez obtenidas las relaciones suplementarias, se procede a construir el programa transformado. Para ello, a partir de las reglas con sus suplementos, se generan nuevas reglas. La primera relación suplementaria de cada regla se substituirá por el predicado de entrada asociado a la cabeza de la regla, y la última relación suplementaria se substituirá por el predicado en cabeza. Tras realizar estas substituciones por cada relación suplementaria de cada regla se generará una nueva regla cuya cabeza sera la relación suplementaria y cuyo cuerpo contendrá dos predicados. El cuerpo de esta regla estará formado por el predicado Bi y la relación suplementaria anterior. Además, se generarán reglas para los predicados de entrada cuyo cuerpo estará formado por la relación suplementaria que preceda al predicado decorado en cuestión. De esta forma el programa transformado se puede observar en la Figura 5.3. Para este ejemplo se han generado un total de **3** hechos temporales, al igual que con la selección de esa partición para resolver el problema. Sin embargo, cabe resaltar como se puede observar en el programa transformado que uno de esos **3** hechos

[8] Una variable está libre si es la primera vez que aparece en el cuerpo y no aparece en ligada en la cabeza.

$$input\_uncle^{bf}(luis). \quad (1)$$

$$ancestor^{bf}(X,Y) \ :- \ input\_ancestor^{bf}(X), parent(X,Y). \quad (2)$$

$$sup_1^2(X,Z) \ :- \ input\_ancestor^{bf}(X), ancestor^{bf}(X,Z). \quad (3)$$

$$ancestor^{bf}(X,Y) : -sup_1^2(X,Z), parent(Z,Y). \quad (4)$$

$$sup_1^3(X) : -input\_uncle^{bf}(X), man(X). \quad (5)$$

$$sup_2^3(X,Z) : -sup_1^3(X), brother(Z,X). \quad (6)$$

$$uncle^{bf}(X,Y) : -sup_2^3(X,Z), ancestor^{bf}(Z,Y). \quad (7)$$

$$input\_ancestor^{bf}(Z) : -sup_2^3(X,Z). \quad (8)$$

Figura 5.3: Programa transformado utilizando la técnica de los *Magic-Sets*

no se ha generado mediante una refutación sino como parte del proceso de transformación. Los beneficios de utilizar la técnica se comentan a continuación:

- Nos permite manejar consultas utilizando métodos de evaluación ascendente.
- Aprovecha la "Transmisión de información lateral" y la propagación a a través de los adornos para simular la evaluación de técnicas descendentes y tratar de reducir el coste de evaluación.
- Un programa transformado con la técnica de los *magic-sets* puede satisfacer un conjunto de consultas.

Problemas:

- La propagación de adornos puede acabar duplicando dominios y haciendo que, debido a ello, el coste de resolver la consulta sea mayor que el de resolver el problema.
- El uso de la técnica no permite elegir la descomposición óptima y se debe resolver aplicando alguna de las técnicas ya expuestas.

5.3. Otras técnicas de optimización

En esta sección se van a presentar técnicas que tratan de reducir la complejidad del programa sobre el que se aplican. Es decir, permiten obtener el mismo número de soluciones con un coste computacional menor. En concreto vamos a presentar dos técnicas. La primera se aprovecha de las propiedades que existen entre los diferentes dominios representados en un programa para evitar calcular soluciones redundantes. La segunda trata de aplicar la ya conocida técnica de la evaluación parcial al ámbito de programas lógicos en el que estamos trabajando.

5.3.1. Estudiando las relaciones entre los conjuntos

Un predicado puede verse como una relación de grado igual a la aridad del átomo al que representa sobre conjuntos representados por los diferentes dominios que intervienen en la creación de dicho predicado. En esta sección vamos a tratar de explotar propiedades de relaciones sobre conjuntos para reducir la complejidad de resolver un problema lógico. Para esta técnica no vamos a utilizar más que el propio programa como fuente de información, ya que únicamente estamos interesados en las relaciones que subyacen a la especificación del problema.

Desde un punto formal, una relación sobre conjuntos se define de la siguiente manera. Una relación R, sobre los conjuntos A_1, A_2, ..., A_n es un subconjunto del producto cartesiano:

$$R \subseteq A_1 \times A_2 \times \ldots \times A_n$$

Un caso particular se da cuando todos los conjuntos de la relación son iguales: $A_1 = A_2 = \ldots = A_n$. En este caso se representa $A \times A \times \ldots \times A$ como A^n, pudiéndose decir que la relación pertenece a A a la n.

$$R \subseteq A^n$$

Para mostrar como se puede utilizar esta información para reducir la complejidad de los programas lógicos, vamos a centrarnos en la propiedad simétrica de las relaciones. Una relación cumple la propiedad de simetría siempre que un elemento de K está relacionado con otro, entonces ese otro elemento también se relaciona con el primero. Es decir:

$$\forall x, y \in K,\ xRy \ \Rightarrow\ yRx$$

Partiendo de esta propiedad podemos reducir el coste de resolución de un programa lógico; si durante la resolución de un programa lógico se obtiene un determinado hecho representado como xRy y sabemos que el predicado que representa dicho hecho cumple la condición de simetría no hace falta computar la refutación que indique que también existe el hecho yRx.

Vamos a ilustrar este concepto usando el programa de ejemplo anterior. En dicho programa se utiliza el predicado `brother`. Dicho predicado cumple la condición de simetría. En el ejemplo se ha utilizado como un predicado extensional. A continuación se presenta la regla que permite obtener dicho conocimiento:

$$brother(X, Y)\ :-\ parent(Z, X), parent(Z, Y).$$

De esta forma si durante la resolución del problema obtenemos que José es el hermano de Luis se puede añadir también el hecho de que Luis es hermano de José sin tener que obtener dicha refutación.

En esta sección se ha presentado el estudio de las propiedades de las relaciones sobre conjuntos (representadas por los predicados de un programa) para reducir la complejidad de la evaluación. En esa línea se ha presentado el uso de una relación que cumple la condición de simetría y se ha mostrado cómo aprovecharse de esa condición para reducir la complejidad de problemas lógicos. El estudio de otro tipo de propiedades y cómo influyen en la evaluación del tipo de programas lógicos presentados queda como trabajo futuro.

5.3.2. Evaluación parcial

La Evaluación Parcial (EP) es una técnica de transformación de programas que consiste en la especialización de programas respecto a ciertos datos de entrada, conocidos en tiempo de compilación, por lo que también se la denomina especialización de programas. La EP establece cómo ejecutar un programa cuando sólo conocemos parte de sus datos de entrada. De forma más precisa, dado un programa P y parte de sus datos de entrada **in1**, el objetivo de la EP es construir un nuevo programa P_{in1} que cuando recibe el resto de los datos de entrada **in2**, computa el mismo resultado que produce P al procesar toda su entrada **in1** + **in2**. Esto último asegura la corrección de la transformación efectuada.

En esta sección se va a aplicar esta técnica al ámbito de los lenguajes lógicos en el que que venimos trabajando. En concreto se va a aplicar en el mismo contexto que la técnica de los *magic-sets*, es decir en el caso de que partamos como fuentes de información de un **programa**, una **base de hechos** y un objetivo para el **programa** sobre dicha **base de hechos**. Al igual que con la técnica de los *magic-sets*, utilizaremos esta técnica cuando queramos aproximar una evaluación descendente. Vamos a utilizar el evaluador parcial ***ECCE*** [9] para obtener un programa especializado para el programa de las líneas genealógicas y la siguiente consulta:

$$uncle(luis, Y)?$$

Utilizando la técnica de evaluación parcial "*Conjunctive Fast*" se obtiene el siguiente programa especializado:

$$uncle(luis, A) \ :- \ parent\_\_2(A). \quad (1)$$
$$uncle\_\_1(A) \ :- \ parent\_\_2(A). \quad (2)$$

Y la siguiente base de hechos:

$$parent\_\_2(miguel). \quad parent\_\_2(ana).$$

A simple vista se puede observar que se ha generado un programa transformado óptimo, pero el uso de la técnica plantea los siguientes problemas:

- Al aplicar la especialización, la técnica toma decisiones sobre qué particiones elegir de manera desinformada. En la práctica esto supone que, en algunos casos, el coste de resolver el problema transformado sea peor que el de resolver el problema original.
- La especialización sólo sirve para una consulta determinada y el coste computacional puede ser peor que utilizar el motor de evaluación para obtener todas las soluciones.

Por estas razones no resulta interesante el uso de la evaluación parcial tradicional de forma automatizada en este ámbito de programas lógicos.

[9] http://www.stups.uni-duesseldorf.de/~asap/asap-online-demo/mecce.php

CAPÍTULO 6

Estructura de datos

En este capítulo se presenta la estructura de datos que va a dar soporte a las operaciones descritas por la estrategia en el Capítulo 4. Comencemos recordando cuáles son dichas operaciones: inicialización de conjuntos, obtención del dominio de un conjunto **dom**(M), obtención de la imagen de un conjunto $M\{X\}$, comprobación de conjuntos vacíos $S = \{\}$ y $S \neq \{\}$, recuperación de elementos de conjuntos y adición de elementos $S \cup := \ X$. Teniendo en cuenta esta serie de operaciones, la estructura de datos se ha diseñado para que permita un acceso eficiente a sus elementos y sea extremadamente compacta en memoria.

Tras haber presentado en detalle la estructura de datos, se comparará con otras soluciones encontradas en la literatura. Posteriormente se discutirán los pros y contras de cada alternativa matizándose las razones por las cuales la estructura de datos propuesta es más eficiente.

6.1. Antecedentes

Consideremos la posibilidad de utilizar una lista enlazada para representar un conjunto e igualmente para representar los conjuntos dominio e imagen de un mapa. Además, cada elemento en la lista enlazada del conjunto dominio contiene un puntero a la lista enlazada de su imagen. En otras palabras, representemos un conjunto como una lista enlazada y un mapa como una lista enlazada de listas enlazadas. Es fácil observar que, si el acceso asociativo se puede realizar en tiempo constante en el peor caso $O(1)$, también lo pueden hacer el resto de primitivas. Un acceso asociativo tendría un coste lineal si la lista tuviese que ser atravesada para localizar un elemento. En la literatura se puede encontrar referencias [Pai89] [CFH+91] que describen técnicas para diseñar estructuras enlazadas que soportan accesos asociativos constantes para el peor de los casos $O(1)$ para programas basados en conjuntos. Consideremos el siguiente fragmento de código que obtiene elementos del conjunto W y comprueba si dicho elemento existe en el conjunto S:

for X **in** $W, \ldots \ X$ **in** $S \ \ldots$
$M\{X\}$ donde el dominio del conjunto M es S

Nuestra intención es localizar el valor de X en el conjunto S después de que haya sido localizado en el conjunto W. La idea es usar un conjunto B, llamado base, para almacenar los valores tanto de W como de S, de tal manera que la obtención del valor a partir de W también localiza el valor de S. La base B es un conjunto (este conjunto es puramente conceptual) de registros, con un campo K que almacena la clave.

– El conjunto S se representa usando un campo de B: los registros de B cuyas claves pertenecen a S están conectadas por una lista enlazada cuyos enlaces se almacenan

en el campo S. Los registros de B cuyas claves no existen en S almacenan un valor especial de indefinición en el campo S.

– El conjunto W se representa separadamente como una lista enlazada de punteros a registros de B cuyas claves pertenecen a W.

Por lo tanto, un elemento de S se representa como un campo del registro, y se dice que S está *fuertemente basado* en B. Un elemento de W se representa como un puntero al registro y se dice que W está *débilmente basado* en B. La Figura 6.1 muestra la representación de base para el conjunto W conteniendo los elementos x_1 y x_3, y el conjunto S conteniendo los elementos x_1 y x_2. Dicha representación permite un número indefinido de conjuntos que se basen débilmente, pero sólo un número definido de conjuntos que se basen fuertemente. Esencialmente, la base B provee un tipo de indexado a los elementos de S partiendo de los elementos de W.

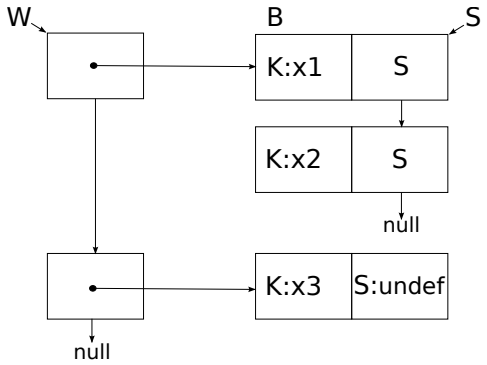

Figura 6.1: Representación de base para los conjuntos S y W usando la base B

6.2. Descripción

Dada la estrategia propuesta en el capítulo anterior, es necesario proporcionar dos tipos de accesos asociativos, uno de los cuales también conllevará la navegación de los elementos que contiene. A partir de esta información, en nuestro formalismo, necesitamos proporcionar los siguientes tipos de accesos:

1. conjunto de resultados: se necesita comprobar si un hecho ya ha sido computado antes de añadirlo al conjunto. Por ejemplo, operaciones como $Q \cup := \{(Ds)\}$; y

2. mapeado de relaciones auxiliares: para buscar todas las tuplas, $(Ys \cdot \sigma, Xis, Cis)$ (Figura 4.5) que ya han sido previamente computadas, dado un conjunto de argumentos constantes (Ys, Cis) y una relación H_i. Por ejemplo, operaciones como $\forall\, Xis \mid (Ys \cdot \sigma, Xis, Cis) \in H_i$.

A continuación describiremos un método uniforme para representar todos esos conjuntos y mapas. Utilizaremos un *array de punteros* y un *Trie* para los conjuntos que requieren

acceso asociativo, una *lista enlazada* para cada conjunto que es atravesado en busca de nuevas soluciones y un *mapa de bits* qué es útil si se necesita comprobar si una solución ya existe.

Trie. Uno de los aspectos claves para el funcionamiento eficiente de la estructura es la implementación del *Trie*. Un *Trie* o árbol de prefijos, es una estructura de datos basada en un árbol ordenado que se utiliza para almacenar un array asociativo. La posición en el árbol muestra cuál es la clave que tiene asociada un nodo. Todos los descendientes de un nodo tienen un prefijo común de la clave. El término *Trie* viene de la palabra inglesa *"retrieval"*, que quiere decir obtención. Hemos construido el *Trie* utilizando una estructura llamada Judy. A continuación se proporciona más información sobre dicha estructura y las razones por las que se ha escogido como base para implementar el *Trie*. Esta estructura, propuesta por Doug Baskins [Bas04], es un array asociativo que puede almacenar y buscar valores utilizando enteros o cadenas como claves. Está construido utilizando árboles digitales [FS86] con nodos adaptables y varios mecanismos de compresión. El tamaño de Judy no está predefinido y crece dinámicamente con respecto al numero de claves que almacena. Comparado con los árboles digitales clásicos, Judy consume mucha menos memoria al escoger apropiadamente las estructuras de datos para cada nodo teniendo en cuenta los sucesores de dicho nodo.

Existen cuatro tipo de arrays Judy en la familia Judy: Judy1, JudyL, JudySL y JudyHS. Para la estructura actual, con el array Judy1, que asocia enteros largos a bits, representamos los mapas de bits y utilizamos el array JudyL, que mapea enteros largos a punteros para construir el *Trie*.

Desde el punto de vista lógico, Judy es un árbol digital de 256 caminos que codifica las claves en bytes. Cada nivel almacena al menos un byte de la clave y las diferentes partes de la clave se almacenan en diferentes nodos a lo largo del árbol. Aunque Judy no es un árbol equilibrado, la profundidad máxima se puede predecir fácilmente. Tiene como máximo cuatro niveles para claves de 32 bits y ocho niveles para claves de 64 bits. Al contrario que en los árboles B^+ y T, la búsqueda no depende del número de claves almacenadas en el árbol, lo que hace que la operación sea muy rápida. El proceso de actualización también se puede beneficiar de esta virtud ya que la búsqueda es parte del proceso de actualización. Otros factores prácticos que favorecen su elección son que no necesita ningún tipo de adaptación y que en su diseño se ha tenido en cuenta el *principio de localidad* para maximizar su eficiencia.

En definitiva, esta estructura nos permite garantizar un tiempo de acceso constante, independientemente del número de claves y está optimizada para las dos operaciones básicas que se utilizan durante la gestión del *Trie* por parte de la estrategia descrita en el capítulo anterior, la búsqueda y la actualización.

Representación. Consideremos todos los dominios a partir de los cuales los argumentos de las relaciones toman valores. Para cada dominio D, podemos poner en correspondencia los valores de cualquiera de ellos con un entero en el *Trie* y utilizar dicho entero para referirnos a los elementos de los diferentes dominios. Esto nos permite compactar los dominios

que puede contener un determinado problema. Es importante señalar que la estructura se actualiza de una manera perezosa, es decir, los dominios (representados en la estructura mediante enteros) únicamente tienen correspondencia en la estructura si resulta necesario. Por ejemplo, un elemento del dominio ha sido referenciado en una solución. En la estructura representamos las soluciones y los conjuntos de prefijos ($PiYsXis$) de la siguiente manera:

- Cada solución de n componentes se representa mediante un camino de longitud $n-1$ en el *Trie*, siendo el primer elemento la raíz del *Trie*. Aprovechándonos de la actualización perezosa de la estructura, la comprobación del camino termina si alguna de sus transiciones no existe en el *Trie*. En el nivel $n-1$, es decir, en las hojas existe un mapa de bits que sirve para comprobar si la solución ya ha sido computada.

- Para los conjuntos de prefijos ($PiYsXis$), guarda en cada nodo una lista enlazada con sus sucesores. Esta lista nos servirá para iterar sobre los conjuntos deseados y obtener nuevas soluciones.

Modificaciones. Dada la estructura propuesta, se pueden realizar diversas modificaciones que permiten configurar la estructura en determinados casos. Una pequeña mejora que se puede realizar es evitar utilizar diferentes estructuras para las diferentes tuplas utilizadas. Esto no cambia la complejidad del problema pero permite tener un operador de acceso único. En el caso de que se tenga un programa Datalog cuyos predicados intensionales participan en otras reglas, se puede eliminar la duplicidad de información existente en la estructura eliminando el mapa de bits y manteniendo la lista ordenada. De manera análoga, aprovechando la representación del mapa de bits mediante un Judy array, se puede sustituir la lista de las hojas que representan a las soluciones y utilizar el mapa de bits para iterar en busca de nuevas soluciones. Estas medidas permiten ahorrar memoria pero a costa de empeorar el tiempo de resolución del problema.

Otra modificación interesante para la estructura de datos propuesta sería añadirle un gestor de conjuntos. Dicho gestor estaría encargado de hacer de intermediario entre los accesos a los conjuntos en las hojas del *Trie* y la representación de dichos conjuntos. Esta modificación persigue mantener la eficiencia de acceso a la estructura proporcionada por los mapas de bits, que garantizan un acceso constante, a la vez que intenta reducir el consumo de memoria.

Vamos a ilustrar la optimización anterior mediante un ejemplo, supongamos que al resolver un cierto problema se han obtenido como solución todos los hechos posibles. Esto quiere decir que, en el estado final de la estructura descrita, todas las hojas contendrían un mapa de bits en el que se haría referencia a todos elementos del dominio. En ese caso un gestor de conjuntos eficiente mantendría una sola copia de dicho mapa de bits. En el diseño de esta estructura nos enfrentamos a los siguientes problemas:

- identificar un determinado conjunto y;

- gestionar la creación, borrado y modificación de los diferentes conjuntos.

Para el primer problema, y sabiendo que los elementos del conjunto van a ser enteros, se puede implementar una función de dispersión especializada que permita identificar cada conjunto. El segundo problema es más complejo y conlleva el estudio en profundidad de la gestión de los conjuntos. Una primera aproximación podría ser la siguiente, en la que cada conjunto viene caracterizado por:

- Los valores del conjunto, que se pueden representar como hasta ahora utilizando un mapa de bits.
- Un contador de referencias que indica cuántos punteros tiene asociados el conjunto; es decir, cuántos nodos hoja apuntan a dicho conjunto.
- Un clave *hash* que permite identificar dicho conjunto.

El funcionamiento es el siguiente: cuando se accede a un conjunto que pertenece a una hoja del *Trie* para añadir un elemento, el gestor comprueba si dicho elemento ya forma parte del conjunto. En caso de que el valor no pertenezca al conjunto, se genera un valor de dispersión que servirá para referenciar al nuevo conjunto [1]. Si el valor de dispersión generado ya existe y, por lo tanto, sirve para referenciar otro conjunto, se actualizarán los contadores de referencias y punteros asociados. Es decir, el puntero del nodo hoja se modificará para que apunte al conjunto referenciado por el nuevo valor de dispersión y se actualizarán los contadores de referencias de ambos conjuntos. Si no existe el valor de dispersión calculado y el nodo hoja ya apunta a un conjunto, entonces se crea uno que es una copia del anterior más el nuevo elemento añadido y se actualizan los contadores de referencias y los punteros. Se actuará de forma análoga si el nodo hoja no tiene asociado ningún conjunto. Si durante la actualización de un conjunto su contador de referencias llega a 0, se borra [2]. En la Figura 6.2 se puede observar cuál sería el estado final del gestor para el conjunto vP de la Figura 6.3. El recuadro de la izquierda representa los diferentes punteros provenientes del *Trie* mientras que el recuadro de la derecha representa al propio gestor.

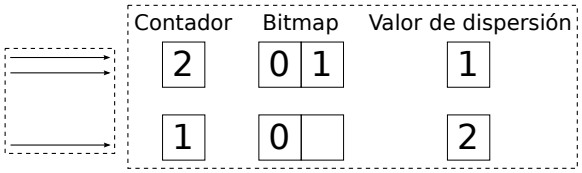

Figura 6.2: Estado final del gestor para el ejemplo del análisis de punteros

Ejemplo. A partir del ejemplo desarrollado en el capítulo anterior para el análisis de punteros de Andersen descrito en la Figura 4.4 y del conjunto de hechos generados a partir del programa Java que se puede encontrar en la Figura 4.7, describimos en la Figura 6.3 el estado final de la estructura tras haber resuelto el problema. Por simplicidad de exposición se ha creado una estructura para los predicados con 2 argumentos y otra para los predicados

[1] En efecto, los conjuntos también deben poder ser referenciados a partir del valor de dispersión.
[2] La política de borrado puede ser diferente.

con 3 argumentos. Como ya habíamos comentado, consideramos que el tamaño de los diferentes dominios del problema es el mismo. Las soluciones se obtienen navegando a través de la estructura utilizando la lista de nodos sucesores si fuese necesario. Por ejemplo, la solución $hP(0,0,0)$ se obtiene a partir del nodo raíz con un 0: en este nodo iteramos sobre hP, predicado del cual deseamos conocer su sucesores, y sólo obtenemos uno, el 0. Ya en el nodo hoja obtenemos el 0 de la lista otra vez. Para comprobar si la solución ya ha sido obtenida se realizaría un proceso análogo, pero el último valor se comprobaría en el mapa de bits. En la Figura 6.3 el mapa de bits está representado como el último elemento del array de punteros. Conviene recordar en este punto que el mapa de bits crece dinámicamente, por lo que alguno de ellos está vacío. Para iterar sobre un determinado conjunto para buscar nuevas soluciones, simplemente debemos indicar el prefijo del camino y el predicado sobre el cual queremos iterar. Por ejemplo temp2(0,0) nos devolvería la lista que contiene el elemento 3.

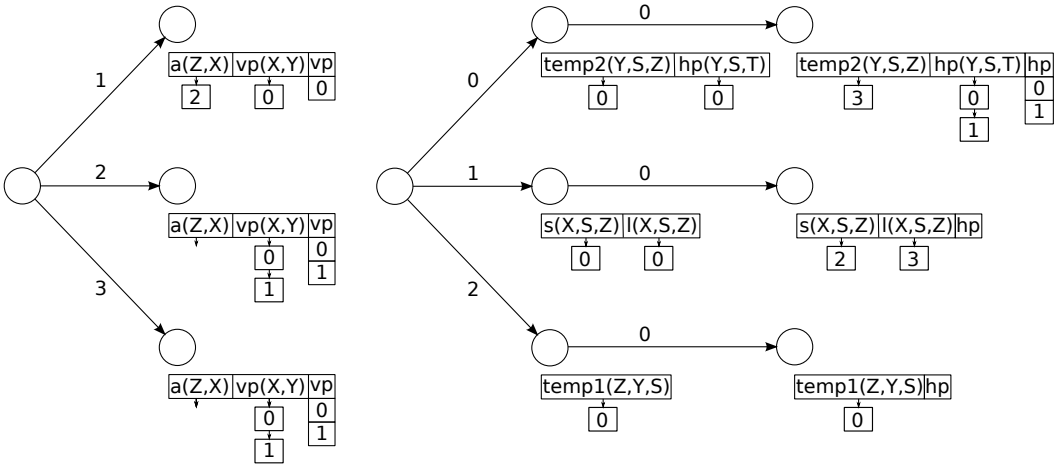

Figura 6.3: Estado final de la estructura de datos propuesta para el análisis de punteros de Andersen y un conjunto de hechos dados

6.3. Comparación con otras estructuras

En esta sección se va a comparar la estructura descrita con otras estructuras que se pueden encontrar en la literatura. Primero se comparará con la estructura especializada descrita en el artículo [LS09]. Posteriormente se comparará con las tablas de dispersión (*hash*) una estructura ampliamente utilizada, tanto para los evaluadores Datalog como para la resolución de sistemas de ecuaciones booleanas.

6.3.1. Estructura descrita por Liu y Stoller

La estructura descrita por Liu y Stoller en su artículo [LS09] se basa en la representación descrita en la Sección 6.1. Esta estructura, que sólo puede representar tuplas de

dos componentes, utiliza un array de punteros para representar la primera componente. La segunda componente se representa dependiendo del tipo de acceso que se quiera proporcionar. Si se quiere garantizar un acceso constante, los elementos de la segunda componente de la tupla se representan mediante un mapa de bits. Si por contra se pretende facilitar la navegación a través de un conjunto, la segunda componente se representa utilizando una lista enlazada. Puede darse el caso de que se necesiten ambas representaciones para un determinado conjunto. Con este diseño se asegura un acceso constante para la comprobación de la existencia de soluciones y un acceso eficiente para la navegación de conjuntos. Las tuplas con más de dos elementos se reemplazan por tuplas de dos elementos anidadas por la cola. Por ejemplo, $[X\ Y\ Z\ V]$ pasaría a ser $[X\ [Y\ [Z\ V]]]$. Para cada dominio D, el array de punteros proyecta los valores en D uno a uno desde 1 hasta $\#D$.

En la Figura 6.4 se puede observar el estado final de la estructura para una versión reducida del ejemplo estudiado. En la figura, las componentes R_i representan conjuntos de soluciones y las componentes W_i representan conjuntos de trabajo. Una vez se ha evaluado una componente del conjunto de trabajo, se traslada al de soluciones. También se utilizan listas para indexar las primeras componentes de los predicados.

Como ya se ha comentado, para facilitar la representación de la estructura se ha reducido el ejemplo que venimos utilizando hasta ahora a las dos siguientes reglas:

$$vP(X,Y) \ :- \ vP0(X,Y). \qquad (1)$$
$$vP(X,Y) \ :- \ A(X,Z),\ vP(Z,Y). \qquad (2)$$

y el conjunto de hechos dados a los tres siguientes elementos:

$$vP0(1,0). \quad vP0(2,1). \quad A(2,1).$$

Aunque la estructura descrita garantiza un acceso constante, en la práctica no se puede utilizar con instancias grandes de programas Datalog. Esto es debido a que, desde un inicio, la estructura debe representar todos los elementos del dominio independientemente de si son necesarios, y a que la estructura es altamente ineficiente cuando tiene que manejar tuplas con más de dos elementos. Por contra, la estructura descrita en el presente capítulo no posee estas restricciones. El hecho de utilizar un *Trie* permite compactar los datos de manera muy eficiente, utilizando prefijos, además de permitirle trabajar con tuplas de cualquier longitud, mientras que, como ya se ha comentado, la estructura de Liu y Stoller sólo puede manejar tuplas de 2 elementos. Otra mejora que permite la estructura propuesta es la capacidad de representar los diferentes dominios de un determinado problema utilizando una sola instancia de la estructura, mientras que la de Liu y Stoller necesita una instancia de la estructura por cada dominio diferente. Además, el hecho de utilizar un resolutor propio para el sistema de ecuaciones descritas en el formalismo permite a la estructura guardar únicamente la información esencial para generar nuevas soluciones. En definitiva, la estructura propuesta en este capítulo realiza un uso mucho más eficiente de la memoria, además de asegurar un acceso promedio mejor gracias a su capacidad de actualizarse de manera perezosa. Todo ello garantizando el acceso constante a los elementos de la estructura.

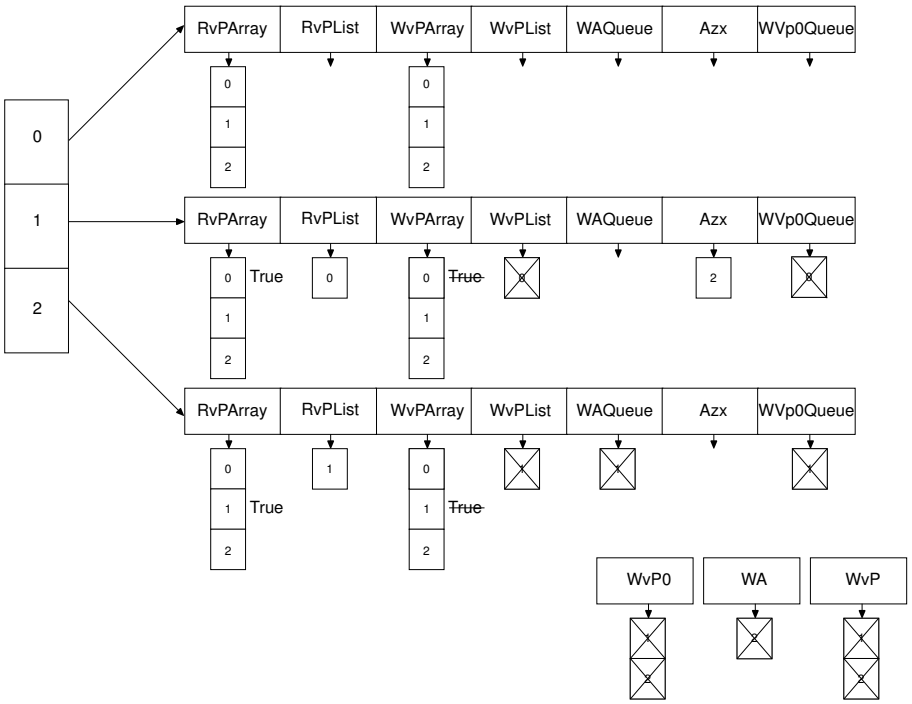

Figura 6.4: Estado final de la estructura descrita por Liu y Stoller para el ejemplo del análisis de punteros

6.3.2. Tablas de dispersión

Una aproximación clásica para resolver este problema es utilizar tablas de dispersión. Una tabla o mapa de dispersión es una estructura de datos que asocia claves con valores. Las tablas de dispersión se han diseñado para permitir realizar búsquedas eficientes sobre conjuntos de elementos. Los elementos se almacenan en una tabla, cuyo índice se obtiene partir de la clave generada por la función de dispersión. En caso de que dos o más claves distintas generen el mismo índice y se produzca una colisión, se debe proporcionar un mecanismo para ubicar los diferentes elementos que producen la colisión. Normalmente se utiliza una lista ordenada.

Comparando las tablas de dispersión con la estructura propuesta, se puede comprobar que la tabla de dispersión proporciona un acceso promedio constante $O(1)$, en lugar de para el caso peor. Además, las tablas de dispersión tienen el sobre coste de computar la función de dispersión.

Conclusión En definitiva, en este capítulo se han presentado los fundamentos de la estructura que se ha desarrollado para dar soporte a las operaciones descritas por la estrategia de resolución. Esta estructura es extremadamente compacta, permite un acceso constante a sus elementos y soporta una política de actualización perezosa. Además se han presentado

mejoras y variantes al diseño y se ha comparado con otras aproximaciones.

CAPÍTULO 7

Arquitectura del prototipo

En este capítulo se describe el prototipo que se ha desarrollado y en el que se han plasmado gran parte de las ideas detalladas en el presente manuscrito. Durante el capítulo se da una visión general del prototipo y se explica tanto su funcionamiento externo, en el que se detalla su uso como aplicación, tanto como su funcionamiento interno, en el que se detalla cada uno de los componentes que se han creado durante el proceso de desarrollo.

7.1. Visión general

En la Figura 7.1 puede verse la estructura general del prototipo desarrollado y el entorno en el que actúa.

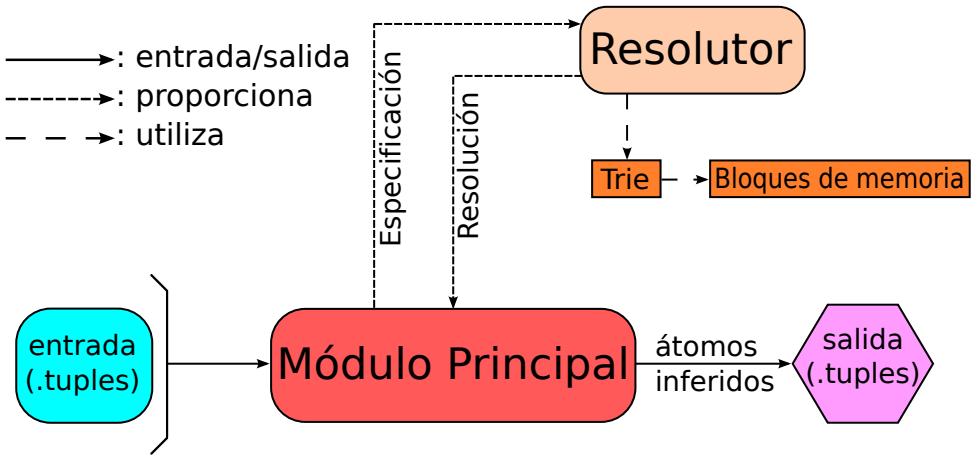

Figura 7.1: Visión general del resolutor

En el marco propuesto, un análisis queda especificado mediante un programa lógico Datalog. Dicho programa es un conjunto de reglas, hechos y consultas. Las reglas dependen del análisis que se desea hacer y se concretan en un fichero ".datalog". Por el contrario, el conjunto de hechos es dependiente de la instancia del problema a resolver. Un ejemplo son los hechos para un determinado programa a analizar. Se concretan en los diferentes ficheros ".tuples" de entrada.

Las consultas no se especifican ya que al tratarse de un resolutor ascendente, calcula todas las posibles soluciones que se pueden inferir. Por el momento, la codificación del sistema de ecuaciones especializado no se genera de forma automática, sino que se realiza a mano. En este caso se ha codificado el ejemplo de la Figura 4.4 que ha sido el ejemplo guía

utilizado durante todo el manuscrito. Como trabajo futuro se propone añadir un módulo al prototipo que permita automatizar dicha tarea incorporando las técnicas desarrolladas en el Capítulo 5, con lo cual, además, se podría dar soporte al uso de consultas.

En las siguientes secciones ahondaremos un poco más en las distintas partes de la herramienta.

7.2. Visión Externa

En esta sección se van a detallar tanto el conjunto de ficheros que el resolutor utiliza como entrada, como el conjunto de ficheros que genera como salida. Especificaremos su formato y las condiciones que deben cumplir para poder ser utilizadas por el resolutor. Es importante señalar que se van a detallar el conjunto de ficheros que maneja el resolutor en la actualidad, sin tener en cuenta los que serían necesarios de estar completo para automatizar completamente el proceso de análisis de un programa Datalog y su especificación en un sistema de ecuaciones especializado.

7.2.1. Entrada del programa

Tal y como se explicó al comienzo del presente capítulo, la entrada al resolutor está formada por un conjunto de hechos. A continuación analizaremos el tipo y el formato de los ficheros que maneja el resolutor.

7.2.1.1. Ficheros con hechos

Un fichero de hechos (*.tuples*) representa todos los hechos de un predicado determinado. Dicho fichero ha de llamarse "*nombre_del_predicado.tuples*" para que el resolutor pueda acceder a él. Para el ejemplo concreto implementado son: "*vP0.tuples*", "*A.tuples*", "*S.tuples*" y "*L.tuples"*. En las siguientes líneas, cuando hablemos de argumento o de aridad estaremos haciendo referencia a los argumentos o a la aridad del predicado cuyos hechos están representados en un fichero determinado.

Como se puede ver en la Figura 7.1, un fichero de hechos es un fichero de texto plano en el que cada línea consta de valores numéricos separados por espacios en blanco. Cada línea del fichero representa un hecho del predicado asociado al fichero. La columna i-ésima de una línea representa al elemento del dominio[1] que constituye el argumento i-ésimo en el hecho representado por dicha línea. Es decir, la cantidad de números presentes en cada línea será la misma para todo el fichero y será igual a la *aridad* del predicado de interés. Por ejemplo, la Figura 7.1 podría respresentar el predicado de interes vP cuya aridad es dos, por lo tanto, cada línea del fichero contiene dos columnas

7.2.2. Salida del programa

La salida del prototipo está formada por un conjunto de ficheros con los hechos inferidos por el resolutor. Cada fichero contendrá los hechos inferidos para un determinado

[1] Se entiende que es el dominio asociado al argumento i-ésimo del predicado.

1	0
2	1
2	2
⋮	

Cuadro 7.1: Un fichero de hechos (*.tuples*) asociado a un predicado.

predicado y su nombre será el de dicho predicado. En el ejemplo codificado se generan los ficheros "*vP.tuples*" y "*hP.tuples*".

7.3. Visión Interna

Esta sección presenta con mayor profundidad las diferentes partes de las que consta la estructura general del prototipo que se puede observar en la Figura 7.1. Dicho prototipo consta de un parser (150 líneas de código C) que se encarga de manejar los datos de entrada. Hemos desarrollado un resolutor específico para el sistema de ecuaciones descrito en la Sección 4.3.2 (550 líneas de código C). Para finalizar, la estructura de datos principal que utiliza el resolutor es la descrita en el Capítulo 6. Se presenta como un API que esconde la implementación [2] de 400 líneas de código C, que está conectada a la librería Judy [3].

7.3.1. Estructura de datos

En la Figura 7.2 se puede observar la API que se ofrece al resto de la aplicación para gestionar la estructura de datos, y que básicamente ofrece una interfaz sencilla para interactuar con el *Trie*. Como ya se ha comentado anteriormente, se ha utilizado el array Judy como base para implementar el *Trie*. La estructura de datos distingue entre dos tipos de nodos diferentes, los nodos internos y los nodos hoja. En la Figura 7.3 se puede observar la estructura codificada en el lenguaje C que se ha utilizado para representar los nodos hoja en el prototipo. El nodo contiene un array de punteros para acceder a los conjuntos sobre los que se desea iterar [4] y un puntero al mapa de bits. Los nodos internos son similares, con la única diferencia de que el puntero al mapa de bits se convierte en un puntero al siguiente nivel del *Trie*.

Para la implementación de la estructura se ha desarrollado un gestor de bloques de memoria. El gestor de bloques se ha implementado con la finalidad de reservar de forma eficiente únicamente la memoria estrictamente necesaria para la computación del problema. Ciertamente, la evaluación de un programa Datalog puede implicar millones de llamadas para reservar memoria, lo que genera un coste prohibitivo en términos de consumo de memoria. En el peor caso, se genera el doble de la memoria necesaria. Esto se debe al hecho de que el sistema operativo, en este caso Linux, reserva unos bytes adicionales para la gestión de cada bloque que reserva. Para solucionar esto, nuestro gestor de bloques

[2] La implementación actual no incluye el gestor de conjuntos.

[3] http://judy.sourceforge.net.

[4] Dichos conjuntos son $S(x,s,z)$, $L(x,s,z)$, $temp1(x,s,t)$, $temp2(y,s,z)$ y $hP(y,s,t)$.

```
extern void Trie_init();
extern void Trie_free();

extern void Trie_insert2(int, int, int);
extern void Trie_insert3(int, int, int);

extern intList * Trie_get_intList1(int, int);
extern intList * Trie_get_intList2(int, int);

extern void Trie_append_solutionVP(int, int);
extern void Trie_append_solutionHP(int, int, int);

extern int Trie_contains_solutionVP(int, int);
extern int Trie_contains_solutionHP(int, int, int);
```

Figura 7.2: API del trie.

```
struct TrieLeaf{
    intList *m[5];
    Pvoid_t RhP;
};
typedef struct TrieLeaf TrieLeaf;
```

Figura 7.3: Estructura que representa un nodo hoja en la aplicación.

reserva bloques de memoria de diferentes tamaños [5] [6]. Con el gestor de bloques se obtiene una ganancia tanto en tiempo, ya que se ahorran llamadas al sistema, como en memoria, con menos espacio dedicado a la gestión. Otra optimización que se consigue al utilizar el gestor de bloques proviene del hecho de que la mayoría de las operaciones soportadas por la estructura de datos se basan en iteraciones sobre conjuntos. Con el fin de hacer un mejor uso de las CPUs actuales, nuestro gestor de bloques sigue el mismo criterio que los arrays Judy, colocando los valores de los conjuntos en posiciones contiguas de la memoria para incrementar la eficiencia de la memoria cache [7].

7.3.2. Resolutor para el sistema de ecuaciones

Otro de los elementos que se han diseñado e implementado especialmente para el prototipo es un resolutor especializado para el sistemas de ecuaciones descrito en la Sección 4.3.2. Durante la fase de diseño del resolutor se han definido los siguientes requisitos:

[5] De 5 MB a 10 KB.

[6] Éste y otros parámetros pueden modificarse como parte del proceso de configuración del gestor de bloques.

[7] Explotando de esta manera el principio de localidad de las memorias cache.

- El resolutor debe poder incorporar un planificador que permita establecer y evaluar el orden de evaluación descrito en la Sección 4.5.1. Este orden de evaluación permite ahorrar operaciones para resolver el problema y da soporte a la evaluación de programas lógicos estratificados.

- El resolutor debe poder hacer uso de la estructura de datos descrita en el Capítulo 6. Esta estructura como ya se ha comentado anteriormente soporta de manera eficiente las operaciones descritas en el formalismo para la estrategia de evaluación.

En definitiva, el haber desarrollado un motor de resolución propio nos ha permitido aplicar muchas de las aportaciones descritas en este trabajo. En concreto, nos ha permitido tanto utilizar la estructura explicada en el Capitulo 6 como el orden de evaluación de la Sección 4.5.1. Otra importante optimización que se ha conseguido gracias al hecho de haber desarrollado un resolutor para el sistema de ecuaciones es la capacidad de construir el grafo de resolución de forma implícita, manteniendo en memoria únicamente la información necesaria para generar nuevos sucesores.

Conclusión En este capítulo se ha presentado el prototipo en el que se han implementado muchas de las ideas vertidas a lo largo del manuscrito. Durante el capítulo se detallada tanto su uso como aplicación como cada uno de los componenetes que lo conforman. Presentando en cada caso las mejoras que se han conseguido, como por ejemplo poder utilizar la estructura especializada o construir las soluciones de forma implicita. Todo ello acaba resultando en un resolutor extremadamente eficiente, tanto en consumo de memoria, como en tiempo de ejecucón.

CAPÍTULO 8

Experimentos

Este capítulo investiga sobre el rendimiento, medido en términos de tiempo de ejecución y memoria consumida, del prototipo desarrollado en este trabajo. Hemos comprobado la eficiencia y optimalidad de nuestra implementación comparándola con diferentes resolutores Datalog existentes en la literatura: XSB 3.2 [1], el prototipo de Liu y Stoller [2], al que llamaremos TOPLAS a partir de ahora y BDDBDDB [3]. Para alcanzar este objetivo, hemos desarrollado dos conjuntos de pruebas, uno utilizando conjuntos de hechos de entrada generados aleatoriamente y otro ejecutando conjuntos de hechos extraídos de proyectos Java reales.

Para el primer conjunto de experimentos, se ha desarrollado un script de Python que genera hechos aleatorios para el análisis de punteros de Andersen. Esto nos permite controlar el número de hechos de entrada generados y el tamaño de sus dominios. Las pruebas de referencia generadas ha resultado ser muy útiles para comprobar los límites de los resolutores en términos de tiempo de ejecución y consumo de memoria.

Para el segundo conjunto de pruebas, se han utilizado el conjunto de programas Java incluido en el marco de pruebas DACAPO, los hechos se han extraído usando el sistema SOOT [4]. Para este conjunto de experimentos también se ha añadido una versión del prototipo hecha en Python para compararlo con el prototipo de TOPLAS (también hecho en Python). Esta versión del prototipo se ha desarrollado utilizando tipos nativos de Python [5], por lo tanto, no se han podido implementar todas las técnicas aplicadas a la versión desarrollada en C. El haber desarrollado este prototipo en Python nos ha permitido resaltar directamente los beneficios que aportan nuestras nuevas estrategias con respecto a la herramienta TOPLAS.

El primer conjunto de experimentos se han realizado en un portátil AMD Athlon(tm) XP2000+ con 685 Megabytes de RAM ejecutando Linux Ubuntu 10.04. El segundo conjunto de experimentos se han realizado en un ordenador Intel Core 2 duo E4500 2.2 GHz (utilizando un solo core), con 2048 KB cache, 4 GB de RAM y ejecutando Linux Ubuntu 10.04. Como compilador se ha utilizado gcc 4.4.1. Los prototipos de Python se han ejecutado utilizando la versión 2.6.5 de Python y los resultados se han comprobado comparando experimentalmente la salida generada por todos los resolutores.

[1] http://xsb.sourceforge.net
[2] Proporcionado por los autores de [LS09]
[3] http://bddbddb.sourceforge.net
[4] http://www.sable.mcgill.ca/soot
[5] Diccionarios, listas y conjuntos

8.1. Experimentos con datos aleatorios

La Figura 8.1 compara el tiempo de usuario de CPU (en segundos) para conjuntos crecientes de hechos de entrada. La Figura 8.2 compara la memoria consumida (en megabytes) para conjuntos crecientes de hechos inferidos. En la Tabla 8.1 se muestran los detalles desglosados para cada conjunto de entrada. Las seis primeras entradas de la tabla corresponden a la primera gráfica de la Figura 8.1 mientras que las tres últimas corresponden a la segunda gráfica de la Figura 8.1 y a la Figura 8.2. Todos los hechos se han generado dentro un dominio de 10,000 elementos. En la Figura 8.1, la entrada para Xsb 4 corresponde a la ejecución de la versión del análisis de punteros de 4 reglas (Figura 4.2) con el resolutor XSB, mientras que la entrada Xsb 6 corresponde a la versión descompuesta (Figura 4.4). Si un test ha tardado más de 20 minutos en concluir no se ha incluido en las gráficas. Como se puede observar en las gráficas, el prototipo que hemos desarrollado se comporta mucho mejor, tanto en términos del tiempo de resolución como en consumo de memoria. Además es el resolutor que menor penalización obtiene, tanto en tiempo como en memoria, al resolver un conjunto mayor de hechos de entrada. En la gráfica de la Figura 8.1 se observan tiempos 150 veces más rápidos que XSB y 50 veces más rápidos que TOPLAS. También se puede observar que mientras el conjunto de entrada de 23750 se resuelve en $6,74$ segundos, TOPLAS tarda $325,81$ segundos y XSB tarda $1066,44$ segundos. Además, en ambas gráficas, se observa que la pendiente es mucho más pronunciada para el resolutor XSB. En cuanto al consumo de memoria mostrado en la Figura 8.2, se ha obtenido un consumo de memoria $3,5$ veces menor XSB y $7,5$ veces menor que TOPLAS. Para el conjunto de hechos más grande el prototipo acaba consumiendo $44,3$ MB, mientras que XSB consume $153,4$ MB y TOPLAS acaba consumiendo $337,03$ MB. Como se observa en la gráfica, el prototipo sufre un impacto mucho menor al aumentar la cantidad de hechos inferidos. Esto es debido a la estructura de datos descrita en el Capítulo 6. El prototipo gasta aproximadamente 20 MB [6] en añadir un millón de hechos inferidos, mientras que XSB gasta aproximadamente 120 MB y TOPLAS gasta aproximadamente 175 MB.

Entrada	vP0	A	S	L	Soluciones (vP + hP)
500	160	200	70	70	161
1000	320	400	140	140	335
2000	640	800	280	280	712
5000	1600	2000	700	700	2022
10000	3200	4000	1400	1400	5071
20000	6400	8000	2800	2800	51832
23500	7520	9400	3290	3290	783769
23650	7570	9460	3310	3310	1376876
23750	8000	9500	3325	3325	1878547

Cuadro 8.1: Conjuntos de hechos de entrada desglosados por predicados y soluciones.

[6] En realidad esta cifra algo es menor ya que el gestor de bloques suele reservar memoria por adelantado.

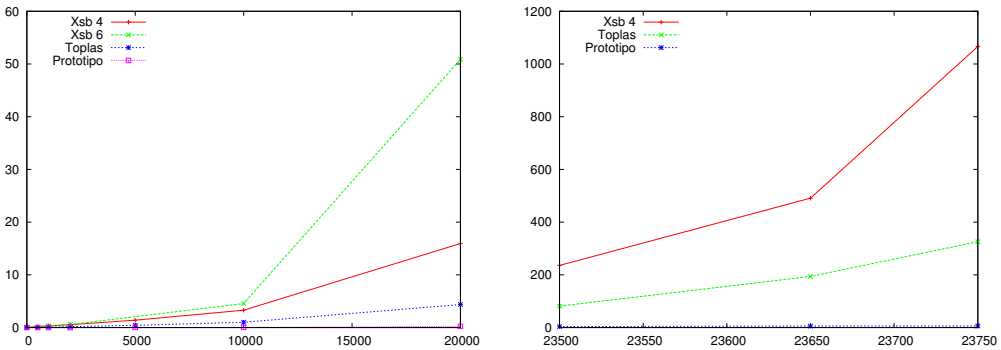

Figura 8.1: Tiempo de análisis (seg.) para un número creciente de hechos de entrada

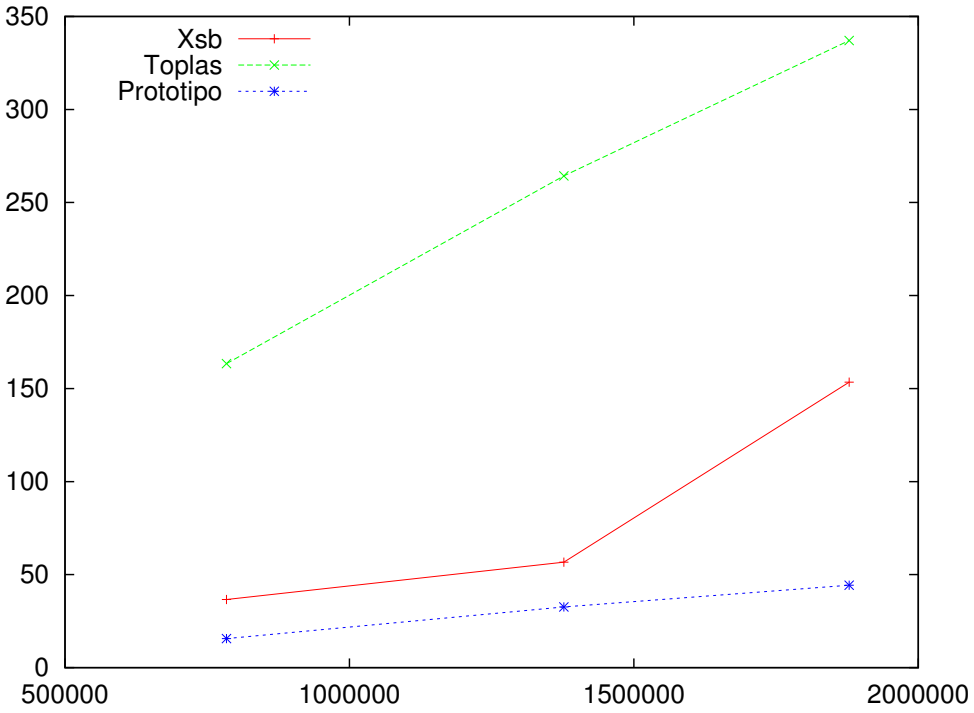

Figura 8.2: Consumo de memoria (MB.) para un número creciente de hechos inferidos

8.2. Experimentos con datos reales

En el segundo conjunto de experimentos, hemos utilizado el resolutor para resolver el conjunto de programas Java incluido en el marco de pruebas DACAPO, programas que son utilizados por miles de usuarios. El prototipo evalúa el marco de pruebas completo en solo 3 segundos con un tiempo medio por programa de 0.3 segundos. Como se puede ver en la gráfica de la Figura 8.3, donde el eje-y (seg.) está expresado usando una escala logarítmica, el prototipo es un orden de magnitud más rápido que el prototipo de TOPLAS y el resolutor BDDBDDB que a su vez son un orden de magnitud más rápidos que XSB. Para la prueba realizada sobre el programa *jython*, XSB evalúa el análisis de punteros en 482 segundos, BDDBDDB en 34 segundos, TOPLAS en 10 segundos, el prototipo hecho en Python en 5 segundos y el prototipo hecho en C en 0.406 segundos. En la Figura 8.4 dónde el eje-y (MB.) está expresado usando una escala logarítmica podemos observar que el prototipo hecho en C consume 5 veces menos memoria que XSB y un orden de magnitud menos que el el prototipo hecho en Python, TOPLAS y BDDBDDB. Para la prueba realizada sobre el programa *lusearch*, BDDBDDB ha requerido 326 MB. de memoria para completar el análisis, TOPLAS 116 MB., el prototipo hecho en Python 90 MB., XSB 34 MB., y el prototipo hecho en C 7 MB. Es importante destacar que los resultados de consumo de memoria del prototipo hecho en Python pueden mejorarse enormemente evitando algunos de los tipos nativos de Python que se han utilizado para construir la estructura de datos [7]. Este conjunto de pruebas nos permite mostrar que las técnicas utilizadas escalan muy bien para programas realmente grandes dentro del contexto del análisis de punteros de Andersen.

Conclusión En este capítulo se han presentado los experimentos realizados y se han mostrado los resultados obtenidos. En los experimentos realizados se ha comparado el prototipo con otros resolutores que se pueden encontrar en el estado del arte. Dicho prototipo se desarrolló en un breve período de tiempo (unos pocos días), incluyendo algunas de las ideas descritas en el manuscrito y como se puede comprobar por los resultados ha mejorado el estado de arte actual. Como trabajo futuro se plantea compararlo con soluciones comerciales como DOOP [8] utilizando diferentes análisis.

[7] En concreto el haber usado *sets* para representar los mapas de bits

[8] http://doop.program-analysis.org

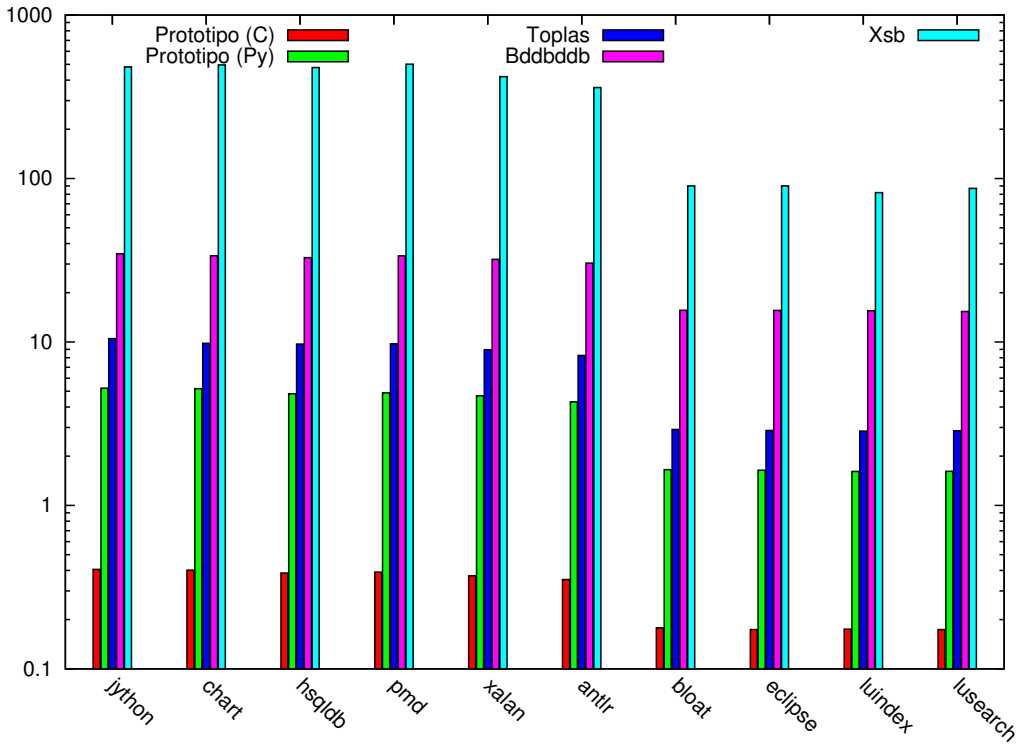

Figura 8.3: Tiempo de análisis (seg.) para el conjunto de pruebas DACAPO.

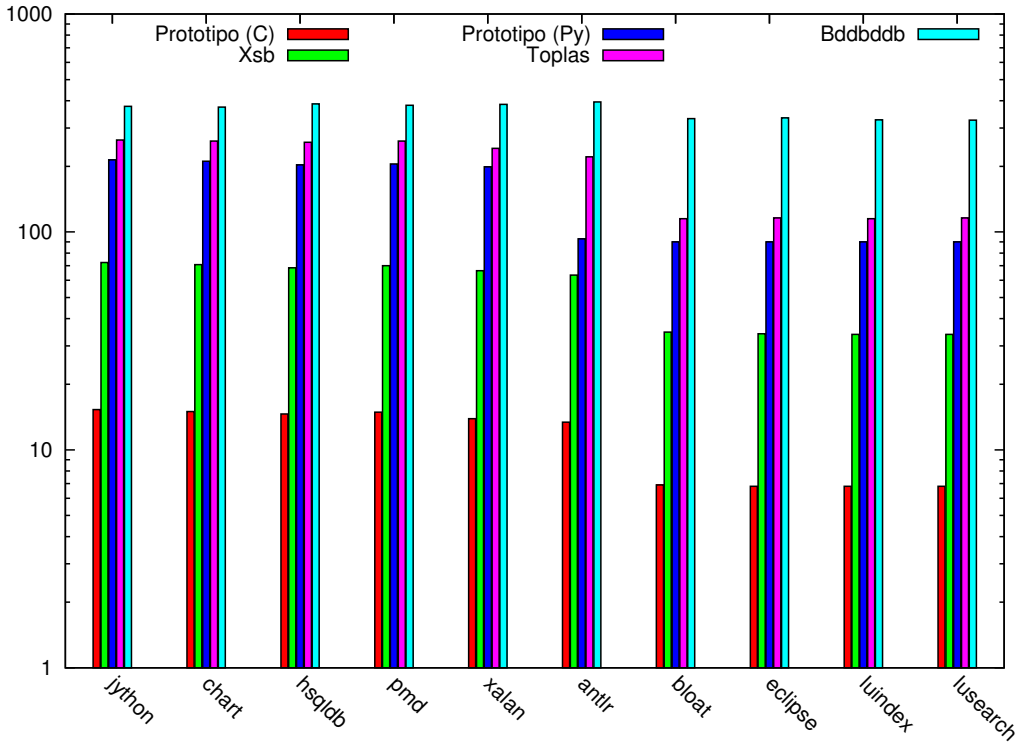

Figura 8.4: Consumo de memoria (MB.) para el conjunto de pruebas DACAPO.

Conclusiones y trabajo futuro

Se han desarrollado un conjunto de metodologías para realizar de forma eficiente la resolución ascendente de programas especificados en Datalog, incluyendo programas negados. Estas metodologías incluyen una estrategia de evaluación optimizada [9], un resolutor especifico para dicha evaluación y una estructura de datos especializada para soportar el conjunto de operaciones descritas por la estrategia y tratar de mitigar la explosión de estados inherente a todo programa Datalog. También se han presentado diferentes técnicas para manejar la complejidad de los problemas lógicos.

Además se ha desarrollado un prototipo para poder comparar la eficiencia de dichas aproximaciones con respecto a los resolutores existentes en el estado de la técnica. En concreto se ha experimentado [10] con el análisis de punteros de Andersen. En este prototipo se han plasmado gran parte de las ideas detalladas durante el manuscrito. Además, en las pruebas realizadas, el prototipo ha demostrado un comportamiento mucho mejor que el resto de resolutores, tanto en términos de tiempo de resolución como en términos de consumo de memoria, llegando incluso a solventar instancias que el resto han demostrado ser incapaces de manejar.

Trabajo relacionado

Datalog y los métodos para optimizarlo han sido ampliamente estudiados en las áreas de las bases de datos y la programación lógica. Las técnicas para la evaluación de los programas Datalog y de forma más general los programas lógicos pueden clasificarse atendiendo a varios criterios [AHV95]. Un criterio clave es la división entre las técnicas ascendentes y descendentes. Las técnicas descendentes razonan desde la raíz, mientras que las ascendentes lo hacen desde las hojas. Otro criterio clave es la división entre técnicas directas y métodos de transformación. Las técnicas directas evalúan directamente la consulta, mientras que los métodos de transformación transforman la consulta para posteriormente evaluarla utilizando una técnica directa. La evaluación *semi-naive* [CGT90] es una técnica directa y ascendente que evita la computación repetida de átomos básicos. Sin embargo, cada argumento de los átomos es instanciado sobre todos los valores del dominio durante la computación incremental, en vez de instanciarlos sobre los valores obtenidos de los átomos básicos inferidos, como en [LS09]. La transformación *magic sets* [BMSU86b] es una técnica de transformación para evaluaciones ascendentes. En esta técnica, las restricciones de la consulta se propagan a través del programa, obteniendo una evaluación igual de eficiente que una descendente para el mismo programa. Liu y Stoller [LS09] desarrollaron una aproximación que compila la especificación Datalog en una implementación especia-

[9] Especificada mediante un sistema de ecuaciones.

[10] Incluyendo el conjunto de programas Java ofrecido por en el marco de pruebas DACAPO.

lizada. Esta implementación especializada realiza una evaluación ascendente con garantías de coste en tiempo y memoria.

El uso de bases de datos deductivas y la programación lógica para el análisis de programas tiene una larga historia [Rep94][Ull89][DRW96]. La descripción de análisis sensibles al flujo como consultas a bases de datos fue aplicada por primera vez por Ullman [Ull89] y Reps [Rep94], quienes utilizaron la implementación ascendente con *magic-sets* de Datalog para derivar automáticamente una implementación local. Existen otros análisis de programas y métodos de *model-checking* que utilizan ecuaciones, restricciones, autómatas y lenguajes formales [CC77][HJ94][Rep98][EHRS00]. Sin embargo, la aproximación basada en reglas es típicamente más directa y general. Además, nuestra aproximación genera implementaciones y algoritmos derivados formalmente utilizando un método sistemático, en contraste al desarrollo ad-hoc de otros métodos; esto preserva la corrección y la complejidad asegurada para las implementaciones y algoritmos generados. PADDLE[11] fue el primer marco para el análisis de punteros de programas Java. Especifica un análisis en el lenguaje Jedd, una extensión del lenguaje Java que utiliza diagramas de decisión binarios (BDDs). BDDBDDB introdujo posteriormente (hasta cierto punto) Datalog para la especificación de análisis de programas, proporcionando un resolutor eficiente basado en la traducción a BDDs. Recientemente DOOP[12] se ha convertido en el marco de referencia para el análisis de punteros de programas Java. DOOP especifica su análisis exclusivamente en Datalog, convirtiéndose de esta manera en el primer análisis de punteros completamente declarativo.

Trabajos Futuros

Además de los problemas que se han dejado como posible trabajo futuro a lo largo del manuscrito, a partir de las ideas plasmadas en el proyecto realizado se pueden seguir varias líneas de trabajo que pueden complementar o profundizar en lo ya realizado. A continuación se enumeran éstas, comentando brevemente el enfoque así como sus ventajas e inconvenientes.

- Aplicar técnicas de distribución: Consistiría en obtener un algoritmo distribuido a partir de la estrategia actual. Parece la aproximación natural para solventar el mayor problema que plantea la resolución de los programas Datalog aplicados a un gran dominio, que es la ingente cantidad de memoria necesaria para almacenar las soluciones. Por contra, se puede plantear que si no se encuentra un método que escale de forma adecuada éste puede llegar a ser más lento que la metodología actual.

- Aplicar técnicas de paralelización: Consistiría en obtener una versión del trabajo actual que pueda funcionar de manera paralela. De esta manera, se seguiría la tendencia actual del mercado del hardware y de las empresas que comercializan soluciones basadas en Datalog. Esto permitiría obtener un mayor rendimiento computacional. Por contra, se debe mencionar que seguiría existiendo el problema de la memoria pa-

[11] http://www.sable.mcgill.ca/paddle/
[12] http://doop.program-analysis.org/

ra programas aplicados a un gran dominio y que en términos computacionales la solución actual es muy competitiva.

- Construir una herramienta de más alto nivel para el estudio de programas Datalog: Aunque la estrategia actual ya permite generar programas Datalog óptimos, esta parte del trabajo se realiza a mano. Teniendo en cuenta que ya existen herramientas para calcular los costes de cada posible solución, se podría desarrollar una propuesta para automatizar el análisis y la transformación. Este aporte permitiría generar un resolutor completamente automático.

- Completar el estudio centrándonos en problemas de interés alternativos: El conjunto de pruebas realizadas para comparar nuestra solución con el resto de resolutores existentes se ha centrado en el ejemplo del análisis de punteros de Andersen. Esta propuesta se centra en el desarrollo de un conjunto de pruebas más amplio para tener una mejor casuística. Dichos conjuntos de pruebas pueden centrarse tanto en la comparación sobre diferentes clases de análisis de punteros como en pruebas más generales para programas Datalog diversos.

- Profundizar en el análisis de punteros: Está propuesta trata de trabajar a un nivel anterior al de la resolución del programa Datalog, trabajando en la extracción de información de los programas y tratando de optimizar el funcionamiento del proceso global.

- Aplicar los algoritmos desarrollados a la búsqueda de fallos en aplicaciones reales: Esta propuesta se centra en aplicar los algoritmos desarrollados y los que se pudiesen desarrollar en alguno de los puntos anteriores a la resolución de fallos en aplicaciones reales. En particular, se podría comenzar realizando un estudio en profundidad sobre el análisis de recursos y pérdidas de memoria (*memory leaks*) en la aplicación PANGEA[13].

Es interesante comentar que ninguna de estas propuestas es incompatible con las demás. Como ejemplo, las dos primeras propuestas se podrían desarrollar en una misma solución para obtener un resolutor más sofisticado y eficiente.

[13] http://pangea.upv.es/

Bibliografía

[ABM09] Serge Abiteboul, Pierre Bourhis, and Bogdan Marinoiu. Efficient maintenance techniques for views over active documents. In *Proc. 12th Int'l Conf. on Extending Database Technology EDBT'09*, volume 360 of *ACM International Conference Proceeding Series*, pages 1076–1087. ACM Press, 2009. (Cited on page 1.)

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison Wesley, 1995. (Cited on pages 1 and 77.)

[And94] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, 1994. DIKU report 94/19. (Cited on page 27.)

[Bas04] Doug Baskins. Judy array. Available at: `http://judy.sourceforge.net/`, 2004. (Cited on page 57.)

[BB79] Catriel Beeri and Philip A. Bernstein. Computational problems related to the design of normal form relational schemas. *ACM Trans. Database Syst.*, 4(1):30–59, 1979. (Cited on pages 2 and 24.)

[BFG01] Robert Baumgartner, Sergio Flesca, and Georg Gottlob. The elog web extraction language. In *8th Int'l Conf. on Logic for Progr., Artificial Intelligence, and Reasoning LPAR'01*, volume 2250 of *LNCS*, pages 548–560. Springer-Verlag, 2001. (Cited on page 1.)

[BMSU86a] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proc 5th ACM SIGACT-SIGMOD Symp. on Principles of Database Systems PODS'86*, pages 1–15. ACM Press, 1986. (Cited on page 43.)

[BMSU86b] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proc. 5th Symp. on Principles of Database Systems PODS'86*, pages 1–15. ACM Press, 1986. (Cited on page 77.)

[CC77] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. 4th ACM Symp. POPL*, pages 238–252, 1977. (Cited on page 78.)

[CFH+91] Jiazhen Cai, Philippe Facon, Fritz Henglein, Robert Paige, and Edmond Schonberg. Type analysis and data structure selection. In Möller, editor, *Constructing Programs from Specifications*, pages 125–164. North-Holland, May 1991. (Cited on page 55.)

[CGT90] Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic Programming and Databases*. Springer, 1990. (Cited on page 77.)

[CP89] Jiazhen Cai and Robert Paige. Program derivation by fixed point computation. *Sci. Comput. Program.*, 11(3):197–261, 1989. (Cited on page 25.)

[DRW96] Steven Dawson, C. R. Ramakrishnan, and David S. Warren. Practical program analysis using general purpose logic programming systems—a case study. In *Proc. ACM SIGPLAN Conf. on Programming language design and implementation PLDI'96*, pages 117–126, New York, NY, USA, 1996. ACM. (Cited on page 78.)

[EHRS00] Javier Esparza, David Hansel, Peter Rossmanith, and Stefan Schwoon. Efficient algorithms for model checking pushdown systems. In *Proc. 12th Conf. on Comp. Aid. Verif. CAV'00*, volume 1855 of *LNCS*, pages 232–247. Springer-Verlag, 2000. (Cited on page 78.)

[FJT10a] Marco A. Feliú, Christophe Joubert, and Fernando Tarín. Efficient BES-based Bottom-Up Evaluation of Datalog Programs. In *In Proc. X Jornadas sobre Programación y Lenguajes PROLE 2010, pp. XX-YY, 2010.*, 2010. Revised version in *Electronic Notes in Theoretical Computer Science*, To appear 2011. (Cited on page 3.)

[FJT10b] Marco A. Feliú, Christophe Joubert, and Fernando Tarín. Evaluation Strategies for Datalog-based Points-To Analysis. In *In J. Bendisposto, M. Leuschel, M. Roggenbach (eds.), Proc. 10th International Workshop on Automated Verification of Critical Systems AVoCS 2010, pp. 88-103*, 2010. Technical Report of Düsseldorf University, 2010. (Cited on page 3.)

[FJT10c] Marco A. Feliú, Christophe Joubert, and Fernando Tarín. Evaluation Strategies for Datalog-based Points-To Analysis. In *In J. Bendisposto, M. Leuschel, M. Roggenbach (eds.), selected papers from the 10th International Workshop on Automated Verification of Critical Systems AVoCS 2010*, 2010. *Electronic Communications of the EASST:35*, 2010. (Cited on page 3.)

[FS86] Philippe Flajolet and Robert Sedgewick. Digital search trees revisited. *SIAM J. Comput.*, 15(3):748–767, 1986. (Cited on page 57.)

[GM78] Hervé Gallaire and Jack Minker, editors. *Logic and Data Bases*, Advances in Data Base Theory. Plemum Press, 1978. (Cited on page 1.)

[HJ94] Nevin Heintze and Joxan Jaffar. Set constraints and set-based analysis. In *2nd Int'l Work. on Principles and Practice of Constraint Programming PPCP'94*, volume 874 of *LNCS*, pages 281–298. Springer-Verlag, 1994. (Cited on page 78.)

[LS09] Yanhong A. Liu and Scott D. Stoller. From datalog rules to efficient programs with time and space guarantees. *ACM Trans. Program. Lang. Syst.*, 31(6), 2009. (Cited on pages 2, 4, 24, 30, 37, 60, 71 and 77.)

[LWL05] Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection analysis for Java. In Kwangkeun Yi, editor, *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems*, volume 3780 of *Lecture Notes in Computer Science*. Springer-Verlag, November 2005. (Cited on page 27.)

[Pai89] Robert Paige. Real-time simulation of a set machine on a ram. volume II, pages 69–73. Canadian Scholars Press, May 1989. (Cited on page 55.)

[Rei87] R. Reiter. *On closed world data bases*, pages 300–310. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987. (Cited on page 10.)

[Rep94] Thomas W. Reps. Solving Demand Versions of Interprocedural Analysis Problems. In *Proc. 5th Int'l Conf. on Compiler Construction CC'94*, volume 786, pages 389–403. Springer LNCS, 1994. (Cited on page 78.)

[Rep98] Thomas W. Reps. Program analysis via graph reachability. *Infor. & Soft. Tech.*, 40(11-12):701–726, 1998. (Cited on page 78.)

[SDSD86] Jacob T. Schwartz, Robert B. Dewar, Edmond Schonberg, and Edward Dubinsky. *Programming with sets; an introduction to SETL*. Springer-Verlag New York, Inc., New York, NY, USA, 1986. (Cited on page 25.)

[SF09] Manu Sridharan and Stephen J. Fink. The complexity of andersen's analysis in practice. In *SAS '09: Proceedings of the 16th International Symposium on Static Analysis*, pages 205–221, Berlin, Heidelberg, 2009. Springer-Verlag. (Cited on page 27.)

[SSD+05] Jon Sneyers, Tom Schrijvers, Bart Demoen, Jon Sneyers, Tom Schrijvers, Bart Demoen, Jon Sneyers, Tom Schrijvers, and Bart Demoen. — constraint and logic languages; d.3.4 processors — code generation, compilation, 2005. (Cited on page 24.)

[SSN+10] Seok-Won Seong, Jiwon Seo, Matthew Nasielski, Debangsu Sengupta, Sudheendra Hangal, Seng Keat Teh, Ruven Chu, Ben Dodson, and Monica S. Lam. Preserving privacy with prpl: a decentralized social networking infrastructure. In *Proc. 9th Int'l Symp. on Privacy Enhancing Technologies PETS'10*, volume 5672 of *LNCS*. Springer-Verlag, 2010. To appear. (Cited on page 1.)

[Ull89] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I and II, The New Technologies*. Computer Science Press, 1989. (Cited on pages 1 and 78.)

[WACL05] J. Whaley, D. Avots, M. Carbin, and M. S. Lam. Using Datalog with Binary Decision Diagrams for Program Analysis. In *Proc. Third Asian Symp. on Programming Languages and Systems APLAS'05*, volume 3780, pages 97–118. Springer LNCS, 2005. (Cited on page 35.)

Apéndice A

Marco A. Feliú, Christophe Joubert, and Fernando Tarín.
Evaluation Strategies for Datalog-based Points-To Analysis
In J. Bendisposto, M. Leuschel, M. Roggenbach (eds.), selected papers from the 10th International Workshop on Automated Verification of Critical Systems AVoCS 2010, *Electronic Communications of the EASST:35*, 2010.

Electronic Communications of the EASST
Volume 35 (2010)

Proceedings of the 10th International Workshop on Automated Verification of Critical Systems (AVoCS 2010)

Evaluation Strategies for Datalog-based Points-To Analysis

Marco-A. Feliú, Christophe Joubert, and Fernando Tarín

18 pages

Guest Editors: Jens Bendisposto, Michael Leuschel, Markus Roggenbach
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Evaluation Strategies for Datalog-based Points-To Analysis

Marco-A. Feliú*, Christophe Joubert, and Fernando Tarín†

mfeliu,joubert,ftarin@dsic.upv.es
Universidad Politécnica de Valencia, DSIC / ELP
Camino de Vera s/n, 46022 Valencia, Spain

Abstract: During the last decade, several hard problems have been described and solved in Datalog in a sound way (points-to analyses, data web management, security, privacy, and trust). In this work, we describe novel evaluation strategies for this language within the context of program analyses. We first decompose any Datalog program into a program where rules have at most two atoms in their body. Then, we show that a specialized bottom-up evaluation algorithm with time and memory guarantees can be described as the on-the-fly resolution of a Boolean Equation System (BES). The resolution computes all ground atoms in an efficient way thanks to a compact data structure with constant time access that has so far not been used in the Datalog or the BES literature. A prototype has been developed and tested on a number of real Java projects in the context of Andersen's points-to analysis. For this specific points-to analysis, experimental results show that our prototype is several orders of magnitude faster than state-of-the-art solvers like BDDBDDB or XSB, and an order of magnitude better that the novel strategy of Liu and Stoller [LS09] both in execution time and memory consumption.

Keywords: datalog; bottom-up evaluation; boolean equation system; program analysis

1 Introduction

The subset of Prolog that is called Datalog was first used in the late 70's [GM78]. It was further popularized in the late 80's by Ullman [Ull89] and Abiteboul [AHV95] with a direct application to database queries. At the time it was defined, Datalog was a very simple language that had a higher expressiveness than other standardized relational languages, such as Sql, which only supported recursive queries in its fourth revision (Sql:1999). Several extensions of the initial Datalog language have been made since then, namely the inclusion of negation, disjunctions in the head of the rules (DLV) [1], constraints, as well as new Datalog-based languages like Axlog [ABM09], Elog [BFG01], and SociaLite [SSN+10].

We have observed a resurgence of Datalog in different computer science communities over the last decade. Datalog has been used in a number of non-trivial analysis problems referenced

* This author is partly sponsored by the Spanish MEC FPU grant AP2008-00608.
† This work has been partially supported by the EU (FEDER), the Spanish MEC/MICINN under grants TIN 2007-68093-C02 and TIN 2010-21062-C02-02 and the Generalitat Valenciana under grant Emergentes GV/2009/024.
[1] http://www.dbai.tuwien.ac.at/proj/dlv/

in [LS09] such as pointer analysis, simplification of regular tree grammar-based constraints, trust management, path queries, model checking, security frameworks, queries of semi-structured data and social networking. Datalog is now popular in companies that currently sell solutions based on Datalog, such as LogicBlox, Semmle Ltd., Lixto, and Exeura[2].

In this paper, we are interested in the use of Datalog to detect critical errors in system software (such as memory leaks or Sql injections) by static analysis of conservative approximate pointer information. Static analysis techniques [ALSU07] like abstract interpretation are a powerful means for finding code errors before actually running the program. They have also been successfully used in conjunction with model checking, by making the program more abstract in order to save time and memory during its verification [GJMS07]. Among the most renowned static analyses are the *pointer analyses*, which compute relationships between program pointers and memory locations. These analyses appear in many program verification and optimization problems. In this paper, we will introduce novel evaluation strategies for solving Datalog programs and apply them to the problem of pointer analysis.

Since the advent of the first *naïve* algorithm to solve Datalog programs, numerous optimizations have appeared in the literature. BDDBDDB [WL04] is a recent tool that uses an implicit BDD-based representation to solve Datalog programs. It is an efficient solver in practice, yet its worst case complexity is linear with the cardinality of the cartesian product of the argument domains. Liu and Stoller have recently defined an evaluation strategy based on an explicit representation of Datalog programs that offers (lower) complexity guarantees [LS09]. The strategy is a generalization of the systematic algorithm development method of Paige et al. [PK82], which transforms extensive set computations like set union, intersection, and difference into incremental operations. Incremental operations are supported by sophisticated data structures with constant time access. An imperative resolution algorithm is derived, and it computes a fixed point over all (preformatted) rules by first considering input predicates, then considering rules with one subgoal, and finally considering rules with two subgoals.
We propose novel evaluation strategies based on the following:

1. A declarative description of Liu and Stoller's bottom-up resolution strategy that is separate from the fixed-point computation. This is achieved by transforming Datalog programs to *Boolean Equation Systems* (BESs) and evaluating the resulting BESs by standard solvers. BESs have been used successfully in the formal verification of asynchronous systems [And94a, VL92, Mad97]. The main features of this formalism are the following: it is concise (simple list of boolean equations); it relies on fixed-point operators; and there exist linear time and memory complexity algorithms to solve alternation-free BESs [Mat06].

2. A simplification of the resulting BES based on the dependency between predicate symbols for a given Datalog program. This predicate order graph allows us to remove various set operations during the construction of the BES.

3. A sophisticated data structure with worst-case constant access and compact representation of the underlying data. This efficient data structure is based on prefix tree structures, lists and bitmaps. This structure has faster look-up keys than binary search trees and imperfect hash tables commonly used in the Datalog literature.

[2] www.logicblox.com, www.semmle.com, www.lixto.com, www.exeura.com

The rest of the paper is organized as follows: Section 2 introduces both the syntax and semantics of the Datalog language considered in this paper. It also gives a running example, namely Andersen's points-to analysis, which illustrates our approach throughout the different sections. Section 3 introduces our BES approach, and Section 4 presents an algorithm that computes an evaluation order over the predicates. Section 5 details the data structure that efficiently supports the evaluation strategy. Section 6 compares the experimental results for the points-to analysis of real Java programs performed by different state-of-the-art Datalog solvers. Finally, Section 7 concludes and gives future directions of research.

2 Datalog Programs

Datalog has a simple and clear syntax and semantics. A Datalog program is composed of a finite set of declarative rules to both describe and query a deductive database.

Definition 1 (Syntax of Rules)

$$p_0(a_{0,1},\ldots,a_{0,n_0}) \ :- \ p_1(a_{1,1},\ldots,a_{1,n_1}),\ \ldots,\ p_m(a_{m,1},\ldots,a_{m,n_m}).$$

where each p_i is a predicate symbol of arity n_i with arguments $a_{i,j}$ ($j \in [1..n_i]$) that are either constant or variable.

The atom $p_0(a_{0,1},\ldots,a_{0,n_0})$ in the left-hand side of the rule is the rule's *head*. The finite conjunction of *subgoals*, also called *hypotheses*, in the right-hand side of the formula is the rule's *body*, *i.e.*, atoms that contain all variables appearing in the head. Following logic programming terminology, a rule with empty body ($m = 0$) is called a *fact*. The set of facts is called the *extensional database*, whereas the set of ground atoms *inferred* from rules is called the *intensional database*. For ease of exposition, we will consider Datalog programs with finite sets of rules and facts, and with rules that do not contain negated hypotheses, although the evaluation strategy would work for rules with stratified negation [LS09]. In the rest of the paper, we will follow the logic programming criteria where variables are described in capital letters and where predicate symbols and constants are described in lower-case letters.

Definition 2 (Fixed point semantics) [Ull89, AHV95] Let R be a Datalog program. The least *Herbrand model* of R is a Herbrand interpretation I of R defined as the least fixed point of a monotonic, continuous operator $T_R : \mathscr{I} \to \mathscr{I}$ known as the *immediate consequences operator* and defined by:

$$\begin{aligned} T_R(I) \quad = \quad & \{q \in B_R \mid \ q :- b_1,...,b_m \text{ is a ground instance of a rule in } R, \\ & \text{with } B_R \text{ the Herbrand base, } \ b_i \in I, i = 1..m, m \geq 0\} \end{aligned}$$

Following the approximation of [LS09], some auxiliary definitions are necessary: a variable that occurs multiple times in a hypothesis is called an *equal card*, and a variable that occurs only once and in only one hypothesis but not in the head of a rule is called a *wild card*. In the remainder of the paper, our derivation of the specialized Datalog evaluation algorithm will be applied to a restricted form of Datalog defined as follows: rules have at most two hypotheses; *equal cards* and *wild cards* can only occur in rules with one hypothesis; and facts must be ground.

There exists a decomposition from any non-restricted Datalog program into a restricted one. This decomposition does not affect the guaranteed worst-case time or space complexities of the original program evaluation. Grouping and reordering of predicate arguments do not affect the complexities, either. We will use this rule format to describe the bottom-up evaluation strategy in terms of BES in the next section.

Definition 3 (Restricted Datalog Rules) Rules and facts have exactly the following forms:

form 2: $\mathbf{q_2}(\mathbf{X_{1s}}, \mathbf{X_{2s}}, \mathbf{Y'_s}, \mathbf{c_{3s}}) : -\mathbf{h_1}(\mathbf{X_{1s}}, \mathbf{Y_s}, \mathbf{c_{1s}}), \mathbf{h_2}(\mathbf{X_{2s}}, \mathbf{Y_s}, \mathbf{c_{2s}})$.

form 1: $\mathbf{q_1}(\mathbf{Z'_s}, \mathbf{b_s}) : -\mathbf{h}(\mathbf{Z_s}, \mathbf{a_s})$.

form 0: $\mathbf{q}(\mathbf{c_s})$.

where each X_{1s},X_{2s},Y_s,Y'_s,Z_s, and Z'_s abbreviates a group of variables; each c_{1s}, c_{2s}, c_{3s}, a_s, b_s, and c_s abbreviates a group of constants; variables in Y'_s and Z'_s are subsets of the variables in Y_s and Z_s respectively.

Example 1 (The Datalog-based Andersen points-to analysis.) *In his thesis [And94b], Andersen gave a type inference system of a flow- and context-insensitive, inclusion-based pointer analysis for* C *programs. This analysis is based on four kinds of assignment statements that involve pointers (object allocations* **vp0***, variable assignments* **a** *and store* **s** *or load* **l** *operations into structures and fields of structures). They can be represented as four declarative rules as follows [ALSU07]:*

```
vp(X, Y) :- vp0(X, Y). (1)
vp(X, Y) :- a(X, Z), vp(Z, Y). (2)
hp(Y, S, T) :- s(X, S, Z), vp(X, Y), vp(Z, T). (3)
vp(Z, T) :- l(X, S, Z), vp(X, Y), hp(Y, S, T). (4)
```

where **vp0**, **a**, **s**, *and* **l** *are extensional predicates and* **vp** *(variables that point to heap locations),* **hp** *(heap locations that point to other heap locations through object fields) are intensional predicates.*
We can note that Datalog *rules (3) and (4) are not in one of the three forms accepted by our formal derivation since their bodies have three hypotheses.*

Rules with more than two hypotheses must be decomposed first into rules with two hypotheses. This can be achieved by repeatedly inserting auxiliary predicates in rules with more than two hypotheses in order to convert them into rules with at most two hypotheses. For a given rule with `h` hypotheses, there exist $(2h-3)!!$ ways of decomposing it by means of a simple algorithm [LS09], with:

$$n!! = \begin{cases} 1, & \text{if } n = 0 \text{ or } n = 1 \\ n \times (n-2)!!, & \text{if } n \geq 2 \end{cases}$$

Time complexity of a Datalog program is given by the sum of firings over all rules. The total number of times a rule is fired can be computed in terms of predicate size, domain size, argument size, and relative argument size. Given a Datalog program with k rules r_i ($i \in [1..k]$) with h_i hypotheses ($h_i > 2$), the time complexity of searching the optimal decomposition process can be bounded by $O(k.(h_{max}!!))$, with h_{max} being the maximum number of hypotheses in a rule of the program.

Assuming that all sets are of the same size, a key aspect in minimizing the evaluation cost is the process of looking up rules of `form 2` based on the set of common variables in the hypotheses. If a decomposition maximizes the size of that set for every rule, then fewer sets have to be checked in order to obtain new solutions. In addition to that, if those sets are in the same order as in the original program, no reordering, also called *views*, of the sets of variables belonging to a hypothesis would be needed to solve the problem. The concept of views will be expanded in section 3 and section 5.

Example 2 *Given the* Datalog*-based Andersen points-to analysis of Example 1, one decomposition that satisfies the strategy presented above, is described as follows:*

```
vp(X, Y) :- vp0(X, Y). (1)
vp(X, Y) :- a(X, Z), vp(Z, Y). (2)
temp1(Z, Y, S) :- s(X, S, Z), vp(X, Y). (3)
hp(Y, S, T) :- temp1(Z, Y, S), vp(Z, T). (4)
temp2(Y, S, Z) :- l(X, S, Z), vp(X, Y). (5)
vp(Z, T) :- temp2(Y, S, Z), hp(Y, S, T). (6)
```

where `temp1` *and* `temp2` *are two newly inserted auxiliary predicates.*
All rules are now either in `form 1` *or* `form 2` *formats. As an example, we have decomposed rule* **hp** *as follows:*

1. *Choose a pair of predicates in* **hp**. *In this case:* $\mathbf{s}(\mathbf{X},\mathbf{S},\mathbf{Z})$ *and* $\mathbf{vp}(\mathbf{X},\mathbf{Y})$.
2. *Create a new rule with the chosen pair as body and a new predicate as header, whose variables are those that are not repeated in both predicates. In this case:* $\mathbf{temp1}(\mathbf{Z},\mathbf{Y},\mathbf{S}) : -\mathbf{s}(\mathbf{X},\mathbf{S},\mathbf{Z}),\mathbf{vp}(\mathbf{X},\mathbf{Y})$.
3. *Substitute the chosen pair with the head of the new created rule. In this case:* $\mathbf{hp}(\mathbf{Y},\mathbf{S},\mathbf{T}) : -\mathbf{temp1}(\mathbf{Z},\mathbf{Y},\mathbf{S}),\mathbf{vp}(\mathbf{Z},\mathbf{T})$.
4. *Repeat the process from step 1 if the rule still has more than two hypotheses.*

Since the original rule **hp** *has only three hypotheses, the rule's decomposition terminates after one iteration. Only one new auxiliary predicate has been created and both new and modified rules have exactly two hypotheses in their body.*

The problem considered in this paper is to efficiently compute the least set of ground atoms that can be inferred using the rules. In the case of the points-to analysis, we want to derive all ground atoms for predicates **vp** and **hp**.

3 BES Approach

In this section, we describe the bottom-up evaluation strategy for any restricted Datalog program at a high level by using parameterized Boolean equation systems (PBESs). We illustrate this evaluation strategy on the example of Andersen's points-to analysis. The PBES is instantiated

into a parameterless BES that is later solved. Finally, we propose an algorithm to compute a predicate evaluation order for a given Datalog program and show its usefulness to optimize the BES resolution.

3.1 Parameterized Boolean Equation System (PBES)

In this paper, we will use a restricted fragment of the PBES formalism [GM98], namely, single equation block PBESs. We first recall the definition of parameterless BES to later introduce its parameterized extension.

Definition 4 (Single block BES) Given $\mathscr{W}$ a set of boolean variables, a *Boolean Equation System* (BES) $B = (W_0, M)$ is defined as follows: $W_0 \in \mathscr{W}$ is a boolean variable whose value is of interest in the context of the local resolution methodology; M is a set of n fixed point equations of the form $W_i \stackrel{\sigma}{=} \phi_i$, where all W_i are different ($i \in [0..n]$). $\sigma \in \{\mu, \nu\}$ is the least (μ) or greatest (ν) fixed point operator. Each W_i is a boolean variable from $\mathscr{W}$ whose value is the value of the respective *boolean formula* ϕ_i. A *boolean formula* ϕ, defined over an alphabet of boolean variables $\mathscr{W}$, is an expression built with the following syntax given in positive form: $\phi ::= \mathsf{true} \mid \mathsf{false} \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid W$ where boolean constants and operators have their usual semantics, ϕ_1 and ϕ_2 are boolean formulas, and W is a boolean variable. Empty conjunction (resp. disjunction) corresponds to boolean constant true (resp. false).

Definition 5 (Valuation of boolean variables and formulas) A valuation is a function v from a set of boolean variables $\mathscr{W}$ to $\{\mathsf{true}, \mathsf{false}\}$. It is extended over boolean formulas so that $v(\phi)$, being ϕ a boolean formula, is the value of the formula after substituting each free variable W in ϕ by $v(W)$.

Definition 6 (Semantics of a BES) The semantics of a BES B, denoted by $[\![B]\!]$, is the least (resp., greatest) fixed point valuation of the n boolean formulas ϕ_i of B.

There exist algorithms for solving single block BESs with a temporal and spatial complexity that is linear with respect to the total number of boolean variables and boolean operators of the BES [Mat06]. Thus, BESs allow us to simplify the definition of Datalog evaluation algorithms by only describing declarative aspects of the algorithm, and not operational aspects like fixed-point computation.

Definition 7 (Single block PBES) Single block PBESs are single block BESs where boolean variables have typed value parameters $D \subseteq \mathscr{D}$. Hence, boolean formulae have the following extended syntax given in positive form:

$$\phi, \phi_1, \phi_2 ::= \mathsf{true} \mid \mathsf{false} \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid X(e) \mid \forall d \in D.\ \phi \mid \exists d \in D.\ \phi$$

where e is a data term (constant or variable of type D), $X(e)$ denotes the *call* of a boolean variable X with parameter e, and d is a term of type D.

The main advantages of specific symbolic encodings such as Binary Decision Diagrams (BDDs) over explicit encodings such as PBESs in the context of program analysis, is that BDDs

can efficiently compact the program predicates. However, the compaction highly depends on the regularity of the points-to analysis domain, the redundancy of the relations, and the boolean variable ordering. These factors may vary the execution time of the analysis by several orders of magnitude as we observed with the Andersen's points-to analysis with standard BDD and data mining libraries. The Datalog relations can be represented compactly with well adapted data structures (prefix trees structures, lists and bitmaps) for any relation ordering in the Datalog program. Moreover, explicit encodings may facilitate the design of distributed solutions and the integration of further optimizations.

3.2 PBES Evaluation Strategy

The meaning of a restricted Datalog program corresponds to the fixed-point computation of a set of ground atoms over a given set of rules and facts. In this evaluation strategy, we are interested in all the ground atoms that can be inferred from the Datalog program. Hence, our PBES will be described in terms of a greatest fixed-point computation (ν). Rules of the program are in one of the three forms described in Section 2. As in [LS09], we would like to impose a bottom-up evaluation in which only inferred ground atoms of the Datalog program are generated. Therefore, the PBES should start the computation from boolean variable W_0 by generating boolean variables that hold the given facts. This will constitute the first boolean formula of our PBES description. We will represent each ground atom $\mathbf{p}(\mathbf{c_s})$., namely facts and inferred ground atoms, as a parameterized boolean variable $W_1(p : D_p,\ c_s : D_{c_s})$ identified by predicate symbol p defined over a domain D_p of predicates, and constant arguments c_s, defined over a domain $D_{c_s} = D_{a_{p,1}} \times \ldots \times D_{a_{p,p_n}}$, which is the composition of the argument domains. Then, new boolean variables $W_1(q,\ d_s)$, with q, a predicate symbol, and d_s, constant arguments, will be generated for each parameterized boolean variable $W_1(p : D_p,\ c_s : D_{c_s})$ and each rule where predicate p appears as a hypothesis. This will constitute the second boolean formula of our PBES description. Both boolean equations are defined as follows:

$$
\begin{aligned}
W_0 &\overset{\nu}{=} \bigwedge_{\mathbf{q}(\mathbf{c_s}). \in \mathit{Facts}} W_1(q,\ c_s) \\
W_1(p : D_p,\ c_s : D_{c_s}) &\overset{\nu}{=} \bigwedge_{p\ \mathit{of}\ \mathbf{h}(\mathbf{Z_s},\mathbf{a_s})} W_1(q_1,\ (Z'_s, b_s)) \quad (\star 1 \star) \\
&\quad \wedge \bigwedge_{p\ \mathit{of}\ \mathbf{h_1}(\mathbf{X_{1s}},\mathbf{Y_s},\mathbf{c_{1s}})} \ \bigwedge_{\mathbf{X_{2s}} \in \mathbf{h_2}(\mathbf{X_{2s}},\mathbf{Y_s},\mathbf{c_{2s}}).} W_1(q_2,\ (X_{1s}, X_{2s}, Y'_s, c_{3s})) \quad (\star 2 \star) \\
&\quad \wedge \bigwedge_{p\ \mathit{of}\ \mathbf{h_2}(\mathbf{X_{2s}},\mathbf{Y_s},\mathbf{c_{2s}})} \ \bigwedge_{\mathbf{X_{1s}} \in \mathbf{h_1}(\mathbf{X_{1s}},\mathbf{Y_s},\mathbf{c_{1s}}).} W_1(q_2,\ (X_{1s}, X_{2s}, Y'_s, c_{3s})) \quad (\star 3 \star)
\end{aligned}
$$

Boolean formulas on the right hand-side of both equations are conjunctions of distinct W_1 boolean variables. In the first equation, one variable W_1 is generated for each fact of the Datalog program. In the second equation, for each hypothesis $\mathbf{h}$ of the Datalog program that matches the predicate parameter p on the left hand-side of the equation, one variable W_1 is generated for each conclusion q that can be inferred by the given rule. Since there are three types of hypotheses in the Datalog programs, the second boolean equation is divided into three parts:

$(\star 1 \star)$ In rules of `form 1`, $\mathbf{q_1}(\mathbf{Z'_s}, \mathbf{b_s})\ :-\ \mathbf{h}(\mathbf{Z_s}, \mathbf{a_s})$., a new ground atom $\mathbf{q_1}(\mathbf{Z'_s}, \mathbf{b_s})$. is inferred when p appears as the hypothesis $\mathbf{h}$ of the rule, and arguments $(\mathbf{Z_s}, \mathbf{a_s})$ of $\mathbf{h}$ can be sub-

stituted by constant parameters c_s. For each new ground atom, a new boolean variable $W_1(q_1, (Z'_s, b_s))$ is generated.

(⋆2⋆) In rules of `form 2`, $\mathbf{q_2}(\mathbf{X_{1s}}, \mathbf{X_{2s}}, \mathbf{Y'_s}, \mathbf{c_{3s}}) :- \mathbf{h_1}(\mathbf{X_{1s}}, \mathbf{Y_s}, \mathbf{c_{1s}}), \mathbf{h_2}(\mathbf{X_{2s}}, \mathbf{Y_s}, \mathbf{c_{2s}})$., where p appears in the first hypothesis $\mathbf{h_1}$, arguments $(\mathbf{X_{1s}}, \mathbf{Y_s}, \mathbf{c_{1s}})$ of $\mathbf{h_1}$ are substituted by constant parameters c_s. Then, for each value of $\mathbf{X_{2s}}$ such that $\mathbf{h_2}(\mathbf{X_{2s}}, \mathbf{Y_s}, \mathbf{c_{2s}})$. is a previously computed ground atom, a new ground atom $\mathbf{q_2}(\mathbf{X_{1s}}, \mathbf{X_{2s}}, \mathbf{Y'_s}, \mathbf{c_{3s}})$. is inferred, and $W_1(q_2, (X_{1s}, X_{2s}, Y'_s, c_{3s}))$ is generated.

(⋆3⋆) If p appears in the second hypothesis $\mathbf{h_2}$ of a rule of `form 2`, new boolean variables $W_1(q_2, (X_{1s}, X_{2s}, Y'_s, c_{3s}))$ are generated symmetrically.

It can be observed that there is a one-to-one correspondence between boolean variables W_1 and solutions (ground atoms) inferred from the Datalog program. The solution of the greatest fixed-point computation over this PBES always gives that all boolean variables are true. Indeed, this PBES is actually a tautology whose only purpose is to incrementally compute all possible boolean variables of the PBES starting from boolean variable W_0. In order to obtain an efficient implementation of this PBES-based evaluation strategy, expensive set operations like $\mathbf{X_{is}} \in \mathbf{h_i}(\mathbf{X_{is}}, \mathbf{Y_s}, \mathbf{c_{is}})$ (set membership) must be replaced by constant time incremental computation based on auxiliary maps that will be described in Section 5. For instance, our evaluation strategy will construct an auxiliary map (also called view) $h_i(X_{is}, Y_s, c_{is})$ for every hypotheses pertaining to a rule of the `form 2`. The arguments of hypothesis $\mathbf{h_i}$ will be reordered if they are not strictly grouped with the following order: Y_s, X_{is}, c_{is}. Using this approach and a standard BES resolution algorithm, the temporal and spatial complexity of our evaluation strategy is linear with the number of ground atoms. In the rest of the section, we use a canonical form to specify the hypothesis, which is agnostic to any order related to the internal data structures, to ease the exposition of the transformation.

Example 3 (PBES evaluation of Andersen's points-to analysis) *For a given* Datalog *program, the* PBES *evaluation algorithm can be solved by instantiating the parameterized boolean variables over the given predicate domain [GM98].*

Figure 1 describe a simple example in Java*, from which a set of facts are extracted.*

Example a = new Example();	vp0(v_a, h_1).
Example b = new Example();	vp0(v_b, h_2).
b = a;	a(v_b, v_a).
a.x = b;	s(v_a, x, v_b).
c = a.x;	l(v_a, x, v_c).
(a)	(b)

Figure 1: (a) Example of Java program. (b) Corresponding input relations (facts).

In order to efficiently represent the domains, we represent data values as natural numbers. With the example above, we choose the following mapping between domains and values: v_a*=1,* v_b*=2,* v_c*=3* h_1*=0,* h_2*=1* *x=0*
The computed five facts are now written as follows:

```
vp0(1,0).  vp0(2,1).  a(2,1).  s(1,0,2).  l(1,0,3).
```

Given the Datalog*-based Andersen's points-to analysis in Example 2 and the facts above, we can instantiate the* PBES *by incrementally constructing a* BES *from variable* W_0 *[Mat98], which would give the following instantiated* BES*:*

$$
\begin{array}{lcl}
W_0 & \overset{\nu}{=} & W_{1,vp0(1,0)} \wedge W_{1,vp0(2,1)} \wedge W_{1,a(2,1)} \wedge W_{1,s(1,0,2)} \wedge W_{1,l(1,0,3)} \\
W_{1,vp0(1,0)} & \overset{\nu}{=} & W_{1,vp(1,0)} \\
W_{1,vp0(2,1)} & \overset{\nu}{=} & W_{1,vp(2,1)} \\
W_{1,a(2,1)} & \overset{\nu}{=} & \mathsf{true} \\
W_{1,s(1,0,2)} & \overset{\nu}{=} & \mathsf{true} \\
W_{1,l(1,0,3)} & \overset{\nu}{=} & \mathsf{true} \\
W_{1,vp(1,0)} & \overset{\nu}{=} & W_{1,vp(2,0)} \wedge W_{1,temp1(2,0,0)} \wedge W_{1,temp2(0,0,3)} \\
W_{1,vp(2,1)} & \overset{\nu}{=} & \mathsf{true} \\
W_{1,vp(2,0)} & \overset{\nu}{=} & \mathsf{true} \\
W_{1,temp1(2,0,0)} & \overset{\nu}{=} & W_{1,hp(0,0,0)} \wedge W_{1,hp(0,0,1)} \\
W_{1,temp2(0,0,3)} & \overset{\nu}{=} & \mathsf{true} \\
W_{1,hp(0,0,0)} & \overset{\nu}{=} & W_{1,vp(3,0)} \\
W_{1,hp(0,0,1)} & \overset{\nu}{=} & W_{1,vp(3,1)} \\
W_{1,vp(3,0)} & \overset{\nu}{=} & \mathsf{true} \\
W_{1,vp(3,1)} & \overset{\nu}{=} & \mathsf{true}
\end{array}
$$

It can be observed that parameters p *and* c_s *of variable* W_1 *in the* PBES *appear as a subscript in the* BES*, like* $W_{1,vp0(1,0)}$ *with the fact* $\mathbf{vp0}(\mathbf{1},\mathbf{0})$*.* W_0 *is now defined as the conjunction of parameterless boolean variables, one per fact in the* Datalog *program. Then, all boolean variables except* W_0 *are defined by the second equation of the* PBES*. For instance, variable* $W_{1,vp0(1,0)}$ *holds the fact* $\mathbf{vp0}(\mathbf{1},\mathbf{0})$*. Now,* $\mathbf{vp0}$ *is only used in one rule of the* Datalog *program, namely* $\mathbf{vp}(\mathbf{X},\mathbf{Y}) :- \mathbf{vp0}(\mathbf{X},\mathbf{Y})$.*, which is of* `form 1`*, so only the* $(*1*)$ *part of the second boolean equation in the* PBES *applies. From the* Datalog *rule, we can infer only one ground atom, namely* $\mathbf{vp}(\mathbf{1},\mathbf{0})$*. As a result, only one boolean variable* $W_{1,vp(1,0)}$ *is generated on the right hand-side of the equation defining* $W_{1,vp0(1,0)}$ *as shown above. Another example is the boolean variable* $W_{1,a(2,1)}$*. Predicate* $\mathbf{a}$ *is only used in one* Datalog *rule, namely* $\mathbf{vp}(\mathbf{X},\mathbf{Y}) :- \mathbf{a}(\mathbf{X},\mathbf{Z}), \mathbf{vp}(\mathbf{Z},\mathbf{Y})$.*, which is of* `form 2`*. Since* $\mathbf{a}$ *appears as the first hypothesis in this rule, so only the* $(*2*)$ *part of the boolean equation applies. Now, there does not exist any value* $\mathbf{Y}$ *such that* $\mathbf{vp}(\mathbf{1},\mathbf{Y})$*. is a previously computed ground atom. Indeed, only the ground atoms* $\mathbf{vp0}(\mathbf{1},\mathbf{0})$*. and* $\mathbf{vp0}(\mathbf{2},\mathbf{1})$*. have been computed so far. Therefore, variable* $W_{1,a(2,1)}$ *is defined in terms of an empty conjunction, which is by definition the constant* true*. Finally, we can comment on the equation defining variable* $W_{1,hp(0,0,0)}$*. Predicate* $\mathbf{hp}$ *only appears in one rule, namely* $\mathbf{vp}(\mathbf{X},\mathbf{Y}) :- \mathbf{temp2}(\mathbf{Y},\mathbf{S},\mathbf{Z}), \mathbf{hp}(\mathbf{Y},\mathbf{S},\mathbf{T})$.*, which is of* `form 2`*. Since it is the second hypothesis in the rule, only the* $(*3*)$ *part of the boolean equation applies. Now, there exists a value* $\mathbf{Z}$ *such that* $\mathbf{temp2}(\mathbf{0},\mathbf{0},\mathbf{Z})$*. is a previously computed ground atom. Indeed, the ground atom* $\mathbf{temp2}(\mathbf{0},\mathbf{0},\mathbf{3})$*. has been previously generated in the equation that defines variable* $W_{1,vp(1,0)}$*. Hence, the ground atom* $\mathbf{vp}(\mathbf{3},\mathbf{0})$*. can be inferred from the rule and a boolean variable*

$W_{1,vp(3,0)}$ *is generated on the right hand-side of the equation defining* $W_{1,hp(0,0,0)}$.

The least set of ground atoms that can be inferred using the rules of the Datalog*-based points-to analysis is the set of all computed boolean variables, minus* W_0. *In the case of pointer analysis, we are interested in the intentional database, namely the inferred ground atoms for* **vp** *and* **hp**. *These atoms are given by the computed boolean variables* $W_{1,vp(...)}$ *and* $W_{1,hp(...)}$, *from which we directly obtain the solutions* $\mathbf{vp}(\mathbf{1},\mathbf{0})$., $\mathbf{vp}(\mathbf{2},\mathbf{0})$., $\mathbf{vp}(\mathbf{3},\mathbf{0})$., $\mathbf{vp}(\mathbf{2},\mathbf{1})$., $\mathbf{vp}(\mathbf{3},\mathbf{1})$., $\mathbf{hp}(\mathbf{0},\mathbf{0},\mathbf{0})$., *and* $\mathbf{hp}(\mathbf{0},\mathbf{0},\mathbf{1})$. *Since there exists a mapping between the natural number values and the* Java *program's domains, we can describe the solutions in terms of the program's elements as follows:* $vp(v_a, h_1)$. $vp(v_b, h_1)$. $vp(v_c, h_1)$. $vp(v_b, h_2)$. $vp(v_c, h_2)$. $hp(v_a, x, v_a)$. *and* $hp(v_a, x, v_b)$.

4 Evaluation Order based on Predicate Dependency Graphs

Predicate order evaluation has a direct impact on performance results while not affecting time complexity of the evaluation problem. Depending on the predicate sorting, several operations can be discarded in the evaluation algorithm. In this section, we propose an algorithm to compute a predicate evaluation order for a given Datalog program and show its usefulness to optimize the BES resolution. The algorithm constructs a predicate evaluation order from a *predicate dependency graph* (PDG). A PDG is a directed graph that describes how predicates are dependent on each other. Each node is a predicate and each edge indicates that the start node appears as a hypothesis in a rule whose conclusion is the end node.

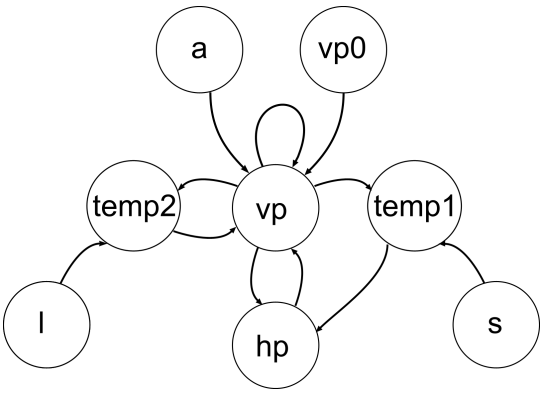

Figure 2: Predicate Dependency Graph for Andersen's Points-to Analysis

Example 4 *Given the decomposed* Datalog *program for Andersen's points-to analysis in Section 1, Figure 2 shows its corresponding* PDG. *In this example, it can be observed that only the extensional predicates, namely* **vp0**, **s**, **l**, *and* **a**, *are not part of a cycle.*

From a PDG, a topological order can be constructed based on the number of incident edges for each node as follows:

1. A node with no incident edges means that it does not appear as the conclusion of any rule in the program. Thus, it is preferable to start the evaluation of the program by propagating the ground atoms of these nodes;

2. Then, all successor nodes that are not part of a cycle, can be ordered and evaluated topologically; and

3. Finally, the remaining nodes that pertain to a cycle cannot be ordered topologically. In order to optimize the execution, we propose to order and evaluate them by decreasing

number of incident edges.

This algorithm forms three *levels* of predicate.

Example 5 *From the* PDG *in Figure 2, predicates* **vp0**, **s**, **l**, *and* **a** *will be evaluated first since the corresponding nodes do not have incident edges. The construction of a topological order over these predicates allows them to be evaluated in any order. All other predicates are part of a cycle. By decreasing the number of incident edges, we get the following predicate order:* **vp**, *and then in any order* **hp**, **temp2** *and* **temp1**.

In the PBES evaluation strategy described in Section 3.2, some expensive computations are performed, like $\mathbf{X_{2s}} \in \mathbf{h_2}(\mathbf{X_{2s}}, \mathbf{Y_s}, \mathbf{c_{2s}})$., which looks up previously computed ground atoms for predicate $\mathbf{h_2}$ with arguments $(\mathbf{X_{2s}}, \mathbf{Y_s}, \mathbf{c_{2s}})$. Another expensive operation is the update of the auxiliary data structure with new ground atoms. This operation does not explicitly appear in the PBES description. The data structure can be updated when a new ground atom is found, or when this ground atom is used as a hypothesis to compute other ground atoms. We will consider this approach later in this paper. By considering the predicate evaluation order described above, we can determine the smallest set of look-up and update operations that should be performed in order to obtain all solutions for a given Datalog program. For instance, if a predicate participates in a rule of `form 2` that generates a predicate that has a higher evaluation order, then the following hold:

- If the other hypothesis in the rule has a lower order or the same order and it has already been processed, then it is only necessary to look up the data structure to test the existence of ground atoms.
- If the other hypothesis in the rule has a higher order or the same order and it has not been processed yet, then it is only necessary to update the data structure by adding the new computed ground atom.

Example 6 *The predicate evaluation order can be used while instantiating the* PBES *described in Section 3.2 for Andersen's points-to analysis. Instead of choosing the facts in an unordered way, we can choose all facts that pertain to a predicate at the same time as indicated in the evaluation order. Then, the simplifications specified in the previous example can be made dynamically to prevent having to look up or update the data structure. For instance, given the fact* $\mathbf{a}(\mathbf{2},\mathbf{2})$., *the predicate* **a** *participates in a rule of* `form 2`, *namely rule* $\mathbf{vp}(\mathbf{X},\mathbf{Y}) :- \mathbf{a}(\mathbf{X},\mathbf{Z}), \mathbf{vp}(\mathbf{Z},\mathbf{Y})$.. *This rule generates a predicate, namely* **vp**, *from another hypothesis which is also* **vp**. *From the evaluation order,* **vp** *is evaluated after* **a**. *Thus, only an update of the data structure with the ground atom* $\mathbf{a}(\mathbf{2},\mathbf{2})$. *is necessary. Thanks to the proposed sorting algorithm, we know in advance that no look-up to previously computed ground atoms in the data structure is necessary. This means that the conjunction* $\bigwedge_{X_{2s} \in \mathbf{h_2}(\mathbf{X_{2s}},\mathbf{Y_s},\mathbf{c_{2s}})}$ *can be simply discarded for facts of predicate* **a**. *The resulting boolean formula that computes all the conclusions inferred from* $\mathbf{a}(\mathbf{X},\mathbf{Y})$ *can be reduced to* true. *The same applies to predicates* **s** *and* **l**, *which appear as hypotheses in rules of* `form 2`. *Hence, we could reduce the instantiated* BES *by the number of facts in* **a**, **s**, *and* **l**, *which would only generate* true *boolean variables. Note that any* BES *resolution algorithm*

could be used to solve our problem. However, in order to capture the preferred evaluation order, a specialized algorithm based on multiple queues is required. Such an algorithm could use one queue for all predicates or one queue per predicate level.

Even though they have less impact on the execution time, there are other kinds of optimizations that are based on the predicate evaluation order such as:

- removing useless rules from strongly connected components in the graph.
- preventing the addition of new ground atoms if there only exists a unique intensional predicate in each strongly connected component of the graph.

5 Efficient and Compact Auxiliary Data Structure

In this section, we describe a sophisticated data structure that has quicker access time and lower memory consumption than the dedicated structure in [LS09]. This efficient and compact data structure plays an essential role in the drastic improvements that can be observed in the experimental results of Section 6.

5.1 Data Structure Representation

We have defined a data structure that is extremely compact in memory while ensuring constant access to previously computed ground atoms and boolean variables. In our evaluation algorithm, we need to perform constant time access over two structures:

1. auxiliary mapping relations, to search all tuples (X_{is}, Y_s, c_{is}) that were previously computed given a set of constant arguments $(\mathbf{Y_s}, \mathbf{c_{is}})$ and a predicate $\mathbf{h_i}$, *i.e.*, operations $\mathbf{X_{is}} \in \mathbf{h_i}(\mathbf{X_{is}}, \mathbf{Y_s}, \mathbf{c_{is}})$ of our PBES evaluation strategy; and
2. result sets, to test if a new generated ground atom has already been computed before adding a new boolean variable.

The auxiliary mapping relations are represented by linked lists, one per node in the data structure representation. This allows us to access all existing ground atoms for a given predicate in a constant time. However, we can no longer use linked lists to retrieve one particular ground atom in a constant time. Instead, result sets are represented by adding bitmaps to the leaf nodes of the data structure representation. These bitmaps are based on digital trees to allow sparse domains. Note that both linked lists and bitmaps grow dynamically.

As a result, our data structure representation is based on the following: linked lists to represent sets; a representation of a `trie` (currently smart digital trees are used) to represent a layered indexed map; and bitmaps to test whether or not a ground atom has been previously computed. In the worst-case, associative access to elements of this data structure are done in O(1) time. Update and look-up operations to the structure are also done in O(1) time. If we relax the constraint about constant access, we could remove the bitmaps of the structure for certain specific problems. An ordered auxiliary mapping relation could be used in their place. In practice, this would save most of the memory used to solve the evaluation problem.

Example 7 *Figure 3 gives the final state of our data structure representation after all the possible ground atoms have been inferred with Andersen's points-to analysis and the five initial facts of Example 3. For example, the ground atom* $\mathbf{hp}(\mathbf{0},\mathbf{0},\mathbf{0})$. *is identified in the trie representation by*

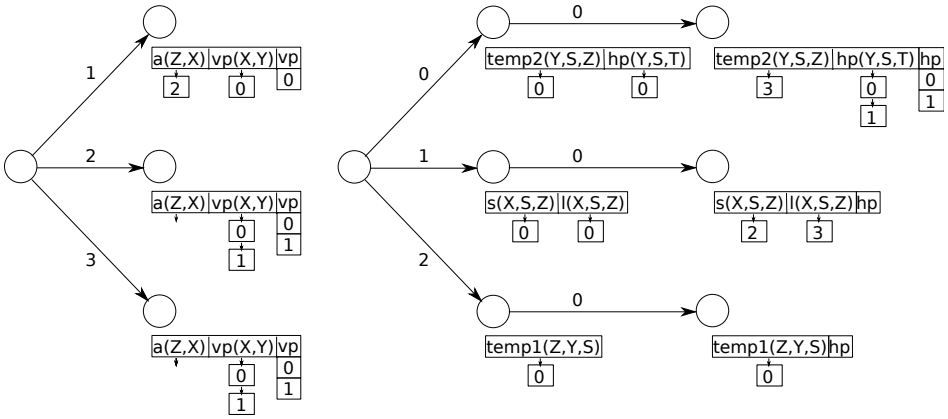

Figure 3: Data structure representation

its prefix, which corresponds to the path $0 \rightarrow 0$ *from root node. In the leaf node, a linked list indicates all values* $\mathbf{X}$ *such that* $\mathbf{hp}(\mathbf{0},\mathbf{0},\mathbf{X})$. *is a ground atom. Actually, there are only two values* (0, 1) *that have been computed for* **hp** *with the prefix* $(\mathbf{0},\mathbf{0})$. *To test whether or not* $\mathbf{hp}(\mathbf{0},\mathbf{0},\mathbf{0})$. *has been previously computed without iterating through the whole linked list, we use the bitmap for* **hp** *in the leaf node as well.*

5.2 Classical Auxiliary Structures

A classical approach for solving a Datalog program is to use `hash tables`. Access in hash tables is done in average, rather than worst-case, O(1) time. Moreover, hash tables have an overhead for computing the hash related functions for every operation. If all keys are known in advance, a perfect hash function can be used to avoid collisions. This enables having constant time look-ups in the worst case. Even if perfect hash tables could be used in our context, since keys are numbers (*i.e.*, no need to use a hash function) and sizes of domains are known a priori, our approach has two advantages. First, our structure is extremely compact, due to the fact that the information is stored by prefixes, which allows memory to be managed in a very efficient way; then, even if domain sizes are known, they are usually so large that they have to be managed dynamically. Therefore, perfect hash tables cannot be used in practice, and regular hash tables cannot assure constant access. Other solutions can be found in the literature, namely [LS02]. This describes a technique to design linked structures that support associative arrays in worst-case O(1) access time with little space overhead for a general class of set-based programs. To achieve this, it uses a representation of an arbitrary number of sets using a base. This approach guarantees a worst-case O(1) access time, but it cannot efficiently manage large sparse domains. Therefore, this structure does not scale for large instances of Datalog programs. Compared to the data structure detailed in [LS09], our structure does not need to manage several sets for one

domain since this information is propagated through the boolean variables. Moreover, our data structure representation does not have the limitation to work with tuples with more than two components. With the structure of [LS09], this would be very inefficient in terms of memory. Finally, in comparison with the usage of tries in XSB [RRS+99], our data structure does not make use of tries in the classical sense. XSB allows to use tries to represent Datalog's tabled subgoals and their answers, while we store dynamic lists and bitmaps to compute solutions to the original Datalog specification in an efficient way.

6 Experimental Results

In this section, we compare an implementation of our novel evaluation strategies with standard Datalog solvers by evaluating Andersen's points-to analysis on the DACAPO [3] benchmark.

Prototype We have extended the DATALOG_SOLVE [AFJV09] prototype, implemented in C, with the new evaluation strategy encoded as BES. Facts are extracted by SOOT [4] from the Java programs of the DACAPO benchmark with JDK 1.6. The extraction time varies between 115 seconds for the `lusearch` Java project and 315 seconds for the `fop` Java project. The complete benchmark is processed by SOOT in 39 minutes and 36 MB. of facts are generated. DATALOG_SOLVE first parses (150 lines of C code) the facts, then calls a BES solver (550 lines of C code) with an implicit description of the BES-based points-to analysis in terms of a successor function. This function is supported by our data structure representation that is connected to the Judy[5] library. Such an architecture brings the following advantages: constant access to check if a solution has already been computed and to browse sets for new solutions; compact domains; and lazy updating. In order to efficiently allocate the memory space that is strictly necessary for the computation, our system is connected to a specific memory pool and set values are put in contiguous memory positions in order to increase cache efficiency.

The new BES solver implements a very simple BES resolution algorithm based on a `queue` and our data structure representation. The `queue` enables the resolution graph associated to the Datalog program to be constructed on-the-fly, whereas our data structure representation is used to store and retrieve the required information to create new boolean variables. Only the boolean variables that are necessary to expand the successors are kept in the `queue`. The BES algorithm starts with boolean variable X_0 for which all successors are enumerated through the successor function provided by DATALOG_SOLVE. Each generated successor is checked in our data structure representation to see if it has already been computed. If it is a new solution, then it is added to the `queue`. Otherwise, it is ignored. Upon completion of the whole evaluation, the DATALOG_SOLVE tool extracts the inferred ground atoms from the computed boolean variables.We can remark that our simple BES solver is not specific to solve Datalog programs. It can be used for any problem that can be represented in terms of a BES with a unique block of conjunctive boolean formulas.

[3] http://voxel.dl.sourceforge.net/sourceforge/dacapobench/dacapo-2006-10-MR2-xdeps.zip
[4] http://www.sable.mcgill.ca/soot
[5] http://judy.sourceforge.net

Experiments We tested the efficiency and feasibility of our implementation by comparing it to three state-of-the-art Datalog solvers BDDBDDB[6], XSB 3.2[7] and the prototype of Liu and Stoller[8], which in the rest of the paper we will call TOPLAS. We also have implemented a pure Python version of our DATALOG_SOLVE prototype to compare it with the TOPLAS prototype (also written in pure Python). This version of the prototype has been developed using built-in types (dictionaries, lists and sets). Therefore, not all the techniques applied to our C prototype have been applied to the Python version. Nevertheless, this allows us to highlight the direct benefits of our new strategies. In Figure 4, performance results are presented in terms of evaluation user time (seconds) and memory consumption (MB.). All experiments were performed on an Intel Core 2 duo E4500 2.2 GHz (only one core used), with 2048 KB cache, 4 GB of RAM, and running Linux Ubuntu 10.04. DATALOG_SOLVE and XSB solvers were compiled using gcc 4.4.1. Python 2.6.4 was used for the TOPLAS solver. The measures do not include the time needed by XSB to precompile the facts. BDDBDDB is executed with the best variable ordering that we have found for the Andersen's analysis example, namely: V V H H F. XSB and BDDBDDB prototypes have been tested on both the original Andersen's analysis with 4 rules and the decomposed analysis with 6 rules. Since their execution time and memory consumption were not better with the decomposed analysis, only experimental results for the original analysis are presented in the figure. The analysis results were verified by comparing the outputs of all solvers.

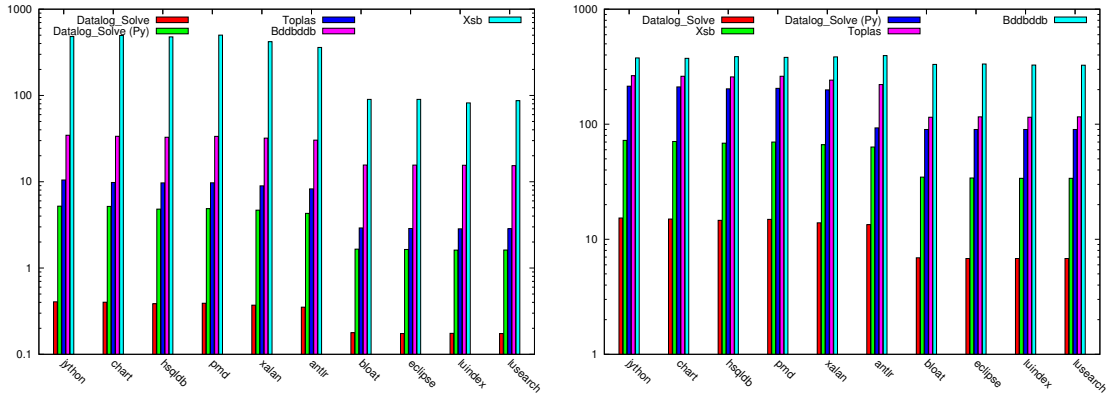

Figure 4: Analysis time (sec.) (left) and memory usage (MB.) (right) on the DACAPO benchmark

DATALOG_SOLVE evaluates the whole benchmark in only 3 seconds with a mean-time of 0.3 seconds per program. On the left part of Figure 4, where the y-axis (sec.) has a logarithmic scale, we can observe that DATALOG_SOLVE is an order of magnitude faster than TOPLAS, two orders faster than BDDBDDB, and three orders faster than XSB. For the *jython* example, XSB evaluated the points-to analysis in 482 seconds, BDDBDDB in 34 seconds, TOPLAS in 10 seconds, DATALOG_SOLVE (Python) in 5 seconds and DATALOG_SOLVE in 0.406 seconds. On the right part of Figure 4, where the y-axis (MB.) has a logarithmic scale, we can

[6] http://bddbddb.sourceforge.net
[7] http://xsb.sourceforge.net
[8] Provided by the authors of [LS09]

observe that DATALOG_SOLVE consumes five times less memory than XSB and an order of magnitude less memory than DATALOG_SOLVE (Python), TOPLAS and BDDBDDB. For the *lusearch* example, BDDBDDB required 326 MB. of memory to solve the analysis, TOPLAS 116 MB., DATALOG_SOLVE (Python) 90 MB., XSB 34 MB, and DATALOG_SOLVE 7 MB. It should be mentioned that the memory results of the DATALOG_SOLVE (Python) prototype can be highly improved avoiding some of the built-in types to build the data structure representation. These performance results show that our novel evaluation strategies scales really well for very large programs in the context of Andersen's points-to analysis.

7 Conclusions and Future Work

This paper presents new evaluation strategies to solve Datalog programs inspired by Liu and Stoller [LS09]. These strategies are based on: the separation of the evaluation strategy from the fixed point computation; an evaluation order over the predicates to avoid useless set operations; and a sophisticated data structure with worst-case constant access and compact representation of the underlying data. The first strategy is based on Boolean Equation Systems (BESs) to derive a specialized bottom-up evaluation algorithm with time and memory guarantees for any Datalog program. The algorithm can then be evaluated by independent optimized fixed-point solvers. The second strategy extracts a topological order over the predicates based on the dependencies of the Datalog program to simplify the BES. Finally, the third strategy exploits the features of prefix tree structures, together with linked lists and bitmaps to enable efficient accesses to inferred tuples whose values are sparse over the finite domains. The overall approach is faster and less memory-consuming, by some orders of magnitude, than state-of-the-art Datalog solvers. It was implemented in the DATALOG_SOLVE tool and was successfully tested on the Datalog-based Andersen points-to analysis of real Java projects.

An interesting future work would be to study the impact of our efficient data structure on more complex applications, such as flow- and context-sensitive pointer analyses, where more than 10^{14} contexts have to be stored and retrieved from memory during the evaluation of the problem. Recently, DOOP[9] became the new state-of-the-art framework for performing context-sensitive pointer analyses of Java programs. It is based on a commercial Datalog solver, called LogicBlox[10]. We would like to compare our prototype with DOOP/LogicBlox over such a set of complex points-to analyses. Other Datalog applications are also foreseen, like model checking and database benchmarks. We are also interested in the adequacy of the BES formalism to distribute the bottom-up Datalog resolution over interconnected machines. Several distributed BES resolution algorithms exist in the literature and have been applied with success for formal verification problems [JM06]. A promising alternative approach to explore would be to distribute the workload directly at the Datalog level by using Map-Reduce-based algorithms such as [AU10].

Acknowledgements We are grateful to Annie Liu and Scott Stoller for providing us with the generated Python code with guaranteed time and memory complexities for Andersen's points-to analysis example.

[9] http://doop.program-analysis.org
[10] http://www.logicblox.com

Bibliography

[ABM09] S. Abiteboul, P. Bourhis, B. Marinoiu. Efficient maintenance techniques for views over active documents. In *Proc. 12th Int'l Conf. on Extending Database Technology EDBT'09*. ACM Int'l Conf. Proc. Series 360, pp. 1076–1087. ACM Press, 2009.

[AFJV09] M. Alpuente, M. A. Feliú, C. Joubert, A. Villanueva. DATALOG_SOLVE: A Datalog-Based Demand-Driven Program Analyzer. *Electr. Notes Theor. Comput. Sci.* 248:57–66, 2009.

[AHV95] S. Abiteboul, R. Hull, V. Vianu. *Foundations of Databases*. Addison Wesley, 1995.

[ALSU07] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Publishing Co., Inc., Boston, MA, USA, 2007.

[And94a] H. R. Andersen. Model checking and boolean graphs. *Theo. Comp Sci* 126(1):3–30, 1994.

[And94b] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, 1994. DIKU report 94/19.

[AU10] F. Afrati, J. Ullman. Optimizing joins in a map-reduce environment. In *Proc. 13th Int'l Conf. on Extending Database Technology EDBT'10*. ACM Int'l Conf. Proc. Series 426, pp. 99–110. ACM, 2010.

[BFG01] R. Baumgartner, S. Flesca, G. Gottlob. The Elog Web Extraction Language. In *8th Int'l Conf. on Logic for Progr., Artificial Intelligence, and Reasoning LPAR'01*. LNCS 2250, pp. 548–560. Springer, 2001.

[GJMS07] M. Gallardo, C. Joubert, P. Merino, D. Sanán. C.OPEN and ANNOTATOR: Tools for On-the-Fly Model Checking C Programs. In *Proc. 14th Int'l Work. on Model Checking of Software SPIN'07*. LNCS 4595, pp. 268–273. Springer, 2007.

[GM78] H. Gallaire, J. Minker (eds.). *Logic and Data Bases*. Adv. in DB Theo. Plemum, 1978.

[GM98] J. F. Groote, R. Mateescu. Verification of Temporal Properties of Processes in a Setting with Data. In *Proc. 7th Int'l Conf. Algebraic Methodology and Software Technology AMAST'98*. LNCS 1548, pp. 74–90. Springer, 1998.

[JM06] C. Joubert, R. Mateescu. Distributed On-the-Fly Model Checking and Test Case Generation. In *Proc. 13th Int'l Workshop on Model Checking of Software SPIN'06*. LNCS 3925, pp. 126–145. Springer, 2006.

[LS02] Y. A. Liu, S. D. Stoller. Program optimization using indexed and recursive data structures. In *Proc. ACM SIGPLAN Work. on Partial Evaluation and Semantics-Based Program Manipulation PEPM'02*. SIGPLAN Notices 3, pp. 108–118. ACM, 2002.

[LS09] Y. A. Liu, S. D. Stoller. From datalog rules to efficient programs with time and space guarantees. *ACM Trans. Program. Lang. Syst.* 31(6), 2009.

[Mad97] A. Mader. *Verification of Modal Properties Using Boolean Equation Systems*. Bertz Verlag, 1997.

[Mat98] R. Mateescu. Local Model-Checking of an Alternation-Free Value-Based Modal Mu-Calculus. In *Proc. 2nd Int'l Workshop on Verication, Model Checking and Abstract Interpretation VMCAI'98*. 1998.

[Mat06] R. Mateescu. CAESAR_SOLVE: A Generic Library for On-the-Fly Resolution of Alternation-Free Boolean Equation Systems. *Springer Int'l Journal on Soft. Tools for Tech. Transfer (STTT)* 8:37–56, 2006.

[PK82] R. Paige, S. Koenig. Finite Differencing of Computable Expressions. *ACM Trans. Program. Lang. Syst.* 4(3):402–454, 1982.

[RRS+99] I. V. Ramakrishnan, P. Rao, K. F. Sagonas, T. Swift, D. S. Warren. Efficient Access Mechanisms for Tabled Logic Programs. *Journal of Logic Prog.* 38(1):31–54, 1999.

[SSN+10] S.-W. Seong, J. Seo, M. Nasielski, D. Sengupta, S. Hangal, S. K. Teh, R. Chu, B. Dodson, M. S. Lam. Preserving Privacy with PrPl: a Decentralized Social Networking Infrastructure. In *Proc. 9th Int'l Symp. on Privacy Enhancing Technologies PETS'10*. LNCS 6205. Springer, 2010.

[Ull89] J. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I and II, The New Technologies*. Computer Science Press, 1989.

[VL92] B. Vergauwen, J. Lewi. A linear algorithm for solving fixed-point equations on transition systems. In *Proc. 17th Colloquium on Trees in Algebra and Progr. CAAP'92*. LNCS 581, pp. 322–341. Springer, 1992.

[WL04] J. Whaley, M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation PLDI'04*. Pp. 131–144. ACM Press, 2004.