Análisis Estático de Programas para la Generación de Computación Ubicua: Inferencia de Tipos

Claudio Vaucheret
Departamento de Ciencias de la Computación
Universidad Nacional del Comhaue
vaucheret@gmail.com

Resumen

Este trabajo expone una linea de investigación en análisis estático de programas lógicos. El objetivo general de la computación ubicua es la generación automática de software en el contexto de recursos limitados. Un aspecto importante para este objetivo es lograr la información de los tipos en lenguajes no tipados como Prolog. Esta linea de investigación abarca el estudio de todas las consideraciones para una inferencia automática eficiente y precisa de la información de tipos en el contexto de la programación lógica.

1. Introducción

La tendencia actual hacia la creación de ambientes inteligentes (robótica, domótica, electrodomótica, etc.) supone la universalización de la capacidad de cómputo: la computación no se localiza sólo en los ordenadores, sino que puede atañer también a muchos artefactos de uso habitual en la vida diaria. La computación tiende a ser, cada vez más, ubicua [19]. Esta tendencia requiere un esfuerzo importante de adaptación del software, entendido a la manera clásica, para su incorporación a las nuevas unidades de computación: teléfonos móviles, agendas personales, coches, electrodomésticos conectados a la redes de comunicaciones, instrumentos portátiles que se confunden con la ropa u otros complementos (*wearable computers*). En general, y al menos con la tecnología actual, es de esperar que estos dispositivos no dispongan de toda la capacidad (velocidad, memoria, posibilidad de consumo y de calentamiento) que tienen los ordenadores hoy en día. Por tanto es necesario desarrollar *software* consciente de la limitación de los recursos y, en consecuencia, de su propia capacidad de consumo de los mismos.

El desafío de generar *software* con consciencia de los recursos es mayor cuando no se trata de su desarrollo desde cero (con el consiguiente coste adicional y retraso en su consecución),

sino de adaptar código ya existente, en el sentido de lograr una reutilización de componentes *software*. En ambos casos (desarrollo completo y adaptación) es necesario disponer de una serie de herramientas de manipulación del código potentes y efectivas, que ayuden a la creación de programas con características propicias para la ubicuidad y que auxilien en la adaptación de código ya existente, automatizando esta adaptación en lo posible. Una de las tareas a realizar incluyen experimentar los dominios y técnicas de análisis de tipos más adecuados para el lenguaje y la aplicación particular: el análisis de consumo de recursos e implementar el análisis de tipos resultante.

2. Interpretación Abstracta

La interpretación abstracta de programas [3] es una tecnología que permite el análisis estático de programas para obtener propiedades de la ejecución de los mismos sin ejecutar realmente los programas. La idea principal es simular la ejecución de los programas utilizando descripciones de los valores concretos, es decir un dominio abstracto de dichos valores, con el fin de capturar propiedades relevantes de la ejecución de los programas. Una simple analogía de este tipo de ejecución abstracta es la obtención del signo del resultado de una expresión algebraica aplicando la regla de los signos. En lugar de calcular la expresión con los valores concretos, aplicamos un cálculo abstracto sólo considerando los signos de dichos valores. La utilización de este dominio abstracto, en este caso, el de los signos, nos permite obtener propiedades del cálculo sin realizarlo realmente. Existen resultados teóricos que establecen las condiciones que deben tener estos dominios abstractos para garantizar la corrección y la terminación del análisis [3], basándose en una semántica formal de partida del lenguaje de programación a analizar.

Una ventaja de los lenguajes de programación lógica (en general, de los lenguajes declarativos) respecto a otros lenguajes más procedurales es la facilidad de definir semánticas formales sencillas basadas en conceptos puramente lógicos. Esto otorga muchas facilidades a la hora de realizar analizadores de programas basados en interpretación abstracta para programas lógicos [5]. La utilización de diferentes dominios abstractos permiten inferir de programas lógicos propiedades como el grado de instanciación de las variables, aliasing y dependencias entre las variables, tipos, etc.

3. Inferencia de Tipos

Dentro del conjunto de propiedades obtenidas por medio de interpretación abstracta de programas se encuentran los tipos, es decir los conjuntos de posibles valores para las variables de un programa.

Los sistemas de tipos en los lenguajes de programación proveen información que es beneficiosa para la eficiencia de los mismos por medio de la optimización de código y la especialización de programas. Y también para la localización de errores de programación en tiempo de compilación.

Existen dos enfoques distintos para los sistemas de tipos en los lenguajes de programación. El primero de ellos es el punto de vista *prescriptivo* originalmente introducido por Church [1] donde las expresiones del lenguaje sin un sentido definido son llamadas no bien tipadas y deben ser rechazadas por el compilador. El segundo enfoque considera los tipos como una diciplina para la clasificación de programas con una semántica definida en forma independiente del lenguaje de tipos. Este enfoque se llama *descriptivo* y es originalmente propuesto por Curry [6].

En programación lógica, este último enfoque asume la existencia de un universo de todos los objetos y los tipos son considerados como subconjuntos de este universo y se describen con diferentes lenguajes de tipos. Una ventaja de este enfoque es que no necesita declaración de tipos y de hecho los tipos pueden ser inferidos del programa. Sin embargo los tipos inferidos pueden no coincidir con el significado intentado del programa. En el tipo de vista prescriptivo no se presupone un universo de objetos, cada tipo diferente se asocia a diferentes dominios. La declaración de tipos forma una parte escencial del programa y se utilizan para determinar la semántica de los mismos. Una ventaja de este enfoque es que el sistema de tipos claramente especifica el significado intentado por el programador. Dentro de la programación lógica este enfoque es aplicado en los lenguajes λProlog y en el lenguaje Gödel .

Si bien originalmente Prolog es un lenguaje no tipado, se han propuesto sistemas de tipos para Prolog [8, 20] y se ha mostrado su utilización tanto para depuración de errores [13, 14] como para la optimización [11] y especialización de programas [15]. En los compiladores de Prolog basados en la máquina abstracta WAM cada cláusula es traducida en una secuencia de instrucciones WAM, con el fin de incrementar la eficiencia, este código debe ser especializado y optimizado durante la compilación. Cuanto más información se tenga sobre los posibles valores de las variables más optimizaciones pueden realizarse, por lo tanto la información sobre modos y tipos es crucial en este aspecto.

Tradicionalmente la información de tipos era una carga más para el programador dado que debía proveerla por medio de declaraciones. La aproximación más usual en lenguajes de programación no tipados como Prolog es sin embargo que los tipos o posibles valores de las variables sean inferidos automáticamente.

4. Objetivos

Para la concreción de los objetivos se necesitan abarcar diferentes tareas como las siguientes:

Implementación Dentro de los objetivos de la investigacion se encuentra el de comparar la precisión y eficiencia de los analizadores top-down para tipos. Para ello es necesario la implementación de un analizador de modo que sea comparable con las implementaciones existentes bottom-up y se le puedan incorporar las diferentes técnicas que se quieren comparar.

Operadores de Widening Los operadores de widening [4] son necesarios para garantizar la terminación del análisis en dominios infinitos como el dominio de tipos. En nuestro caso de estudio, es decir en los analizadores "top-down" es necesaria la aplicación de operadores de widening tanto en las sustituciones de éxito como en las sustituciones de llamadas. Estos operadores tienen gran influencia en la eficiencia y precisión del análisis de tipos, por lo tanto es uno de los temas que deben ser tratados en el trabajo. Un avance en esta línea de ivestigación ha sido presentado en [18].

Dominio de Tipos Existen varias maneras de representar los tipos. Uno de los dominios mas utilizados es el de los tipos regulares [7], debido a que permiten una gran expresividad, manteniendo la posibilidad de implementar eficientemente las operaciones. Los tipos regulares pueden representarse mediante gramáticas regulares deterministas. Las gramáticas deterministicas tienen menor poder expresivo que las no-deterministicas pues las primeras solo pueden expresar un conjunto de términos que sean *tuple-distributivos*, es decir que se pierde la dependencia entre las posiciones de los argumentos. La mayor expresividad de los tipos no-regulares produce resultados de análisis más precisos. Un objetivo de este trabajo es establecer si esa mayor precisión es costosa en términos de eficiencia.

Multivarianza Una diferencia que persiste entre los analizadores Bottom-up y Top-down es que los últimos permiten obtener información discriminada de diferentes variantes, o diferentes versiones de la llamada a predicados del programa, así como información en los distintos puntos de programa. El análisis se fundamenta en la extensión de la semántica operacional SLD en colecciones de semánticas las cuales capturan los patrones de llamadas a predicados y los patrones de éxito de los mismos. En esta investigación estudiaremos cómo esta característica influye en la precisión y eficiencia del análisis.

Combinación de dominios Además de ser importantes en la depuración y optimización de programas, la información de tipos puede ser utilizada para el propio análisis de programas. En [2] se ha mostrado la posibilidad de reutilizar previos analizadores y combinarlos para mejorar las prestaciones de dichos análisis. La información de tipos puede ser útil para aumentar la precisión de otros análisis. Un caso de estudio particular es el tratamiento de meta-llamadas en programas lógicos.

Referencias

[1] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[2] M. Codish, A. Mulkers, M. Bruynooghe, M. García de la Banda, and M. Hermenegildo. Improving Abstract Interpretations by Combining Domains. *ACM Transactions on Programming Languages and Systems*, 17(1):28–44, January 1995.

[3] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[4] P. Cousot and R. Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. Technical report, LIX, Ecole Polytechnique, France, 1991.

[5] P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming*, 13(2 and 3):103–179, July 1992.

[6] H. B. Curry and R. Feys. *Combinatory Logic*, volume 1. North-Holland, Amsterdam, 1958.

[7] P.W. Dart and J. Zobel. A Regular Type Language for Logic Programs. In *Types in Logic Programming*, pages 157–187. MIT Press, 1992.

[8] T. Früwirth, E. Shapiro, M.Y. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *Proc. LICS'91*, pages 300–309, 1991.

[9] J.P. Gallagher and D.A. de Waal. Fast and precise regular approximations of logic programs. In Pascal Van Hentenryck, editor, *Proc. of the 11th International Conference on Logic Programming*, pages 599–613. MIT Press, 1994.

[10] G. Janssens and M. Bruynooghe. Deriving Descriptions of Possible Values of Program Variables by means of Abstract Interpretation. *Journal of Logic Programming*, 13(2 and 3):205–258, July 1992.

[11] A. Marien, G. Janssens, A. Mulkers, and M. Bruynooghe. The Impact of Abstract Interpretation: an Experiment in Code Generation. In *Sixth International Conference on Logic Programming*, pages 33–47. MIT Press, June 1989.

[12] K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming*, pages 49–63. MIT Press, June 1991.

[13] G. Puebla, F. Bueno, and M. Hermenegildo. A Generic Preprocessor for Program Validation and Debugging. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 63–107. Springer-Verlag, September 2000.

[14] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, September 2000.

[15] G. Puebla and M. Hermenegildo. Abstract Multiple Specialization and its Application to Program Parallelization. *J. of Logic Programming. Special Issue on Synthesis, Transformation and Analysis of Logic Programs*, 41(2&3):279–316, November 1999.

[16] H. Saglam and J. Gallagher. Approximating constraint logic programs using polymorphic types and regular descriptions. Technical Report CSTR-95-17, Department of Computer Science, University of Bristol, Bristol BS8 1TR, 1995.

[17] P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Type analysis of prolog using type graphs. *Journal of Logic Programming*, 22(3):179–209, 1995.

[18] C. Vaucheret and F. Bueno. More precise yet efficient type inference for logic programs. In *International Static Analysis Symposium*, number 2477 in LNCS, pages 102–116. Springer-Verlag, September 2002.

[19] M. Weiser. The computer for the twenty-first century. *Scientific American*, 3(265):94–104, September 1991.

[20] E. Yardeni and E.Y. Shapiro. A type system for logic programs. *Journal of Logic Programming*, 10(2):125–154, 1990.