Grafos para fragmentación de programas sensible a los campos que mejoran al usarlos (trabajo en progreso)*

Carlos Galindo[0000−0002−3569−6218], Sergio Pérez[0000−0002−4384−7004], and Josep Silva[0000−0001−5096−0008]

Universitat Politècnica de València, Valencia, Spain
cargaji@vrain.upv.es, serperu@dsic.upv.es, jsilva@dsic.upv.es

Resumen El nivel de granularidad que proporciona el *Program Dependence Graph* (PDG) al representar estructuras de datos complejas (listas, arrays, objetos...) es insuficiente para diferenciar entre sus elementos al aplicar fragmentación de programas. En un trabajo anterior, se propuso un nuevo modelo, el *Constrained-Edges Program Dependence Graph* (CE-PDG), capaz de obtener fragmentos precisos al aplicar fragmentación a programas con estructuras de datos recursivas. En este trabajo, mostramos cómo los mecanismos utilizados por el CE-PDG pueden ser mejorados, proporcionando fragmentos notablemente más precisos de una manera más eficiente. Estas mejoras implican dos cambios distintos sobre el proceso de fragmentación: un nuevo criterio de parada para el algoritmo y una sustitución permanente en las restricciones del CE-PDG que mejora la eficiencia del recorrido cuantos más fragmentos se calculan sobre el grafo.

Keywords: análisis estático · fragmentación de programas · constrained-edges program dependence graph

1. Introducción

La fragmentación de programas [3] (en inglés *program slicing*) es una técnica de análisis de programas que, dado un punto de interés o *criterio de fragmentación*, extrae del programa el subconjunto de sus instrucciones, o *fragmento*, que pueden afectar al valor del criterio durante su ejecución. Su principal aplicación es la depuración de programas: seleccionando como criterio de fragmentación una variable que contenga un valor erróneo, podemos descartar aquellas instrucciones que no puedan haber afectado al cálculo de este, y por tanto, que no puedan contener el error.

* Este trabajo ha sido financiado parcialmente por la ayuda PID2019-104735RB-C41, financiada por el MCIN/AEI y por TAILOR, un proyecto financiado por el programa de investigación e innovación del Horizonte 2020 de la UE número 952215. Carlos Galindo ha sido financiado parcialmente por el Ministerio de Universidades bajo la ayuda FPU20/03861.

Por otro lado, el concepto de sensibilidad a los campos es una propiedad de aquellos análisis de software capaces de diferenciar entre los distintos subelementos de estructuras de datos complejas (como listas, arrays, objetos...). Por ejemplo, si consideramos `A = (1, 2); (B, C) = A` (un encapsulamiento y extracción de valores de una tupla en un lenguaje funcional), un análisis sensible al contexto sería capaz de determinar que `B = 1` y `C = 2`.

Varias son las propuestas que han intentado incorporar la sensibilidad a los campos en fragmentación de programas, siendo las más relevantes la atomización [2] y el más avanzado *Constrained-Edges Program Dependence Graph* (CE-PDG), que permite fragmentar de manera precisa programas con estructuras de datos recursivas [1]. En este trabajo, mostramos cómo el CE-PDG puede ser perfeccionado de distintas formas, aplicando nuevas técnicas que mejoran notablemente su eficiencia y precisión. En concreto, introducimos una nueva condición de parada para el análisis (Apartado 3) y un sistema para mejorar de forma continua la velocidad de los fragmentadores de programas (Apartado 4).

2. Contexto y antecedentes

Aunque los fragmentos de un programa se pueden calcular con varios métodos distintos, el más común es utilizar grafos que modelan las dependencias del programa. Entre ellos encontramos el *Program Dependence Graph* (PDG), que representa funciones aisladas, y el *System Dependence Graph* (SDG), que establece conexiones entre funciones a través de sus llamadas. Ambos son grafos dirigidos que representan cada instrucción del programa con un nodo y cada relación o dependencia entre ellas con arcos de distintos tipos. Una vez construidos los grafos, los fragmentos pueden calcularse recorriendo sus arcos con un coste lineal. Cuando un nodo es alcanzado durante el recorrido por primera vez, se incluye en el fragmento y se continúa con el recorrido. Por el contrario, si el nodo es alcanzado otra vez durante el recorrido, será ignorado. Finalmente, el recorrido termina cuando todos los nodos alcanzables desde el criterio son incluidos en el fragmento. Por tanto, el cálculo de un fragmento tiene un coste lineal $\mathcal{O}(N)$, siendo N el número de instrucciones del programa.

Ejemplo 1 (Extracción de fragmentos en el PDG y SDG). Una vez construido un PDG o SDG, como el que se muestra en la Figura 1a, los fragmentos se obtienen resolviendo un problema de alcanzabilidad sobre este. Por ejemplo, si elegimos como criterio el nodo que contiene la instrucción `(C, D) = A` (con el borde en negrita), se atravesarían todos los arcos hacia atrás, incluyendo los nodos `A = (1, 2)` y `Enter` en el fragmento.

En un trabajo previo [1], se describe cómo construir un PDG que incorpora la sensibilidad a los campos (CE-PDG), de tal manera que se pueda seleccionar de cada estructura de datos solo los elementos imprescindibles para producir un fragmento. La técnica se basa en etiquetar los arcos de datos con restricciones de forma que, al atravesarlos para calcular un fragmento, las restricciones se

acumulen en una pila y sean utilizadas para determinar los caminos que se pueden seguir durante el recorrido.

Hay cuatro restricciones distintas para los arcos. Cada una de ellas con un efecto diferente sobre la pila utilizada durante el recorrido. Se pueden observar sus efectos en la Tabla 1.

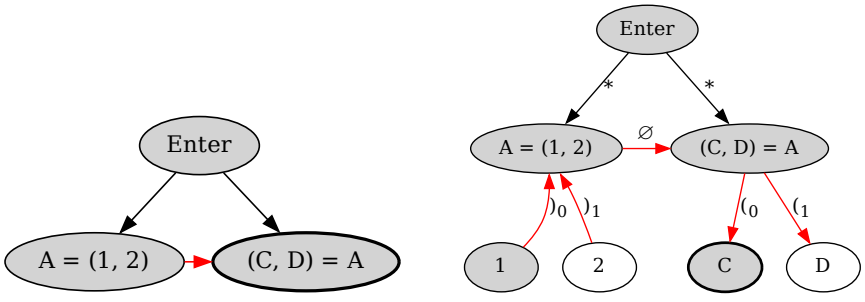
Tabla 1: Reglas de recorrido usando la pila de restricciones [1]. x e y son índices enteros. $\varnothing$ y $*$ son restricciones vacías y asterisco, respectivamente. S es una pila cualquiera y $\perp$ es la pila vacía.

	Pila	Restricción	Resultado
(1)	S	$\varnothing$	S
(2)	S	$*$	$\perp$
(3)	S	$(_x$	$S(_x$
(4)	$\perp$	$)_x$	$\perp$
(5)	$S(_x$	$)_x$	S
(6)	$S(_x$	$)_y$	*error*

(1) Las *restricciones vacías* ($\varnothing$) no tienen efecto sobre la pila.
(2) Las *restricciones asterisco* ($*$) vacían la pila, borrando el estado anterior.
(3) Las *restricciones de acceso de lectura* ($(_i$) se acumulan en la pila.

Las *restricciones de acceso de escritura* ($)_i$) son la clave de la metodología y su comportamiento esta condicionado por el contenido de la pila.

(4) Si la pila está vacía, las restricciones de escritura son ignoradas.
(5) Si la pila no está vacía y la restricción de escritura es "simétrica" con la que hay en la cima de la pila, se elimina la restricción de la cima de la pila y el recorrido continúa.



(a) PDG (no sensible a los campos) (b) CE-PDG (sensible a los campos)

Figura 1: Grafos que representan `A = (1, 2); (B, C) = A`. Los arcos negros representan dependencias de control y los arcos rojos, de datos.

(6) Si la pila no está vacía y la restricción de escritura no es "simétrica" a la de la cima de la pila, se producirá un error y el recorrido de ese arco se detiene.

Ejemplo 2 (Uso de la pila para hacer el recorrido sensible a los campos). En la Figura 1b podemos ver la versión sensible a los campos del PDG que hemos visto en el ejemplo anterior. En este caso, el recorrido es un poco más complejo, ya que debe almacenar el nodo actual y el estado de la pila, que se va modificando según las reglas de la Tabla 1. El recorrido completo se muestra a continuación, donde cada configuración $\langle n, s\rangle$ contiene el nodo alcanzado (n) y la pila resultante al alcanzarlo (s).

$$\langle \texttt{C}, \bot\rangle \xleftarrow{(_0} \langle \texttt{(C, D) = A}, (_0\rangle \tag{1}$$

$$\langle \texttt{(C, D) = A}, (_0\rangle \xleftarrow{*} \langle \textit{Enter}, \bot\rangle \tag{2}$$

$$\langle \texttt{(C, D) = A}, (_0\rangle \xleftarrow{\varnothing} \langle \texttt{A = (1, 2)}, (_0\rangle \tag{3}$$

$$\langle \texttt{A = (1, 2)}, (_0\rangle \xleftarrow{*} \langle \textit{Enter}, \bot\rangle \tag{4}$$

$$\langle \texttt{A = (1, 2)}, (_0\rangle \xleftarrow{)_0} \langle \texttt{1}, \bot\rangle \tag{5}$$

$$\langle \texttt{A = (1, 2)}, (_0\rangle \xleftarrow{)_1} \langle \texttt{2}, \textit{error}\rangle \tag{6}$$

Si empezamos desde el nodo `C` con la pila vacía (1), solo podemos atravesar un arco hacia `(C, D) = A`, siguiendo la regla 3 de la tabla. Una vez alcanzado, podemos atravesar dos arcos ($*$ y $\varnothing$): (2) el primero vacía la pila, alcanzando el nodo *Enter* (regla 2) y (3) el segundo alcanza el nodo `A = (1, 2)` sin afectar a la pila (regla 1). Desde este último nodo, se pueden atravesar tres arcos, etiquetados con $*$, $)_0$ y $)_1$: (5) el primero repite el nodo *Enter* con la pila vacía (regla 2); (4) el segundo alcanza el nodo `1` y la pila queda vacía después de cancelarse $(_0$ con $)_0$ (regla 5); y (6) el tercero no puede ser recorrido, ya que se produce un error por no coincidir la restricción del arco $)_1$ con $(_0$, la que hay en la cima de la pila.

Dado el ejemplo anterior, el recorrido del CE-PDG no es tan sencillo como el PDG tradicional. La condición de parada deja de ser "haber alcanzado un nodo ya visitado" y pasa a ser "haber alcanzado un nodo ya visitado *con la misma pila*". En caso de alcanzar un nodo dos veces con distintos estados, ambos deberían continuar.

La inclusión de la pila en el recorrido lo complica, dejando de ser un proceso lineal. Anteriormente, cada nodo se atravesaba una vez; en cambio, ahora cada nodo puede alcanzarse una vez por cada recorrido que llegue hasta él con una pila distinta, aumentando el espacio de búsqueda. Esto supone un problema cuando aparecen secuencias de arcos cíclicas en el PDG que contengan una o más restricciones de acceso de lectura (que siempre añaden un elemento a la pila), ya que esta situación puede provocar un recorrido infinito, donde la pila crece sin cesar. A estas secuencias las denominamos *bucles crecientes*.

Ejemplo 3 (Bucles crecientes). La secuencia de arcos $a \xleftarrow{(_1} b \xleftarrow{)_1} c \xleftarrow{(_0} a$ es un bucle creciente, ya que si empezamos con una pila cualquiera S, cada vez que

atravesemos los tres arcos añadiremos $(_0$ al tope de la pila, y como la secuencia termina en su nodo inicial, a, podríamos recorrerla un número infinito de veces.

Este problema tiene varias soluciones, desde las más sencillas como restringir el tamaño máximo de la pila (*k-limiting*) a las más complejas, como la propuesta en [1], que utiliza un autómata de pila para detectar bucles crecientes durante el recorrido y vacía la pila para garantizar que el recorrido termine.

Si consideramos de nuevo la secuencia del Ejemplo 3, la pila crecería indefinidamente, y el recorrido no terminaría nunca. *k-limiting* establece un tamaño máximo para la pila, vaciándola al llegar al límite. Una vez vacía, inevitablemente se repetirá el mismo estado de la pila en un mismo nodo, terminando el recorrido. Por otro lado, la propuesta en [1] utiliza un autómata que analiza la secuencia de restricciones que contiene el bucle (en este caso sería "$(_1,)_1, (_0$") y determina si es creciente. En este caso sí que lo es, y en ese momento se vacía la pila. Con respecto a *k-limiting*, el autómata proporciona la misma precisión, pero ahorra tiempo.

En el resto de este trabajo (Apartados 3 y 4) describiremos modificaciones adicionales al CE-PDG que solucionan el problema de una forma más eficiente y precisa, haciendo que el autómata se ejecute muchas menos veces, o incluso eliminando la existencia de estos bucles de dependencias del grafo.

3. Criterio de parada por historial de estados

La introducción de la pila en el recorrido del CE-PDG produce una ralentización de la fase de recorrido, ya que esta no termina hasta que se repita un estado (nodo y pila). La condición de considerar únicamente igualdad/repetición como criterio de parada es demasiado estricta, y desaprovecha la posibilidad de establecer una relación de orden entre pilas haciendo uso de las reglas de la Tabla 1.

Para empezar, veamos cómo el número de restricciones acumuladas en la pila al llegar a una estructura de datos compleja indica la profundidad a la que se encuentra el elemento que estamos buscando dentro de ella. Consideremos que alcanzamos una misma estructura de tuplas anidadas $(((1,2),(3,4)),((5,6),(7,8)))$ con cuatro pilas distintas: (a) pila vacía "$\perp$", (b) la pila "$(_0$", (c) la pila "$(_1$" y (d) la pila "$(_1, (_0$". En esta última, la cima de la pila es $(_0$.

(a) Si la pila esta vacía, estamos interesados en todos los elementos de la estructura.
(b) Si en su lugar la pila tiene una restricción "$(_0$", estaremos interesados en toda la tupla que hay en la posición 0: $((1,2),(3,4))$.
(c) Si por el contrario la pila tiene una restricción "$(_1$", estaremos interesados en toda la tupla que hay en la posición 1: $((5,6),(7,8))$.
(d) Por último, si la pila tiene dos restricciones "$(_1, (_0$", también estaremos interesados en el elemento de la posición 0, pero en este caso no en todo su contenido, sino solo en el elemento en la posición 1: $(3,4)$.

Viendo estas cuatro situaciones es fácil notar las siguientes relaciones entre los elementos seleccionados por cada escenario: $(b) \subseteq (a)$, $(c) \subseteq (a)$, $(d) \subseteq (a)$ y $(d) \subseteq (b)$. Los elementos de una estructura de datos que nos interesan cuando llegamos a ella con un número de restricciones mayor están también contenidos en aquellos que se incluyen al llegar a ella con menos restricciones siempre que coincidan las primeras restricciones (empezando por la cima de ambas pilas). Es decir, algunas pilas subsumen a otras. Tal es el caso de la pila (b), que subsume a la pila (d). Esto se debe a que cada restricción de lectura que hay en la pila debe cancelarse por una restricción de lectura, la cual limita los caminos posibles de acuerdo a la regla (6) de la Tabla 1.

Informalmente, una pila S subsume a otra S' si todos los elementos en la cima de S se encuentran en el mismo orden en la cima de S'. Esta relación puede formalizarse de la siguiente forma:

Definición 1 (Relación de orden entre pilas). *Dadas dos pilas S y S' podemos establecer la relación de orden $S' \subseteq S$ si y solo si S es un sufijo de S'.*

Esta relación es reflexiva ($S \subseteq S$), antisimétrica ($S_1 \subseteq S_2 \wedge S_2 \subseteq S_1 \iff S_1 = S_2$) y transitiva ($S_1 \subseteq S_2 \wedge S_2 \subseteq S_3 \implies S_1 \subseteq S_3$).

Utilizando esta relación de orden, podemos establecer una condición de parada más compleja: dado un nodo n, un conjunto de pilas S_n con las que se ha alcanzado previamente n y con una pila S con la que n se alcanza nuevamente, si $S \subseteq S_i$ para cualquier $S_i \in S_n$ el recorrido termina.

Esta mejora disminuye el tiempo necesario para recorrer el grafo y mejora la precisión del fragmento obtenido. Esto se debe a que evita muchos de los bucles que antes se trataban con el autómata. Además, en algunos casos permite atravesar un bucle sin vaciar la pila, mejorando así la precisión.

4. Mejora continua del CE-PDG durante el recorrido

Pese a reducir el número de veces que se ejecuta el autómata para detectar los bucles crecientes tras aplicar la mejora del Apartado 3, este sigue siendo necesario para tratar escenarios donde estos bucles aparecen durante el recorrido. Al final del Apartado 2 se ha explicado que el modelo presentado en [1] ejecuta un autómata sobre una secuencia de arcos que forma un bucle, analizando si es creciente y vaciando la pila durante el recorrido.

El autómata descrito en [1] recibe la secuencia de restricciones en los arcos del bucle y las analiza dividiéndolas en restricciones de acceso de lectura y escritura mediante el uso de dos pilas de estado. La mejora que proponemos en este apartado hace uso de las pilas de estado del autómata y, mediante un proceso de emparejamiento de restricciones, muestra cómo es posible detectar los arcos de lectura que hacen que el bucle sea creciente, i.e., aquellos arcos que añaden una restricción de lectura en la pila que no se complementa con ninguna restricción de escritura durante la misma iteración del bucle.

La Figura 2 muestra cómo las restricciones de acceso de lectura y escritura en la secuencia de restricciones asociada a un bucle creciente se emparejan fácilmente dejando libres aquellas restricciones (y sus correspondientes arcos) que

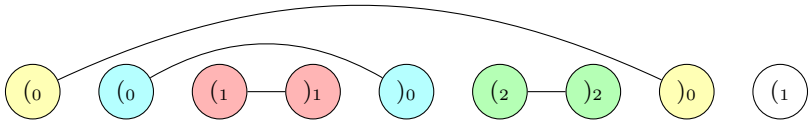


Figura 2: Una secuencia de restricciones "$(_0(_0(_1)_1)_0(_2)_2)_0(_1$" que forma un bucle creciente, quedando emparejadas aquellas que se cancelan mutuamente.

añaden a la pila restricciones de lectura que nunca se resuelven dentro del bucle. Una vez identificados los arcos que suponen un peligro para el recorrido del grafo, podemos reemplazar permanentemente las restricciones de estos por $*$. En la práctica, esto rompe el bucle de manera equivalente al funcionamiento original del autómata explicado en el Apartado 2, sin embargo, con esta modificación el bucle se elimina de forma permanente. Por tanto, el siguiente recorrido que pase por los mismos arcos ya no lanzará el autómata, porque esos arcos no formarán un bucle creciente.

En esta propuesta, el autómata solo analiza cada bucle en el grafo una vez, y, además, el recorrido se hace más eficiente cuantos más fragmentos se calculen sobre un mismo grafo. Dados suficientes fragmentos calculados, llegará un momento en que no habrá ningún bucle creciente en el grafo, y el autómata dejará de usarse, aumentando la eficiencia del recorrido.

5. Conclusiones y trabajo futuro

La fragmentación de programas sensible a los campos permite diferenciar los elementos de estructuras de datos complejas. Este trabajo plantea dos mejoras aplicadas al modelo definido en [1] que mejoran el coste temporal de la fase de recorrido y producen fragmentos de programa más precisos utilizando el CE-PDG como base.

Por un lado, mejoramos la condición de parada del recorrido, estableciendo una relación de orden entre los estados de recorrido y reduciendo el coste temporal de éste. Por otro lado, utilizamos la información extraída por un autómata de pila para eliminar paulatinamente los bucles del CE-PDG, acelerando todos los futuros recorridos. Esta variación del CE-PDG mejora con cada recorrido que se ejecuta sobre él, hasta que no queden más bucles en él y la detección de bucles resulte irrelevante.

Referencias

1. Galindo, C., Krinke, J., Pérez, S., Silva, J.: Field-sensitive program slicing. In: Software Engineering and Formal Methods: 20th International Conference, SEFM 2022, Berlin, Germany, September 26–30, 2022, Proceedings. vol. 13550, pp. 74–90. Springer Nature (2022)

,

2. Ramalingam, G., Field, J., Tip, F.: Aggregate structure identification and its application to program analysis. In: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 119–132. POPL '99, Association for Computing Machinery, New York, NY, USA (1999). https://doi.org/10.1145/292540.292553, https://doi.org/10.1145/292540.292553
3. Silva, J.: A vocabulary of program slicing-based techniques. ACM Computing Surveys **44**(3) (June 2012)