Instituto Tecnológico de Costa Rica.

Área Académica de Ingeniería en Computadores,

Lenguajes, Compiladores e Intérpretes.

Investigación:

“Una representación intermedia para integrar análisis de ingeniería inversa.”

Autores:

Acuña José Daniel : 2018145020,

Esquivel Sanchez Jonathan : 2018167983,

Ortiz Vega Angelo : 2017239551,

Solís Ávila Iván : 2018209698.

Responsable Técnico:

Marco Hernández

Cartago, Costa Rica.

2019

CONTENIDO

INTRODUCCIÓN **2**

OBJETIVOS **3**

REQUERIMIENTOS DE IR PARA INGENIERÍA INVERSA **4**

REPRESENTACIONES INTERMEDIAS PROPUESTAS (IR) **6**

REPRESENTACIÓN INTERMEDIA INTEGRADA IIR **8**

GAST con GSA 8

ERG con vistas 10

DETALLES TÉCNICOS **12**

Representación ERG 12

Implementación de vistas 12

Conjuntos 13

Vectores de bits 13

Información de vista integrada 13

EJEMPLO DE APLICACIÓN **14**

Horseshoe 14

Nivel de estructura de código 14

Niveles de funciones y arquitectura 14

Abstracción extensible 15

Recuperación de ADT utilizando IIR 15

CONCLUSIONES **16**

BIBLOGRAFÍA **16**

RESUMEN

Las representaciones intermedias (IR) son un tema clave tanto tanto para compiladores como para herramientas de ingeniería inversa para permitir análisis eficientes. Investigación en el campo de los compiladores ha propuesto muchos IR sofisticados que pueden usarse en el dominio de la ingeniería inversa, especialmente en el caso de análisis profundos, pero la ingeniería inversa también tiene su propio requisitos para representaciones intermedias no cubiertas por tecnología de compilación tradicional. Este artículo discute requisitos de los IR para ingeniería inversa. Muestra entonces cómo se pueden cumplir la mayoría de estos requisitos extendiendo e integrando los IR existentes. Estas extensiones incluyen un AST generalizado y un mecanismo que admite múltiples puntos de vista sobre los programas. Además, el documento muestra cómo estos las vistas se pueden implementar de manera eficiente.

INTRODUCCIÓN

Para iniciar con la definición de Ingeniería Inversa, es necesario remontarnos a términos más básicos, se sugiere iniciar con la pregunta: ¿Alguna vez ha estado jugando con un juguete mecánico y lo ha desmontado y vuelto a montar para ver cómo funcionaba? Pues esa es más o menos la base de la ingeniería inversa. Este proceso "hacia atrás" se sigue usando en muchas ocasiones de manera legal para crear nuevos sistemas o mejorar los ya existentes.

Los orígenes de este proceso se remontan a la Segunda Guerra Mundial, cuando la tecnología armamentística empezó a ser tan importante que podía hacer que la balanza de la victoria se decantara por unos países o por otros. Tanto el bando de los Aliados como las Potencias del Eje se dedicaban a capturar aviones, máquinas y armas del enemigo para estudiarlas y buscar puntos débiles de su tecnología, para conseguir una ventaja estratégica frente al bando contrario.

De esta época, en la que la ingeniería inversa servía para analizar hardware, pasamos a la actualidad en la que estamos más centrados en el uso del software. Actualmente la ingeniería

inversa tiene muchas aplicaciones: desde analizar la tecnología de la competencia, para mejorar la nuestra o saber si las otras empresas infringen alguna de nuestras patentes; pasando por desarrollar productos compatibles con otros sistemas de los que no tenemos los detalles técnicos; hasta comprobar que un programa informático no cuenta con ninguna brecha de seguridad.

OBJETIVOS

- Comprender los requerimientos intermedios para poder aplicar ingeniería inversa en los problemas relacionados a compiladores, con soluciones simples y eficientes.
- Determinar la importancia de la abstracción al momento de optimizar las Representaciones Intermedias.
- Explicar cómo se pueden integrar representaciones intermedias más simples en representaciones intermedia más complejas.
- Analizar la utilidad de la ingeniería inversa en el ámbito de trabajo de un ingeniero de software, principalmente en el uso de representaciones intermedias en los compiladores.

REQUERIMIENTOS DE IR PARA INGENIERÍA INVERSA

Una representación intermedia debería soportar desde las tareas más detalladas hasta las más generales. En general, una representación intermedia para ingeniería inversa que soporta análisis profundo debería tener el mismo análisis de profundidad que la representación intermedia de los compiladores tradicionales pero también debería tener un análisis superficial para mayor abstracción. Los requerimientos de las representaciones intermedias son muy parecidas a las tareas de ingeniería inversa. Estos requerimientos pueden dividirse en los que comparten las representaciones intermedias con la ingeniería inversa y los requerimientos compartidos para representaciones intermedias con compiladores.

El análisis general debería incluir seis objetivos comunes: Debe ser independiente de un lenguaje de programación, en otras palabras no debería tener una sintaxis específica de un lenguaje y su análisis se podría aplicar a diferentes lenguajes de la misma familia. La semántica de la representación intermedia tiene que estar bien definida, esto es necesario para un análisis más exacto.

Su recorrido debe ser eficiente ya que el análisis implica recorrer la representación intermedia al menos una vez, pero puede que se tenga que recorrer varias veces como es el caso de los algoritmos iterativos. La representación intermedia debe ser construida eficientemente para realizar un análisis de sistemas más grandes en un tiempo razonable. Otro objetivo es que tenga un tamaño proporcional al tamaño del código fuente. El último objetivo general es que la representación intermedia debería tener un control eficiente y un flujo de datos para el análisis.

Seguidamente veremos los objetivos específicos de la ingeniería inversa, estos incluyen: la representación intermedia debería preservar un mapeo al código fuente, esto permite un análisis que le brinda retroalimentación al usuario en términos del programa original. El siguiente requerimiento es que la representación intermedia debería poder representar un sistema hecho de varios programas, por ejemplo debería poder distinguir entre dos instancias de la misma variable, con el mismo nombre y en el mismo archivo. Otra requerimiento específico de la ingeniería inversa la representación intermedia debería soportar diferentes niveles de

“granularidad”, ya que debería ser capaz de diferenciar los datos y procesos más importantes de los menos importantes.

Al ser utilizado en un ambiente interactivo de ingeniería inversa, la representación intermedia debería tomar información proporcionada por el usuario además de datos derivables directamente de el código fuente, esta información se guarda en anotaciones, atributos o aserciones. Otro objetivo es que se debería poder guardar la representación intermedia ya que a diferencia de los compiladores la ingeniería inversa involucra intervenciones del usuario y análisis computacional más costosos. En los ambientes de ingeniería inversa las representaciones intermedias deberían capturar un alto nivel de abstracciones. Las representaciones intermedias también deberían proporcionar los medios para expresar cualquier relación entre los conceptos, un ejemplo de esto sería una relación de comunicación entre dos subsistemas en una descripción arquitectónica, la representación intermedia debería ser capaz de relacionar estos atributos. Por último, en un ambiente de ingeniería inversa que incluye a varios usuarios, la representación intermedia debería poder capturar los resultados de los análisis de estos distintos usuarios y representarlos como diferentes perspectivas del sistema.

Los requerimientos de eficiencia de la construcción y el recorrido son muy importantes para los compiladores y aún más para la ingeniería inversa, ya que este tiene que analizar todo un sistema en lugar de módulos individuales. Esto implica que la escalabilidad (capacidad de adaptarse) es uno de los problemas más grandes. La representación intermedia debería ser capaz de analizar códigos fuente de diferentes tipos de lenguajes y debe ser lo suficientemente general y amplia para cubrir esos programas. Si no se cumple con lo anterior, eso implicaría análisis más específicos para cada lenguaje y esto puede llevar a diversos problemas más adelante, como problemas de mantenimiento de la representación intermedia.

REPRESENTACIONES INTERMEDIAS PROPUESTAS (IR)

Las representaciones intermedias que se van a discutir son los Árboles Sintácticos Abstractos, Grafos de Relaciones de Entidades y Formulario de Asignaciones Individuales Cerrado.

Árbol Sintáctico Abstracto (AST)

En el AST los nodos son constructos del lenguaje de programación y los bordes representan la jerarquía entre estos constructos. El Árbol es una representación más simple de la gramática del lenguaje. Si es un árbol de atributos puede contener el nivel jerárquico, información de bordes de más adelante como un identificador que apunta a su declaración.

Cualquier borde que define la estructura del AST se le llama borde sintáctico, los bordes extra son considerados bordes semánticos, estos extienden el AST de un árbol a un posible grafo cíclico, aumentando la versatilidad y complejidad del lenguaje.

La estructura del árbol sintáctico de un lenguaje lo define, pero estos árboles pueden ser generalizados para representar a diversos lenguajes. Los ASTs generados por diversas unidades de compilación se pueden unir para hacer un análisis general del sistema compuesto por diversos programas y mantener separado únicamente los programas principales y variables globales.

Debido a que los ASTs son una representacion del codigo fuente solo permiten ver un nivel de complejidad, no son buenos para representar conceptos más abstractos o diferentes vistas a las relaciones.

Formulario de Asignaciones Individuales Cerrado (GSA)

Los GSA (Gate Single Assignment) están basados en los SSA (Static Single Assignment) que fueron creados para acelerar el análisis de referencias a variables y sus declaraciones que las puedes alcanzar. Esto se logra mediante funciones que combinan múltiples definiciones que

convergen donde diferentes flujos de control mergen. GSA extienden esta función de los SSA utilizando más de una funciones de control de acuerdo al flujo de control.

Los GSA no son formas de Representación Intermedia estrictamente, sino más como una extensión de una IR para representar definiciones y usos. Como los grafos de control de flujo son extendidos usando SSA. Otro uso es en los ASTs, mientras un AST puro solo permite recorrerlo de arriba a abajo, un AST con GSA permite moverse de una definición a usos y viceversa, esto aumenta la eficiencia del flujo de análisis de datos.

En espacio requerido para construir un GSA también es eficiente porque es casi lineal, en ciclos anidados la complejidad es de O(n^2), pero en programas típicos las medidas son lineales al tamaño del programa.

Grafos de Relaciones de Entidades (ERG)

Si se quiere una representación más amplia del sistema se puede usar un ERG en lugar de ASTs y GSA. Las entidades de un ERG son conceptos del lenguaje de programación, como: funciones, tipos, variables y conceptos más abstractos como componentes, tipos de datos abstractos o subsistemas. Las entidades se representan como nodos y sus relaciones como bordes, algunos ejemplos de relaciones entre entidades pueden ser llamadas a funciones o algo más abstracto como la comunicación entre un cliente servidor.

Una aplicación de un ERG es en los Grafos de Flujo de Recursos (RFG) que consiste de funciones, tipos y variables. Los RFG son usados para detectar subsistemas y tipos de datos abstractos. El fin de los ERG es describir la arquitectura de un sistema.

REPRESENTACIÓN INTERMEDIA INTEGRADA IIR

La representación intermedia propuesta (IIR) es una integración de representaciones intermedias existentes las cuales consisten en dos niveles: el nivel inferior es un Árbol Abstracto Generalizado de Sintaxis (GAST) extendido con GSA para permitir el análisis de flujo de datos. El nivel más alto es un ERG que permite análisis "granulares", además soporta conceptos de alto nivel.

GAST con GSA

El GAST está diseñado para representar todas las construcciones del lenguaje de programación C y para un subconjunto de Ada, es decir, su parte secuencial y procesal. Para mantener el número de construcciones distintas en la representación intermedia para evitar que los análisis se distingan en muchos casos diferentes, un principio de diseño era proporcionar construcciones simples que se pueden combinar para representar cualquiera de las construcciones de lenguajes. Esto puede ilustrarse con el ejemplo de los bucles while. En lugar de tener un bucle while explícito con una condición asociada, un bucle while se puede representar usando un bucle general, un if-then-else, y una declaración de salida para salir del bucle como lo ilustra los dos programas equivalentes en la Figura 3 (a) y (b).

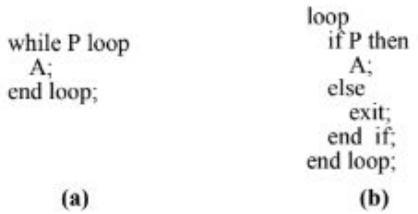


Figure 3. Equivalent loops.

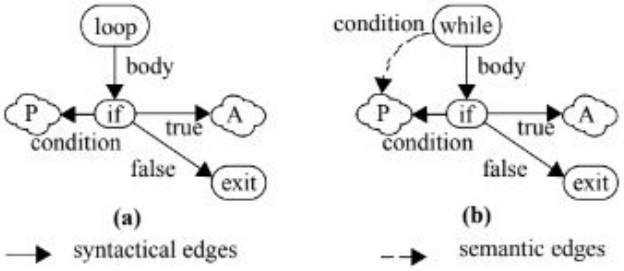


Figure 4. GASTs for loops.

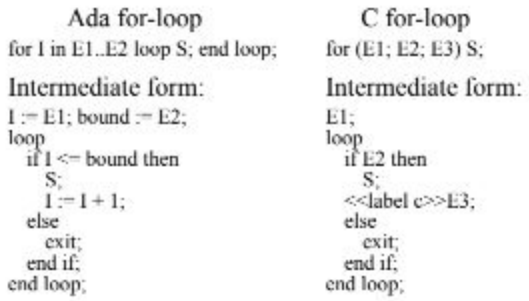
La representación gráfica de la declaración de bucle en la figura 3b se muestra en la figura 4a. Este mapeo tiene la desventaja de que la organización del código fuente original se pierde. Para evitar esto, una clase de nodo while se deriva de la clase de nodo de bucle general con un valor semántico adicional que apunta a la condición del bucle while (Figura 4b). Ahora, dado que while es un bucle y el adicional el atributo de condición es solo semántico, el análisis ve el bucle en la Figura 4a pero el código fuente sigue siendo apropiado representado por el gráfico en la Figura 4b. El mismo principio de diseño se aplica para capturar la semántica de diferentes construcciones en C y Ada, como el bucle for.

- En Ada, el valor inicial y final de un ciclo for son obligatorios. Ambas expresiones se evalúan sólo una vez. El valor del paso 1 es implícitamente.
- En C, todas estas partes son opcionales y pueden ser cualquier presión (que incluye tareas en C). La inicialización se ejecuta una vez, las expresiones de la condición y el paso se ejecutan para cada iteración.

A pesar de estas diferencias, ambos bucles pueden ser representados de la misma manera mediante la representación intermedia.

- En el caso de Ada, se agrega el código implícito del bucle for a la representación intermedia. La figura 5 ilustra esto (la forma intermedia se escribe como Ada equivalente programa de legibilidad).
- En el caso de C, se puede elegir la misma estructura (pero E2 tiene que ejecutarse en cada iteración y, por lo tanto, es movido al cuerpo). A continuar dentro del cuerpo S del for loop original está representado por un nodo continuo (derivado de declaraciones goto) cuyo objetivo es la etiqueta c en el código en la figura 5.

Para preservar la fuente original, el bucle es en realidad representado por un nodo for-loop (derivado del general construcción de bucle) en la forma intermedia. Similar al ejemplo del ciclo while anterior, un nodo for-loop tiene tres atributos semánticos adicionales en las respectivas expresiones del bucle: inicio, fin y expresión de paso. las conexiones semánticas denotan sólo la organización del código fuente.



Ada for-loop	C for-loop
for I in E1..E2 loop S; end loop;	for (E1; E2; E3) S;
Intermediate form:	Intermediate form:

I := E1; bound := E2;
loop
  if I <= bound then
  S;
  I := I + 1;
  else
  exit;
  end if;
end loop;
	E1;
	loop
	  if E2 then
	  S;
	<<label c>>E3;
	  else
	  exit;
	  end if;
	end loop;

Figure 5. Unified loop representation.

ERG con vistas

Para análisis “granular”, usamos el ERG. Sin embargo, agregamos vistas a la ERG para representar las perspectivas de diferentes usuarios, así como también resultados naturales de diferentes análisis. Las vistas también pueden ser solo la parte relevante de un gráfico global a un análisis de manera uniforme y eficiente. Una vista es un subgrafo del ERG, es decir, un gráfico consistente del subconjunto de los nodos y las conexiones de manera que ambos el objetivo y la fuente de cualquier conexión en la vista pertenecen a la vista. Las vistas son un medio de

modelado conveniente en diferentes contextos como se muestra en las siguientes secciones. Entre ellas se encuentran:

1. Vistas para representar diferentes aspectos del código fuente. (Tabla 2)
2. Vistas como entrada y salida uniforme de análisis.
3. Vistas para representar resultados intermedios. (Figura 6)
4. Vistas para representar resultados alternativos.

Table 2. Common graphs in reverse engineering.

Graph	Nodes	Edges
call graph	routines	call relationship
type graph	types	part-of, is-a, or subtype relationships
variable reference graph	routines, variables	set and use relationships

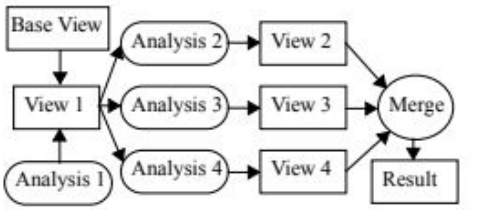


Figure 6. A network of analyses connected by views.

DETALLES TÉCNICOS

Lo básico que se asume de la implementación es:

1. El número promedio de conexiones de un nodo es relativamente pequeño, por eso el grafo es muy disperso.
2. Rara vez se agregan nodos y/o conexiones y es aún más raro eliminarlos. Es más importante el acceso rápido que la adición o eliminación eficiente.
3. El número de vistas es bastante pequeño

Representación ERG

Como se dijo antes, el grafo ERG es muy disperso, por esta razón, este se implementa utilizando listas de adyacencia. Esto implica que cada uno de los nodos tiene una lista de sus sucesores y una lista de sus predecesores. Nodo es una clase abstracta que dice qué entidades concretas (ej: variables y rutinas) pueden ser derivadas. La clase abstracta conexión es extendida por clases derivadas con atributos adicionales. Cada conexión tiene como mínimo dos atributos: el origen y el destino. Los nodos y las conexiones se almacenan en dos tablas dinámicas.

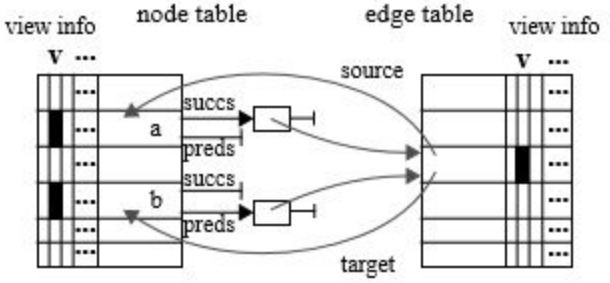


Figure 7. ERG implementation.

Implementación de vistas

Las vistas son conceptualmente conjuntos de nodos y conexiones y son una parte integral de la implementación de ERG, ya que estas permiten remover un nodo de todas las vistas cuando este nodo es removido del ERG.

Conjuntos

Una de las formas de implementar vistas es a través de una representación en conjunto, o en otras palabras, una lista enlazada. El problema principal de esta implementación es la complejidad del tiempo; esto se puede optimizar utilizando un árbol ordenado o un hash, sin embargo, esto no arregla por completo este problema.

Vectores de bits

Otra de las maneras de implementar vistas es como vectores de bits. Cada uno de estos bits indica si una conexión o un nodo está presente en una vista. De esta manera se optimiza el tiempo de acceso a O(1). La desventaja de los vectores de bits es que agregar o remover nodos es muy exigente en cuanto a recursos, esto debido a que cada vez que se agregue o remueva una entidad del ERG, se tiene que actualizar el vector.

Información de vista integrada

Esta es una implementación basada en la anterior. La diferencia principal es que a la hora de guardar la información de las vistas, esta queda almacenada en conjunto con las tablas de nodos y conexiones. La forma de hacer esto es guardar las entradas de la tabla como una entrada doble, donde el primer ítem corresponde a los atributos del nodo o la conexión y el segundo corresponde al vector de bits del que se habló en la implementación anterior. Las ventajas que trae esta implementación son: los recorridos en un ERG son más eficientes (esto si no se tiene que iterar todo el grafo) y la unión de vistas es más eficiente.

EJEMPLO DE APLICACIÓN

Horseshoe

El framework llamado horseshoe, presentado por Kazman et al, se basa en acomodar análisis y procesos de transformación en la recuperación de arquitectura. Este framework consiste de cuatro niveles diferentes:

1. Nivel fuente: código fuente en representación textual
2. Nivel de estructura de código: el AST (Abstract syntax tree) enriquecido con control y flujo de información
3. Nivel de funciones: relaciones entre funciones, datos y archivos
4. Nivel de arquitectura: elementos de arquitectura

Dos de estos niveles corresponden a niveles de IIR: el nivel de estructura de código está representado en el GAST y el nivel de funciones está representado en el ERG.

Nivel de estructura de código

En sistemas de gran tamaño, por lo general, se pueden encontrar diversos componentes escritos en diversos lenguajes. Para simplificar esto, todos estos elementos deberían de estar representados en el nivel de estructura de código. Para esto se creó el GAST, el cual es un AST.

Niveles de funciones y arquitectura

Kazman menciona que una representación intermedia para reingeniería de arquitectura debe representar conceptos en todos los cuatro niveles de abstracción mencionados anteriormente. IR debe de de soportar el mapeo sin ninguna complicación en todos los niveles. IIR soporta estas necesidades y aporta inclusive más beneficios gracias a su mecanismo de vistas. Entre las funciones adicionales del mecanismo de vistas están: la captura de alternativas causadas por las diferentes opiniones de expertos en las bases de datos y analizarlas con mayor facilidad; como las vistas capturan versiones específicas del sistema, estas se pueden analizar para poder monitorear la evolución del sistema; etc.

Abstracción extensible

El ERG permite la adición de conceptos al sistema de abstracción más allá del nivel de código.

Recuperación de ADT utilizando IIR

En el contexto de la recuperación de arquitectura, IIR es usado para detectar abstract data types (ADT) y encapsulaciones de estado y subsistemas. Las vistas facilitan el manejo de diferentes perspectivas de usuario así como la manipulación de los resultados de diferentes análisis. Ya que hay diferentes análisis propuestos para reconocer ADTs, el usuario puede escoger su preferido basado en su experiencia; los resultados de estos análisis pueden ser almacenados en una vista que se le presenta al usuario para que este lo valide. El usuario puede escoger si desea excluir algunas entidades de ADTs o que deben aparecer juntas. Luego de este proceso, las vistas se pueden combinar para formar la entrada de la siguiente iteración del análisis seleccionado.

CONCLUSIONES

Se aprendió algunos de los requerimientos de las representaciones intermedias para la ingeniería inversa, como la importancia de la eficiencia y de un alto nivel de abstracción, también se vieron las similitudes y diferencias con las representaciones intermedias para optimizar compiladores. Se aprendió como los requerimientos de las representaciones intermedias se pueden integrar y cómo se pueden extender a las representaciones intermedias ya existentes, un ejemplo de cómo se pueden extender los ya existentes son por medio de árboles de sintaxis abstracta. También leyó sobre una propuesta de cómo implementar una representación intermedia y su utilidad en el contexto de trabajo de la recuperación de datos abstracta y como podría ser soportada por un modelo de horseshoe.

BIBLOGRAFÍA

Koschke, R. and Martin, J. (1998). *An Intermediate Representation for Integrating Reverse Engineering Analyses*. Stuttgart, Germany: IEEE Computer Society., pp.241-250.