Universidad Nacional Autónoma de México

Facultad de Ciencias

Alojamiento en registros por solución de rompecabezas

T E S I S

QUE PARA OBTENER EL TÍTULO DE:
LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

PRESENTA:
DIEGO ALEJANDRO VELÁZQUEZ CERVANTES

DIRECTOR DE TESIS:
DRA. ELISA VISO GUROVICH

2012

Hoja de Datos del Jurado

1. Datos del alumno
Velázquez
Cervantes
Diego Alejandro
0445522440384
Universidad Nacional Autónoma de México
Facultad de Ciencias
Ciencias de la Computación
305022970

2. Datos del tutor
Dra
Elisa
Viso
Gurovich

3. Datos del sinodal 1
Dr
José David
Flores
Peñaloza

4. Datos del sinodal 2
Dra
María de Luz
Gasca
Soto

5. Datos del sinodal 3
Dr
José de Jesús
Galaviz
Casas

6. Datos del sinodal 4
Dra
Amparo
López
Gaona

7. Datos del trabajo escrito
Alojamiento en registros por solución
de rompecabezas
78 p.
2012

Agradecimientos

Primero agradezco a mis padres: Pedro y Concepción, por haberme dado la vida y haberme apoyado siempre en mis decisiones.

A Edith, que me dió su apoyo y me obligó a trabajar cuando ya no quería hacerlo, brindándome tanto el cariño como la confianza que necesité en cada momento.

A mis hermanos: Toni, Claus y Lili, porque aún cuando tenemos muchas diferencias hemos aprendido a soportarnos (a veces). A pesar de todo los quiero.

A Elisa, por haberme invitado tan temprano a dar clases, y después por guiarme en la elaboración de éste trabajo, al mismo tiempo de subsidiar el café consumido en el proceso.

A mis amigos, pues de cierta forma he crecido por sus logros y reflexionado por sus errores.

Y por último al pueblo mexicano, porque de no existir nuestra universidad quizá yo no estaría escribiendo estos agradecimientos.

Índice general

Índice de figuras

Introducción

Este trabajo tiene como objetivo la implementación de una biblioteca que modela el problema de alojamiento en registros mediante solución de rompecabezas basado en el artículo *Register allocation by puzzle solving* de Fernando Magno Quintão Pereira y Jens Palsberg[13].

Un rompecabezas es una colección de subespacios a llenar en los que se divide el espacio total de registros. Para hacer uso de esta reducción primero se hacen transformaciones al programa original para lograr que los rangos de vida de las variables interfieran lo menos posible y así optimizar el uso de los registros; después se mapea el conjunto de registros de la arquitectura específica para la cuál se quiera compilar, a un tablero de rompecabezas[1], y la colección de variables del programa en su forma elemental a piezas que servirán para solucionar el tablero de rompecabezas.

En este trabajo se exponen dos tipos de solucionadores de rompecabezas: los de tipo 0, que se usan para arquitecturas que no tienen nivel de alias entre registros y, los de tipo 1, que se usan para las arquitecturas que tienen un nivel de alias entre registros, como es el caso de los registros AX, AH y AL en la arquitectura x86. Es importante decir que los distintos solucionadores se pueden combinar para resolver tableros de arquitecturas de tipo híbrido como la x86 en sus registros de propósito general.

La estructura del trabajo se presenta como sigue:

En el primer capítulo se da una introducción al tema de coloración de gráficas, que es una manera de solucionar el problema que nos ocupa. Después se da un panorama general de alojamiento en registro, cuáles son las soluciones clásicas al problema y las diferencias entre éstas. En seguida se da una pequeña introducción acerca de la representación SSI. Finalmente, se termina con una breve explicación de lo que es el problema de alojamiento en registros por solución de rompecabezas.

El segundo capítulo toca los temas referentes a los rompecabezas, cuáles son los rompecabezas de tipo 0, los de tipo 1, los de tipo 2 y los híbridos. Después se muestra cómo hacer el mapeo entre las variables del programa y las piezas del rompecabezas, primero

[1]Más adelante se expone la forma de hacer esta relación.

transformando el programa a su forma elemental[2] y luego identificando las piezas para la solución del rompecabezas. A continuación se muestra cómo se reduce el problema de alojamiento en registros al problema de coloración y qué tipos de gráficas de interferencia tienen los programas transformados a su forma elemental. Finalmente, se define la coloración 1-2 alineada y cómo es que de este problema podemos pasar al de solución de rompecabezas para abordar el problema de alojamiento en registros.

El tercer capítulo introduce una forma visual para representar programas de solución de rompecabezas dando las definiciones de reglas, enunciados y programas además de mencionar la complejidad en tiempo de un programa solucionador. Después se presenta un programa solucionador de rompecabezas y por último se demuestra que el programa es correcto.

Para terminar, se presentan dos apéndices. En el apéndice A se expone a detelle el programa auxiliar que se usa para la demostración del lema de preservación del capítulo 3. En el apéndice B se presenta un modelo de implementación del solucionador de rompecabezas y un ejemplo de uso de la biblioteca para la arquitectura x86 de 16 bits.

[2]Se dará la definición precisa más adelante; por ahora basta con saber que es una forma de representación de un programa, tal como SSA y SSI.

Capítulo 1

Antecedentes

En este capítulo se da una breve introducción a la coloración en gráficas, que es una forma de resolver el problema de alojamiento en registros. En seguida se muestra un panorama general a cerca de los alojadores en registros y su implementación. Después se presenta un breve resumen de la representación en forma *Static Single Information*, que es una representación intermedia del programa en el proceso de compilación. Por último se expone brevemente el tema de alojamiento en registros por solución de rompecabezas.

1.1. Coloración de gráficas

El término de coloración en una gráfica se refiere a dar colores a los vértices de la misma. Una k-coloración por vértices es una coloración que usa k colores; la coloración se llama *propia* si ningún par de vértices adyacentes tienen el mismo color; entonces, una k-coloración de $G = (V, E)$ es una partición $(V_1, V_2, \ldots, V_k)$ de V en k conjuntos ajenos, donde en el subconjunto V_i están todos los vértices coloreados con el i-ésimo color. El número cromático de una gráfica es la mínima k con la que se puede tener una coloración propia de G. En computación sabemos que encontrar el número cromático de una gráfica es un problema NP-completo y es importante señalarlo ya que un alojador en registros debe minimizar el número de registros de procesador usados por un programa. El mínimo absoluto resulta ser el número cromático de la gráfica.

1.2. Asignación de registros

Al ser los registros del procesador los elementos más rápidos en una computadora, en cuanto a almacenamiento, operación y recuperación de la información, es de vital impor-

tancia para el desempeño del programa un buen manejo de aquéllos. En un compilador el encargado de hacer este trabajo es el *alojador en registros*.

El alojador en registros determina qué valores deben alojarse en los registros y si en determinado punto del programa esto no es posible, se encarga de pasar el valor al siguiente nivel de memoria.

Conceptualmente el alojador en registros toma como entrada un cierto programa que usa un número arbitrario de registros y produce un programa equivalente que se ajusta al conjunto finito de registros de la máquina objetivo. Además de esto necesita insertar código de lectura y escritura a la memoria, lo cual debe minimizar para un mejor rendimiento del programa.

En un compilador moderno el alojador de registros resuelve dos problemas distintos: alojamiento en registros y la asignación de registros. El alojamiento en registros mapea un conjunto de registros de la máquina objetivo a un conjunto ilimitado de espacios de nombres también llamados registros virtuales, mientras que la asignación de registros mapea un conjunto de registros virtuales a los registros físicos de la máquina objetivo. La asignación de registros supone que el alojamiento en registros ha sido realizado, por lo que el código se ajustará al conjunto de registros físicos provistos por la máquina objetivo. También proveerá los verdaderos nombres requeridos por el código ejecutable. En seguida daremos dos definiciones que usaremos un poco más adelante.

Definición. Un *rango de vida* es un conjunto cerrado de definiciones y usos relacionados de una variable, que sirve como la base de nombres para el alojamiento de registros. La figura 1.1 ejemplifica gráficamente los rangos de vida de varias variables.

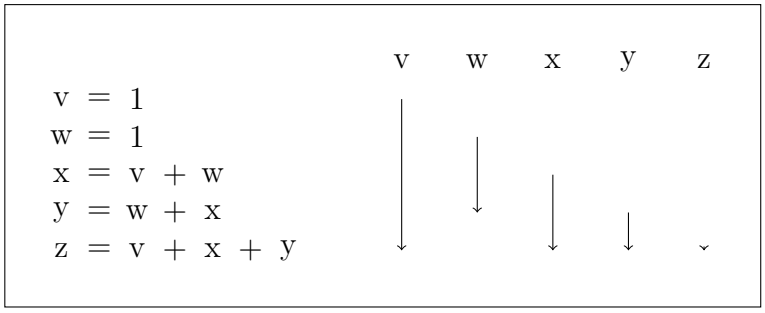


Figura 1.1: Rangos de vida de las variables del programa en el lado izquierdo

Definición. La *gráfica de intervalos* es una gráfica que representa la sobreposición entre múltiples intervalos, en donde cada nodo de la gráfica representa un intervalo y una arista entre dos nodos significa que la intersección entre los dos intervalos no es vacía. Un intervalo puede ser representado por *[i,j]*, donde la i-ésima operación es la que define una variable x y la j-ésima es la que tiene el último uso de la variable x. La figura 1.2 muestra la gráfica de intervalos correspondiente a la figura 1.1.

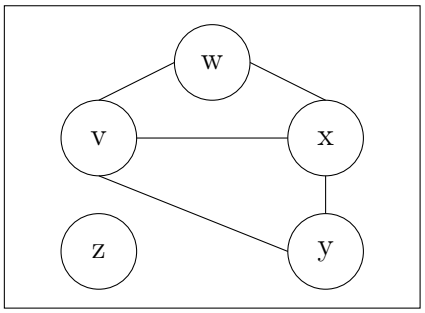


Figura 1.2: Gráfica de intervalos correspondiente a la figura 1.1

El alojamiento en registros es un problema difícil, ya que las formalizaciones generales del problema son NP-completas.[7] Por otro lado, la asignación de registros en muchos casos se puede resolver en tiempo polinomial mediante la coloración de la gráfica de intervalos, si es que habiendo hecho la repartición de registros no se excede el número de registros físicos de la máquina.

Los registros físicos provistos por la mayoría de los procesadores no forman un conjunto homogéneo de recursos intercambiables. Comúnmente se tienen diversas clases de registros para diferentes tipos de variables, por ejemplo de propósito general y de punto flotante. El compilador debe colocar cada valor en una clase de registro apropiada, lo cual se logra haciendo la asignación de las clases de registros más especializadas a las más generales; por ejemplo, primero alojando los registros de punto flotante y después los de propósito general.

Tanto el alojamiento como la asignación se puede hacer a dos escalas: local y global. El alojador local trabaja sobre un solo bloque[1] que consiste en una serie de operaciones de tres direcciones, donde cada operación tiene la forma *op* vr_1 vr_2 vr_3, donde vr_i corresponde

[1]Un bloque es una sucesión de código donde ninguno de los rangos de vida de las variables involucradas se extiende fuera de la sucesión.

a un registro virtual. Un alojador global trabaja con el programa completo, representando a éste también con código de tres direcciones.

Primero daremos una breve precisión a cerca de los alojadores en registros locales y en seguida haremos lo respectivo con los alojadores globales.

Los alojadores en registros locales pueden ser ascendentes o descendentes. Un alojador local de registros descendente *(top-down)* trabaja bajo el principio de que los valores más frecuentemente usados deben estar en los registros; para esto cuenta el número de veces que un registro virtual aparece en el bloque y después asigna registros virtuales a registros físicos en orden de uso descendente. Si hay más registros virtuales que físicos, denotando con k el número de registros físicos, se deben reservar suficientes registros físicos para leer y escribir de memoria, y poder usar los valores que no estén guardados en los registros[2]. El número de registros que se deben reservar depende del procesador y lo denotaremos con F. Si el bloque usa más de k registros virtuales el compilador aplica el algoritmo que se muestra en la figura 1.3.

```
1 Calcular la prioridad de cada registro virtual.
2 Ordenar los registros virtuales por prioridad.
3 Asignar registros en orden de prioridad.
4 Reescribir el código.
```

Figura 1.3: Algoritmo del alojador local de registros descendente

Un alojador de registros local ascendente *(bottom-up)* se enfoca en los detalles de cómo se define cada valor y cómo se usa a lo largo de cada operación. Comienza con cero registros ocupados y para cada operación se encarga de que sus operandos estén en los registros antes de ser ejecutada. Itera sobre las operaciones del bloque tomando decisiones de asignación sobre demanda. La mayoría de las complicaciones del algoritmo ocurren en las rutinas *Asegura*, *Aloja* y *Libera*. La rutina *Asegura* toma dos argumentos: un registro virtual *vr* que tiene el valor deseado y una representación de la clase apropiada de registro, *clase*. Si *vr* no está en un registro físico, asigna *vr* a uno de tipo *clase* y genera código para mover el valor de *vr* al registro asignado. *Aloja* y *Libera* exponen los detalles del problema de asignación. *Aloja* se encarga de regresar un registro físico de una clase dada. En caso de haber un registro libre lo devuelve, en otro caso selecciona el valor guardado en la clase que se usará más a futuro, derrama el valor y regresa el registro correspondiente. El proceso *Libera* se encarga de liberar registros.

[2]A esto es a lo que se llama *"spilling"* pues se derraman los registros a memoria.

La figura 1.4 muestra el algoritmo de alojamiento en registros local ascendente y la figura 1.5 las subrutinas *Aloja* y *Asegura*[3]; la función *Dist(vr)*, usada en el algoritmo, regresa el índice de la siguiente referencia a *vr*.

```
1  para cada operación, i, en orden de 1 a N donde i tiene
       la forma op vr_i1 vr_i2 => vr_i3
2    r_x = Asegura(vr_i1,clase(vr_i1))
3    r_y = Asegura(vr_i2,clase(vr_i2))
4    si vr_i1 no se necesita después de i
5      entonces Libera(r_x, clase(r_x))
6    si vr_i2 no se necesita después de i
7      entonces Libera(r_y, clase(r_y))
8    r_z = Aloja(vr_i3,clase(vr_i3))
9    reescribe i como op_i r_x, r_y => r_z
10   si vr_i1 no se necesita después de i
11     entonces clase.Siguiente[r_x] = Dist(vr_i1)
12   si vr_i2 no se necesita después de i
13     entonces clase.Siguiente[r_y] = Dist(vr_i2)
14   clase.Siguiente[r_z] = Dist(vr_i3)
```

Figura 1.4: Algoritmo del alojador en registros ascendente

En este punto son importantes los conceptos de *valor sucio* y *valor limpio*, que se refieren a si un valor debe o no ser escrito a memoria respectivamente. Este concepto es importante ya que dependiendo de si un valor está limpio o sucio, el código generado por *Asegura* puede optimizarse en menor o mayor grado.

En cuanto a la asignación global se puede decir que difiere de la asignación local en dos aspectos fundamentales.

1. La estructura de un rango de vida global puede ser más compleja que una local.

2. En un rango de vida local, todas las referencias se ejecutan una vez por ejecución del bloque, por lo que el costo es uniforme, mientras que en un rango de vida global, las distintas referencias se pueden ejecutar distinto número de veces por lo que el costo depende de si hay o no código de derrame *(spill)*.

Los alojadores globales toman decisiones acerca de si cada rango de vida residirá o no en un registro, cuándo puede o no compartir un registro con otro y cuál es el registro físico para alojarlo. Para tomar estas decisiones muchos compiladores hacen la asignación de registros

[3]Para mayor detalle consultar *Engineering a Compiler*[7].

```
1  Asegura(vr,clase)
2    si (vr ya está en clase)
3      entonces resultado = registro físico de vr
4    en otro caso
5      resultado = Aloja(vr,clase)
6    produce código para mover vr a resultado
7    regresa resultado
8
9  Aloja(vr,clase)
10   si (clase.Top >= 0)
11     entonces i = pop(clase)
12   en otro caso
13     i = j que maximiza clase.Siguiente[j]
14     guarda el contenido de j
15   clase.Nombre[i] = vr
16   clase.Siguiente[i] = -1
17   clase.Libera[i] = falso
18   regresa i
```

Figura 1.5: Subrutinas *Asegura* y *Aloja*

usando una analogía de coloración de gráficas, construyendo la gráfica de interferencias para modelar conflictos entre rangos de vida. La gráfica de interferencia es una gráfica donde los nodos representan rangos de vida y una arista *(i,j)* indica que las variables en el rango de vida i y las del rango de vida j no pueden compartir un registro. Se trata de construir una *k-coloración* para la gráfica de interferencias donde k es el número de registros físicos disponibles. Una *k-coloración* se traduce como un alojamiento de los valores de las variables asociadas a los rangos de vida en registros físicos. Si no es posible, se modifica el código derramando algunos valores a la memoria e intentando de nuevo.

La forma que asuma el código intermedio influye en las etapas de alojamiento y asignación. La forma SSA *(Static Single-Assignment)* es una representación intermedia de código de tres direcciones que tiene un sistema de nombres basado en valores creados por renombramiento y por el uso de funciones ϕ. Una función ϕ toma como argumento nombres asociados a valores para mezclarlos en los puntos de reunión del programa. Podemos ver un ejemplo de un programa en forma SSA en la figura 1.6.

Para construir los rangos de vida de las variables de un programa en forma SSA, un alojador global usa los algoritmos de unión y búsqueda sobre conjuntos disjuntos haciendo una pasada sobre el código. El alojador trata a cada nombre o definición en el programa como un conjunto en el algoritmo. Examina cada función-ϕ en el programa y une los con-

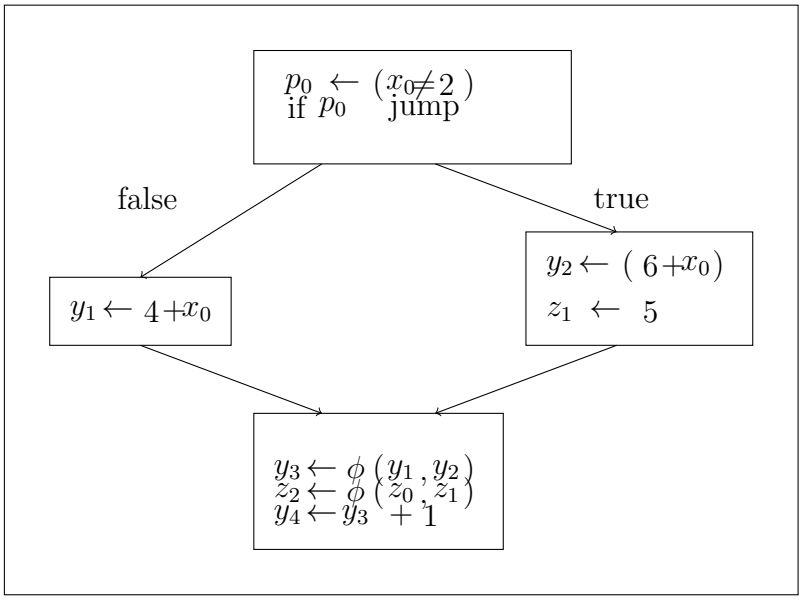


Figura 1.6: Programa en forma SSA

juntos asociados con cada parámetro de la función-ϕ con el conjunto asociado al resultado de esta función. Después de que cada función-ϕ se procesa, los conjuntos representan los rangos de vida en el código.

Para hacer la estimación del costo de derrame, el asignador global necesita un estimado del costo de derrame de cada valor. Para calcular este costo se tiene que tomar en cuenta la dirección de cálculo, la operación de memoria y un estimado de frecuencia de ejecución. Para estimar la frecuencia muchos compiladores suponen que cada ciclo se ejecuta diez veces y la presencia de una condicional *(if-then-else)* decrecerá la frecuencia de su bloque a la mitad.

Para construir un alojador global basado en el paradigma de coloración de gráficas, se necesitan dos mecanismos adicionales. Primero se necesita una técnica eficiente para descubrir k-coloraciones. Desafortunadamente el problema de determinar si una gráfica G es k-coloreable es NP-completo[7], por lo que los alojadores globales usan heurísticas que no siempre garantizan una k-coloración. Posteriormente es necesario un mecanismo que maneje el caso de que un rango de vida no pueda ser coloreado. A coninuación definimos lo que es una gráfica de interferencia cuyo concepto utilizaremos en seguida.

Definición. Una *gráfica de interferencias* es aquella en la que los nodos representan rangos de vida y las aristas representan interferencia entre ellos. Dos rangos de vida interfieren si uno está vivo en la definición del otro y tienen diferentes valores. La figura 1.7 muestra la gráfica de interferencia de los rangos de vida de la figura 1.1 en la página 10.

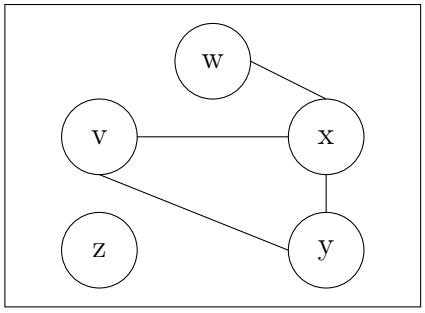


Figura 1.7: Gráfica de interferencia para el programa de la figura 1.1

Un alojador descendente de registros por coloración usa información de bajo nivel para asignar el color a rangos de vida e información de alto nivel para seleccionar el orden en que son coloreados. Se trata de colorear los rangos de vida en un orden determinado por una función que arroja un resultado análogo al costo de derrame. Un asignador ascendente de registros por coloración usa las mismas técnicas que uno descendente, siendo la diferencia principal el mecanismo de ordenamiento de los rangos de vida: mientras que un asignador descendente usa información de alto nivel para seleccionar el orden, un asignador ascendente calcula un orden por el conocimiento detallado de la estructura de la gráfica de interferencia.

1.3. Forma SSI *(Static Single Information)*

La forma SSI extiende a la forma SSA para obtener simetría en el flujo de datos hacia delante o hacia atrás. La forma SSI reconoce que la información acerca de las variables es generada en las bifurcaciones del flujo y genera nuevos nombres en esos puntos. Esto nos provee de un mapeo uno a uno entre los nombres de las variables y la información de las variables en cada punto del programa. Los análisis pueden entonces asociar información con nombres de variable y propagarla eficientemente con y contra la dirección del control de flujo.

La construcción de la forma SSI involucra añadir pseudo asignaciones para una variable v:

1. Funciones ϕ al mezclar dos trayectorias provenientes de una condicional, cuando al menos una de las dos trayectorias contiene una definición de v.

2. Funciones π en los puntos donde el control de flujo se divide y por lo menos una de las dos trayectorias usa el valor de v.

Podemos apreciar la comparación entre un programa en forma SSA y el mismo en forma SSI en la figura 1.8 en la siguiente página.

A continuación nos referimos a una trayectoria de longitud mayor o igual a 1 que va de X a Y con la notación $X \xrightarrow{+} Y$.

Formalmente un programa en forma SSI satisface las siguientes condiciones:

1. **Colocación de funciones** ϕ. Si dos trayectorias $X \xrightarrow{+} Z$ y $Y \xrightarrow{+} Z$ son ajenas excepto por el nodo Z y los nodos X e Y tienen asignaciones a la variable v en el programa original, o una función ϕ o π para v en el nuevo programa, entonces se insertó una función ϕ para v en Z en el nuevo programa.

2. **Colocación de funciones** π. Si dos trayectorias $Z \xrightarrow{+} X$ y $Z \xrightarrow{+} Y$ existen teniendo sólo el nodo Z en común donde se separan, y los nodos X e Y tienen usos de la variable v en el programa original o funciones ϕ o π para v en el nuevo programa, entonces se insertó una función π para v en Z en el nuevo programa.

3. **Uso posterior a las funciones** ϕ. Para cada nodo X que tenga una definición de la variable v en el nuevo programa y para cada nodo Y que haga uso de esa variable, existe por lo menos una trayectoria $X \xrightarrow{+} Y$ y en esa trayectoria no hay otra definición de v más que la de X.

4. **Uso posterior a las funciones** π. Para cada par de nodos X e Y que usen la variable v definida en el nodo Z en el nuevo programa, toda trayectoria $Z \xrightarrow{+} X$ contiene a Y o toda trayectoria $Z \xrightarrow{+} Y$ contiene a X.

5. **Condiciones de frontera**. Se supone que el nodo de inicio contiene una definición y que el nodo final contiene un uso para cada variable en el programa original.

6. **Correctud**. A lo largo de cada posible trayectoria de control de flujo de un programa en ejecución, considerar cualquier uso de la variable v en el programa original y el uso correspondiente de v_i en el nuevo programa. Entonces en cada presencia de uso en el trayectoria, v y v_i tienen el mismo valor.

1.4. Alojamiento en registros por solución de rompecabezas

El alojamiento en registros por solución de rompecabezas es una abstracción del problema de alojamiento en registros introducido por Fernando Magno Quintão Pereira y Jens Palsberg[13]. En esta abstracción se modela al archivo de registros como un tablero de rompecabezas y las variables del programa como piezas de rompecabezas. El resultado es una colección de rompecabezas, con un rompecabezas por cada instrucción en la representación intermedia del programa.

El alojador en registros consiste en los siguientes pasos:

1. Transformar el programa en un programa elemental aumentándolo con funciones ϕ, funciones π y copias paralelas.

2. Transformar el programa elemental en una colección de rompecabezas.

3. Resolver los rompecabezas, hacer código de derrame y fusión de valores *(coalescing)*.

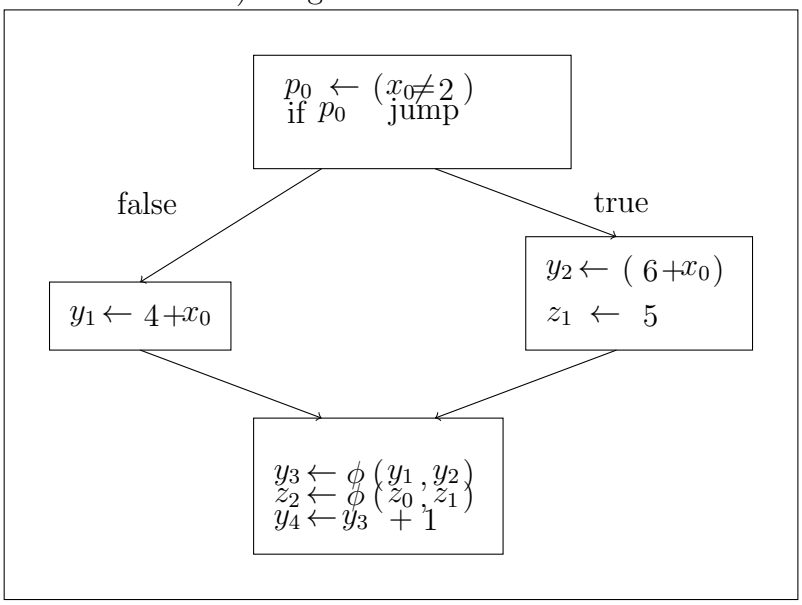
4. Transformar el programa elemental y el resultado del alojamiento en registros.

La idea principal de esta aproximación es el uso de programas elementales. En los programas elementales los rangos de vida son cortos lo que permite definir y resolver un rompecabezas por instrucción. La figura 1.9[4] muestra un ejemplo de un programa elemental, las piezas que se obtienen del programa y un tablero con tres áreas correspondientes a tres registros X, Y y Z con alias de tipo 1.

Hasta ahora hemos introducido brevemente el problema que nos ocupa. En los siguietes capítulos se trata más a detalle cada aspecto del trabajo, como la definición de lo que llamamos rompecabezas, la forma en que representamos un programa, el programa solucionador en cuestión y la demostración de que el algoritmo es correcto.

[4]Esta imagen proviene de la página personal de Fernando Magno Quintão Pereira en la UCLA[1]; aquí se puede encontrar una presentación completa del proyecto.

a) Programa en forma SSA



b) Programa en forma SSI

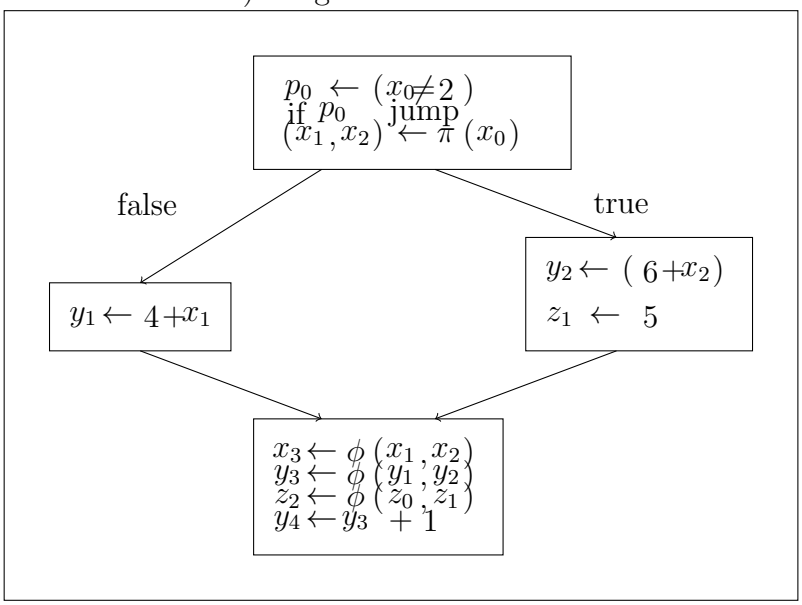


Figura 1.8: a) un programa en forma SSA, b) el mismo programa en forma SSI

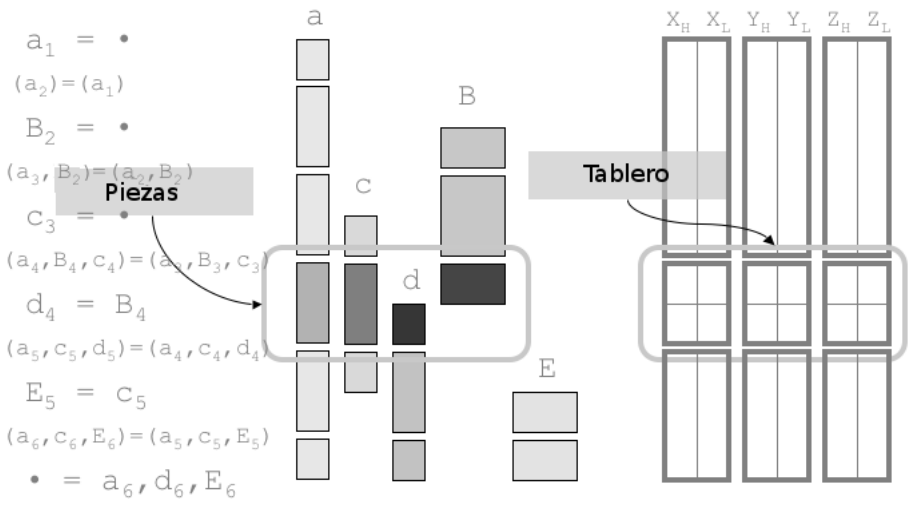


Figura 1.9: Un programa elemental y su correspondiente rompecabezas

Capítulo 2

Rompecabezas

En este capítulo daremos la definición de lo que llamamos rompecabezas. En seguida veremos cómo representar el conjunto de registros de una arquitectura como un tablero de rompecabezas y revisaremos algunos tipos de tableros que podemos obtener de diferentes arquitecturas. Después trataremos el proceso de obtención de las piezas del rompecabezas, primero transformando el programa a su forma elemental y después mapeando cada variable del programa transformado a una pieza de rompecabezas. Finalmente, se presenta la demostrción de que el problema de alojamiento en registros y la solución de rompecabezas son problemas equivalentes.

Definición. Un *rompecabezas* para el alojamiento en registros consiste en un tablero y un conjunto de piezas. Las piezas no se pueden sobreponer en el tablero; además, puede haber piezas ya acomodadas en el tablero que no se pueden mover. El problema de solución de rompecabezas consiste en acomodar las piezas faltantes en el tablero.

2.1. Del conjunto de registros al tablero del rompecabezas

El conjunto de registros en la arquitectura objetivo determina la forma del tablero. Cada tablero tiene un número de *áreas* separadas, donde cada área consta de dos renglones de cuadrados, más adelante se explicará el por qué de los dos renglones. El archivo de registros puede soportar alias, lo que determina el número de columnas en cada área, las formas válidas de las piezas y las reglas para acomodar las piezas en el tablero. Se distinguen tres tipos de rompecabezas: los de tipo 0, los de tipo 1 y los de tipo 2, donde cada área de un rompecabezas de tipo n tiene 2^n columnas. La figura 2.1 muestra la forma de los

tableros de tipo 0, de tipo 1 y de tipo 2, así como los tipos de piezas válidas para cada tipo de tablero.

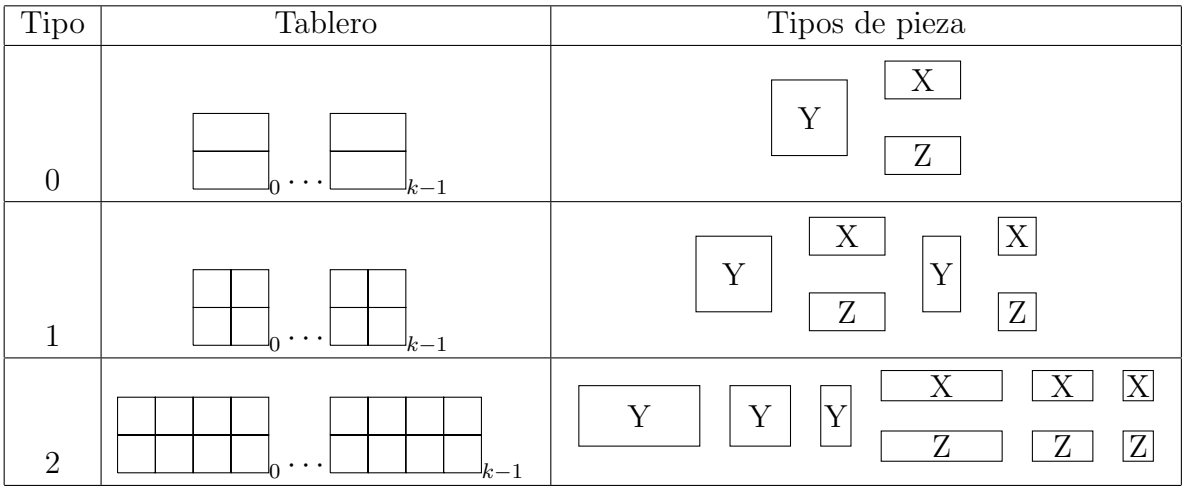


Figura 2.1: Tipos de tablero y piezas.

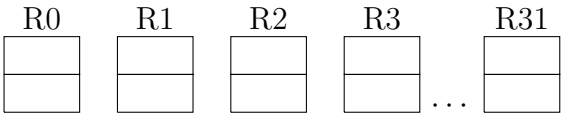
2.1.1. Rompecabezas de tipo 0

El conjunto de registros de propósito general en la arquitecturas powerPC es simple ya que no soportan alias en sus registros. En la figura 2.2 (a) se muestra el tablero de rompecabezas correspondiente a los registros de propósito general de la arquitectura PowerPC. Cada área, que corresponde a cada registro, tiene una sola columna y las piezas válidas son las correspondientes a los tres tipos mostrados en la figura 2.1. Se puede ver que la solución de rompecabezas de tipo cero se puede hacer en tiempo lineal en el número de áreas, acomodando primero las piezas de tipo Y, luego las de tipo X y, por último, las de tipo Z.

2.1.2. Rompecabezas de tipo 1

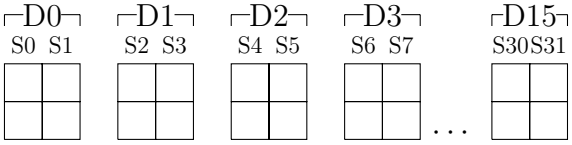
La figura 2.2 (b) muestra el tablero de rompecabezas correspondiente a los registros de punto flotante usados en la arquitectura ARM. El conjunto de registros contiene treinta y dos registros de precisión simple que se pueden combinar en dieciséis registros de doble precisión. Cada área del tablero tiene dos columnas que corresponden a los dos registros que se pueden combinar. El conjunto de registros de punto flotante de la arquitectura

a)



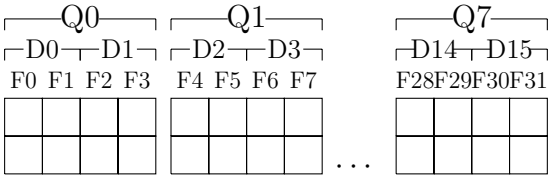
PowerPC: 32 registros de propósito general.

b)



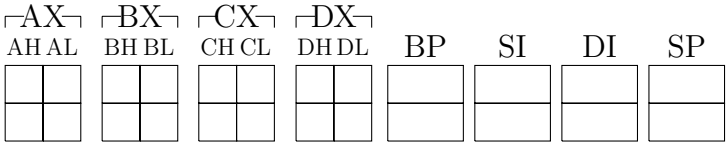
ARM: 16 registros de punto flotante y doble precisión.

c)



SPARC V8: 8 registros de punto flotante y precisión cuádruple.

d)



x86: 8 registros enteros.

Figura 2.2: Ejemplos de tableros

ARM produce rompecabezas de tipo 1 y sus piezas válidas son las correspondientes en la figura 2.1.

Es importante dar la definición de tamaño de una pieza de rompecabezas, ya que el concepto se usará de aquí adelante.

Definición. El *tamaño* de una pieza es el número de cuadrados que ocupa en el tablero.

Ahora mencionaremos los lugares donde se pueden acomodar los diferentes tipos de piezas. Las piezas de tipo X y tamaño 1 se pueden acomodar en cualquier cuadrado del renglón su-

perior; una pieza de tipo X y tamaño 2 en los dos cuadrados superiores de cualquier área; una pieza de tipo Z y tamaño 1 en cualquier cuadrado del renglón inferior; una pieza de tipo Z y tamaño 2 en los dos cuadrados inferiores de un área; se trabaja de forma análoga con las demás piezas.

2.1.3. Rompecabezas de tipo 2

La arquitectura SPARC V8 soporta dos niveles de alias: dos registros de punto flotante de 32 bits pueden ser combinados para formar un registro para valores de 64 bits; además, dos de estos registros pueden ser combinados para formar registros de 128 bits. La figura 2.2 (c) muestra el tablero de rompecabezas para los registros de punto flotante de SPARC V8. Cada área tiene cuatro columnas correspondientes a cuatro registros que pueden ser combinados. Esta arquitectura genera un rompecabezas de tipo 2 con los nueve tipos de piezas correspondientes mostrados anteriormente en la figura 2.1.

2.1.4. Rompecabezas híbridos

La arquitectura x86 produce un híbrido de rompecabezas de tipo 0 y de tipo 1. La figura 2.3 muestra el conjunto de registros enteros de 8 y 16 bits para la arquitectura x86. La figura 2.2 (d) muestra el tablero de rompecabezas correspondiente. Los registros AX, BX, CX y DX producen rompecabezas de tipo 1, mientras que los registros BP, SI, DI, y SP producen un rompecabezas de tipo 0. Es importante notar que la arquitectura x86 no genera rompecabezas de tipo 2, ya que aunque se pueden alojar valores de 8 bits en los registros de 32 bits, la arquitectura no provee alias para la porción de los 16 bits superiores.

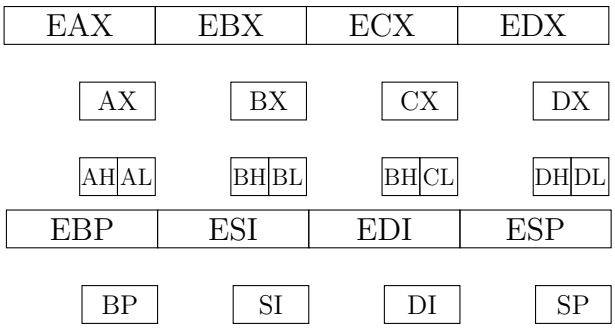


Figura 2.3: Arhivo de registros x86

2.2. Obtención de las piezas del rompecabezas

Para otener las piezas de rompecabezas primero se transforma el programa a su forma elemental y después se mapean las variables del programa transformado a piezas de rompecabezas. A continuación se detalla el proceso mencionado.

2.2.1. Transformación de un programa a un programa elemental

Para convertir el código de programa en un programa elemental primero se convierte el código a forma SSA. En seguida se transforma la forma SSA en la forma SSI. En la forma SSI, cada bloque termina con una función π[1] que renombra las variables que siguen vivas al salir del bloque. Finalmente se transforma la forma SSI en un programa elemental insertando una copia paralela entre cada par de instrucciones consecutivas. En la figura 2.4, de la siguiente página, se puede ver la conversión de un programa a un programa elemental. Se adopta la convención de que las letras minúsculas representan variables que pueden ser guardadas en registros simples y las letras mayúsculas denotan variables que deben ser guardadas en dos registros. Las variables en fuente *typewriter*, por ejemplo `AL`, denotan registros precoloreados. Se usa también la notación $x = y$ que significa el uso de y para la definición de x, no sólo la copia de valores. Una instrucción como $v_1 = \bullet$ define una variable pero no usa una variable o registro para su definición.

2.2.2. Mapeo entre las variables de un programa elemental y las piezas de rompecabezas

En esta sección se definen los conceptos punto de programa y rango de vida, para poder formalizar el cuándo una variable entra o sale viva de una instrucción del programa. Finalmente se menciona la manera en que se hace el mapeo de las variables del programa elemental a piezas de rompecabezas.

Definición. Un *punto del programa* es un punto entre cualquier par de instrucciones consecutivas. En la figura 2.4 b) de la siguiente página podemos observar los puntos de programa $p_0, \ldots, p_{11}$.

[1]Las funciones π se llaman también funciones σ *(σ-functions)* u operadores de cambio *(switch operators)*

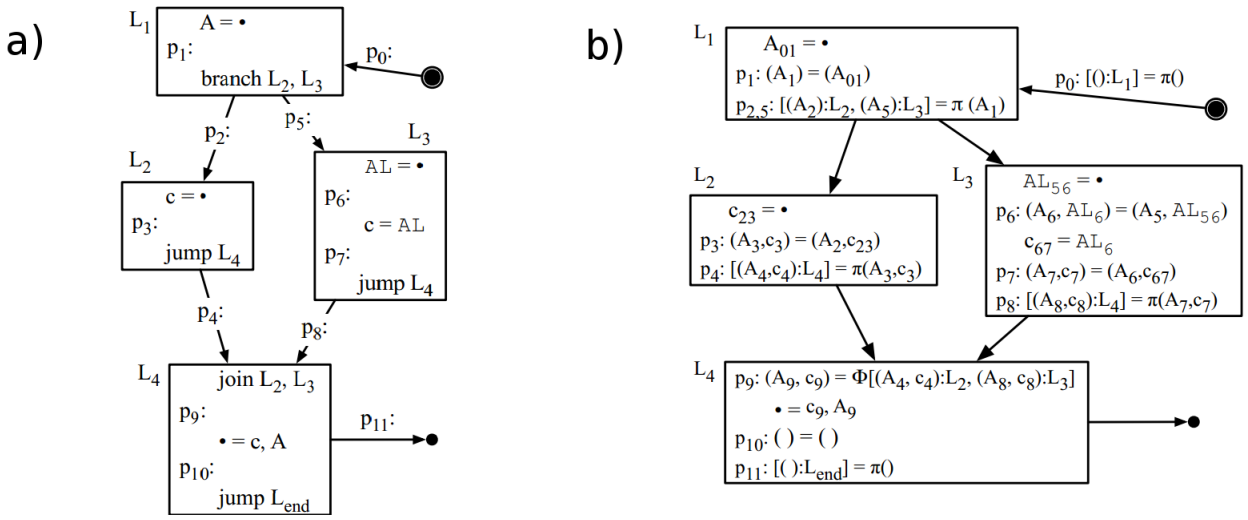


Figura 2.4: Conversión de un programa a un programa elemental

Definición. El *rango de vida* de una variable v es la colección de puntos del programa donde la variable v está viva.

Es importante mencionar que los rangos de vida en un programa elemental contienen a lo más dos puntos del programa.

Se dice que una variable *entra viva* a la instrucción i si su rango de vida contiene un punto del programa que precede a i. Similarmente, se dice que v *sale viva* de i si su rango de vida contiene un punto de programa que sigue a la instrucción i. De esto es intuitivo el hecho de que cada tablero de rompecabezas tenga dos renglones.

Para hacer el mapeo de variables del programa elemental a piezas de rompecabezas, para cada instrucción i en un programa elemental se crea un rompecabezas que tiene una pieza por cada variable que entra o sale viva de la instrucción i. Las variables cuyos rangos de vida terminan en i se convierten en piezas de tipo X; las variables cuyos rangos de vida comienzan en i se convierten en piezas de tipo Z y las variables cuyos rangos de vida cruzan i se convierten en piezas de tipo Y. La figura 2.5 muestra un ejemplo de un fragmento de programa que usa seis variables, sus rangos de vida y las piezas de rompecabezas resultantes.

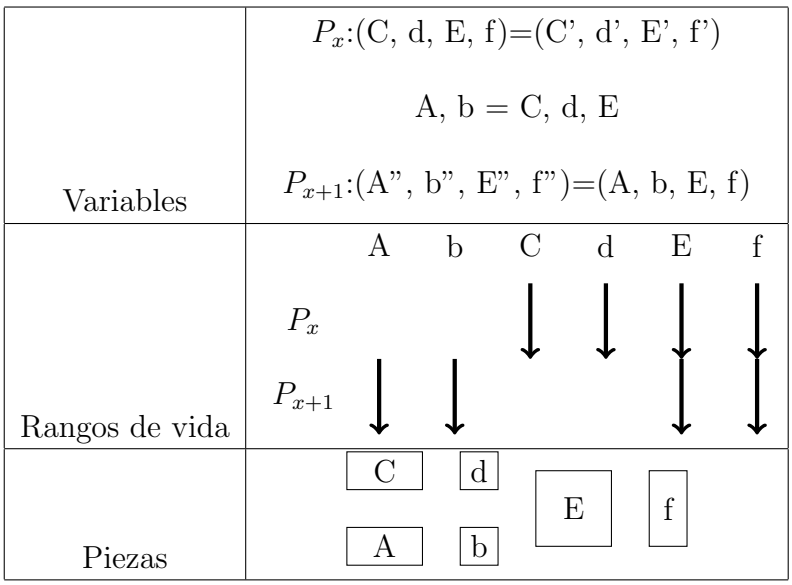


Figura 2.5: Mapeo de variables de programa a piezas de rompecabezas

Debemos notar que el tamaño de las piezas está dado por el tipo de las variables. Por ejemplo, para la arquitectura x86 una variable de 8 bits con un rango de vida que termina en la instrucción i se convierte en una pieza de tipo X y tamaño 1, mientras que una de 16

o 32 bits cuyo rango de vida termina en la instrucción i se convierte en una pieza de tipo X y tamaño 2. La figura 2.6 (a) muestra los rompecabezas producidos por el programa en la figura 2.4 b).

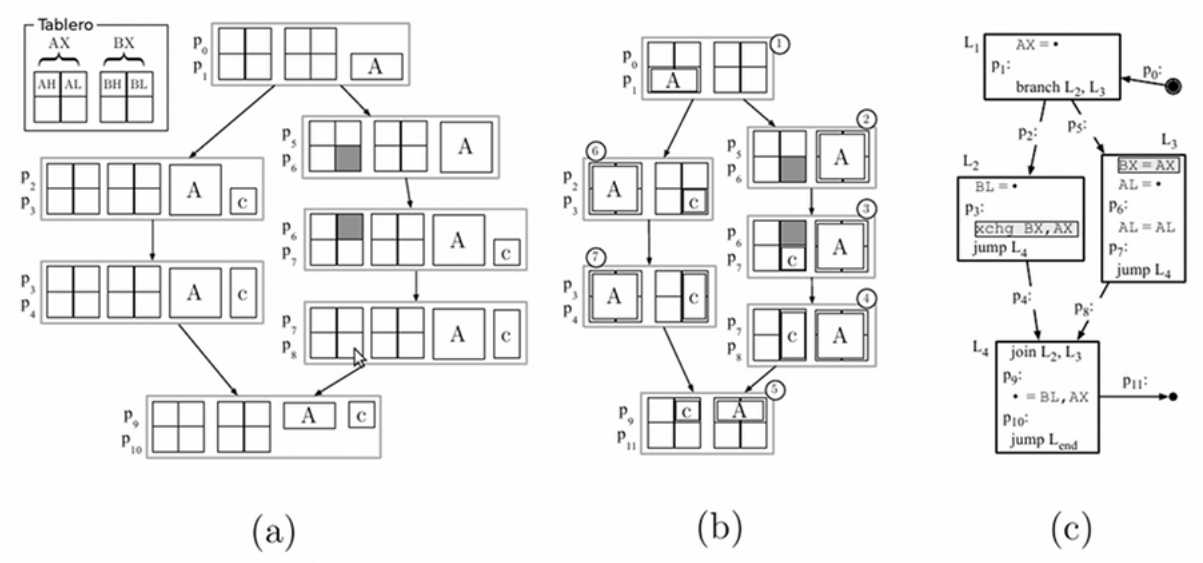


Figura 2.6: (a) Los rompecabezas producidos por el programa de la figura 2.4 b). (b) Un ejemplo de solución. (c) El programa final.

2.3. Relación entre el alojamiento en registros y la solución de rompecabezas

En en esta sección se presenta la demostración de que el alojamiento en registros y la solución de rompecabezas son problemas equivalentes. Primero explicaremos algunos conceptos que se utilizarán en la demostración. Después se expone la forma de reducir el problema de alojamiento en registros a la coloración de gráficas.

Definición. Dada una gráfica G con sus vértices etiquetados con pesos w, $w \in \{1, 2\}$, una *coloración 1-2 alineada* de G es una coloración con $2k$ colores de G, con la particularidad de que los vértices con peso 2 son coloreados con dos *colores alineados*, entendiendo por colores alineados a los colores $2i$ y $2i + 1$ con $0 \leq i < k$.

Definición. Sean $G = (V, E)$ una gráfica, $V_R \subseteq V$ y C un conjunto de colores. La función $\phi : V_R \to C$ es una *coloración parcial* de G que asigna a los vértices de V_R un color en C.

Denotemos como $[x, y]$ a un conjunto de dos colores y como $[x]$ a un conjunto de un solo color. Esta notación será utilizada en las definiciones que siguen.

Definición. Sean $G = (V, E)$ una gráfica con sus vértices etiquetados con pesos w, con $w \in \{1, 2\}$, $V_R \subseteq V$, C un conjunto de colores, y $C' = \{[x] | x \in C\} \cup \{[x, y] | x, y \in C\}$. Una *coloración parcial 1-2 alineada* de G es una función $\phi : V_R \to C'$ que asigna un color, $[x]$, a los vértices de peso 1, o un par de colores alineados, $[x, y]$, a los vértices de peso 2 en V_R.

Definición. Supongamos que se tienen $2k$ colores, una gráfica G con sus vértices etiquetados con pesos w, $w \in \{1, 2\}$ y una coloración parcial 1-2 alineada ϕ de G. El problema de la extensión de la coloración 1-2 alineada consiste en extender ϕ a una coloración 1-2 alineada de G. Se denota a un ejemplar del problema como $\langle 2k, G, \phi \rangle$. Además cualquier vértice v de G que esté en el dominio de ϕ se dice que está *precoloreado*.

Definición. Sea $G = (V, E)$ una gráfica, $K_V = (V_{K_V}, E_{K_V})$ la gráfica completa de $|V|$ vértices, definimos el *complemento de* G, como $\overline{G} = (V, E_{K_V} - E)$.

Definición. Sean H_0 una gráfica con n vértices $v_1, v_2, \ldots, v_n$ y $H_1, H_2, \ldots, H_n$ n gráficas disjuntas arbitrarias. Construimos la *composición de gráficas* $H = H_0[H_1, H_2 \ldots H_n]$ de la siguiente forma: para toda $0 \leq i, j \leq n$ se reemplaza al vértice v_i en H_0 con la gráfica H_i y se hace adyacente a cada vértice de H_i con cada vértice de H_j, siempre que v_i sea adyacente a v_j en H_0. Se muestra un ejemplo en la figura 2.7, donde se compone a K_2 con $\overline{K_2}$ y $\overline{K_2}$.

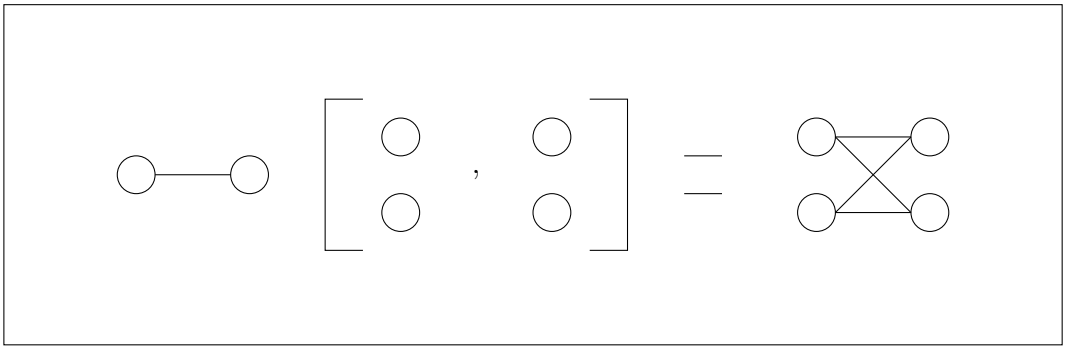


Figura 2.7: Composición de gráficas

Definición. Dada una gráfica $G = (V, E)$, decimos que $H = (V', E')$ es una *subgráfica inducida* de G si $V' \subseteq V$ y dados dos vértices $u, v \in V'$, $uv \in E'$ si y sólo si $uv \in E$.

Definición. Un *clan* en una gráfica no dirigida $G(V, E)$ es una subgráfica inducida $G'(V', E')$ de G donde $V' \subseteq V$ y para todo par de vértices $u, v \in V'$, existe una arista $e \in E'$ que los conecta.

Definición. Una *sustitución clan* de P_3 se define como $P_{X,Y,Z} = P_3[K_X, K_Y, K_Z]$ donde P_3 es la trayectoria de tres vértices y cada K_s es la gráfica completa con s nodos.

Definición. Una gráfica G es *gráfica elemental* si y sólo si cada componente conexa de G es una substitución clan de P_3.

Definición. Dada una gráfica F, decimos que G *es libre de* F si ninguna de sus subgráficas inducidas es isomorfa a F. En este caso decimos que F es una *subgráfica inducida prohibida* de G.

2.3.1. Del alojamiento en registros a la coloración

En seguida se demuestra la relación que hay entre el alojamiento en registros libre de derrame con precoloración para programas elementales y la coloracion 1-2 alineada en las gráficas de interferencia correspondientes, haciendo uso de una demostración que se presenta en *Register allocation via coloring*[6].

Lema 2.3.1 *El alojamiento en registros libre de derrame con precoloración para un programa elemental P es equivalente a la extensión de una coloración 1-2 alineada para la gráfica de interferencia de P.*

Demostración En [6] se demuestra que el alojamiento en registros libre de derrame para un programa P es equivalente a colorear la gráfica de interferencia de P, donde cada color representa un registro físico en la máquina. Para extender el alojamiento en registros libre de derrame a una arquitectura con un banco de registros de tipo 1, se asignan pesos a cada variable en la gráfica de interferencias de tal forma que las variables que ocupan un registro tengan peso 1 y las que ocupan dos registros tengan peso 2. Para tomar en cuenta la precoloración se define $\phi(v) = r$ si el vértice v representa una variable precoloreada y el color r representa el registro físico asignado a esa variable. En otro caso $\phi(v)$ no está definido. ∎

2.3.2. Programas elementales y gráficas

Se mostrará en tres pasos que un programa elemental tiene una gráfica de interferencia elemental. Primero se da una caracterización de las sustituciones clan de P_3. Después se demostrará que una gráfica G es elemental si y sólo si G tiene una representación de intervalos elemental. Finalmente, se mostrará que la gráfica de interferencia de un programa elemental tiene una representación de intervalos elemental y por tanto es una gráfica elemental.

2.3.2.1. Una caracterización de las sustituciones clan de P_3

En seguida se dan algunas definiciones que se utilizan más adelante:

Definición. Un *conjunto independiente* es un subconjunto de n vértices de una gráfica G los cuales inducen $\overline{K_n}$ en G.

Definición. Un *conjunto independiente máximo* es un conjunto independiente máximo por contención. Denotamos como $\alpha(G)$ al número de vértices en un conjunto independiente máximo de G.

Sean $m(G)$ el número de clanes máximos de una gráfica no dirigida G. Claramente

$$\alpha(G) \leq m(G)$$

ya que debe haber $\alpha(G)$ clanes distintos que contienen a los elementos de un conjunto independiente máximo.

Definición. Una gráfica $G = (V, E)$ es es *trivialmente perfecta* si para todo $A \subseteq V$, la subgráfica inducida de G por A, G_A, cumple que $\alpha(G_A) = m(G_A)$.

Se dará una caracterización de las sustituciones clan en términos de sus subgráficas inducidas prohibidas. Comencemos la caracterización describiendo las clases de las *gráficas trivialmente perfectas*[8] para lo que necesitamos la definición de conjunto independiente máximo.

En una gráfica trivialmente perfecta el tamaño de los conjuntos independientes máximos es igual al número de los clanes máximos.

Para la demostración del lema 2.3.3 se hace uso del siguiente teorema demostrado en *Trivially perfect graphs*[8], por lo que se omite su demostración.

Teorema 2.3.2 *Una gráfica G es una gráfica trivialmente perfecta si y sólo si G no contiene subgráficas inducidas isomorfas a C_4 o P_4.*

Enunciamos ahora el lema que nos da la caracterización de las sustituciones clan de P_3.

Lema 2.3.3 *Una gráfica G es una sustitución clan de P_3 si y sólo si G no contiene subgráficas inducidas isomorfas a C_4, P_4 o $\overline{K_3}$*

Demostración ($\Rightarrow$) Sea G una sustitución clan de P_3 de la forma $P_{X,Y,Z}$. Nótese que G tiene a lo más uno o dos clanes máximos. Si G contiene un clan máximo, tenemos que G es de la forma $P_{\phi,Y,\phi}$ ya que $P_{X,Y,\phi}$ y $P_{\phi,Y,Z}$ son isomorfas a $P_{\phi,Y',\phi}$, y el conjunto independiente máximo tiene tamaño 1. Si G contiene dos clanes máximos, esos clanes deben ser $X \cup Y$ y $Y \cup Z$. En este caso el conjunto independiente máximo tiene dos vértices, un elemento de $X - Y$ y otro elemento de $Z - Y$. De esto, G es trivialmente perfecta. Por lo tanto G no contiene C_4 ni P_4 como subgráficas inducidas. Además como el máximo conjunto independiente de G tiene tamaño 1 o 2, $\overline{K_3}$ no puede ser una subgráfica inducida de G.

($\Leftarrow$) Si C_4 y P_4 no son subgráficas inducidas de G entonces G es trivialmente perfecta. Como $\overline{K_3}$ no es subgráfica inducida de G, su conjunto independiente máximo tiene 1 o 2 vértices. Si G no es conexa, tenemos que G consiste de dos clanes no conectados, por lo que $G = P_{X,\phi,Y}$. Si G es conexa puede tener uno o dos clanes máximos. En el primer caso tenemos que $G = P_{\phi,Y,\phi}$. En el segundo caso, tenemos que $G = P_{C_1-C_2,C_1\cap C_2,C_2-C_1}$, donde C_1 y C_2 son los clanes máximos de G. ∎

2.3.2.2. Una caracterización de las gráficas elementales

Una gráfica de intervalos es una gráfica de intersección de una familia de subintervalos sobre un intervalo de los números reales. La gráfica de intersección de trayectorias dirigidas de vértices en un árbol con raíz se llama *gráfica de trayectorias de vértices*, o *RDV*[2]. La familia gráficas de cuerdas incluye a la de gráficas *RDV* que a su vez incluye a las gráficas de intervalos. Ejemplares de estos conjuntos y sus contenciones se muestran en la figura 2.8.

Se denota un conjunto de n trayectorias dirigidas en un árbol dirigido T como $L = \{\overline{v_1}, \ldots, \overline{v_n}\}$. La gráfica *RDV* que corresponde a L es $G = (\{v_1, \ldots, v_n\}, E)$ donde $v_iv_j \in E$ si y sólo si $\overline{v_i} \cap \overline{v_j} \neq \varnothing$. Llamamos a L la representación de trayectorias de G. Como T es un árbol dirigido cada trayectoria $\overline{v_i}$ tiene un punto de inicio $s(\overline{v_i})$ y un punto final $e(\overline{v_i})$ bien definidos: $s(\overline{v_i})$ es el nodo de v_i más cercano a la raíz de T y $e(\overline{v_i})$ es el punto más lejano de la raíz.

[2]Del inglés *Rooted Directed Vertex path graph*

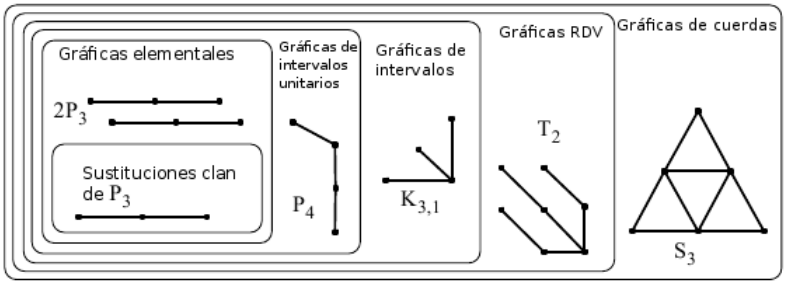


Figura 2.8: Contensiones en conjuntos de gráficas

*Definición.*Una gráfica tiene una *representación de intervalos elemental* si:

1. G es una gráfica RDV.
2. Si $uv \in E$, entonces $s(\overline{u}) = s(\overline{v})$ o $e(\overline{u}) = e(\overline{v})$.
3. Si $uv \in E$, entonces $\overline{u} \subseteq \overline{v}$ o $\overline{v} \subseteq \overline{u}$

Supongamos que $G = (V, E)$ tiene una representación de intervalos elemental. Probaremos algunas propiedades de G que se usarán en la demostración del lema 2.3.10, el cual será enunciado más adelante.

Proposición 2.3.4 *Si $a, b, c \in V$, $ab \in E$, $bc \in E$ y $ac \notin E$ entonces $(\overline{a} \cup \overline{c}) \subseteq \overline{b}$*

Demostración Es importante ver que $\overline{b} \not\subseteq \overline{a}$ ya que de lo contrario, como $ac \notin E$ tendríamos que $bc \notin E$, lo que es una contradicción. Dado que $ab \in E$ y que $\overline{b} \not\subseteq \overline{a}$ tenemos que $\overline{a} \subseteq \overline{b}$. Por simetría tenemos que $\overline{c} \subseteq \overline{b}$ por lo que $(\overline{a} \cup \overline{c}) \subseteq \overline{b}$. ∎

Proposición 2.3.5 *G no tiene a P_4 como subgráfica inducida.*

Demostración Por contrapositivo, supongamos que G tiene cuatro vértices x, y, z, w que inducen la trayectoria $\{xy, yz, zw\}$ en G. Por la proposición 2.3.4 tenemos que $(\overline{x} \cup \overline{z}) \subseteq \overline{y}$, en particular $\overline{z} \subseteq \overline{y}$. Similarmente tenemos que $(\overline{y} \cup \overline{w}) \subseteq \overline{z}$ y en particular $\overline{y} \subseteq \overline{z}$. Por lo tanto, $\overline{y} = \overline{z}$. Como $zw \in E$ y $\overline{y} = \overline{z}$, tenemos que $yw \in E$ contradiciendo que el conjunto $\{x, y, z, w\}$ induce a P_4 en G. ∎

Proposición 2.3.6 *Sea $C = (V_C, E_C)$ una componente conexa de G. Si $a, b \in V_C$ tal que $ab \notin E_C$ entonces $\exists v$ tal que $av \in E_C$ y $vb \in E_C$.*

Demostración De la proposición 2.3.5 tenemos que G no tiene a P_4 como subgráfica inducida, por lo que toda trayectoria mínima entre dos vértices conectados contiene una o dos aristas. Se tiene que $a, c \in V_C$, por lo que a y b están conectados. Como $ab \notin E_C$, tenemos una trayectoria de longitud mínima $\{av, vb\}$ para algún vértice $v \in V_C$. ∎

Proposición 2.3.7 *G no tiene a $K_{1,3}$ como subgráfica inducida.*

Demostración Sea L la representación de trayectorias dirigidas de G. Supongamos que G tiene cuatro vértices x, y, z, w que inducen a $K_{1,3} = \{xy, xz, xw\}$. Supongamos sin pérdida de generalidad que $s(\overline{x}) = s(\overline{y})$. Como G es una gráfica RDV tenemos que $e(\overline{x}) = e(\overline{z})$. Sin embargo, x y w interfieren por lo que w no puede compartir su punto de inicio con x pues interferiría con y, ni puede compartir su punto de fin con x pues interferiría con z. Por lo tanto $K_{1,3}$ no es subgráfica inducida de G. ∎

Proposición 2.3.8 *Si C es una componente conexa de G entonces $\overline{K_3}$ no es subgráfica inducida de C.*

Demostración Sea $C = (V_C, E_C)$ una componente conexa de G. Supongamos que existen $\{a, b, c\} \in V_C$ tal que $ab \notin V_C$, $ac \notin V_C$ y $bc \notin V_C$ Por la proposición 2.3.6 existe un vértice v_{ab} que es adyacente a a y a b y existe otro vértice v_{bc} que es adyacente a b y a c. Por la proposición 2.3.4 tenemos que en cualquier representación de trayectorias de G, $(\overline{a} \cup \overline{b}) \subseteq \overline{v_{ab}}$. Por lo tanto $\overline{b} \subseteq (\overline{v_{ab}} \cap \overline{v_{bc}})$; entonces $\overline{v_{ab}v_{bc}} \in E_C$, por lo que $\overline{v_{ab}} \subseteq \overline{v_{bc}}$ o $\overline{v_{bc}} \subseteq \overline{v_{ab}}$. Si lo primero sucede $\{a, b, c, v_{bc}\}$ induce a $K_{1,3}$ en G; si sucede lo segundo $\{a, b, c, v_{ab}\}$ induce a $K_{1,3}$ en G. Pero por la proposición 2.3.7 esto no puede suceder. ∎

A continuación se dan dos definiciones que se usan en la demostración de la siguente proposición.

Definición. Una *cuerda* es una arista que une dos vértices que no son adyacentes en un ciclo elemental.

Definición. Una gráfica es una *gráfica de cuerdas* si cada uno de sus ciclos de cuatro o más nodos tiene una cuerda.

Proposición 2.3.9 *C_4 no es una subgráfica inducida de G.*

Demostración Por definición las gráficas RDV son gráficas de cuerdas, las cuales no tienen a C_4 como subgráfica inducida. ∎

Lema 2.3.10 *Una gráfica G es una gráfica elemental si y sólo si G tiene una representación de intervalos elemental.*

Demostración ($\Rightarrow$) Sea G una gráfica con k componentes conexas tal que cada uno de ellos es una sustitución clan de P_3. Sea $P_{X,Y,Z}$ una de sus componentes conexas. Probaremos primero que $P_{X,Y,Z}$ tiene una representación de intervalos elemental.
Sea T un árbol dirigido isomorfo a $P_4 = (\{a, b, c, d\}, \{ab, bc, cd\})$, y sea a su raíz. Construiremos una gráfica elemental G_P isomorfa a $P_{X,Y,Z}$ utilizando los intervalos en T. Sea $\overrightarrow{v_1, v_2, \ldots, v_n}$ la trayectoria dirigida que comienza en el nodo v_1 y termina en el nodo v_n. Construimos la representación de intervalos elemental de $P_{X,Y,Z}$ de la siguiente manera: para cada $x \in X$, sea $\overline{x} = \overrightarrow{ab}$. Para toda $y \in Y$, sea $\overline{y} = \overrightarrow{abcd}$. Para cada $z \in Z$, sea $\overline{z} = \overrightarrow{cd}$. Mostremos ahora que G tiene una representación de intervalos elemental. Para cada componente conexa C_i de G, sea T_i el árbol dirigido que representa su trayectoria dirigida y sea r_i su raíz. Construyamos un árbol dirigido T como $r \cup T_i$, $1 \leq i \leq k$, donde r es un nuevo nodo que no es parte de ningún T_i. Las trayectorias dirigidas en cada rama de T cumplen con los requerimientos para tener una representación de intervalos elemental, por lo que T constituye una representación de intervalos elemental.

($\Leftarrow$) Supongamos que G tiene una representación de intervalos elemental. Para probar que G es una gráfica elemental basta demostrar que cada componente conexa de G es una sustitución clan de P_3. Tenemos que una caracterización mínima de las sustituciones clan de P_3 en términos de sus gráficas prohibidas consiste de C_4, P_4, y $\overline{K_3}$. C_4 no es subgráfica inducida de G por la proposición 2.3.9; P_4 no es subgráfica inducida de G por la proposición 2.3.5; $\overline{K_3}$ no es subgráfica inducida de cualquier componente conexa de G por la proposición 2.3.8. Por lo tanto, G es una gráfica elemental. ∎

*Definición.*Una *gráfica de intervalos unitarios* es una gráfica de intervalos que tiene una representación en la que todo intervalo tiene la misma longitud.

Corolario 2.3.11 *Una gráfica elemental es una gráfica de intervalos unitarios.*

Demostración Primero se demostrará que una sustitución clan es una gráfica de intervalos unitarios. Sea i un entero. Dada $P_{X,Y,Z}$ definimos una gráfica de intervalos unitaria I de la siguiente forma. Para cada $x \in X - Y$, sea $\overline{x} = [i, i + 3]$; para cada $y \in (Y - (X \cup Z))$, sea $\overline{y} = [i + 2, i + 5]$; y por cada $z \in Z - Y$, sea $\overline{z} = [i + 4, i + 7]$. Esos intervalos representan $P_{X,Y,Z}$ y constituyen una gráfica de intervalos unitarios.
Por la definición de gráfica elemental, tenemos que cada componente conexo de G es una sustitución clan de P_3. De cada componente conexo de G podemos construir gráficas de intervalos unitarios y por tanto juntarlas en una que represente una gráfica de intervalos unitarios de G. ∎

2.3.2.3. Programas elementales y gráficas de interferencia elementales

En seguida se describe cómo se puede obtener un programa elemental de programas ordinarios mediante la división de sus rangos de vida, y el renombramiento de sus variables. Pimero se dan las definiciones de programa estricto, programa simple y programa elemental. Después se hace mención del proceso para transformar un programa elemental a un programa estricto. Finalmente se dan algunas definiciones que serán de utilidad para demostrar que un programa elemental tiene una gráfica de interferencia elemental.

Definición. Sea $LR(v)$ el rango de vida de la variable v y $d(v)$ la instrucción que define a v. Un programa P es un *programa estricto* si todo camino en la gráfica de control de flujo de P desde el nodo de inicio hasta un uso de la variable v pasa por una de las definiciones de v.

Definición. Un programa P es un *programa simple* si P es estricto en su forma SSA y para cualquier variable v de P, $LR(v)$ contiene a lo más un punto de programa fuera del bloque básico que contiene a $d(v)$.

Definición. Para una variable v contenida en un bloque básico B en un programa simple, definimos $k(v)$ como la única instrucción fuera de B que usa a v o si v sólo se usa en B como la última instrucción en B. Cabe destacar que como P es simple, $LR(v)$ consiste de los puntos del programa en la única trayectoria de $d(v)$ a $k(v)$.

Definición. Un programa P producido por la gramática de la figura 2.9 está en su forma elemental si y sólo si tiene las siguientes propiedades:

1. P es un programa simple.
2. Si dos variables u, v de P interfieren, entonces $d(u) = d(v)$ o $k(u) = k(v)$.
3. Si dos variables u, v de P interfieren, entonces $LR(u) \subseteq LR(v)$ o $LR(v) \subseteq LR(u)$.

$$
\begin{array}{rcl}
P & ::= & S(L\phi(m,n)i * \pi(p,q)) * E \\
L & ::= & L_{start}, L_2 \ldots L_{end} \\
v & ::= & v_1, v_2, \ldots \\
r & ::= & \texttt{AX, AH, AL BX, } \ldots \\
o & ::= & \bullet \\
 & & \mid v \\
 & & \mid r \\
S & ::= & L_{start} : \pi(p,q) \\
E & ::= & L_{end} : halt \\
i & ::= & o = o \\
 & & \mid V(n) = V(n) \\
\pi(p,q) & ::= & M(p,q) = \pi V(q) \\
\phi(n,m) & ::= & V(n) = \phi M(m,n) \\
V(n) & ::= & (o_1, \ldots, o_n) \\
M(m,n) & ::= & V_1(n) : L_1, \ldots, V_m(n) : L_m
\end{array}
$$

Figura 2.9: Gramática de los programas elementales.

Podemos producir un programa elemental de un programa estricto como sigue:

- Insertar funciones ϕ al inicio de bloques básicos con múltiples predecesores;
- Insertar funciones π al final de bloques básicos con múltiples sucesores;
- Insertar copias paralelas entre instrucciones consecutivas en el mismo bloque básico.
- Renombrar variables dadas por las funciones ϕ, las funciones π y las copias paralelas.

Un programa elemental P generado por la gramática de la figura 2.9 es una secuencia de bloques básicos. Un bloque básico, representado con una etiqueta L, es una secuencia de instrucciones, empezando con una función ϕ y terminando con una función π. Se supone que un programa P tiene dos bloques básicos especiales: L_{start} y L_{end}, que son, respectivamente el primero y el último bloque que se visitan durante la ejecución de P. Las instrucciones ordinarias definen o usan un operando, como en $r_1 = v_1$; una instrucción como $v_1 = \bullet$ define una variable pero no usa otra variable o registro. Las copias paralelas se representan como $(v_1 \ldots v_n) = (v'_1 \ldots v'_n)$.

Con el fin de dividir los rangos de vida de las variables, los programas elementales usan funciones ϕ y funciones π. Las funciones ϕ son abstracciones usadas en la forma SSA para

unir los rangos de vida de las variables. Una asignación como:

$$(v_1, \ldots, v_n) = \phi[(v_{11}, \ldots, v_{n1}) : L_1, \ldots, (v_{1m}, \ldots, v_{nm}) : L_m]$$

contiene n funciones ϕ tal que $v_i \leftarrow \phi(v_{i1} : L_1, \ldots, v_{im} : L_m)$, donde cada v_{ij} pertenece al j-ésimo bloque básico.

Las funciones π son las funciones duales de las funciones ϕ, ya que las funciones π se encargan de copiar el valor de una variable en otras que se usarán cuando el flujo del programa se separe, y las funciones ϕ se encargan de calcular el valor de una variable cuando el flujo del programa converge. Una asignación de la siguiente forma:

$$[(v_{11}, \ldots, v_{n1}) : L_1, \ldots, (v_{1m}, \ldots, v_{nm}) : L_m] = \pi(v_1, \ldots, v_n)$$

asigna a la variable v_{ij} del bloque L_j el valor en v_i si el flujo del programa cae dentro del bloque L_j.

Decimos que un bloque básico L_i domina a otro bloque básico L_j si todo camino desde L_{start} a L_j pasa por L_i.

Definición. Un bloque básico L_d es el dominador inmediato de otro bloque L si L_d domina a L, y para cualquier otro bloque L_i en el programa, si L_i domina a L entonces L_i también domina a L_d. Todos los bloques básicos de un programa, excepto el bloque inicial, tienen un bloque dominador inmediato, el cual es único.

Definición. Construimos un árbol $T = (V, E)$ de tal forma que sus vértices son los bloques básicos de un programa y $L_iL_j \in E$ si y sólo si L_i es el bloque dominador inmediato de L_j. Este árbol se llama *árbol de dominancia* del programa.

Lema 2.3.12 *Un programa elemental tiene una gráfica de interferencia elemental.*

Demostración Sean P un programa elemental, $G = (V, E)$ su gráfica de interferencia y T_P el árbol de dominancia de P. Recordemos que $LR(v)$ consiste de los vértices en la única trayectoria que inicia en $d(v)$ y que termina en $k(v)$. Todos los vértices de esa trayectoria están dentro de un mismo bloque, excepto tal vez $k(v)$. Por lo tanto, todo vértice en esa trayectoria domina los siguientes vértices en la trayectoria y $LR(v)$ determina una trayectoria dirigida en T_P; entonces G es una gráfica RDV. Dada una variable v, sea $s(LR(v)) = d(v)$, y sea $e(LR(v)) = k(v)$. Los requisitos segundo y tercero en la definición de la página 33 se siguen inmediatamente del segundo y tercer requisito de la definición de programa elemental. ∎

2.3.2.4. Relación entre las gráficas elementales y las gráficas de interferencia de programas elementales

En seguida se demuestran dos lemas de importancia. Primero se demuestra que una sustitución clan de P_3 es la gráfica de interferencia de una secuencia de instrucciones. Después se demuestra que una gráfica elemental es la gráfica de interferencia de un programa elemental. Los resultados mencionados se usarán más adelante.

Lema 2.3.13 *Una sustitución clan de P_3 es la gráfica de interferencia de una secuencia de instrucciones.*

Demostración Sea $G = P_{X,Y,Z}$ una sustitución clan de P_3. Sea $m = |X|$, $n = |Y|$ y $p = |Z|$. Se construye una secuencia de $2(m+n+p)$ instrucciones $i_1 \dots i_{2(m+n+p)}$ que usa $m+n+p$ variables, tal que cada instrucción define o usa una variable. La secuencia de intrucciones se muestra a continuación:

$$
\begin{array}{llcll}
i_j & v_j & = & \bullet & j \in \{1 \dots n\} \\
i_{n+j} & v_{n+j} & = & \bullet & j \in \{1 \dots m\} \\
i_{n+m+j} & \bullet & = & v_{n+j} & j \in \{1 \dots m\} \\
i_{n+2m+j} & v_{n+m+j} & = & \bullet & j \in \{1 \dots p\} \\
i_{n+2m+p+j} & \bullet & = & v_{n+m+j} & j \in \{1 \dots p\} \\
i_{n+2m+2p+j} & \bullet & = & v_j & j \in \{1 \dots n\}
\end{array}
$$

La figura 2.10 ilustra las instrucciones y es fácil ver que $P_{X,Y,Z}$ es la gráfica de interferencia de la secuencia de instrucciones. ∎

Lema 2.3.14 *Una gráfica elemental es la gráfica de interferencia de un programa elemental.*

Demostración Sea G una gráfica elemental y sean $C_1 \dots C_n$ los componentes conexos de G. Cada C_i es una sustitución clan de P_3. Por el lema 2.3.13 tenemos que cada C_i es la gráfica de interferencia de una secuencia de instrucciones s_i. Podemos construir un programa elemental P con $n+2$ bloques básicos $B_{start}, B_1, \dots, B_n, B_{end}$, tal que B_{start} contiene solamente un salto a B_1, cada B_i consiste de s_i seguido de un salto a B_{i+1}, con $i \in \{1 \dots n-1\}$, y B_n consiste de s_n seguido de un salto a B_{end}. La gráfica de interferencia del programa construido es G. ∎

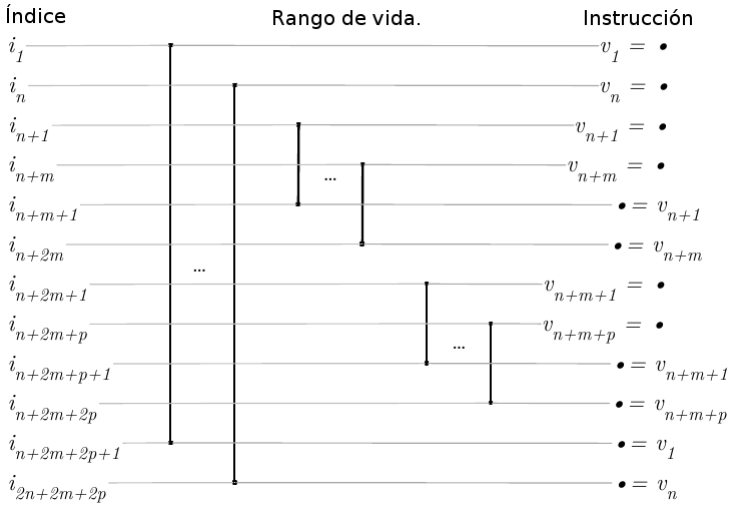


Figura 2.10: Programa elemental representando una sustitución clan de P_3

2.3.3. De la coloración 1-2 alineada a la resolución de rompecabezas

Ahora se mostrará que el problema de extensión de coloración 1-2 alineada y la solución de rompecabezas son equivalentes bajo reducciones en tiempo lineal. Primero se mostrará cómo reducir el problema de extensión de la coloración 1-2 alineada rellenando una gráfica y después se mostrará cómo mapear una gráfica a un rompecabezas.

En seguida se demuestran dos lemas que se utilizarán para demostrar que el alojamiento en registros libre de derrame con precoloración para un programa elemental es equivalente a resolver una colección de rompecabezas. Antes de ello se da la definición de gráfica $2k$ balanceada, el proceso de rellenado de una gráfica y el concepto de gráfica de comparabilidad.

Definición. Una gráfica $P_{X,Y,Z}$ *está* $2k$ *balanceada* si el peso de X iguala al peso de Z y el peso de $X \cup Y$ es $2k$, siendo el peso de un conjunto de vértices la suma de los pesos de sus elementos.

Una gráfica $P_{X,Y,Z}$ se rellena añadiendo conjuntos X' y Y' de vértices nuevos a $P_{X,Y,Z}$ de tal manera que $P_{(X\cup X'),Y,(Z\cup Z')}$ está $2k$ balanceada. La figura 2.11 muestra un ejemplo de rellenado. Después del rellenado cada clan máximo tiene peso 6.

Figura 2.11: Ejemplo de rellenado. Los vértices cuadrados tienen peso 2 y los demás peso 1.

Definición. Una *gráfica de comparabilidad* es una gráfica que conecta pares de elementos que son comparables en algún orden parcial.

Lema 2.3.15 *Para cualquier coloración parcial 1-2 alineada ϕ cuyo dominio es un subconjunto de $X \cup Y \cup Z$, $\langle 2k, P_{X,Y,Z}, \phi\rangle$ tiene solución si y sólo si $\langle 2k, P_{(X\cup X'),Y,(Z\cup Z')}, \phi\rangle$ tiene solución.*

Demostración ($\Rightarrow$) Sea $G = P_{X,Y,Z}$, y sea $G_P = P_{(X\cup X'),Y,(Z\cup Z')}$. Sea c una función que asocia vértices de G con colores y sea c_p una función que asocia vértices de G_P con colores. Sea v un vértice de G y sea v_P el vértice de G_P que corresponde a v. Sea S el conjunto de vértices de G_P creados por el rellenado. Dada c, definimos $c_P(v_P) = c(v)$. Después de colorear los vértices de $V(G_P)$ coloreamos los vértices en S. Sean M_1 y M_2 los clanes máximos de G. Como M_1 es una gráfica de comparabilidad, la coloración de M_1 usa un número de colores igual a su peso w_1 [9]. Lo mismo pasa con M_2. Sean Q_1 el clan máximo de G_P que corresponde a M_1, S_1 el conjunto de los vértices de Q_1 generados por el rellenado y V_1 el conjunto de los vértices de Q_1 que tienen vértices correspondientes en M_1. Definimos w_2, Q_2, V_2 y S_2 de la misma forma. El número de vértices de S_i es igual a $2k - w_i$, por la definición del rellenado. La coloración de los vértices V_i requiere w_i colores. Los vértices de S_i se colorean con los colores restantes. La coloración obtenida para S_1 y S_2 es válida debido a que los vértices de S_1 no son adyacentes a los de S_2 .

($\Leftarrow$) Dado c_P construimos c de la siguiente forma:

$$c(v) = c_P(v_P)$$

siempre que $v_P \in V(G)$. ∎

Ahora definimos una biyección F del problema de extensión de coloración 1-2 alineada para sustituciones clan de P_3 a la solución de rompecabezas. Veremos un tablero con k áreas como una tabla bidimensional de $2 \times 2k$, en la cual la i-ésima área consiste de los cuadradados con índices $(1, 2i)$, $(1, 2i + 1)$, $(2, 2i)$, $(2, 2i + 1)$.

Sea $\langle 2k, G, \phi\rangle$ un ejemplar del problema de extensión de coloración 1-2 alineada, donde G es una sustitución clan de P_3 $2k$ balanceada. Definimos un rompecabezas $F(\langle 2k, G, \phi\rangle)$ con k áreas y las siguientes piezas.

1. $\forall v \in X$ tal que el peso de v es 1, se agrega una pieza de tipo X y tamaño 1. Si $\phi(v)$ está definida y $\phi(v) = i$, entonces la pieza se coloca en el cuadrado $(1, i)$.

2. $\forall v \in X$ tal que el peso de v es 2, se agrega una pieza de tipo X y tamaño 2. Si $\phi(v)$ está definida y $\phi(v) = \{i, i+1\}$, entonces la pieza se coloca en el renglón superior del área i.

3. $\forall v \in Y$ tal que el peso de v es 1, se agrega una pieza de tipo Y y tamaño 2. Si $\phi(v)$ está definida y $\phi(v) = i$, entonces la pieza se coloca en la columna i-ésima.

4. $\forall v \in Y$ tal que el peso de v es 2, se agrega una pieza de tipo Y y tamaño 4. Si $\phi(v)$ está definida y $\phi(v) = \{i, i+1\}$, entonces la pieza se coloca en el área i.

5. $\forall v \in Z$ tal que el peso de v es 1, se agrega una pieza de tipo Z y tamaño 1. Si $\phi(v)$ está definida y $\phi(v) = i$, entonces la pieza se coloca en el cuadrado $(2, i)$.

6. $\forall v \in Z$ tal que el peso de v es 2, se agrega una pieza de tipo Z y tamaño 2. Si $\phi(v)$ está definida y $\phi(v) = \{i, i+1\}$, entonces la pieza se coloca en el renglón inferior del área i.

Dado que ϕ es una coloración parcial 1-2 alineada de G, tenemos que las piezas en la tabla no se sobreponen. Dado que G es $2k$ balanceada tenemos que las piezas tienen un tamaño total de $4k$ y que el tamaño total de las piezas de tipo X iguala al de las piezas de tipo Z. Es fácil ver que F es inyectiva y suprayectiva, por lo que F es una biyección. También es fácil ver que la aplicación tanto de F como de F^{-1} es en orden $O(k)$.

Lema 2.3.16 *La extensión de coloración 1-2 alineada para sustituciones clan de P_3 es equivalente a la solución de rompecabezas.*

Demostración ($\Rightarrow$) Reducimos la extensión de coloración 1-2 alineada a la solución de rompecabezas. Sea $\langle 2k, G, \phi\rangle$ un ejemplar del problema de extensión de coloración 1-2 alineada donde G es una sustitución clan de P_3. Podemos suponer sin pérdida de generalidad que G está $2k$ balanceada, si no es así la rellenamos. Usemos la biyección F definida anteriormente para construir el rompecabezas $F(\langle 2k, G, \phi\rangle)$. Supongamos que $\langle 2k, G, \phi\rangle$ tiene solución. La solución acomoda las piezas restantes en el tablero y con F^{-1} podemos construir una coloración 1-2 alineada de G que extiende a ϕ.

($\Leftarrow$) Reducimos la solución de rompecabezas a la coloración 1-2 alineada. Sea P_Z y usemos la reducción F^{-1} para construir un ejemplar del problema de extensión de coloración 1-2 alineada $F^{-1}(P_z) = \langle 2k, G, \phi \rangle$, donde G es una sustitución clan de P_3. Supongamos que P_Z tiene solución. La solución coloca todas las piezas en el tablero y podemos usar F^{-1} para definir una coloración 1-2 alineada de G que extiende a ϕ. Suponiendo que $F^{-1}(P_z)$ tiene solución, la solución extiende a ϕ a una coloración 1-2 alineada de G, por último usamos F para colocar todas las piezas en el tablero. ∎

Teorema 2.3.17 *El alojamiento en registros libre de derrame con precoloración para un programa elemental es equivalente a resolver una colección de rompecabezas.*

Demostración De los lemas 2.3.1, 2.3.12 y 2.3.14 se tiene que el alojamiento en registros con precoloración para un programa elemental es equivalente a la extensión de coloración 1-2 alineada para gráficas elementales. Por el lema 2.3.16 tenemos que la extensión de coloración 1-2 alineada para gráficas elementales es equivalente a resolver un conjunto de rompecabezas. Por lo tanto el alojamiento en registros libre de derrame con precoloración es equivalente a resolver un conjunto de rompecabezas. ∎

Pasamos ahora al siguiente capítulo, donde se toca el tema de la solución de rompecabezas de tipo 1.

Capítulo 3

Solución de rompecabezas de tipo 1

En este capítulo se pretende explicar cómo es que trabaja un programa solucionador de rompecabezas. Primero se define un lenguaje visual de programas de solución a rompecabezas. Después de explicar la semántica se observan pequeños cambios que hacen incorrecto al algoritmo.

Para que la propuesta funcione se tiene que *rellenar* al rompecabezas antes de comenzar el proceso. Si un rompecabezas tiene un conjunto de piezas con un área total menor al área total del tablero de rompecabezas, el rompecabezas se rellena añadiendo piezas de tipo X y tamaño 1 y piezas de tipo Z y tamaño 1, hasta que el área total de las piezas de tipo X se iguale con el área total de las piezas de tipo Z y el área total de todas las pieza sea $4k$, donde k es el número de áreas en el tablero.

3.1. Un lenguaje visual para programas de solución de rompecabezas

En esta sección se presenta un lenguaje visual para la descripción de programas solucionadores de rompecabezas. Primero se presentan las definiciones de área completa y programa. Después se presenta la gramática para generar programas solucionadores de rompecabezas. Luego se explica la semántica del lenguaje. Finalmente se trata la complejidad en tiempo de ejecución de algoritmos solucionadores de rompecabezas generados por la gramática del lenguaje.

Definición. Decimos que *un área está completa* cuando sus cuatro cuadrados están cubiertos por piezas.

Definición. Un *programa* es un conjunto de enunciados donde un enunciado puede ser una regla r o un enunciado condicional $r : s$. Los conceptos de regla y enunciado se tratan más adelante.

Como ya se dijo, el lenguaje para solucionadores de rompecabezas que se usa en este trabajo es visual. La gramática en la figura 3.1 define un lenguaje visual para programar solucionadores de rompecabezas de tipo 1.

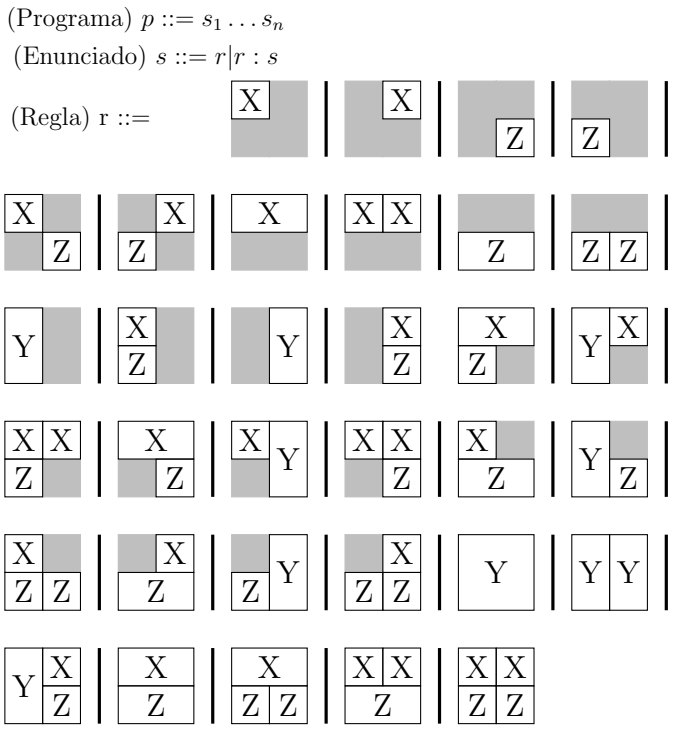


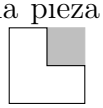
Figura 3.1: Lenguaje visual para programar solucionadores de rompecabezas

3.1.1. Reglas

Una regla explica cómo completar un área y se escribe como un diagrama de 2×2 con dos facetas: un *patrón*, que es un conjunto de áreas oscuras que muestran los cuadrados que ya deben estar llenos para la aplicación de la regla, y una *estrategia*, que es una descripción de cómo completar el área, incluyendo cuáles piezas usar y dónde colocarlas. Se dice que el

patrón de la regla empareja un área a si el patrón y el área tienen exactamente los mismos cuadrados llenos. Para una regla r y un área a, donde el patrón de r empareja a a, la aplicación de r a a se lleva a cabo si las piezas que se requieren por la estrategia de r están disponibles. El resultado es que las piezas requeridas por la estrategia de r se acomodan en a.

Por ejemplo, la regla 

tiene un patrón que consiste en el cuadrado superior derecho y su estrategia es colocar una pieza de tipo X y tamaño 1 en la esquina superior izquierda y una pieza de tipo Z y tamaño 2 en el renglón inferior. Si se aplica la regla anterior al área  y una pieza de tipo X y tamaño 1 y una pieza de tipo Z y tamaño 2 están disponibles, el resultado de la ejecución de la regla es que las dos piezas se colocan en el área y la regla tiene éxito; en otro caso la regla falla.

3.1.2. Enunciados

Ya se mostró cómo aplicar un enunciado de la forma r. Para un enunciado condicional $r : s$ se requiere que r y s tengan el mismo patrón, que se llamará el patrón de $r : s$. Para un enunciado condicional $r : s$ y un área a donde el patrón de $r : s$ se empareja con a, la aplicación de $r : s$ a a se ejecuta aplicando primero la regla r a a y solo en caso de que falle r se le aplica la regla s a a.

3.1.3. Programas

La ejecución de un programa $s_1 \ldots s_n$ sobre un rompecabezas R se ejecuta de la siguiente manera:

1. Para cada $i \in 1 \ldots n$

    a) Para cada área a en R tal que el patrón de s_i empareja a a

        1) Aplicar s_i a a
        2) Si la aplicación de s_i a a falla, terminar la ejecución y reportar falla.

3.1.4. Complejidad de tiempo

Es sencillo implementar la aplicación de una regla a un área en $O(1)$. Por lo tanto la ejecución de un programa en un tablero con k áreas toma $O(k)$.

3.2. Programa solucionador de rompecabezas

En esta sección se presenta un programa para solucionar rompecabezas y se muestran ejemplos de pequeños cambios al programa que hacen incorrecto el algoritmo. Finalmente se expone la demostración de correctud del algoritmo presentado.

La figura 3.2 muestra el programa solucionador de rompecabezas. Tiene 15 enunciados numerados y cada uno de ellos completa áreas con diferente patrón. Aun cuando el programa parece sencillo, en la mayoría de los casos el orden de los enunciados y el orden de las reglas en los enunciados condicionales es crucial para la correctud del algoritmo. En general el algoritmo trata de llenar los patrones en orden de mayor a menor restricción.

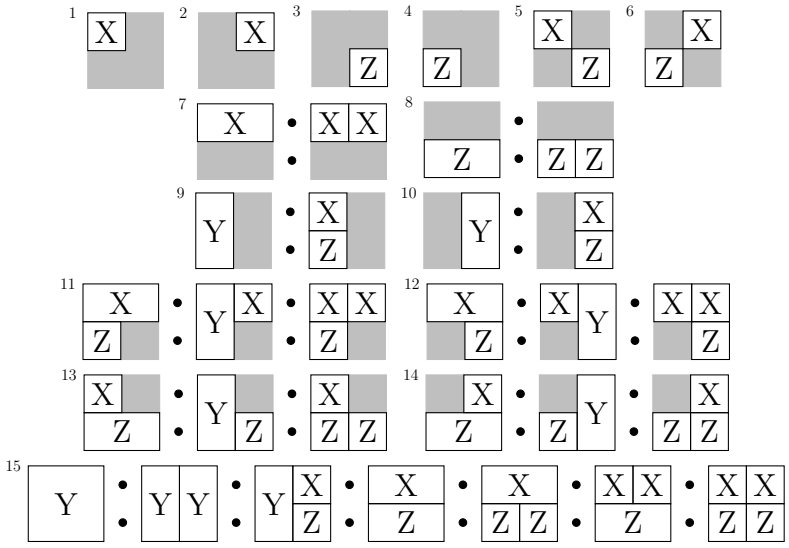
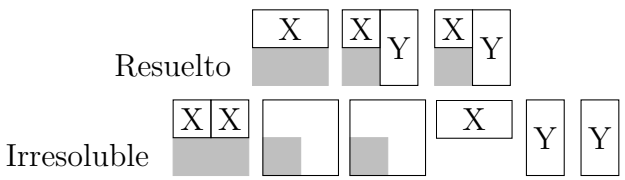


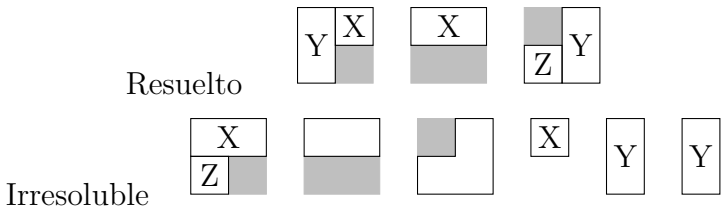
Figura 3.2: Programa solucionador de rompecabezas

Es importante mostrar que en el enunciado 7 el programa prefiere colocar piezas de tipo X y tamaño 2 en lugar de las de tipo X y tamaño 1, ya que al hacerlo de esta última manera, la aplicación del enunciado convertiría a un rompecabezas con solución en uno sin solución. Por ejemplo:

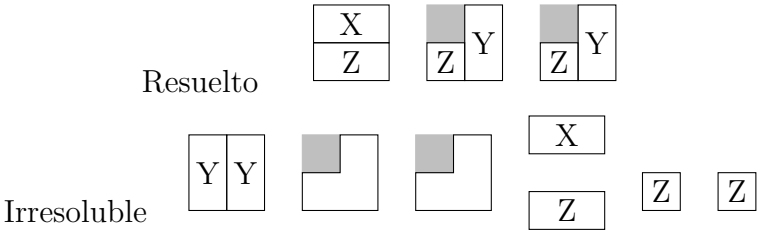


En los enunciados del 11 al 14 se prefieren fichas de tamaño 2 antes de las de tamaño 1 por el mismo motivo que en el enunciado 7.

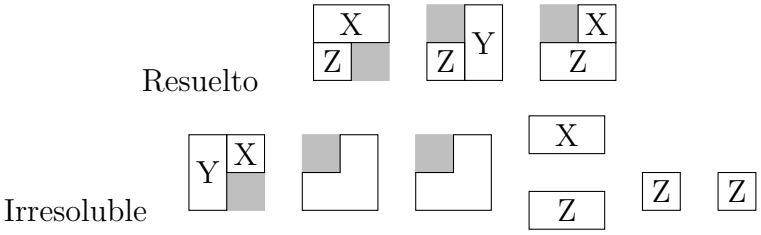
También es importante que los enunciados del 7 al 10 estén antes que los enunciados del 11 al 14, ya que si se cambia el orden de los bloques, nuevamente rompecabezas con solución se harían irresolubles. Por ejemplo:



Análogamente los enunciados del 11 al 14 deben estar antes del 15. Por ejemplo:



Igualmente es importante el orden de las reglas en el enunciado 11, ya que si cambiamos por ejemplo la primera y segunda regla, obtendremos el siguiente caso:



El programa hace elecciones similares en las reglas de la 12 a la 14.

3.3. Correctud del algoritmo

Para demostrar que el algoritmo es correcto primero probamos que la aplicación de una regla del algoritmo preserva la solubilidad de un rompecabezas. Después probamos por inducción sobre el tamaño del tablero que el algoritmo que se presenta encuentra solución para el rompecabezas si y sólo si la hay.

Lema 3.3.1 Preservación. *Sea P_z un rompecabezas y sea $i \in \{1, \ldots, 15\}$ el número de un enunciado del programa. Sea a un área de P_z tal que el patrón del enunciado i es igual al de a. Si P_z tiene solución, entonces la aplicación del enunciado i a a tiene éxito y resulta en un rompecabezas con solución.*

Demostración Notemos que el enunciado i contiene una regla por cada posible estrategia que puede usarse para completar a a. Sea Sl una solución de P_z. Dado que Sl completa a a, es fácil ver que la aplicación del enunciado i a a tiene éxito. Sin embargo, es posible que se use una estrategia distinta a la usada en Sl. Sea P'_z el resultado de la aplicación del enunciado i a a. Para mostrar que P'_z tiene solución se hará un análisis de casos sobre:

1. La estrategia usada por Sl para completar a a.

2. La estrategia usada por i para completar a a.

Nótese también que lo que interesa aquí es probar que si el enunciado i no usa la misma regla que la solución Sl para completar el área a, podemos construir un P'_z cuya primer área sea completada de la misma forma en que fue completada por i; esto se logrará haciendo un reacomodo de las piezas de Sl, de tal manera que se produzca un rompecabezas cuya primer área se complete de la misma forma que lo haría el enunciado i.

Podemos observar que para que la primer área de cualquier solución sea completada con una estrategia distinta a la del programa, podemos a lo más usar cuatro áreas diferentes de donde tomar las piezas, ya que son cuatro cuadrados de los que se compone el área.

Haciendo uso de la observación anterior, la prueba del lema se reduce a mostrar que para cualquier tablero con n áreas, $n \in \{1, 2, 3, 4, 5\}$, el algoritmo solucionador de rompecabezas es capaz de producir una solución al mismo tablero que el de Sl y con las mismas piezas usadas en Sl.

Para hacer eso codificamos un programa en *ProLog* que construye todas las posibles soluciones que existen, para tableros de los tamaños ya mencionados; dadas estas soluciones, el programa construye un conjunto de piezas, las cuales son las usadas para esa solución. Finalmente, se ejecuta el algoritmo de solución de rompecabezas para encontrar P'_z. El programa en *ProLog* mencionado se trata detalladamente en el apéndice A.

Podemos notar que el programa genera todas las posibles soluciones excepto por permutaciones de áreas, lo cual no afecta el resultado pues la ejecución del programa solucionador va en orden de acuerdo a los patrones del área que se quiere completar. ∎

Teorema 3.3.2 Correctud. *Un rompecabezas de tipo* 1 *tiene solución si y sólo si el programa puede solucionar el rompecabezas.*

Demostración Supongamos que P_z es un rompecabezas con solución. Debemos probar que el programa tiene éxito en la aplicación de los 15 enunciados. Del lema 3.3.1 e inducción sobre el tamaño del tablero, se tiene que en efecto, los 15 enunciados tienen éxito.

Caso base. Si $|P_z| = 1$. Este caso es trivial, ya que no hay forma en que se tenga una solución distinta a la obtenida por el programa solucionador de rompecabezas.

Hipótesis de inducción. Supongamos que la proposición se cumple para rompecabezas de tamaño k.

Paso inductivo. Si $|P_z| = k + 1$. Por el lema 3.3.1, y dado que P_z tiene solución, tenemos que se puede construir un rompecabezas P'_z cuya primer área sea completada de la misma manera que lo es en la ejecución del algoritmo. Sea $P''_z = P'_z - \{a_1\}$ donde a_1 es la primer área de P'_z. Como P''_z tiene tamaño k y P''_z tiene solución, por hipótesis de inducción se sigue que el programa solucionador de rompecabezas tiene éxito.

Por lo tanto un rompecabezas de tipo 1 tiene solución si y sólo si el programa puede solucionar el rompecabezas. ∎

De lo anterior se tiene el siguiente resultado:

Corolario 3.3.3 (Complejidad) *El alojamiento en registros libre de derrame con precoloración para un programa elemental P y $2k$ registros tiene solución en tiempo de $O(|P| \times k)$.*

Demostración Para cualquier programa elemental P se generan $|P|$ rompecabezas que pueden ser solucionados en tiempo lineal sobre el número de registros, por lo que el alojamiento en registros libre de derrame con precoloración para un programa elemental P y $2k$ registros tiene solución en tiempo de $O(|P| \times k)$. ∎

Con lo anterior queda demostrado que el algoritmo del programa solucionador de rompecabezas es correcto. En seguida se presentan las conclusiones del trabajo.

Conclusiones

En este trabajo se presentó una implementación basada en una abstracción del problema de alojamiento en registros que consiste en solucionar un tablero de rompecabezas, que representa una colección de registros de alguna arquitectura, con piezas de rompecabezas, que representan a las variables del programa transformado a su forma elemental. Esta abstracción aprovecha el que en la forma elemental de un programa los rangos de vida de las variables son muy cortos, lo que permite hacer un mejor uso de los registros físicos de la máquina.

En cuanto a la complejidad del solucionador del rompecabezas, tanto de tipo 0 como de tipo 1, tiene orden lineal sobre el número de registros de la arquitectura; lo anterior, sumado a que la cantidad de registros en las arquitecturas de computadoras es pequeño comparado con el número de variables del programa, da como resultado que esta abstracción sea una forma lo suficientemente buena y rápida para dar solución a este problema en un compilador real.

En la implementación se tomaron en cuenta varios casos que muchas veces llevan a un mejor comportamiento del módulo de alojamiento en registros. El rellenado del rompecabezas, que originalmente se hace de tal forma que el área total del rompecabezas sea exactamente igual al área que puedan cubrir las piezas, se implementó de forma tal que si las piezas cubren más área de la que tiene el tablero, el tablero se trata de resolver para que el proceso de derrame de valores a memoria tienda a tomar menos tiempo. Se tomó en cuenta también que algunas variables pueden tener una mayor prioridad para estar en un registro físico que otras, cuestión que no se trata en el trabajo original. La implementación cuenta con un sistema de preferencias para las piezas que trata de modelar que una variable preferentemente tiene que estar en un registro o una familia de registros dada, por ejemplo el caso de las variables que sirven de acumuladores. Sumado a todo lo anterior y como se muestra en el apéndice, el uso de la biblioteca implementada es muy sencillo ya que de no

ser necesaria ninguna optimización adicional, se pueden utilizar todos los componentes de la biblioteca casi sin necesidad de adaptación alguna.[1]

[1]Claramente la transformación a forma elemental del programa corre por cuenta del desarrollador, ya que la implementación de este proceso puede variar de compilador en compilador.

Apéndice A

Programa auxiliar para la demostración del lema de preservación

En este apéndice se presenta detalladamente el programa auxiliar en *ProLog* que se usa para la demostración del lema de preservación. En seguida se describen los predicados usados en el programa.

Primero se necesita una forma de representar una regla de un programa solucionador de rompecabeza. El predicado que modela esto es regla, que relaciona una regla con un identificador y el número de piezas de cada tipo que son usadas por la estrategia de la regla. La figura A.1 muestra la definición del enunciado regla donde se menciona el significado de cada uno de sus parámetros.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                    Definimos las reglas
%regla(Id,px,pX,py,pY,pz,pZ).
%La regla con identificador Id ocupa para su ejecución
%     px piezas de tipo X y tamaño 1.
%     pX piezas de tipo X y tamaño 2.
%     py piezas de tipo Y y tamaño 2.
%     pY piezas de tipo Y y tamaño 4.
%     pz piezas de tipo Z y tamaño 1.
%     pZ piezas de tipo Z y tamaño 2.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Figura A.1: Predicado regla

Como ejemplo del enunciado regla la figura A.2 muestra la representación de la regla del primer enunciado del programa solucionador. A esta regla se le asigna el identificador 1 y para su aplicación se necesita una pieza de tipo X y tamaño 1. Su estrategia consiste en colocar la pieza mencionada en el cuadro superior izquierdo del área.

```
1  % ___  ___
2  %|   |   |
3  %| X | # |
4  %|___|___|
5  %|   |   |
6  %| # | # |
7  %|___|___|
8  %
9  regla(1,1,0,0,0,0,0).
```

Figura A.2: Representación de la regla cuyo patrón consiste en el cuadro superior derecho y el renglón inferior del área. La estrategia de la regla consiste en colocar una pieza de tipo X y tamaño 1 en el cuadro superior izquierdo

Otro ejemplo del predicado regla se muestra en la figura A.3 cuyo patrón consiste en el cuadro inferior izquierdo del área. La estrategia consiste en colocar una pieza de tipo Y y tamaño 2 en la columna derecha del área y una pieza de tipo X y tamaño 1 en el cuadro superior izquierdo del área.

```
1  % ___  ___
2  %|   |   |
3  %| X |   |
4  %|___| Y |
5  %|   |   |
6  %| # |   |
7  %|___|___|
8  %
9  regla(19,1,0,1,0,0,0).
```

Figura A.3: Representación de la regla cuyo patrón consiste en el cuadro inferiror izquierdo del área. La estrategia del área consiste en colocar una pieza de tipo Y y tamaño 2 en la columna derecha del área y una pieza de tipo X y tamaño 1 en el cuadro superior izquierdo de la misma

Para representar los patrones se hace uso del predicado patron. Este predicado representa un patrón mediante la agrupación de los cuatro valores que representan los cuadrados del renglón superior e inferior del área en órden de aparición. También relaciona el patrón

con las reglas de acuerdo al identificador declarado en el predicado regla. En la figura A.4 muestra la representación del patrón que consiste en la esquina superior derecha y el renglón inferior del área, así mismo relaciona el patrón con la regla que tiene identificador 1.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Aquí se definen los diferentes tipos de patrones que
%hay en una arquitectura con rompecabezas de tipo 1.
%El cero significa libre y el uno significa ocupado.
%En las figuras aparecen con # los ocupados.
%El predicado patron(Ids,areaR(X,Y,Z,W)) relaciona a
%los id de las reglas con el patrón representado.
% ___  ___
%|   |   |
%|   | # |
%|___|___|
%|   |   |
%| # | # |
%|___|___|
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
patron([1],areaR(0,1,1,1)).
```

Figura A.4: Predicado patron

La figura A.5 muestra el predicado solucion el cual construye las posibles soluciones de un tamaño dado basándose en el reacomodo de las reglas definidas. Lo anterior es correcto debido a que las estrategias de las reglas definen la única forma de acomodar las piezas en un área con cierto patrón, excepto por el caso mostrado en la figura A.6 cuyas piezas se pueden reacomodar para igualar la estrategia de la regla mostrada en la misma figura.

```
solucion(1,[Id]):-regla(Id,_,_,_,_,_,_).
solucion(N,XS):-N>0,
                N1 is N-1,
                solucion(N1,[Id|Xs]),
                regla(Id2,_,_,_,_,_,_),
                Id2>=Id,
                append([Id2],[Id|Xs],XS).
```

Figura A.5: Predicado solucion

En la figura A.7 se presenta la definición y significado del predicado generabolsa. El predicado se encarga de calcular el número de piezas usadas por una solución dada. En

```
1  %  Acomodo distinto     Estrategia de regla
2  %        ---  ---             ---  ---
3  %       |   |   |           |   |   |
4  %       | X |   |           |   | X |
5  %       |---| Y |           | Y |---|
6  %       |   |   |           |   |   |
7  %       | Z |   |           |   | Z |
8  %       |---|---|           |---|---|
9  %
```

Figura A.6: El único caso de acomodo distinto de piezas y su análogo en las estrategias de las reglas

este predicado el primer parámetro es una lista con los identificadores de las reglas que se aplican para la solución, el segundo parámetro es una bolsa que guarda la configuración de partida y, finalmente, el tercer parámetro es otra bolsa que agrupa los números totales de cada tipo de pieza usados por la solución. Una *bolsa* es una estructura que agrupa el número de piezas de cada tipo y tamaño.

La figura A.8 presenta el predicado tablero, que construye el tablero correspondiente a la lista de identificadores de regla usados por una solución. El tablero consiste en una lista de estructuras areaR. Un areaR es una representación de las áreas del tablero que modela si un cuadro del área ya está ocupado o no.

Para modelar los distintos enunciados del programa solucionador de rompecabezas se presentan los predicados enunciadoN, con $N \in \{1, 2, \ldots, 15\}$. Un ejemplo de estos predicados es la figura A.9 que muestra el código en *ProLog* para el enunciado 7 del programa solucionador.

Es necesaria una representación del programa solucionador en *ProLog*. El predicado programa se encarga de ejecutar los enunciados del programa solucionador, y únicamente si la aplicación de todos tiene éxito y al final de su ejecución no queda ninguna pieza que acomodar, el predicado tiene éxito. La programación del predicado mencionado se muestra en la figura A.10 de las páginas siguientes.

Para ejecutar las pruebas para los tableros de rompecabezas con N áreas, se hace uso de los predicados prueba$N$, cuya función es precisamente construir todas las posibles soluciones a un tablero de rompecabezas de tamaño N y escribir las soluciones correspondientes que resultan de ejecutar el programa solucionador. Es importante notar que para la construcción de todas las soluciones se agrega una instrucción fail al final de cada predicado prueba$N$. Lo que se busca es alguna solución de un tablero de rompecabezas que no la pueda solucionar el programa, y si lo anterior sucede, escribirla en el archivo *Salida*. Si después de

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% generabolsa(
%       Ids,
%       bolsa(Px1,PX1,Py1,PY1,Pz1,PZ1),
%       bolsa(Px2,PX2,Py2,PY2,Pz2,PZ2))
% El predicado genera la bolsa de piezas que se ocupan
% en la lista de identificadores de reglas dado.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

generabolsa(
        [],
        bolsa(Px,PX,Py,PY,Pz,PZ),
        bolsa(Px,PX,Py,PY,Pz,PZ)).

generabolsa([Id|Ids],bolsa(Px,PX,Py,PY,Pz,PZ),
        bolsa(PxR,PXR,PyR,PYR,PzR,PZR)):-
        generabolsa(
                Ids,
                bolsa(Px,PX,Py,PY,Pz,PZ),
                bolsa(PxP,PXP,PyP,PYP,PzP,PZP)),
        regla(Id,Pxr,PXr,Pyr,PYr,Pzr,PZr),
        PxR is PxP + Pxr,
        PXR is PXP + PXr,
        PyR is PyP + Pyr,
        PYR is PYP + PYr,
        PzR is PzP + Pzr,
        PZR is PZP + PZr.
```

Figura A.7: Predicado generabolsa

la ejecución de los predicados prueba$N$ el archivo está vacío, entonces podemos concluir que si un tablero de tamaño N tiene solución con ciertas pieza de rompecabezas, entonces el programa solucionador encuentra una solución del mismo tablero de rompecabezas usando las mismas piezas. La figura A.11 muestra un ejemplo de los predicados mencionados para los tableros de tamaño 5.

Para probar lo que queremos se tiene el predicado demostracion, que hace uso de los predicados prueba$N$. La figura A.12 muestra la programación del enunciado demostracion. Después de la ejecución del enunciado, el archivo *Salida* está vacío, que es justamente el resultado que se espera.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%    tablero(Ids, tablero)
%    La relación tablero Indica que la lista de Ids dada
%    corresponde con un tablero dado.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
tablero([],[]).
tablero([Id|Ids],[P|Ps]):-patron(IdsR,P),
        member(Id,IdsR),tablero(Ids,Ps).
```

Figura A.8: Predicado tablero

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                Aplicación del enunciado 7
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

enunciado7([],Bolsa,[],Bolsa).

% Aplicación de la regla 7.1
enunciado7([areaR(0,0,1,1)|Areas],
     bolsa(Px,PX,Py,PY,Pz,PZ),
     [7|AreasR],bolsa(PxR,PXR,Py,PY,Pz,PZ)):-
     enunciado7(Areas,bolsa(Px,PX,Py,PY,Pz,PZ),AreasR,
     bolsa(PxR,PXP,Py,PY,Pz,PZ)),PXR is PXP - 1,PXR >= 0.

% Aplicación de la regla 7.2
enunciado7([areaR(0,0,1,1)|Areas],
     bolsa(Px,PX,Py,PY,Pz,PZ),
     [8|AreasR],bolsa(PxR,PXR,Py,PY,Pz,PZ)):-
     enunciado7(Areas,bolsa(Px,PX,Py,PY,Pz,PZ),AreasR,
     bolsa(PxP,PXR,Py,PY,Pz,PZ)),PxR is PxP - 2,PxR >= 0.

enunciado7([A|Areas],
     bolsa(Px,PX,Py,PY,Pz,PZ),[A|AreasR],
     bolsa(PxR,PXR,PyR,PYR,PzR,PZR)):-
     not(A = areaR(0,0,1,1)),
     enunciado7(Areas,bolsa(Px,PX,Py,PY,Pz,PZ),
     AreasR,bolsa(PxR,PXR,PyR,PYR,PzR,PZR)).
```

Figura A.9: Predicado enunciado7

```
programa(Areas,Bolsa,Solucion):-
        enunciado1(Areas,Bolsa,Areas1,Bolsa1),
        enunciado2(Areas1,Bolsa1,Areas2,Bolsa2),
        enunciado3(Areas2,Bolsa2,Areas3,Bolsa3),
        enunciado4(Areas3,Bolsa3,Areas4,Bolsa4),
        enunciado5(Areas4,Bolsa4,Areas5,Bolsa5),
        enunciado6(Areas5,Bolsa5,Areas6,Bolsa6),
        enunciado7(Areas6,Bolsa6,Areas7,Bolsa7),
        enunciado8(Areas7,Bolsa7,Areas8,Bolsa8),
        enunciado9(Areas8,Bolsa8,Areas9,Bolsa9),
        enunciado10(Areas9,Bolsa9,Areas10,Bolsa10),
        enunciado11(Areas10,Bolsa10,Areas11,Bolsa11),
        enunciado12(Areas11,Bolsa11,Areas12,Bolsa12),
        enunciado13(Areas12,Bolsa12,Areas13,Bolsa13),
        enunciado14(Areas13,Bolsa13,Areas14,Bolsa14),
        enunciado15(Areas14,Bolsa14,Solucion,bolsa
            (0,0,0,0,0,0)),!.
```

Figura A.10: Predicado programa

```
prueba5:-solucion(5,Ids),tablero(Ids,Areas),
        generabolsa(Ids,bolsa(0,0,0,0,0,0),Bolsa),
        not(programa(Areas,Bolsa,_)),
        write(Ids),nl,
        write('No tiene solución'),nl,nl,
        fail.
```

Figura A.11: Predicado demostracion

```
demostracion:-tell('Salida'),
        not(prueba1),
        not(prueba2),
        not(prueba3),
        not(prueba4),
        not(prueba5),
        told.
```

Figura A.12: Predicado demostracion

Apéndice B

Modelo de implementación

Para este trabajo se desarrolló el solucionador de rompecabezas en `C++` . La solución se incorporó a una biblioteca llamada `libpuzzle` . En este apéndice se dará una revisión del diseño y uso de la biblioteca.

B.1. Diagrama de clases

El diagrama de clases para la biblioteca desarrollada se muestra en la figura B.1. Podemos observar las relaciones de herencia y dependencia de las clases involucradas en el modelo.

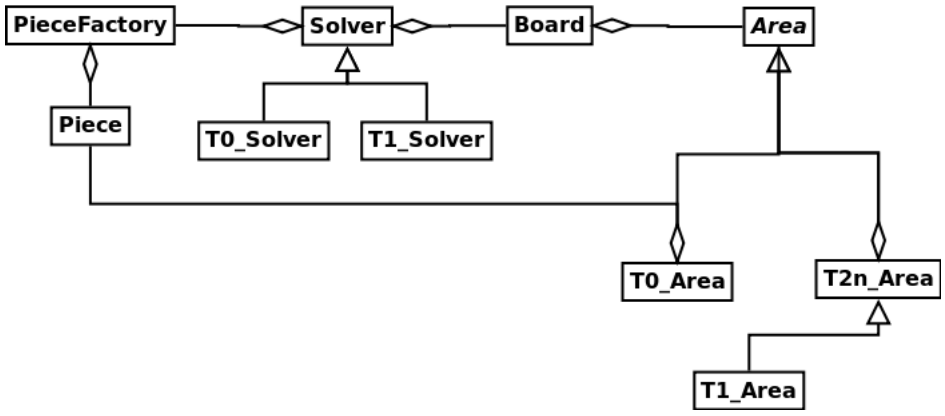


Figura B.1: Diagrama de clases de la biblioteca

B.2. Especificación por clases

En esta sección se detalla cada clase involucrada en la implementación de la biblioteca, el papel que juegan sus principales atributos y métodos, así como sus relaciones con las demás clases.

B.2.1. Piece (Pieza)

Clase que representa a una variable del programa en forma elemental, es la clase principal de todo el modelo, ya que todas las demás dependen principalmente de ella.

La figura B.2 muestra el diagrama de clase de la clase Piece. El atributo `var_info` es un apuntador a la información de la variable y el atributo `priority` da el soporte para implementaciones donde la prioridad de las variables se tome en cuenta para el proceso de alojamiento en registros.

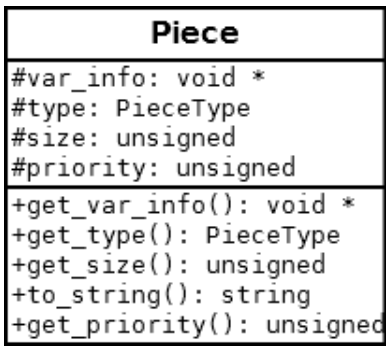


Figura B.2: Diagrama de clases de `Piece`

B.2.2. Area (Área)

Clase abstracta que representa un registro físico en la máquina.

La figura B.3 muestra el diagrama de clase de la clase `Area`; haremos énfasis en el atributo `register_info` , que es un apuntador a la información que representa al registro, y el atributo `preferences` , que es una lista de preferencias que se seguirán para la selección de las piezas en el alojamiento en registros.[1] El atributo `type` es el tipo del área. Su método `to_string` recibe dos parámetros de tipo `DATA_STRING` , que es un apuntador a función que toma como parámetro un apuntador y regresa una representación del

[1] Ver la documentación de la clase PieceFactory.

objeto apuntado. El método `allocate` se encarga de colocar una pieza en la columna dada.[2] El método `get_pattern` regresa una representación numérica del área. Los métodos `get_free_x_number` y `get_free_z_number` regresan el número de espacios vacíos de tipo x y z respectivamente.

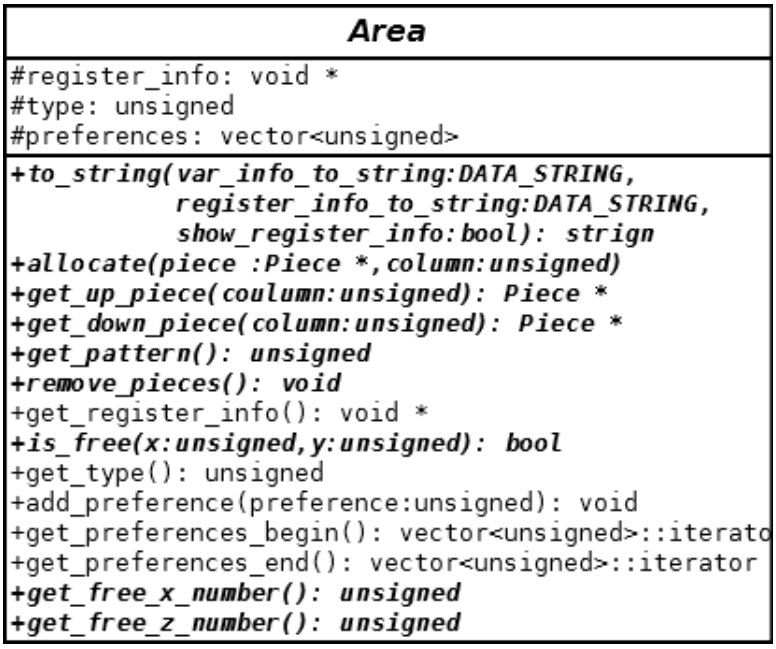


Figura B.3: Diagrama de clases de `Area`

B.2.3. T0_Area (Área de tipo 0)

Especialización de la clase `Area` . Está conformada por un apuntador a la pieza que ocupa el lugar superior y otro a una pieza que ocupa el lugar inferior. La figura B.4 muestra el diagrama de clase de `T0_Area` en el que se ve más a detalle el diseño de la clase. Los atributos `top` y `bottom` son apuntadores a objetos de la clase `Piece` . Se implementan todos los métodos abstractos de `Area` .

[2]El comportamiento depende de la clase. Para documentación más precisa consulte los archivos de encabezado.

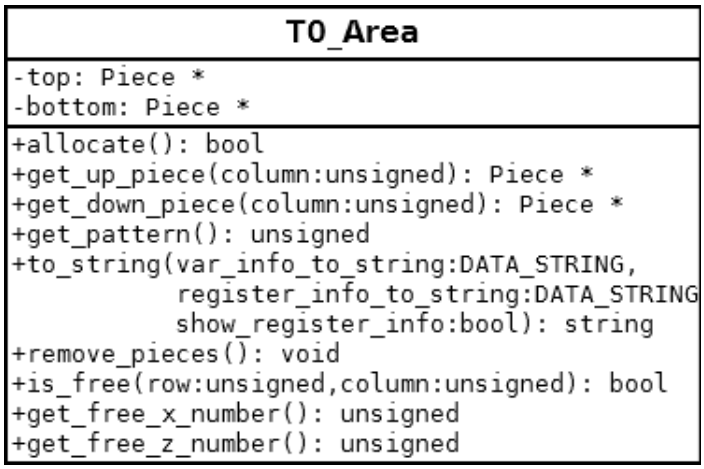


Figura B.4: Diagrama de clase de `T0_Area`

B.2.4. T2n_Area (Área de tipo 2^n)

Clase abstracta que representa a las áreas de 2^n columnas. Sus atributos son dos apuntadores a objetos de la clase `Area` , uno del lado derecho y otro del izquierdo. En implementaciones de `Areas` de tipo 2^{n+1} debe haber apuntadores a objetos de clase `Area` de tipo 2^n. Es importnte señalar que el método `get_pattern` ya está implementado para todas sus subclases. La figura B.5 muestra el diagrama de clase de `T2n_Area` .

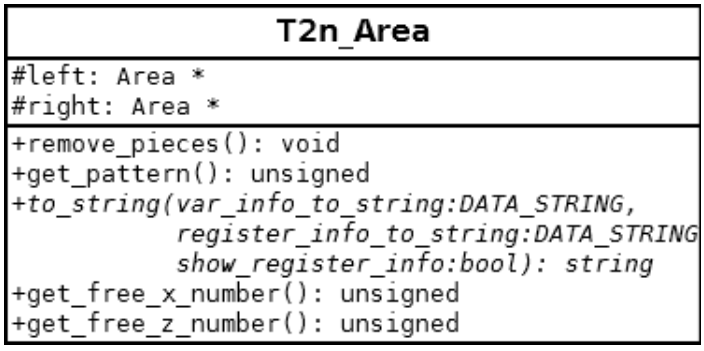


Figura B.5: Diagrama de clase de `T2n_Area`

B.2.5. T1_Area (Área de tipo 1)

El diagrama de clase de `T1_Area` , subclase de `T2n_Area` , se muestra en la figura B.6. Como se puede observar, las subclases de `T2n_Area` sólo deben implementar los métodos `allocate` , `get_down_piece` , `get_up_piece` e `is_free` .

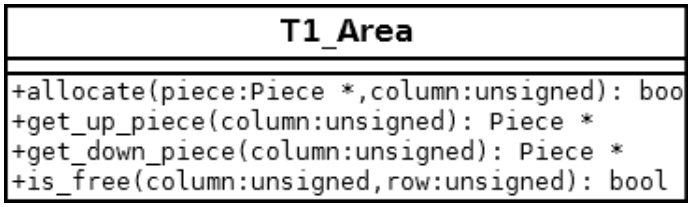


Figura B.6: Diagrama de clase de `T1_Area`

B.2.6. Board (Tablero)

Clase que modela un tablero de rompecabezas. Como principal atributo podemos notar al vector `areas` en el que se alojan las áreas correspondientas al tablero. Es importante ver que el método `to_string` recibe como parámetros, dos `DATA_STRING` , cada uno de los cuales se encargará de dar una "buena" representación de la información de la variable representada por la pieza contenida en el área o la del registro que representa el área.

La figura B.7 muestra el diagrama de clase de `Board`.

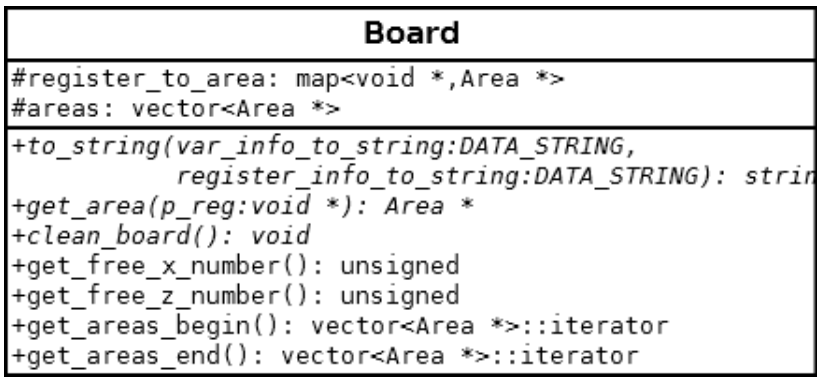


Figura B.7: Diagrama de clase de `Board`

B.2.7. PieceFactory (Fábrica de piezas)

Clase que agrupa a todas las piezas que se utilizan para rellenar el rompecabezas. Como principales atributos tenemos a `x_pieces` , `y_pieces` y `z_pieces` . La figura B.9, de la página siguiente, muestra el diagrama de clase de `PieceFactory` . Podemos ver que los atributos mencionados son mapeos de dos niveles. El primer nivel corresponde a la preferencia de la que ya se había hecho mención y el segundo nivel corresponde al ancho de la pieza que puede ser 2^n. La implementación tiene definida una preferencia por omisión llamada `DEFAULT_PREFERENCE` definida en `preferences.h` , la cual debería usarse al menos para las piezas agregadas en el proceso de rellenado.

B.2.8. Solver (Solucionador)

Las subclases de esta clase asbtracta serán las encargadas de implementar el método `solve` cuyo algoritmo cambiará según el tipo de área que se tenga. Actualmente están disponibles en la solución los algoritmos de solución para rompecabezas de tipo 0 y los de tipo 1. La figura B.8 muestra el diagrama de clase de `Solver` . Podemos notar principalmente los atributos `board` y `pieces` que son apuntadores a ejemplares de la clase `Board` y `PieceFactory` respectivamente.

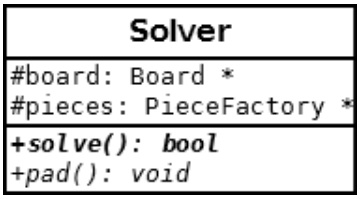


Figura B.8: Diagrama de clase de `Solver`

B.2.9. T0_Solver y T1_Solver (Solucionadores de tipo 0 y 1)

Las subclases de la clase `Solver` sólo implementan el método `solve` . Sin embargo, si es requerido también se puede sobreescribir el método `padd` . La figura B.10, en la página 68, muestra los diagramas de clase correspondientes.

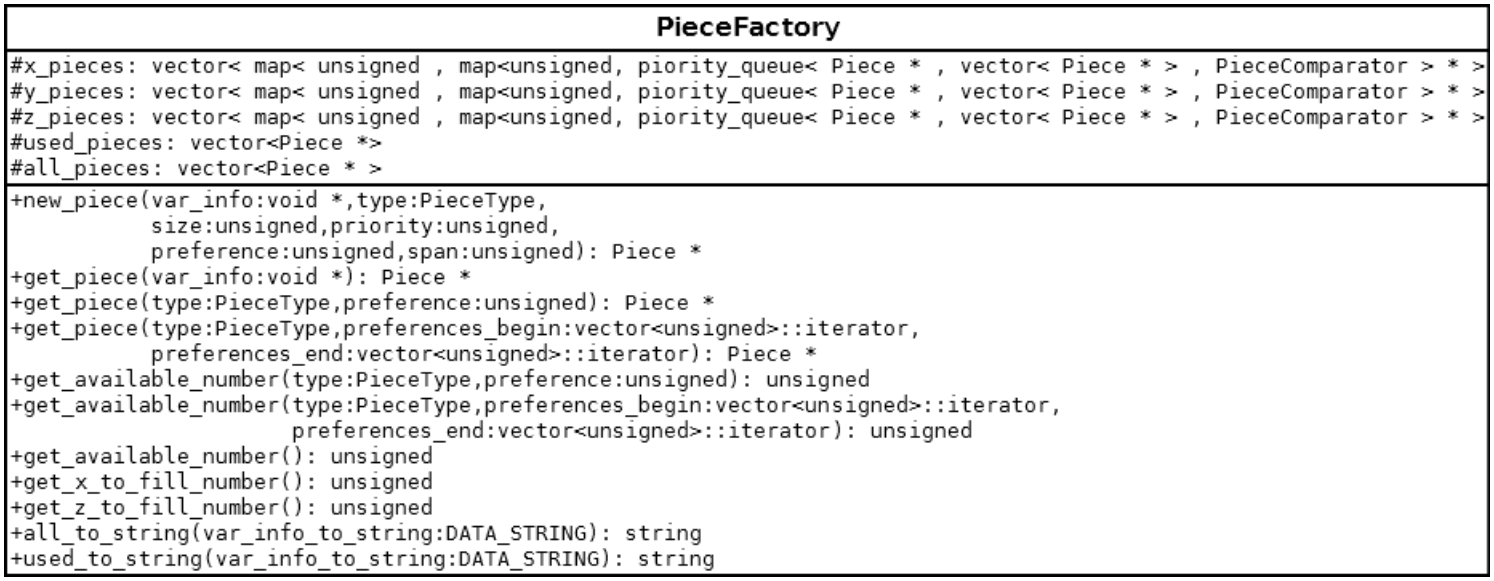


Figura B.9: Diagrama de clase de `PieceFactory`

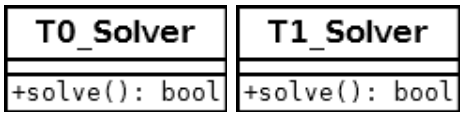


Figura B.10: Diagrama de clase de `T0_Solver` y `T1_Solver`

B.3. Implementación de un solucionador para x86

En esta sección daremos una revisión de manera sencilla a la forma en que se puede utilizar la biblioteca desarrollada para implementar un alojador de registros para `x86` . Tenemos que tener como premisa que la solución desarrollada es independiente del compilador, así que el proceso de transformación del programa a su forma elemental va por cuenta del desarrollador del compilador.

B.3.1. Estructura de directorios

Primero necesitamos construir una estructura adecuada para el proceso de compilación. En el caso de este ejemplo, tendremos la siguiente estructura de directorios.

```
/
  include/
  include_x86/
  src_x86/
  lib/
  Makefile
```

Se decidió separar los encabezados de la biblioteca y del proyecto solamente para hacer énfasis en el poco código que puede potencialmente ser necesario para usar la biblioteca.

B.3.2. Implementación

Para hacer uso de la biblioteca se necesitan sólo dos cosas.

1. Implementar las funciones que se utilizarán en los métodos `to_string()` .
2. Implementar el tablero correspondiente a la arquitectura deseada.

La implementación de la primera depende mucho de la representación de la información que se desea reflejar; en el caso de este ejemplo la información es una cadena, así que la función es muy simple.

Para la segunda parte tenemos que implementar mínimamente el consructor y destructor. En nuestro caso se tienen que implementar el tablero de tipo 0 para los registros `BP` , `SI` , `DI` y `SP` y otro de tipo uno para los regsitros `AX` , `BX` , `CX` y `DX` .[3]

Finalmente, en el ejemplo para `x86` se tiene un *main* que simula aleatoriamente posibles tableros y piezas de rompecabezas con precoloración. Como salida se obtienen las representaciones gráficas del tablero antes de rellenarse y después de rellenarse.

[3]Los códigos tanto para el tablero de tipo 0 como para el de tipo 1 se encuentran en el disco anexo.

Bibliografía

[1] http://compilers.cs.ucla.edu/fernando/projects/puzzles/.

[2] David H. Albonesi, Margaret Martonosi, David I. August, and José F. Martínez, editors. *42st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42 2009), December 12-16, 2009, New York, New York, USA*. ACM, 2009.

[3] C. Scott Ananian. The static single assignment form. Master's thesis, Massachusetts Institute of Technology, Sep 1999.

[4] Preston Briggs, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Coloring heuristics for register allocation. In Wexelblat [17], pages 275–284.

[5] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst.*, 16(3):428–455, 1994.

[6] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Comput. Lang.*, 6(1):47–57, 1981.

[7] Cooper Keith D. and Torkzon Linda. *Engineering a Compiler*. Elsevier, 2012.

[8] Martin Charles Golumbic. Trivially perfect graphs. *Discrete Mathematics*, 24:105–108, 1978.

[9] Martin Charles Golumbic. *Algorithmic Graph Theory and Perfect Graps*. Annals of discrete mathematics. Elsevier, 2004.

[10] Rajiv Gupta and Saman P. Amarasinghe, editors. *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*. ACM, 2008.

[11] Andreas Moshovos, J. Gregory Steffan, Kim M. Hazelwood, and David R. Kaeli, editors. *Proceedings of the CGO 2010, The 8th International Symposium on Code Generation and Optimization, Toronto, Ontario, Canada, April 24-28, 2010*. ACM, 2010.

[12] Fernando Magno Quintão Pereira. *Register Allocation by Puzzle Solving*. PhD thesis, University of California, Los Angeles, 2008.

[13] Fernando Magno Quintão Pereira and Jens Palsberg. Register allocation by puzzle solving. In Gupta and Amarasinghe [10], pages 216–226.

[14] Hongbo Rong. Tree register allocation. In Albonesi et al. [2], pages 67–77.

[15] Peter Steenkiste and John L. Hennessy. A simple interprocedural register allocation algorithm and its effectiveness for lisp. *ACM Trans. Program. Lang. Syst.*, 11(1):1–32, 1989.

[16] Aho Alfred V. et al. *Compiladores: principios, técnicas y herramientas*. Pearson, 2008.

[17] Richard L. Wexelblat, editor. *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation (PLDI), Portland, Oregon, USA, June 21-23, 1989*. ACM, 1989.

[18] Christian Wimmer and Michael Franz. Linear scan register allocation on ssa form. In Moshovos et al. [11], pages 170–179.