“Photoshop Lógico”

Laboratorio 2 Paradigmas

Estudiante: Bastian Brito Garrido (21.165.824-k)
Profesor: Roberto Gonzalez Ibanez
Fecha de entrega: 03-11-2022
Asignatura: Paradigmas de la programación

Índice

Descripción breve del problema

Durante el semestre trabajaremos al respecto de la edición de imágenes, y nuestro objetivo es hacer una especie de photoshop, pero simplificado. En esta ocasión, el segundo laboratorio usará el paradigma lógico para implementar ciertos hechos, reglas y cláusulas que podremos aplicar para la creación o modificación de imágenes, tales como: invertir la imagen, recortar la imagen, comprimir, entre otras. Trabajaremos con 3 tipos de imágenes: Bitmaps, Pixmaps y Hexmaps. Además se trabajará con el concepto de profundidad de las imágenes

Descripción breve del paradigma y los conceptos del mismo que se ven aplicados en este proyecto

Como ya lo habíamos mencionado anteriormente, el paradigma que ocuparemos en esta ocasión será, el paradigma lógico, este se basa principalmente en darle conocimiento al intérprete a través de tu código, este código está compuesto de hechos, estos son cláusulas que describen una relación entre 1 o más términos, estos se asumen como verdad, estos están para modelar un problema generando relaciones entre sus conceptos.

Las reglas también son un tipo de cláusula que relaciona términos, pero esta depende de la conjunción de los elementos o también las condiciones que tienen que tener los elementos para que la regla sea verdadera, para entenderlo mejor, los hechos son un tipo especial de regla, solo que no tienen que cumplir ninguna condición para declarar que eso es verdadero.

Para al momento de hacer una consulta, el intérprete revisa toda esta base de conocimiento para entregarte la respuesta. Este proceso de búsqueda automática está hecho por backtracking y busca unificar la consulta realizada, si se encuentra una coincidencia entre su base de conocimiento, dirá que la consulta es verdadera si no la encuentra, significa que la consulta no se pudo comprobar que es verdadera o no. Pueden haber hechos o reglas en donde con una consulta hallan más de un posible verdadero, gracias al backtracking se muestran todas las relaciones que coincidan

Por ejemplo, en vez de tener una función que cree una imagen, en nuestro código le decimos que una imagen es una lista que contiene ciertos elementos, entonces al preguntar si una entrada es una imagen, el intérprete busca la definición de una imagen que le dimos y te dice si la entrada es verdadera o falsa. En este paradigma no hay salidas en si, solo hay dominios, excepto cuando es una consulta de true o false, así como es la función Bitmap o isCompres. En este paradigma no se puede

Análisis del problema

Para empezar, dividiremos el problema en 4 TDAs distintos, TDA imagen, TDA pixbit, TDA pixrgb, TDA pixhex. La imagen de proporción Alto x Ancho la dividimos en cuadrantes trabajando como una especie de matriz (anexo 1), donde cada cuadrante representa un píxel con un color dado. Con esto como base empezamos a desglosar, los pixeles los dividimos en 3 tipos: Pixbits, PixRGB, PixHex, como bien dicen sus nombres, pixbits representa un pixel binario, pixrgb representa un píxel con color en formato rgb, y pixHex representa un píxel con color en formato hexadecimal, siendo esto lo único que diferencia a estos 3 formatos, ya que en común, cada pixel es representado también por sus coordenadas “x” e “y”, y su profundidad “d”. Una imagen bien construida, está hecha solamente de un tipo de pixel, de los antes mencionados.

Para crear en el programa imágenes y pixeles, se trabajó con la representación de listas, la lista de una imagen es de la forma: Color Comprimido Alto (int) X Ancho (int) X lista de pixeles (pixel bit-d* | pixrgb-d* | pixhex-d*), en esta última se puede definir una lista de lista de tamaño N. Y la lista de cada pixel es de forma: coordenada x (int) X coordenada y (int) X color (int | sting | lista de int) X profundidad.

Las funciones a crear bajo el paradigma actual son: Constructores de cada tda, como serían image, pixbit-d, pixrgb-d, pixhex-d, la gracia en este paradigma, es que los mismos constructores los podemos ocupar de selectores o de pertenencia. Por lo que no era estrictamente necesario crear este tipo de funciones, pero por comodidad y para evitar posibles errores de arrastre si se llegaba a cambiar algun TDA, se crearon igualmente selectores y modificadores por comodidad, tales como, getAncho, getAlto, getx, gety, get bit, setx, sety entre otros.

De las reglas que nos piden crear, tenemos funciones de pertenencia tales como, pixmap?, bitmap?, hexmap?, compressed?. Además tendremos modificadores como rotate90, flipH, flipV, crop, edit, compress. Y por último tendremos funciones de otro tipo que no entra en las descripción ni de constructor, modificador, ni selector, como bien sería, el histograma, el cual no se creó un TDA aparte para él, ya que se entregará como una representación de listas.

Diseño de la solución

Como se trabajo con representaciones de listas, para todos los constructores de los tdas se enlistan los datos de la forma [CabezaLista|RestodeLista] que junta una serie de datos dados

Los selectores nacen a partir del mismo constructor, pero se implementaron igualmente, consultando qué elemento era el que estaba en X posición.

Para las funciones de pertenencia, en imagen se consultaba si el primer pixel era del tipo que se pedía (Ispixmap?, Isbitmap? y Ishexmap?) o si estaba comprimida(compressed?), se consultaba el primer elemento de la imagen, que corresponde al color comprimido, si era un string vacío es porque no estaba comprimido.

Para modificadores tales como flipV, flipH, rotate 90, crop, en donde no cambiaba el color de los pixeles, se trabajó de la siguiente manera, se comparaba la imagen original con la imagen modificada por la función, para así buscar alguna regularidad en el comportamiento de los pixeles y fijar alguna función matemática para la creación de las nuevas coordenadas. Haciendo que para cada pixel sea remplazado por uno que mantuviera los mismos valores de color y profundidad, cambiando solo las cordenadas. Para estas funciones el uso de maplist en la creación de los modificadores fue esencial

Para ir modificando la listas de pixeles en las distintas funciones, se apoyó bastante en el maplist, para realizar cierta modificación a cada pixel y dejarlos en una nueva lista, también se usó bastante el include tanto como el exclude para así filtrar los pixeles que cumplieran las condiciones que se requerían para cada regla.

Agregar que el uso de la recursividad también fue necesitado, especialmente la natural, para la construcción de listas en el histograma y en el depth Layers

Aspectos de implementación

Para este laboratorio se hizo uso del lenguaje Prolog versión 8.4.3. Aparte de las funciones creadas, solo se hizo uso de la documentación que estaba en la página oficial de prolog (manual, s. f.), no se tuvo que importar ningún tipo de módulo. y como intérprete se usó SWISH-Prolog ya que, fue el visto en clases y no se busco otro, se trabajó mayormente en la versión web debido a que el debugger era mucho más completo.

Instrucciones de uso

Con este código es posible crear imágenes y píxeles, pudiendo así realizar rotaciones, invertir tanto la posición como los colores, recortar la imagen, verificar de qué tipo de imagen es y cambiar el tipo de píxel que se usa. Pero para que todo funcione correctamente hay que ingresar bien los datos al momento de crear un pixel o una imagen

Se asume que en cada imagen siempre se tendrá que mantener una homogeneidad con el tipo de píxeles

Para usar se tiene que ocupar el intérprete SWISH-Prolog, en cualquiera de sus formatos, ya sea el ejecutable o la versión online.

Para probar el código, se tiene que consultar el archivo tdaImagen_21165824_BritoGarrido, en donde se encuentran todos los requerimientos funcionales, desde ahí se tiene que copiar alguna línea del script de pruebas y pegar en la caja de consultas.

Resultados y autoevaluación:

	Funcion	Puntaje	Comentario
1	TDAs	0.75	La elección de TDAs fue adecuada para las funciones creadas, pero no se pensó en cómo funciona para el descompress
2	image - constructor.	1	La creacion de imagenes es adecuada y permite una entrada de n elementos
3	TDA image imageIsBitmap	1	Reconoce en cualquier caso si la imagen es bitmap o no
4	TDA image - imageIspixmap	1	Reconoce en cualquier caso si la imagen es pixmap o no
5	TDA image - imageIshexmap	1	Reconoce en cualquier caso si la imagen es hexmap o no
6	TDA image - imageIscompressed	1	Reconoce en cualquier caso si la imagen está comprimida
7	TDA image - imageFlipH:	1	Invierte horizontalmente Correctamente la imagen para cualquier caso entregado usando recursividad
8	TDA image - imageFlipV	1	Invierte verticalmente Correctamente la imagen para cualquier caso entregado usando recursividad
9	TDA image - imagecrop:	1	Recorta la imagen dejando las nuevas dimensiones y coordenadas correctamente
10	TDA image - imageRGBToHex	1	Convierte en cualquier caso la imagen del tipo pixmap a hexmap
11	TDA image - imageToHistogram:	1	Crea correctamente un histograma, y la representación es adecuada para lo requerido en el código
12	TDA image - imageRotate90:	1	Rota correctamente la imagen en 90 grados para cualquier caso (imagen cuadrada, rectangular un solo píxel, etc)
13	TDA image - imageCompress	1	Entrega la imagen comprimida, para todos los casos (todos los colores con la misma frecuencia o distinta)
14	TDA image- imageChangePixel	1	Cambia cualquier píxel por uno que tenga su misma coordenada sin perder ningún dato
15	TDA image - invertColorRGB:	1	invierte correctamente el valor del rgb del pixel
16	TDA image - imageToString	1	Se genera adecuadamente para cualquier largo de imagen
17	TDA image - imageDepthLayers:	1	Se crean correctamente las listas de imágenes y se rellenan todas con los pixeles blancos, crea una imagen por cada profundidad distinta a los píxeles originales.
18	TDA image - imageDecompress	0	Complejidad para implementar debido a tener que modificar el tda imagen

Conclusiones del trabajo

La principal dificultad de este paradigma fue entender cómo era qué hacía el proceso de búsqueda para comprobar su respuesta, osea, entender bien el backtracking. Bajo mi perspectiva los puntos a favor que tienes es que tanto modificadores, selectores y constructores se pueden resumir en una sola regla, cosa que en el paradigma funcional se tenía que implementar una función para cada tipo de función simplificando bastante el trabajo, por otro lado, el recorrer listas o construirlas se llegaba complicar, al igual que el hacer recursión de tipo cola no le es tan simple, encuentro que el manejo de variables era más cómodo en el paradigma funcional, ya que se podían realizar llamados a funciones, dentro de otras funciones, dejando acotado el código, en cambio acá era más lento. El paradigma es muy bueno para modelar problemas, pero a la hora de implementar las soluciones, no resulta de lo más práctico. Hay que notar también que el backtracking es bastante útil al momento de encontrar varías soluciones a una misma consulta, pero en este laboratorio no se pudo notar mucho esta característica

Referencias

manual. (s. f.). Recuperado 1 de noviembre de 2022, de https://www.swi-prolog.org/pldoc/doc_for?object=manual

Anexos

Anexo 1: