Magic Tutorial #S-2: Boxes and labels

Rajit Manohar

Department of Computer Science
California Institute of Technology
Pasadena, CA 91125

This tutorial corresponds to Magic version 7.

Tutorials to read first:

Magic Tutorial #S-1: The scheme command-line interpreter

Commands introduced in this tutorial:

:getbox, :box.push, :box.pop, :box.move, :label.vert, :label.horiz, :label.rename, :label.search, :label.find-next

Macros introduced in this tutorial:

(None)

1 The current box

The fundamental way scheme programs interact with magic layout is by using magic's **box** command. For instance,

(box 1 1 2 2)

changes the current box to the rectangle defined by the coordinates (1,1) and (2,2) in the current edit cell. This is the standard magic **:box** command. After moving the box to a particular position in the layout, the area can be painted, erased, selected, etc.

The scheme function **getbox** returns the current box as a list of four integers. For instance,

(box 1 1 2 2)
(define x (getbox))

will bind the list **(1 1 2 2)** to variable **x**.

2 Saving and restoring the box

If a scheme function moves the current box around, it is good practice to restore the box back to its original position. This is especially useful when writing a function that the user is likely to type on the command line.

box.push can be used to push a box onto the current stack of boxes. **box.pop** restores the box to the one on the top of the box stack. The sequence

```
(box.push (getbox))
(box 1 1 5 4)
(paint poly)
(box.pop)
```

will paint a rectangle of polysilicon from (1,1) to (5,4), restoring the original position of the box.

3 Moving the box

Magic's built-in **move** command is not entirely reliable. Sometimes move commands are ignored, with disastrous effects. (Think about what might happen if a move command was ignored in the middle of drawing a stack of twenty transistors . . .) The scheme function **box.move** moves the box relative to the current position.

```
(box.move 5 3)
```

will move the box right 5 lambda and up 3 lambda.

4 Labelling vertical and horizontal wires

Datapaths are usually designed by designing cells for a single bit of the datapath, and then arraying those cells to obtain the complete datapath. When simulating such designs, it is usually desirable to label wires in the datapath with names like "name0", "name1", up to "nameN."

There are two functions that can be used to perform such a task. The function **label.vert** returns a function that can be used as a labeller for vertically arrayed nodes. **label.horiz** returns a function that can be used as a labeller for horizontally arrayed nodes.

```
(define lbl (label.vert "name" 6)
```

The command above defines a new function **lbl** that can be used to generate labels beginning with " name0" for nodes that are vertically spaced by 6 lambda. The simplest way to use this function is to bind it to a macro as follows:

```
(macro 1 "lbl")
```

Place the box over the lowest node. Every time key "1" is pressed, a new label "nameM" is created and the box is moved up by 6 lambda. **label.horiz** can be used in a similar fashion for labelling nodes that are horizontally arrayed.

5 Finding and renaming existing labels

The label macros provide functionality to search for all labels that match a particular string. Place the box over the region of interest. Type:

(label.search "label")

To place the box over the first occurrence of the label you searched for, type:

(label.find-next)

Repeatedly executing this function causes the box to move to all the labels that match the search pattern. Typically, one would bind **label.find-next** to a macro.

The command **label.rename** can be used to rename all labels with a particular name. To use this command, place the box over the region of interest. Then type

(label.rename "label1" "label2")

All occurrences of label "label1" in the current box will be renamed to "label2".

6 Writing these functions

The functions discussed in this tutorial are not built-in. They are user-defined functions in the default scheme file loaded in when magic starts.

As you begin to use magic with the scheme command-line interpreter, you will observe that commands for drawing paint on the screen are extremely slow. This time interval is not normally noticeable because editing is interactive. However, when one can write a scheme program to draw twenty transistors on the screen, this delay becomes noticeable. It is worthwhile to minimize the number of magic commands executed, even if this involves writing more scheme code. The **box-pop** command has been tuned a little to not execute the **box** command if the box would not move as a result.

```
(define box.list ())

(define box.move
        (lambda (dx dy)
                (let* ((x (getbox))
                       (nllx (+ dx (car x)))
                       (nlly (+ dy (cadr x)))
                       (nurx (+ dx (caddr x)))
                       (nury (+ dy (cadddr x))))
                (box nllx nlly nurx nury)
                )
        )
)
```

```
(define box.=?
        (lambda (b1 b2)
                (and (and (=? (car b1) (car b2)) (=? (cadr b1) (cadr b2)))
                      (and (=? (caddr b1) (caddr b2)) (=? (caddr b1) (caddr b2)))
                )
        )
)

(define box.push
        (lambda (pos)
                (set! box.list (cons pos box.list))
        )
)

(define box.pop
        (lambda ()
                (if (null? box.list)
                      (echo "Box list is empty")
                      (let ((x (car box.list)))
                              (begin
                              (set! box.list (cdr box.list))
                              (if (box.=? x (getbox)) #t (eval (cons 'box x)))
                              )
                      )
                )
        )
)
```