Bitpict Implements Itself

Furnas, George

http://hdl.handle.net/2027.42/83491
The exercise here was for Bitpict's small pixel rewrites to implement a version of InterDiagrammatic Reasoning (IDR) that was powerful enough to then further a small version of Bitpict itself. More precisely, a 8bit-color-2Dim-Bitpict was used to implement 2Bit-grayscale-1Dim-IDR implementing 2Bit-grayscale-1D-bitpict.

To create a Bitpict version of IDR that can actually do anything serious, it was necessary to create a graphically-represented and graphically-computed high-level language corresponding to the LISP that IDR uses for its conditionals, loops, defuns, etc. The language created is basically a functional programming language, a kind of pseudo-lisp. The videos show various aspects of this machinery in action. The pages here show the "code" in detail, with approximate LISP glosses of the functions used in the Bitpict-IDR-Bitpict implementation.

Note that the correspondence to LISP is not exact. For example, built-in operators with a fixed number of arguments do not use any bracketing syntactic markers (where parentheses would appear in LISP). Funcalls do have an EndBracket, and Loops have a Loop-End marker. Setting a variable is really a function call to a corresponding function. If required, variables can be set to a whole sequence of values; we just present it here as "(Set VarName <values> ...)". Also various syntax elements, e.g., for Loops of various kinds, are represented with special colors, which, like the colors for other operators are treated by the language like reserved words. With the exception of the 'divider marker, they are not quoted in the pseudo-Lisp below. A few columns of color ('divider and emptyspace) appear in the "code" just for visual separation, to help human visual parsing, much as extra line breaks are used in text-code.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Functions Colors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main ()</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>GetMatches()</td>
<td>Green-Red</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Match-FD (LHS, Mask, RHS)</td>
<td>Blue-Red</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>AbsDiff(D1,D2)</td>
<td>Red-Blue</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>ApplyMatches(FD,</td>
<td>Green-Blue</td>
</tr>
<tr>
<td>LHS1, Mask1, RHS1,</td>
<td></td>
</tr>
<tr>
<td>LHS2, Mask2, RHS2, ...)</td>
<td></td>
</tr>
</tbody>
</table>
Saves the Ruleset, and the Field Diagram, then iteratively, as long as there are matches, applies those matches to the Field Diagram.

```plaintext
Main ()

-------- Ruleset -------- ^

^FieldDiagram

^------------------------ MainDoLoop ------------------------^
```

```plaintext

LHS1
Mask1
RHS1
LHS2
Mask2
RHS2
...

...
LHS9
Mask9
RHS9

^divider ;; (purely for visual separation)
{Set RuleSet

LHS1
Mask1
RHS1
LHS2
Mask2
RHS2
...

^divider ;; (purely for visual separation)
^selector

^delimiter

^empty space ;; for results of Loop
```
Note that the part of the WHILE loop that comes after the “DO” is refilled whenever the WHILE condition evaluates to TRUE, using the same basic mechanisms used to restore Defuns after they have been executed. That is, rewrite rules use the small color marks (1-2 pixels) to construct the appropriate operator columns (here, rows, because everything is rotated for commenting). Two-pixel marks are needed for what will become function calls (including Get/Set of variables), since user-defined functions are labeled by color pairs.
GetMatches()  
Takes no arguments.
Returns a sequence containing all the rules that match the Field Diagram.

The MAPk loop feeds k diagrams at a time to the specified function. Its syntax is based on a generic LOOP construct that consists of a LOOP-BEGIN, a LOOP-divider, and a LOOP-end. This generic structure invokes rewrites that do generic loop things – like restoring the body of the loop for each iteration, and cleaning up afterwards. The specific behavior of the MAPk loop is invoked by surrounding the LOOP-divider, and preceding the LOOP-end, with a special MAPk-divider color. Then the correct local rewrite rules can be triggered to produce the MAPk behavior.
Match-FD (LHS, Mask, RHS) ;; Function Colors: Blue-Red

Compares the Field Diagram, masked by the mask, to the LHS of the rule:
If they match, it returns the rule (i.e., the triple: LHS, Mask, RHS).
Otherwise, it returns nothing.

This uses the IF-G (IF for Groups) construct. Its syntax is:

```
IF-G color (a particular yellow)
Predicate (must evaluate to T/F)
'divider
Group_of_Diagrams_1
'divider
Group_of_Diagrams_2
'divider
```

If the predicate evaluates to TRUE, it evaluates to the first group, otherwise it evaluates to the second group. Either group may be just a literal sequence of diagrams, or somehow computed.

FYI: It is a more complicated version of the simple IF:

```
IF color (a slightly different yellow)
Predicate (must evaluate to T/F)
Diagram1
Diagram2
```

The simple IF returns Diagram1 if the predicate is TRUE, and Diagram2 otherwise.
AbsDiff(D1,D2)  ;; Function Colors: Red-Blue

Computes the pixel-by-pixel absolute difference between two diagrams. The result is the null (0) diagram, iff the two are equal.

The arguments are placed correctly by the cross-bar switch, seen in the diagram upper right.
ApplyMatches(FD, LHS1, Mask1, RHS1, LHS2, Mask2, RHS2, ...)

Function Colors: Green-Blue

Applies up to three matched rules to the Field Diagram, FD. Returns the new Field Diagram.

This function uses the new Accumulate loop. This is a different implementation of Accumulate than that shown in Anderson&Furnas (2010), in that it uses much of the general-purpose loop machinery, instead of the one-off mechanism in the paper.

*One other difference from the paper is that Accumulate here is done on an OVERLAY operator not an OR. The result is the same under the assumed condition of disjoint matches after conflict resolution.