

# A Distributed Text Field in Bloom

Hesam Samimi

Draft of 7 August 2013

## 1 Background

Bloom [1] is an extension of the Ruby language implementing a distributed Datalog variant called Dedalus [2]. Bloom programs are distributed processes whose state is defined by a set of *state relations* (internally just Ruby collections). Multiple Bloom processes running on a network can be connected to define a workflow via a set of *channel relations*. Tuples from state relations can be sent through these channels from one process to another, which enables us to build distributed applications. The logic of Dedalus (i.e. Datalog) rules determines what data is sent around and to whom.

Bloom programs are reactive processes. Any time a new tuple is received via the incoming channel relations the specified Dedalus rules (given inside the `bloom do` block) are triggered to a fixpoint, moving the process into the next time step in which the new changes to the state are applied.

## 2 Intro

Our long-standing goal is to have an extensible multi-language programming environment where different languages communicate and collaborate only in a simple, unified, and principled manner.

My current vision towards this goal is to use Bloom-like libraries for each of the supported languages the same way Bloom works on top of the Ruby language. While these languages have no way to directly talk to each other, they would be enabled to do so indirectly, as their Bloom-like library extensions are compatible with each other and all operate on the same set of shared Bloom *state* or *channel relations* and over objects and primitive values.

While the Ruby extension Bloom and its core distributed Datalog variant Dedalus are designed for *distributed* applications in terms of physical separation of processes on a network, I would like to also consider the benefits of these tools towards distribution in terms of processes running different languages, as well as logically separable components that (even if written in the same language) are better off be running independently. This may help towards a better story for encapsulation, modularity, understandability, and language interoperability.

Here we will consider a text field application as our first case study. I try to design logically separated components of this application into separate processes that communicate only through Bloom channels. In theory each such process can be running a different language, but for now in the absence of languages with Bloom-like extensions we shall write all these in Bloom itself.

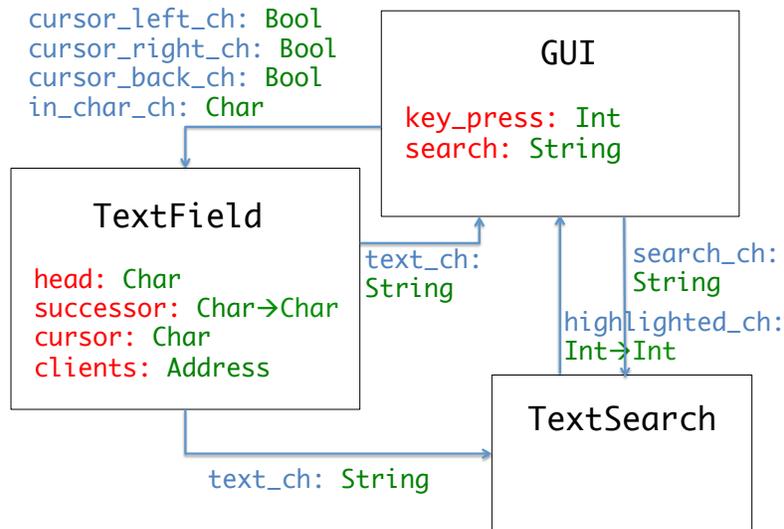


Figure 1: Various communicating processes in the text field application along with the interface channels

### 3 Defining State and Workflow: Relations and Channels

Fig. 1 illustrates the simple design of our application. Each box represents a process in this application. The state representation of each process is shown in red relations, while the channel relations defining the dataflow among processes are given in blue text and arrows. All relations are annotated with types in green color for documentation.

#### 3.1 TextField Process

The `TextField` process represents the logical state of the text field. Assuming each character on the text field is represented as an object of type `Char` the entire state of our text field can be given in terms of a `head` unary relation pointing to the starting character, a `cursor` unary relation representing the character on which the cursor is positioned, as well as a binary `successor` relation representing a linked list of all characters present. We use another relation `clients` to keep track of all users currently viewing the text field.

#### 3.2 GUI Process

The `GUI` process represents a *view* of the text field process above. It's a graphical display of the current state of the text and the cursor position. It also captures any key strokes from the keyboard (stored in `key_press` relation) and sends it over to the `TextField`. This component receives through the unary `text_ch` channel the current text of the text field as a string. We will later use the binary `highlighted_ch` channel to tell this component which positions of the text it should highlight to show the result of a text search. The text requested by the user to search for is stored in the `search` relation.

We make this a distributed application by simply running multiple `GUI` processes.

### 3.3 TextSearch Process

We add a `TextSearch` component that given the text of the text field and a string to search for will compute the positions that match and should be highlighted by the `GUI` process.

### 3.4 The Workflow: Connecting the Processes

Now we use Bloom channels to connect these components together, as shown in Fig. 1. `GUI` processes send keyboard events `cursor_left.ch`, `cursor_right.ch`, `cursor_back.ch`, and `in_char.ch` over to the `TextField` process. The first three indicate movement of the cursor while the last one says what character was typed.

In turn `TextField` computes the string form of its current text from the `head` and `successor` relations and sends it over to all running `GUI` processes.

In order to enable the search feature, we let the `GUI` process send a unary channel `search.ch` over to the `TextSearch` process representing a text to search. This process also receives the current text of the text field in string form from `TextField`. Given this input it computes the `highlighted` relation and sends it to the requesting `GUI` component, which tells it which parts of the text match the search and should be highlighted.

## 4 Defining Change of State: Rules

Now that we have described the state of various processes in our application and the workflow among them, let's use Bloom rules to express how their state can change.

In case the Bloom syntax seems distracting I include comments in English to describe the main rules involved. The syntax and the low level subtleties of the code are not as important for the purposes of this note and can be ignored.

### 4.1 GUI Rules

Fig. 2 describes the rules for each `GUI` process.

### 4.2 TextField Rules

The rules describing how the `TextField` can change are given in Fig. 3.

### 4.3 TextSearch Rules

Finally Fig. 4 describes the rules for the `TextSearch` module.

## References

- [1] Peter Alvaro, Neil Conway, Joseph Hellerstein, and William Marczak. Consistency analysis in bloom: a calm and collected approach. In *Proceedings 5th Biennial Conference on Innovative Data Systems Research*, pages 249–260, 2011.
- [2] Peter Alvaro, William Marczak, Neil Conway, Joseph Hellerstein, and David Maier. Dedalus: Datalog in time and space. In *Datalog Reloaded*. Springer/Verlag, Berlin, 2011.

```

class GUI
  bloom do
    # When a character is typed send it over to TextField as a Char object
    # via in_char_ch channel:
    # (<~ means asynchronous send over a channel)
    in_char_ch <~ key_press { |k| [TEXTFIELD_ADDR, Char.new(k.key)] if is_char?(k.key) }

    # When one of left, right, or back arrows are pressed send one of cursor_left_ch,
    # cursor_right_ch, or cursor_back_ch signals to TextField, respectively:
    cursor_left_ch <~ key_press { |k| [TEXTFIELD_ADDR] if is_left_arrow?(k.key) }
    cursor_right_ch <~ key_press { |k| [TEXTFIELD_ADDR] if is_right_arrow?(k.key) }
    cursor_back_ch <~ key_press { |k| [TEXTFIELD_ADDR] if is_back_arrow?(k.key) }

    # When the string text is received update the displayed text:
    # (using the side effect of set_text method...)
    stdio <~ text_ch { |t| ["setting text..."] if set_text(t.text) }

    # When text to search is requested send it as a string over to TextSearch:
    search_ch <~ search { |c,t| [SEARCH_ADDR, t.text] }

    # When the highlighting info is received highlight the corresponding letters:
    # (using the side effect of set_highlight method...)
    stdio <~ highlighted_ch { |h| ["setting highlighting..."] if set_highlight(h.start, h.len) }
  end
end

```

Figure 2: Main GUI rules

```

class TextField
  bloom do
    # When cursor_right_ch or in_char_ch is received new cursor character becomes
    # the successor of current:
    # (<+- means delete previous tuple of the same key & add new tuple with new value)
    cursor <+- (cursor_right_ch * cursor * successor).pairs
      { |e,c,s| [s.succ] if c.char == s.char }

    # When cursor_left_ch or cursor_back_ch is received new cursor character becomes
    # the predecessor of current:
    cursor <+- (cursor_left_ch * cursor * predecessor).pairs
      { |e,c,p| [p.pred] if c.char == p.char }
    cursor <+- (cursor_back_ch * cursor * predecessor).pairs
      { |e,c,p| [p.pred] if c.char == p.char }

    # When in_char_ch character received insert it in the current position (based on
    # cursor relation) in the successor relation...
    # + The successor of cursor character becomes the incoming character:
    successor <+- (in_char_ch * cursor).pairs { |e,c| [c.char, e.char] }
    # + The successor of incoming char becomes the previous successor of the
    # cursor character:
    # (<+ means add tuple)
    successor <+ (in_char_ch * cursor * successor).pairs
      { |e,c,s| [e.char, s.succ] if c.char == s.char }

    # - When cursor_back_ch received remove character at cursor position from the
    # successor relation...
    # + The successor of cursor's previous character becomes the successor of
    # the cursor character:
    successor <+ (cursor_back * cursor * successor * predecessor).pairs
      { |e,c,s,p| [p.pred, s.succ] if (p.char == c.char && s.char == c.char) }
    # + The cursor character is removed from successor relation:
    # (<- means delete tuple)
    successor <- (cursor_back * cursor * successor).pairs
      { |e,c,s| [s.char, s.succ] if s.char == c.char }
    # + The previous character's successor is removed from the relation:
    successor <- (cursor_back * cursor * predecessor).pairs
      { |e,c,p| [p.pred, c.char] if (p.char == c.char) }

    # The predecessor relation is derived by reversing the successor relation:
    predecessor <= successor { |s| [s.succ, s.char] }

    # Use a Ruby method to compute string text from the head and successor relations:
    text <= (head * successor).pairs { |h,s| [compute_text(h.char, successor)] }

    # Send computed string text to all connected GUI processes and the TextSearch:
    text_ch <~ (text * clients).pairs { |a,t,c| [c.client, t.text] }
  end
end

```

Figure 3: Main TextField rules

```

class TextSearch
  bloom do
    # When both string text and search (text to search) are received perform search
    # and compute highlighted relation and send it to the requesting GUI process:
    highlighted_ch <~ (text_ch * search_ch).pairs
      { |t,s| search_for(s.client, t.text, s.text) }
  end
end

```

Figure 4: Main TextSearch rules