Horus2D

Sebastian Bartlett, Oscar Puente

May, 2020

Contents

1 Vision

We believe that the evolution of language aims to reach beyond a purely textual and auditory modality, towards the visual. Semantics are often clearer when they are seen, and visual cognition, along with story understanding, is a *sine qua non* of our unique capacity for intelligence as humans. In this project, we have taken steps towards developing a platform that enables code to be created, modified, and executed in a visual form using a graph-based evaluator.

2 Steps

2.1 Code Representation

A procedure can be thought of as a topology of sub-components connected by directed control-flow and data paths. These sub-components, in turn, are either self-evaluating primitives such as numbers and character strings, or procedures following the same top-level pattern of sub-components and links. From this it can be noted that the basis of programming is recursive in nature. We aim to make this recursive representation visual, in which sub-components are represented as code blocks, with data and control-flow links represented by directed arrows.

2.2 Built-ins and Arbitrary Code

Our system provides standard language built-ins, such as arithmetic and boolean operators. The list of current built-in functions is as follows: `[+ * % - / * and or not == != < <= > >= len #t #f nil print]`. In addition to these function primitives, numerals are self-evaluating, and can be used in blocks and edge predicates. All functions take the form of code blocks, which can then be connected via links that pass output on to the next block. The user may also insert arbitrary text into a code block. If this text refers to a function name that has been defined in the environment, the function is called with the corresponding code block's inputs. The text may also be tagged with a 'quoted' parameter in the program table representation, which causes the block to be interpreted as a self-evaluating string. If the previous conditions have not been met, the text is taken to be arbitrary Lua code, and is evaluated with the block inputs active within the function scope. In a future version of the system, Compound Blocks and Input Blocks will have labeled inputs, as well as (optionally labeled) outputs. In the system's current state, an input block can refer to its incoming connected values via the `var[]` list that is bound to the block's scope (e.g. `return var[1] + 3*var[2]`).

2.3 Compound Blocks

In order to support increasing complexity in programs while maintaining organizational principles, the contents of sub-components are be represented by the same network-based approach that is seen in the base-level interface. This approach follows the well-known tradition of black-box abstraction in electronic engineering and program prototyping. The system's interface allows users to create these black-box subcomponents, termed Compound Blocks, using the Create editing mode. The system also allows for ascending and descending levels using the Scope mode. There is no arbitrarily imposed limit on depth of Compound Block nesting, and given that Compound Blocks are treated identically to the base-level scope (as are negative scopes, *i.e.* scopes above the base-level), this enables Horus2D to capture the recursive nature of programming.

2.4 Graphical Interface

2.4.1 Modes

The system uses a visual user interface to display and manipulate code, along with an underlying DSL to both draw and get input from the interface. The functionality of the UI is organized in the following modes:

- Create Mode: Creates new blocks or links between them, can be accessed by pressing "c" on the keyboard.
  - Drag the mouse to create new blocks. The system currently supports compound blocks (sub-networks) and code/input blocks (arbitrary inputs or textual Lua code).
  - Drag the mouse from one box to another to make a link between them. Links are one of the two methods of combination that the system uses, together with compound blocks. The user can draw/erase links between blocks to specify the flow of the program.
- Drag Mode: Allows the user to move blocks around the current environment, can be accessed pressing "d" on the keyboard.
- Erase Mode: Erase blocks or links, can be accessed pressing "e" on the keyboard. When an element is removed all its dependences are removed as well (*i.e.* when a block is deleted, all the links associated to it are deleted).
- Scope Mode: Allows the user to navigate through scopes, can be accessed pressing "s" on the keyboard.
  - Left click on a compound block to enter in its scope. Right click anywhere to go to the outside scope.
- View Mode: Can be entered pressing "v" on the keyboard.
  - Drag the mouse to pan the screen.
- Run: the current program can be executed pressing "r" on the keyboard.

2.5 Examples

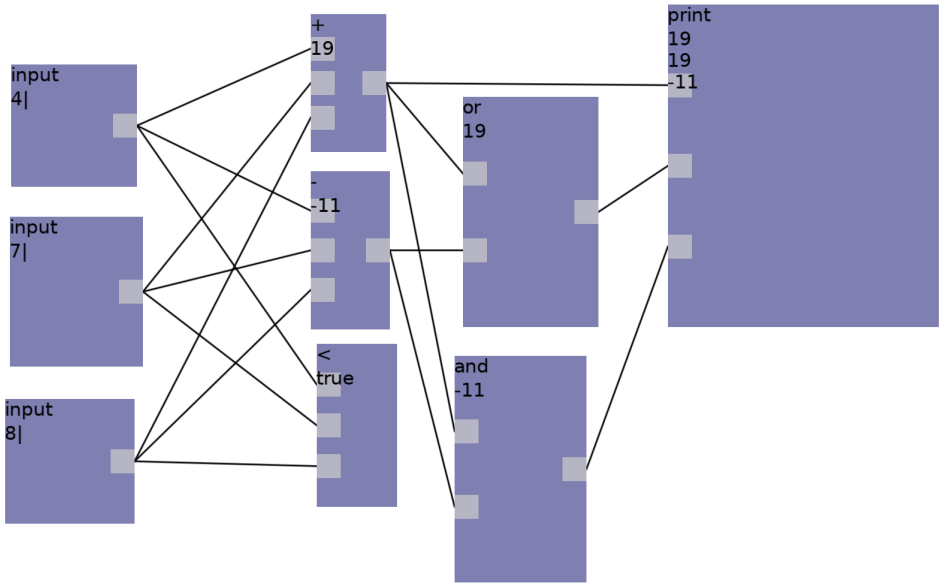

Figure 1: Example of arithmetic and logic built-ins

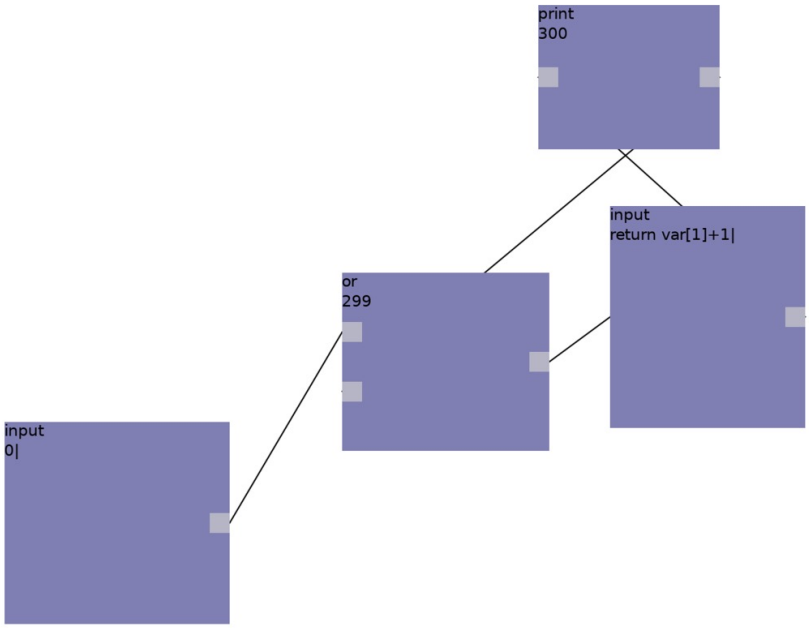

Figure 2: Example of an infinite loop (execution terminates after UserSettings.Eval.maxIterations)

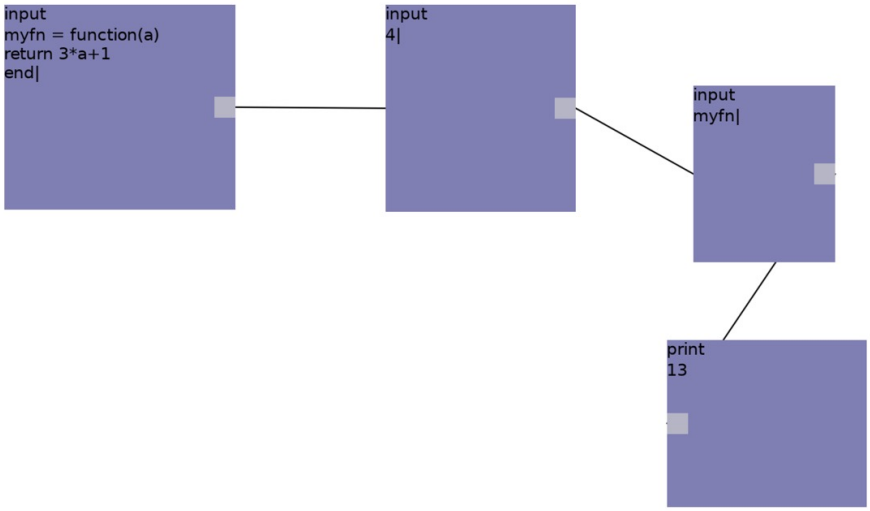

Figure 3: Example of arbitrary user-defined function execution

3 Major Subcomponents

3.1 Evaluator

Horus2D executes code using a graph-based evaluator, beginning at a starting node and proceeding using a breadth-first search through the graph. The evaluator internally enforces block precedence constraints by requiring that any given block's inputs be evaluated before that block can be executed. If this condition is not met, evaluation is delayed until all inputs have been evaluated. As such, the block executes once it has been reached through the longest route in the program graph. Stale inputs are guarded against by clearing the inputs to a block once it has finished executing. A bridge interface is used to connect the nested-table representation required by the evaluator to the object-based system used by the UI system. This interface provides the evaluator with unique names for each block (using the hex memory address for the corresponding Lua object table), such that a block's Next list can refer to any node within the program graph.

For any given block, the evaluator first checks to see if the block is quoted, in which case it returns the literal value of the string held in its contents. If this is not the case, the evaluator proceeds to check various known cases to determine the appropriate evaluation steps to be taken. First, the evaluator attempts to type-cast the input into a number. If this does not succeed, the evaluator enforces precedence constraints, with the understanding that the block is some kind of function or sub-network to be invoked. The evaluator proceeds to check if the block is a Compound Block, in which case it is evaluated recursively by calling `Eval.run()` on the sub-network. If the block is not Compound, the evaluator checks to see if the block's contents refer either to a built-in function or a function defined in the user's environment. If this test succeeds, the corresponding function is executed with the given inputs specific to the execution instance. If the block's contents do not refer to a function name, the evaluator uses Lua's `loadstring` procedure to interpret the text as arbitrary code, with input arguments bound to a `var[]` list. Once a code block has been executed, the values stored in its Next list are interpreted. Next-links can have attached predicates as arbitrary code to test the previous block's output or other bound environment variables, to determine whether the target block should be added to the evaluation queue. These predicates are optional and default to `true` if not explicitly added by the user during editing.

3.2 UI

At the top level of our system is the user interface. The UI serves as the layer of translation between the user and the application. This layer is comprised of two inter-related components, a visual-representation system (system output), and a user-input system (system input). The input system consists of mouse clicks, drags, and keyboard input to modify the blocks and edges on screen.

The visual-representation subsystem is responsible for the screen rendering of all objects in the current environment. Each block has `xy` coordinates, along with a `width` and `height`, so a blocks representation is straightforward. The input system is more complex. First we need to take the raw inputs from the mouse and keyboard, and classify their combinations into events. In the following example we check if the mouse is inside the limits of a rectangle

by iterating through a list of active rectangles), and if we find such a case, we save a reference to that object in the variable `mouseHovering`.

```
for i = #UI.current_compound_block().environment
        .Shapes.Rectangle.instances, 1, -1 do
  local val = UI.current_compound_block().environment
        .Shapes.Rectangle.instances[i]
  if Mouse.isInside.rectangle(val) then
        mouseHovering = val
        mouseHoveringIndex = i
        break
  end
end
```

In the second example, when the mouse is pressed in the UI's "drag" mode, while hovering over a draggable object, we set the UI's current target to the relevant object, and set the dragging-element Boolean to true.

```
Hooks.mousePress.add(function(x,y,button)
                --Drag start
                if UI.mode == 'drag' and is(mouseHovering,'draggable')
                and not UI.current_target then
                  UI.current_target = mouseHovering
                  origin = {x = UI.current_target.x, y = UI.current_target.y}
                  draggingElt=true
                end
end)
```

Once all the raw input data has been processed via system events and variables, we proceed to update the underlying program-structure representation. We feed this data to our UI DSL, that in turn modifies the positions of the blocks and the topology of the program graph.

4 Mechanisms for Extensibility

4.1 Hooks

Building off of the idea of hooks as implemented in Emacs, Horus2D employs the use of general, ad-hoc hooks in order to allow for extensibility in key program functionalities and events. Hooks in Horus2D are similar to hooks in Emacs: they both are a list of procedures to be called at well-defined instances. The key difference lies in the widespread use of hooks—whereas Emacs hooks are typically used to configure user preferences upon the activation of one or more major or minor modes, the Horus backend enables new hooks to be created and executed no differently from normal function. There is no alternate syntax required—hooks are called with a standard `someFunction()` syntax as seen in C, Python, Java, Lua, and so forth. This is a minor yet important point for extensibility. This unity in syntax

allows for low friction in using a hook, in that it is not necessary to differentiate between a hook and function before trying to run one. In addition, hooks (and their contents) can be dynamically added or removed, as well as temporarily disabled on an instance-level using straightforward calls to `enable()` and `disable()` procedures.

The Love2D multimedia framework provides three main procedures for application functionality: love.load, love.update, and love.draw. In addition, Love2D provides a wealth of user-event callbacks, capturing mouse movement, mouse presses and releases, key presses and releases, window resize and focus, as well as support for touch-based mobile devices. A typical hierarchical program structure may implement the functionality associated with these callbacks by creating a top-level procedure or manager for each, within which sub-procedures would be called as delegates for their respective functionality (for instance, UI-related tasks, evaluation tasks, filesystem tasks, and so forth). Hooks allow for great modularity and extensibility in Horus2D by enabling this delegation to be handled dynamically. If a programmer wishes to add new functionality, he or she does not need to trace a line of hierarchy in function calls and alter the code of the appropriate delegation procedure while keeping in mind potentially nested conditionals and other control-flow structures. By adding the code to a hook, the programmer can keep all other code unmodified and simultaneously gain subsystem modularity by having the single line of `Hook.add` code listed directly below the function (regardless of where the Hook itself is defined or called). By maintaining a practice of placing all `Hook.add` calls right after the end of functions, one can scan code and quickly determine where a function is connected in the program's control-flow network.

A remarkable benefit of using hooks over pure functions is that they allow for event-driven or reactive programming styles. For instance, a subcomponent of a UI system may add a function to the mouse-press callback hook, which then can look for subcomponents of its type that might have been clicked. This class-level function can check conditions for whether the program's control flow should be passed along to the appropriate class-instance, say, if the user is in the correct editing mode. The key observation here is that the class-level code can further define a more specialized hook to be called only when circumstances are correct, *e.g.* if a menu box has actually been clicked. In this way, other parts of the system specifically interested in menu box click events need not concern themselves with connecting to a top-level general mouse press event. Instead, they can directly define code to be executed after a menu box click by hooking into the specialized hook provided by the menu box subsystem. The nesting depth of such sub-hooks is not limited, so sub-hook execution can chain as salient events occur and control-flow signals are propagated. As such, a reactive, user-event propagation programming paradigm is easy to achieve, maintain, and debug, with low resource overhead.

Indeed, various system components in Horus2D demonstrate this ability (function subcomponents irrelevant to this illustration have been omitted):

```
PopupMenu.checkForClick = function(x,y,button)
  if ans then --was clicked
        PopupMenu.onClick(ans, target)
  end
  ...
end
Hooks.mouseRelease.add(PopupMenu.checkForClick)

PopupMenu.onClick = Hook()

Environment.new = function(...)
local init = function(name)
        self.drawComponents = Hook()
        -- Used for drawing subcomponents within the graphics transformation
        -- stack (within the virtual xy space of the system rather than the
        -- absolute screen xy coordinates)

        self.drawComponents.add(table_draw(self.Shapes.Line.instances))
        self.drawComponents.add(table_draw(self.Shapes.Rectangle.instances))
        self.drawComponents.add(table_draw(PopupMenu.instances))
  end

  self.drawOverlay = Hook()
  -- Used for drawing things above the graphics transformation stack
  ...
end

;; table_draw is a Curried function with multiple layers
;; defined (with an s-expression syntax) as follows:
(define tableDo
  (lambda (procName)
        (lambda (tbl)
          (foreach (instance in tbl)
                        (do (index procName instance) instance)))))
(define table_draw (tableDo "draw"))
```

When used in this form, hooks allow for a generalization and distribution of control flow logic in the program, which is invaluable towards reducing non-critical complexity. This design choice works particularly well for functions that do not have great precedence requirements, and generally implement independent yet co-operative features. A hierarchical approach is better suited towards enforcing strict precedence constraints, yet here too hooks can help by serving as the nodes of such a hierarchy tree. In brief, Hooks act as a superset of pure procedures in terms of extensibility, while incurring only a minor performance loss.

4.2 Tags

In order to support the UI, we used an extensible object tagging system. In order for the UI to handle the input events that occur at user-level, it uses the tagging system to connect combinations of inputs with the right handlers. As an example, if a rectangle click is received, the following code would be executed inside the `mouseRelease` hook, where `mouseHovering` refers to the relevant object.

```
if UI.mode == 'erase' and is(mouseHovering,'erasable') then
  mouseHovering.erase()
  mouseHovering = nil
end
```

This fast tagging structure allows for flexibility in our implementation. For example, if a user wishes to prevent the erasure of compound blocks, we simply need to add an additional statement checking the object's tags.

```
if UI.mode == 'erase' and is(mouseHovering,'erasable')
and not is(mouseHovering,'compound-block') then
  mouseHovering.erase()
  mouseHovering = nil
end
```

Additionally, using this system, we can change the behavior of the objects we have already defined. As an example, regardless of whether a block's implementation allows for it to be erased, we can change that in a particular object instance by removing the 'erasable' tag in it, providing us with dynamic, instance-specific non-erasable objects. Tags not only allow great flexibility to our implementation, they also improve readability and reduce the complexity of our UI DSL.

5 Next Steps

5.0.1 Meta-code

One possible extension of the project is to include an ability to reprogram the system itself in a dynamic, real-time way, similar to the GNU Emacs editor. For instance, Abstract Block definitions could admit for a piece of metadata specifying a draw procedure within the system, if a user should want to give an Abstract Block a particular visual look or color. In general, we aim to follow the Emacs philosophy of allowing users to directly interact with the system's internals and expand them by connecting new Lua code.

5.0.2 User input improvements

Future versions of the system will improve upon user ergonomics and ease-of-interaction. Features may include complete usability of the system via keyboard alone, a copy-paste feature over text and blocks, as well as the creation of Compound Blocks from an active block selection.

5.0.3 Auto-condense and Variable visual density

In order to improve upon visual code density, we plan to implement a subsystem that will automatically shrink empty space within blocks and links so as to be able to show the visual topology of larger program subcomponents without taking excessive space. Density is a double-edged sword, because while a sparse visual layout makes for code that is difficult to read, it is the right choice for when one wishes to expand code, adding subcomponents and greater connectivity. Although visually dense code is not the right choice for expanding code, it is the right choice for reading it. As such, we plan to have a system that will enable the user to expand and contract the visual code layout as desired, in order to accommodate the functional and aesthetic requirements of both reading and writing complex programs.

5.0.4 Code export and import

Future versions of the system may allow for code export to a target language so as to avoid the overhead of running the top-level interpreter. This would be feasible so long as the Code Blocks are all written in one language, or if there are procedures added that can accurately convert code between languages while maintaining semantics. Translation of visual (top-level) code to a target language should not be difficult, given that the language can be expressed in terms of standard programming constructs such as conditionals, loops, and procedure calls. Future versions of the system may also allow code import (possibly from different languages) to initialize compound blocks with the appropriate program block networks.