Stack Pointer

v0.0.1

https://github.com/SillyFreak/typst-packages/tree/main/stack-pointer

Clemens Koza

Abstract

Stack Pointer is a library for visualizing the execution of (imperative) computer programs, particularly in terms of effects on the call stack: stack frames and local variables therein.

Contents

I Introduction

Stack Pointer provides tools for visualizing program execution. Its main use case is for presentations using frameworks such as Polylux, but it tries to be as general as possible.

There are two main concepts that underpin Stack Pointer: *effects* and *steps*. An effect is something that changes program state, for example a variable assignment. Right now, the effects that are modeled by this library are limited to ones that affect the stack, giving it its name, but more possibilities would be interesting as well. A step is an instant during program execution at which the current program state should be visualized.

Collectively, these (and a few similar but distinct entities) make up *execution sequences*. In this documentation, *sequence item* or just *item* means either an effect or a step, or depending on context also one of these similar entities.

Stack Pointer helps build these execution sequences, calculating the list of states at the steps of interest, and visualizing these states.

II Examples

As the typical use case is presentations, where the program execution would be visualized as a sequence of slides, the example here will necessarily be displayed differently. So, don't let the basic appearance here fool you: it is up to you how to display Stack Pointer's results.

II.a Setting a variable

The possibly simplest example would be visualizing assigning a single variable, then returning . Let's do this and look at what's happening exactly:

```c
void main() {
  int a = 0;
  return 0;
}
```

```typc
execute({
  l(1, call("main"))
  l(2); l(2, push("a", 0))
  l(3); l(none, ret())
})
```

Step 1: line 1
- main

Step 2: line 2
- main

Step 3: line 2
- main: a = 0

Step 4: line 3
- main: a = 0

Step 5: (no line)

A lot is going on here: we're using `execute()` to get the result of an execution sequence, which we created with multiple `l()` calls. Each of these calls first applies zero or more effects, then a step that can be used to visualize the execution state. We use steps without effects to change to lines 2 and 3 before showing what these do, for example. For line 1, we don't do that and instead immediately add the `call("main")` effect, because we want the main stack frame from the beginning. For the final step that returns from main, we don't put a line number because it's not meaningful anymore. Not specifying a line number is also useful for visualizing calls into library functions.

The simple visualization here – listing the five steps (one per `l()` call) in a grid – is *not* part of Stack Pointer itself; it's done directly in this manual in a way that's appropriate for this format. The information in each step – namely, the line numbers, stack frames, and local variables for each stack frame – *are* provided by Stack Pointer though.

II.b Low-level functions for effects and steps

The `l()` function is usually the appropriate way for defining execution sequences, but sometimes it makes sense to drop down a level. Here's the same code, represented with only the low-level functions of Stack Pointer.

```c
void main() {
  int a = 0;
  return 0;
}
```

```typc
execute({
  call("main"); step(line: 1)
  step(line: 2)
  push("a", 0); step(line: 2)
  step(line: 3)
  ret(); step(line: none)
})
```

Step 1: line 1
- main

Step 2: line 2
- main

Step 3: line 2
- main: a = 0

Step 4: line 3
- main: a = 0

Step 5: (no line)

Apart from the individual function calls, one difference can be seen: the `step()` function takes only named arguments; in fact, a bare step doesn't even *have* to have a line; `l()` just has it because it's very common to need it. For use cases where line numbers are not needed but `l()` seems like a better fit, just define your own variant using the `bare-l()` helper, e.g.: `let l(id, ..args) = bare-l(id: id, ..args)` – now you can call this with `id` as a positional parameter.

II.c Calling functions (the manual way)

Regarding the call stack, a single function is not particularly interesting; the fun begins when there are function calls and thus multiple stack frames. Without additional high-level function tools, this can be achieved as follows:

```c
void main() {
  foo(0);
  return 0;
}

void foo(int x) {
  return;
}
```

```typc
execute({
  let foo(x) = {
    l(6, call("foo"), push("x", x))
    l(7); ret()  // (1)
  }
  let main() = {
    l(1, call("main"))
    l(2); foo(0); l(2)  // (2)
    l(3); l(none, ret())  // (3)
  }
  main()
})
```

Step 1: line 1
- main

Step 2: line 2
- main

Step 3: line 6
- main
- foo: x = 0

Step 4: line 7
- main
- foo: x = 0

Step 5: line 2
- main

Step 6: line 3
- main

Step 7: (no line)

As soon as we're working with functions, it makes sense to represent every example function with an actual Typst function. `foo()` comes first because of how Typst resolves names; for mutually recursive functions, we could stick them all into a dictionary. The execution sequence is ultimately constructed by calling `main()` once.

The `foo()` function contains its own `call()` and `ret()` effects, so that these don't need to be handled at every call site. There's a small asymmetry though: the return effect at `(1)` is not added via `l()` and thus not immediately followed by a step. After returning, the line where execution resumes depends on the caller, so the next step is generated by `main()` at `(2)`, with a line within the C `main()` function.

An exception to this is the `main()` function itself: at `(3)`, we generate a step (with line `none`) because here it is clear where execution will resume - or rather, that it *won't* resume.

II.d Calling functions (the convenient way)

Some of this function setup is still boilerplate, so Stack Pointer provides a simpler way, using `func()`:

c

```
1 void main() {
2   foo(0);
3   return 0;
4 }
5
6 void foo(int x) {
7   return;
8 }
```

typc

```
1  execute({
2    let foo(x) = func("foo", 5, l => {
3      l(0, push("x", x))
4      l(1)
5    })
6    let main() = func("main", 1, l => {
7      l(0)
8      l(1); foo(0); l(1)
9      l(2)
10   })
11   main(); l(none)
12 })
```

Step 1: line 1
- main

Step 2: line 2
- main

Step 3: line 5
- main
- foo: x = 0

Step 4: line 6
- main
- foo: x = 0

Step 5: line 2
- main

Step 6: line 3
- main

Step 7: (no line)

`func()` brings two conveniences: one, you put your implementation into a closure that gets an `l()` function that interprets line numbers relative to a first line number (for example, line 5 for `foo()`). This makes it easier to adapt the Typst simulation if you change the code example it refers to. Two, the `call()` and `ret()` effects don't need to be applied manually.

The downside is that this uniform handling means that we needed to add the last step after returning from `main()` manually.

II.e Using return values from functions

The convenience of mapping example functions to Typst functions comes in part from mirroring the logic: instead of having to track specific parameter values in specific calls, just pass exactly those values in Typst calls. Return values are an important part of this, but they need a bit of care:

```c
void main() {
  int x = foo();
  return 0;
}

void foo() {
  return 0;
}
```

```typc
execute({
  let foo() = func("foo", 6, l => {
    l(0)
    l(1)
    retval(0)  // (1)
  })
  let main() = func("main", 1, l => {
    l(0)
    l(1)
    let (x, ..rest) = foo(); rest  // (2)
    l(1, push("x", x))
    l(2)
  })
  main(); l(none)
})
```

Step 1: line 1
- main

Step 2: line 2
- main

Step 3: line 6
- main
- foo

Step 4: line 7
- main
- foo

Step 5: line 2
- main: x = 0

Step 6: line 3
- main: x = 0

Step 7: (no line)

In line `(1)` we have the first piece of the puzzle, the `retval()` function. This function is emitted by the implementation closure as if it was a sequence item, but it must be handled before `execute()`could see it because it isn't actually one. In line `(2)` the caller, who normally receives an array of items, now also receives the return value as the first element of that array. By destructuring, the first element is removed, and then the rest of the array needs to be emitted so that these items are part of the complete execution sequence.

III Module reference

Functions that return sequence items, or similar values like `retval()`, return a value of the following shape: `((type: "...", ...),)` – that is, an array with a single element. That element is a dictionary with some string `type`, and any other payload fields depending on the type. The payload fields are exactly the parameters of the following helper functions, unless specified otherwise.

step(..args: `dictionary`) -> `array`

Sequence item with type `"step"`: a single step at which execution state can be inspected. A step can have any fields associated with it that can be used to visualize the execution state.

Parameters:

`..args` (`dictionary`) – exclusively named arguments to be added to the step

bare-l(..args: `arguments`) -> `array`

A template for creating functions like `l()`. This function takes effects as positional parameters and step fields as named parameters. To Define a custom `l()`-like function, use something like this:

```
1 #let my-l(foo, ..args) = bare-l(foo: foo, ..args)
```
typ

Now `my-l()` lets you specify the `foo` field for your executions as a positional parameter.

Parameters:

`..args` (`arguments`) – the effects to apply before the next step, and the fields for the next step

l(line: `integer`, ..args: `arguments`) -> `array`

High-level step creating function. Emits any number of effects, followed by a single step. This function takes a line number as a positional parameter, then effects as additional positional parameters, and step fields as named parameters.

Parameters:

`line` (`integer`) – the line number to add to the step as a field

`..args` (`arguments`) – the effects to apply before the next step, and the extra fields for the next step

retval(result: `any`) -> `array`

Sequence item-like value with type `"return-value"` for `func()`: a simulated function may generate this as its last "item" to signify its return value. A function that doesn't use this can simply be called by another like this:

```
1 my-func(a, b)
```
typc

which will result in its sequence items being put into the sequence where it is called. A function that calls `retval()` is called like this:

```
1 let (result, ..rest) = my-func(a, b); rest
```
typc

here, the result given to `retval()` is destructured into its own variable, and the real items are emitted so that they appear in the sequence of steps.

`retval()` being called multiple times or not as the last item is an error.

Parameters:

`result` (`any`) – the return value of the function

func(name: `string`, first-line: `int`, callback) -> `array`

A helper for writing functions that produce execution sequences. This function automatically inserts `call()` and `ret()` effects. The implementation used by this function is given as a closure that receives a function similar to `l()`, but which adapts the line number according to the `first-line` parameter.

Usually, the first thing in a function using this will be a step to show the function call, optionally preceded by `push()` effects for the parameters. The last thing may be a `retval()` pseudo sequence item. If it is, then the returned array will have the return value as the *first* element.

The return effect is not part of a step generated by this function; usually the caller will add a step to display the new state at the location (line number) to which execution returned.

Parameters:

`name` (`string`) – the name of the function; used for the call effect

`first-line` (`int`) – the line at which the simulated function starts

- callback: simulates the function, receives an `l()`-like function

execute(sequence: `array`) -> `dictionary`

Simulates the given execution sequence and returns an array of states for each step in the sequence. Each element of the array is a dictionary with two fields:

- `step`: the fields of the step (not including the type); this will often include a line number
- `state`: the execution state according to the executed effects. Currently, the state only contains a `stack` field, which is in turn an array of stack frames with `name` and `vars` fields.

In total, the returned value looks like this:

```
(                                                                    typc
  (
    step: (line: 1, ...),          // any step fields
    state: (                       // currently, executuion state is only the
  stack
      stack: (
        (name: "main", vars: (     // function main is the topmost stack frame
          foo: 1,                  // local variable foo in main has value 1
          ...                      // more local variables
        )),
        ...                        // more stack frames
      )
    )
  ),
  ...                              // execution states for other steps
)

```

Parameters:

`sequence` (`array`) – an execution sequence

III.a Effects

Effects are in a separate module because they have the greatest potential for extension, however they are also re-exported from the main module.

call(name: `string`) -> `array`

Sequence item with type `"call"`: adds a stack frame for calling the named function.

Parameters:

`name` (`string`) – the function name to associate with the stack frame

push(name: `string`, value: `any`) -> `array`

Sequence item with type `"push"`: adds a variable to the current stack frame.

Parameters:

`name` (`string`) – the new local variable being introduced

`value` (`any`) – the value of the variable

assign(name: `string`, value: `any`) -> `array`

Sequence item with type `"assign"`: assigns an already existing variable of the current stack frame.

Parameters:

`name` (`string`) – the existing local variable being assigned

`value` (`any`) – the value of the variable

ret() -> `array`

Sequence item with type `"return"`: pops the current stack frame.