Sea of Nodes

08 October 2015 *Compilers*

Brief intro

This post is going to be about the sea-of-nodes compiler concept that I have recently learned.

While it is not completely necessary, it may be useful to take a peek at the some of my previous posts on JIT-compilers before reading this:

- How to start JIT-ting
- Allocating numbers
- SMIs and Doubles
- Deoptimize me not, v8

Compilers = translators

Compilers are something that every Software Engineer uses several times a day. Surprisingly even people who consider themselves to be far from writing the code, still use a compiler quite heavily throughout their day. This is because most of the web depends on client-side code execution, and many of such client-side programs are passed to the browser in a form of the source code.

Here we come to an important thing: while source code is (usually) human-readable, it looks like complete garbage to your laptop/computer/phone/...'s CPU. On other hand, machine code, that computers **can** read, is (almost always) not human-readable. Something should be done about it, and the solution to this problem is provided by the process called **translation**.

Trivial compilers perform a single pass of *translation*: from the source code to the machine code. However, in practice most compilers do at least two passes: from the source code to Abstract Syntax Tree (AST), and from AST to machine code. AST in this case acts like an *Intermediate Representation* (IR), and as the name suggests, AST is just another form of the same source code. These intermediate representations chain together and essentially are nothing else but the abstraction layers.

There is no limit on the layer count. Each new layer brings the source program closer to how it will look like in machine code.

Optimization layers

However, not all layers are used solely for translation. Many compilers also additionally attempt to optimize the human-written code. (Which is usually written to have a balance between code elegance and code performance).

Take the following JavaScript code, for example:

```
for (var i = 0, acc = 0; i < arr.length; i++)
  acc += arr[i];
```

If the compiler would translate it to the machine code straight out of AST, it may resemble (in very abstract and detached from reality instruction set):

```
acc = 0;
i = 0;
loop {
  // Load `.length` field of arr
  tmp = loadArrayLength(arr);
  if (i >= tmp)
    break;

  // Check that `i` is between 0 and `arr.length`
  // (NOTE: This is necessary for fast loads and
  // stores).
  checkIndex(arr, i);

  // Load value
  acc += load(arr, i);

  // Increment index
  i += 1;
}
```

It may not be obvious, but this code is far from optimal. Array length does not really change inside of the loop, and the range checks are not necessary at all. Ideally, it should look like this:

```
acc = 0;
i = 0;
len = loadArrayLength(arr);
loop {
  if (i >= tmp)
    break;

  acc += load(arr, i);
  i += 1;
}
```

Let's try to imagine how we could do this.

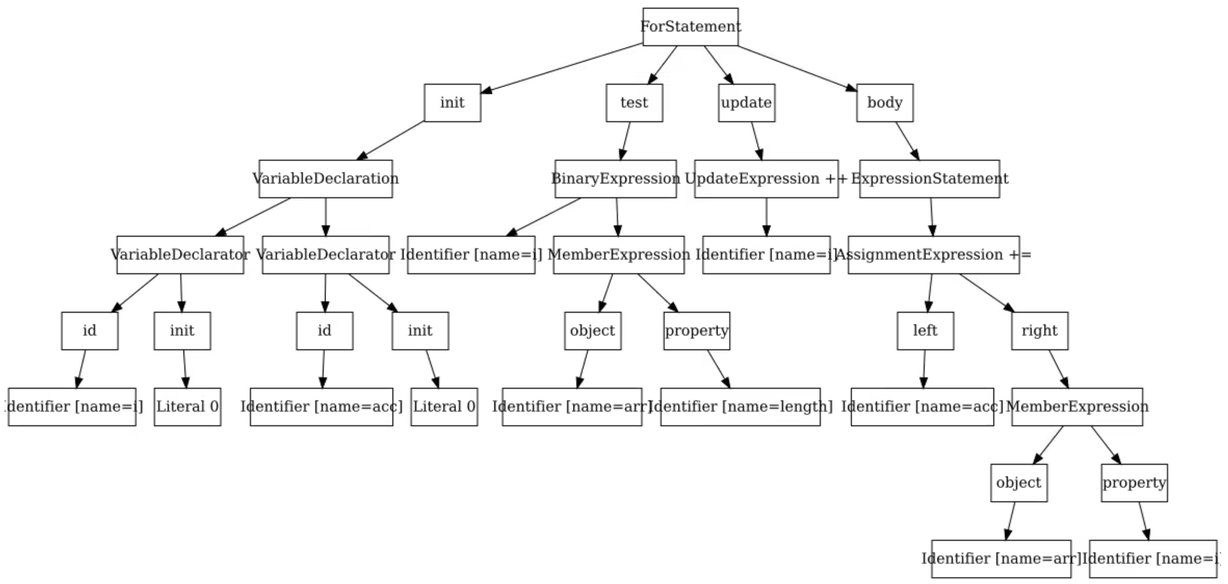
Suppose we have an AST at hand, and we try to generate the machine code straight out of it:

(NOTE: Generated with esprima)

```
{ type: 'ForStatement',

  //
  // This is `var i = 0;`
  //
  init:
   { type: 'VariableDeclaration',
     declarations:
      [ { type: 'VariableDeclarator',
          id: { type: 'Identifier', name: 'i' },
          init: { type: 'Literal', value: 0, raw: '0' } },
        { type: 'VariableDeclarator',
          id: { type: 'Identifier', name: 'acc' },
          init: { type: 'Literal', value: 0, raw: '0' } }],
     kind: 'var' },

  //
  // `i < arr.length`
  //
  test:
```

```
 { type: 'BinaryExpression',
   operator: '<',
   left: { type: 'Identifier', name: 'i' },
   right:
    { type: 'MemberExpression',
      computed: false,
      object: { type: 'Identifier', name: 'arr' },
      property: { type: 'Identifier', name: 'length' } } },

//
// `i++`
//
update:
 { type: 'UpdateExpression',
   operator: '++',
   argument: { type: 'Identifier', name: 'i' },
   prefix: false },

//
// `arr[i] += 1;`
//
body:
 { type: 'ExpressionStatement',
   expression:
    { type: 'AssignmentExpression',
      operator: '+=',
      left: { type: 'Identifier', name: 'acc' },
      right:
       { type: 'MemberExpression',
         computed: true,
         object: { type: 'Identifier', name: 'arr' },
         property: { type: 'Identifier', name: 'i' } } } }
```

This JSON could also be visualized:



This is a tree, so it is very natural to traverse it from the top to the bottom, generating the machine code as we visit the AST nodes. The problem with this approach is that the information about variables is very sparse, and is spread through the different tree nodes.

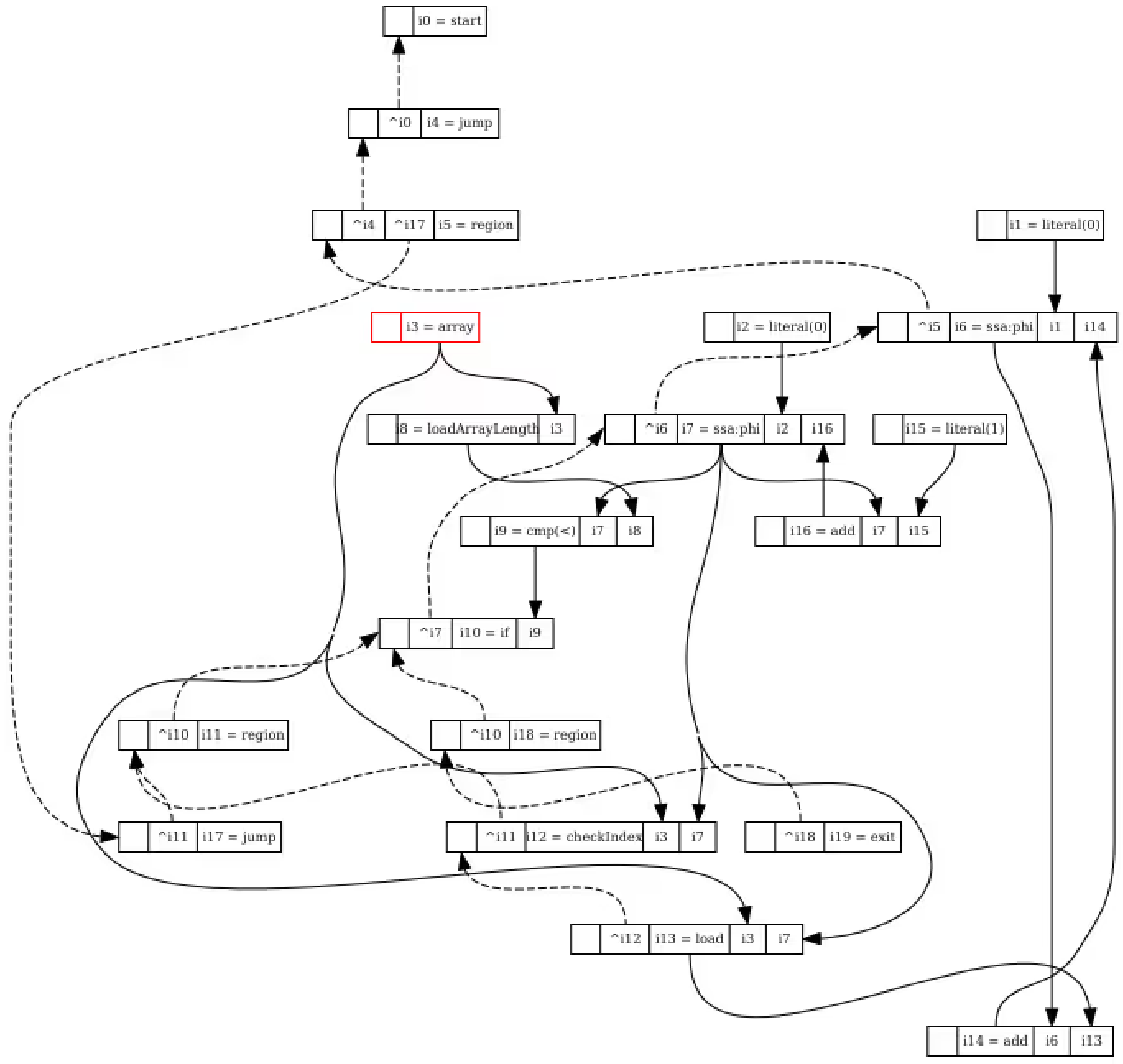
Again, to safely move the length lookup out of the loop we need to know that the array length does not change between the loop's iterations. Humans can do it easily just by looking at the source code, but the compiler needs to do quite a lot of work to confidently extract those facts directly from the AST.

Like many other compiler problems, this is often solved by lifting the data into a more appropriate abstraction layer, i.e. intermediate representation. In this particular case that choice of IR is known as a data-flow graph (DFG). Instead of talking about syntax-entities (like `for loops`, `expressions`, ...), we should talk about the data itself (read, variables values), and how it changes through the program.

Data-flow Graph

In our particular example, the data we are interested in is the value of variable `arr`. We want to be able to easily observe all uses of it to verify that there are no out-of-bounds accesses or any other change that would modify the length of the array.

This is accomplished by introducing "def-use" (definition and uses) relationship between the different data values. Concretely, it means that the value has been declared once (*node*), and that it has been used somewhere to create new values (*edge* for every use). Obviously, connecting different values together will form a **data-flow graph** like this:



Note the red `array` box in this vast graph. The solid arrows going out of it represent uses of this value. By iterating over those edges, the compiler can derive that the value of `array` is used at:

- `loadArrayLength`
- `checkIndex`
- `load`

Such graphs are constructed in the way that explicitly "clones" the array node, if its value was accessed in a destructive manner (i.e. stores, length sizes). Whenever we see `array` node and observe its uses - we are always certain that its value does not change.

It may sound complicated, but this property of the graph is quite easy to achieve. The graph should follow Single Static Assignment (SSA) rules. In short, to convert any program to SSA the compiler needs to rename all assignments and later uses of the variables, to make sure that each variable is assigned only once.

Example, before SSA:

```
var a = 1;
console.log(a);
a = 2;
console.log(a);
```

After SSA:

```
var a0 = 1;
console.log(a0);
var a1 = 2;
console.log(a1);
```

This way, we can be sure that when we are talking about `a0` - we are actually talking about a single assignment to it. This is really close to how people do things in the functional languages!

Seeing that `loadArrayLength` has no control dependency (i.e. no dashed lines; we will talk about them in a bit), compiler may conclude that this node is free to move anywhere it wants to be and can be placed outside of the loop. By going through the graph further, we may observe that the value of `ssa:phi` node is always between `0` and `arr.length`, so the `checkIndex` may be removed altogether.

Pretty neat, isn't it?

Control Flow Graph

We just used some form of data-flow analysis to extract information from the program. This allows us to make safe assumptions about how it could be optimized.

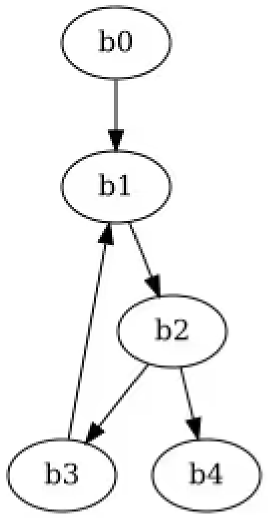
This *data-flow representation* is very useful in many other cases too. The only problem is that by turning our code into this kind of graph, we made a step backwards in our representation chain (from the source code to the machine code). This intermediate representation is less suitable for generating machine code than even the AST.

The reason is that the machine is just a sequential list of instructions, which the CPU executes one-after-another. Our resulting graph doesn't appear to convey that. In fact, there is no enforced ordering in it at all.

Usually, this is solved by grouping the graph nodes into blocks. This representation is known as a Control Flow Graph (CFG). Example:

```
b0 {
  i0 = literal 0
  i1 = literal 0

  i3 = array
  i4 = jump ^b0
}
b0 -> b1

b1 {
  i5 = ssa:phi ^b1 i0, i12
  i6 = ssa:phi ^i5, i1, i14

  i7 = loadArrayLength i3
  i8 = cmp "<", i6, i7
  i9 = if ^i6, i8
}
b1 -> b2, b3
b2 {
  i10 = checkIndex ^b2, i3, i6
  i11 = load ^i10, i3, i6
  i12 = add i5, i11
  i13 = literal 1
  i14 = add i6, i13
  i15 = jump ^b2
}
b2 -> b1

b3 {
  i16 = exit ^b3
}
```

It is called a graph not without the reason. For example, the bXX blocks represent nodes, and the `bXX -> bYY` arrows represent edges. Let's visualize it:



As you can see, there is code before the loop in block `b0`, loop header in `b1`, loop test in `b2`, loop body in `b3`, and exit node in `b4`.

Translation to machine code is very easy from this form. We just replace `iXX` identifiers with CPU register names (in some sense, CPU registers are sort of variables, the CPU has a limited amount of registers, so we need to be careful to not run out of them), and generating machine code for each instruction, line-by-line.

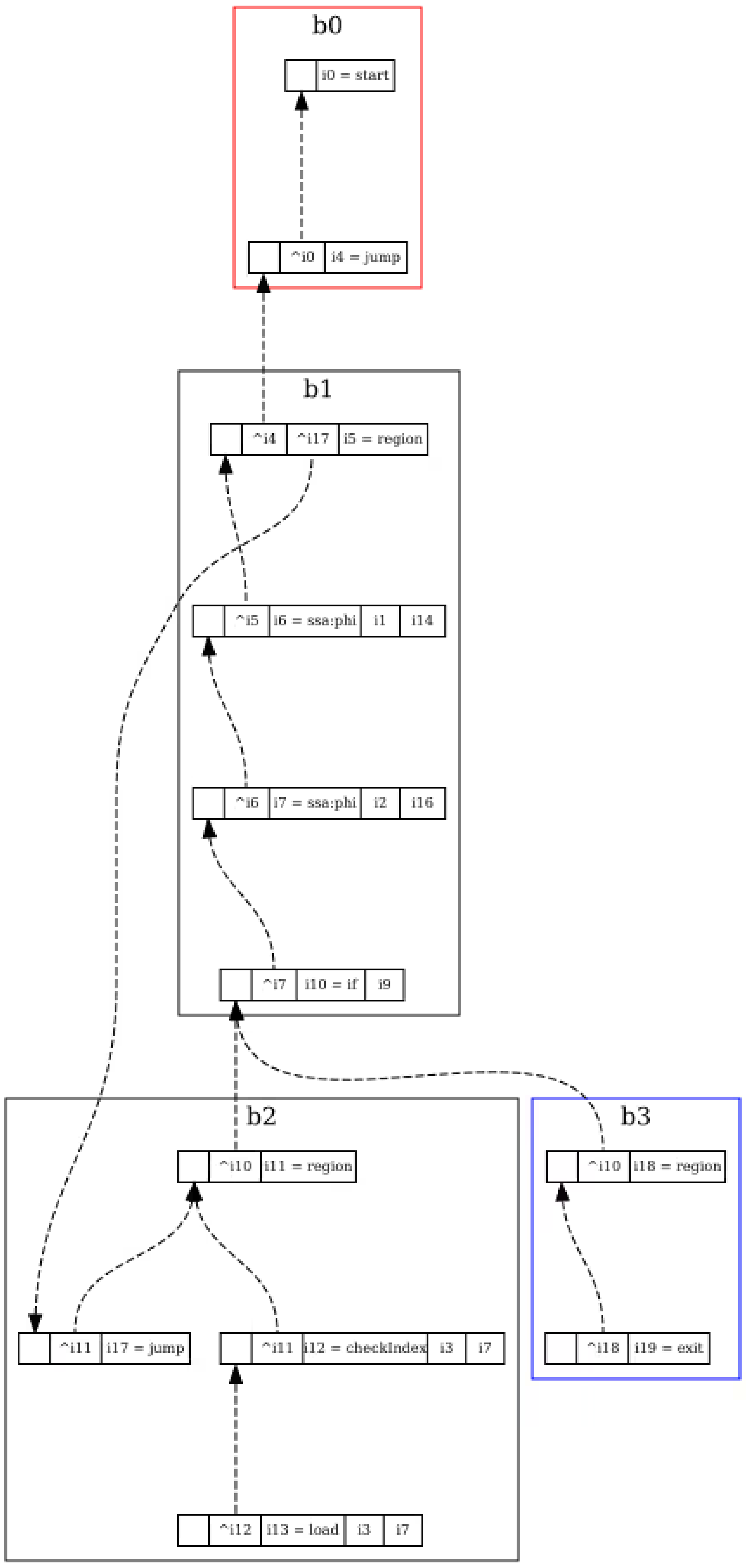
To recap, CFG has data-flow relations and also ordering. This allows us to utilize it for both data-flow analysis and machine code generation. However, attempting to optimize the CFG, by manipulating the blocks and their contents contained within it, can quickly become complex and error-prone.

Instead, Clifford Click and Keith D. Cooper proposed to use an approach called **sea-of-nodes**, the very topic of this blog post!

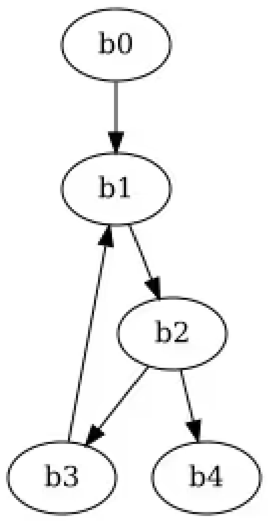
Sea-of-Nodes

Remember our fancy data-flow graph with dashed lines? Those dashed-lines are actually what make that graph a **sea-of-nodes** graph.

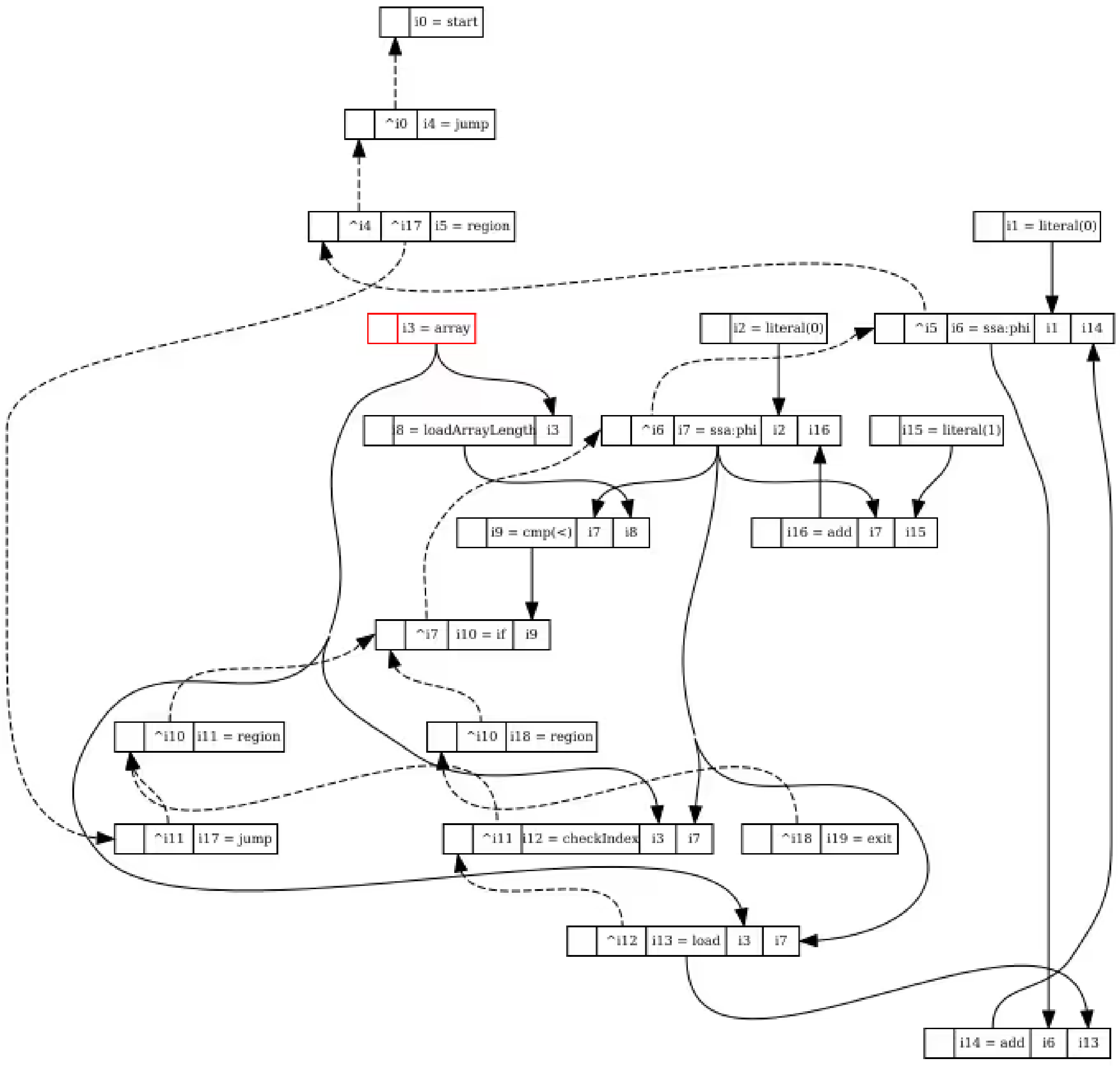
Instead of grouping nodes in blocks and ordering them, we choose to declare the control dependencies as the dashed edges in a graph. If we will take that graph, remove everything **non-dashed**, and group things a bit we will get:


b0
i0 = start
^i0
i4 = jump
b1
^i4
^i17
i5 = region
^i5
i6 = ssa:phi
i1
i14
^i6
i7 = ssa:phi
i2
i16
^i7
i10 = if
i9
b2
^i10
i11 = region
^i11
i17 = jump
^i11
i12 = checkIndex
i3
i7
^i12
i13 = load
i3
i7
b3
^i10
i18 = region
^i18
i19 = exit

With a bit of imagination and node reordering, we can see that this graph is the same as the simplified CFG graphs that we have just seen above:



Let's take another look at the **sea-of-nodes** representation:



The striking difference between this graph and CFG is that there is no ordering of the nodes, except the ones that have control dependencies (in other words, the nodes participating in the control flow).

This representation is very powerful way to look at the code. It has all insights of the general data-flow graph, and could be changed easily without constantly removing/replacing nodes in the blocks.

Reductions

Speaking of changes, let's discuss the way to modify the graph. The sea-of-nodes graph is usually modified by doing graph reductions. We just queue all nodes in the graph. Invoke our reduction function for every node in the queue. Everything that this function touches (changes, replaces) is queued back, and will be passed to the function later on. If you have many reductions, you can just stack them up together and invoke all of them on each node in the queue, or alternatively, you can just apply them one after another, if they depend on the final state of each other. It works like a charm!

I have written a JavaScript toolset for my sea-of-nodes experiments, which includes:

- json-pipeline - the builder and stdlib of the graph. Provides methods to create nodes, add inputs to them, change their control dependencies, and export/import the graph to/from the printable data!
- json-pipeline-reducer - the reductions engine. Just create a reducer instance, feed it several reduction functions, and execute the reducer on the existing json-pipeline graph.
- json-pipeline-scheduler - library for putting back unordered graph in a limited amount of blocks connected to each other by control edges (dashed lines).

Combined together, these tools can solve many problems that could be formulated in terms of data-flow equations.

Example of reduction, which will optimize our initial JS code:

```
for (var i = 0, acc = 0; i < arr.length; i++)
  acc += arr[i];
```

TL;DR

This code chunk is quite big, so if you want to skip it - here are the notes of what we will do below:

- Compute integer ranges of various nodes: literal, add, phi
- Compute limits that apply to branch's body
- Apply range and limit information (`i` is always a non-negative number limited by `arr.length`) to conclude that length check is not necessary and can be removed
- `arr.length` will be moved out of the loop automatically by `json-pipeline-scheduler`. This is because it does Global Code Motion to schedule nodes in blocks.

```
// Just for viewing graphviz output
var fs = require('fs');

var Pipeline = require('json-pipeline');
var Reducer = require('json-pipeline-reducer');
var Scheduler = require('json-pipeline-scheduler');

//
// Create empty graph with CFG convenience
// methods.
//
var p = Pipeline.create('cfg');

//
// Parse the printable data and generate
// the graph.
//
p.parse(`pipeline {
  b0 {
    i0 = literal 0
    i1 = literal 0

    i3 = array
    i4 = jump ^b0
  }
  b0 -> b1

  b1 {
    i5 = ssa:phi ^b1 i0, i12
    i6 = ssa:phi ^i5, i1, i14

    i7 = loadArrayLength i3
    i8 = cmp "<", i6, i7
    i9 = if ^i6, i8
  }
  b1 -> b2, b3
  b2 {
    i10 = checkIndex ^b2, i3, i6
    i11 = load ^i10, i3, i6
    i12 = add i5, i11
    i13 = literal 1
    i14 = add i6, i13
    i15 = jump ^b2
  }
  b2 -> b1

  b3 {
    i16 = exit ^b3
  }
}`, { cfg: true }, 'printable');

if (process.env.DEBUG)
  fs.writeFileSync('before.gv', p.render('graphviz'));

//
// Just a helper to run reductions
//

function reduce(graph, reduction) {
  var reducer = new Reducer();
  reducer.addReduction(reduction);
  reducer.reduce(graph);

}

//
// Create reduction
//
```

```
var ranges = new Map();

function getRange(node) {
  if (ranges.has(node))
    return ranges.get(node);

  var range = { from: -Infinity, to: +Infinity, type: 'any' };
  ranges.set(node, range);
  return range;
}

function updateRange(node, reducer, from, to) {
  var range = getRange(node);

  // Lowest type, can't get upwards
  if (range.type === 'none')
    return;

  if (range.from === from && range.to === to && range.type === 'int
    return;

  range.from = from;
  range.to = to;
  range.type = 'int';
  reducer.change(node);
}

function updateType(node, reducer, type) {
  var range = getRange(node);

  if (range.type === type)
    return;

  range.type = type;
  reducer.change(node);
}

//
// Set type of literal
//
```

```
function reduceLiteral(node, reducer) {
  var value = node.literals[0];
  updateRange(node, reducer, value, value);
}

function reduceBinary(node, left, right, reducer) {
  if (left.type === 'none' || right.type === 'none') {
    updateType(node, reducer, 'none');
    return false;
  }

  if (left.type === 'int' || right.type === 'int')
    updateType(node, reducer, 'int');

  if (left.type !== 'int' || right.type !== 'int')
    return false;

  return true;
}

//
// Just join the ranges of inputs
//
function reducePhi(node, reducer) {
  var left = getRange(node.inputs[0]);
  var right = getRange(node.inputs[1]);

  if (!reduceBinary(node, left, right, reducer))
    return;

  if (node.inputs[1].opcode !== 'add' || left.from !== left.to)
    return;

  var from = Math.min(left.from, right.from);
  var to = Math.max(left.to, right.to);
  updateRange(node, reducer, from, to);
}

//
// Detect: phi = phi + <positive number>, where initial phi is numbe
// report proper range.
//
function reduceAdd(node, reducer) {
  var left = getRange(node.inputs[0]);
  var right = getRange(node.inputs[1]);

  if (!reduceBinary(node, left, right, reducer))
    return;

  var phi = node.inputs[0];
  if (phi.opcode !== 'ssa:phi' || right.from !== right.to)
    return;

  var number = right.from;
  if (number <= 0 || phi.inputs[1] !== node)
    return;

  var initial = getRange(phi.inputs[0]);
  if (initial.type !== 'int')
    return;

  updateRange(node, reducer, initial.from, +Infinity);
}

var limits = new Map();

function getLimit(node) {
  if (limits.has(node))
    return limits.get(node);

  var map = new Map();
  limits.set(node, map);
  return map;
}

function updateLimit(holder, node, reducer, type, value) {
  var map = getLimit(holder);
  if (!map.has(node))
    map.set(node, { type: 'any', value: null });
  var limit = map.get(node);
  if (limit.type === type && limit.value === value)
    return;
  limit.type = type;
  limit.value = value;
  reducer.change(holder);
}

function mergeLimit(node, reducer, other) {
  var map = getLimit(node);
  var otherMap = getLimit(other);

  otherMap.forEach(function(limit, key) {
    updateLimit(node, key, reducer, limit.type, limit.value);
  });
}

//
// Propagate limit from: X < Y to `if`'s true branch
//
function reduceIf(node, reducer) {
  var test = node.inputs[0];
  if (test.opcode !== 'cmp' || test.literals[0] !== '<')
    return;

  var left = test.inputs[0];
  var right = test.inputs[1];

  updateLimit(node.controlUses[0], left, reducer, '<', right);
  updateLimit(node.controlUses[2], left, reducer, '>=', right);
}

//
// Determine ranges and limits of
// the values.
//

var rangeAndLimit = new Reducer.Reduction({
  reduce: function(node, reducer) {
    if (node.opcode === 'literal')
```

```
      reduceLiteral(node, reducer);
    else if (node.opcode === 'ssa:phi')
      reducePhi(node, reducer);
    else if (node.opcode === 'add')
      reduceAdd(node, reducer);
    else if (node.opcode === 'if')
      reduceIf(node, reducer);
  }
});
reduce(p, rangeAndLimit);

//
// Now that we have ranges and limits,
// time to remove the useless array
// length checks.
//

function reduceCheckIndex(node, reducer) {
  // Walk up the control chain
  var region = node.control[0];
  while (region.opcode !== 'region' && region.opcode !== 'start')
    region = region.control[0];

  var array = node.inputs[0];
  var index = node.inputs[1];

  var limit = getLimit(region).get(index);
  if (!limit)
    return;

  var range = getRange(index);

  // Negative array index is not valid
  if (range.from < 0)
    return;

  // Index should be limited by array length
  if (limit.type !== '<' ||
      limit.value.opcode !== 'loadArrayLength' ||
      limit.value.inputs[0] !== array) {
```

```
      return;
    }

    // Check is safe to remove!
    reducer.remove(node);
  }

  var eliminateChecks = new Reducer.Reduction({
    reduce: function(node, reducer) {
      if (node.opcode === 'checkIndex')
        reduceCheckIndex(node, reducer);
    }
  });
  reduce(p, eliminateChecks);

  //
  // Run scheduler to put everything
  // back to the CFG
  //

  var out = Scheduler.create(p).run();
  out.reindex();

  if (process.env.DEBUG)
    fs.writeFileSync('after.gv', out.render('graphviz'));

  console.log(out.render({ cfg: true }, 'printable'));
```

Thank you for reading this. Please expect more information about this sea-of-nodes approach.

Special thanks to Paul Fryzel for proof-reading this, and providing valuable feedback and grammar fixes!

Previous: Diving into C++ internals of node
Next: uv_link_t - libuv pipeline