Optimizing with persistent data structures

Adventures in CPS soup

Andy Wingo ~ wingo@igalia.com

wingolog.org ~ @andywingo

Agenda

SSA and CPS: Tribal lore

A modern CPS

Programs as values: structure

Programs as values: transformation

Evaluation

How we got here

1928 Hilbert: Can has Entscheidungsproblem?

1936 Church: Nope!

Also here is the lambda calculus

For identifiers x and terms t and s, a term is either

- A variable reference: x
- A lambda abstraction: $\lambda x.\ t$
- An application: $(t\ s)$

Computing with lambda

Lambda abstractions bind variables lexically

To compute with the lambda calculus:

❧ take a term and *reduce* it, exhaustively

Sounds like compilation, right?

GOTO?

1958 McCarthy: Hey the lambda calculus is not bad for performing computation!

1965 Landin: Hey we can understand ALGOL 60 using the lambda calculus!

What about `GOTO`?

Landin: `J` operator captures state of SECD machine that can be returned to later

To J or not to J

1964 van Wijngaarden: Not to J!

Just transform your program

1970 F. Lockwood Morris: Re-discovers program transformation

(Iinspired by LISP 1.5 code!)

```
function f()
  local x = foo() ? y() : z();
  return x
end

function f(k)
   function ktest(val)
     function kt() return y(kret) end
     function kf() return z(kret) end
     if val then return kt() else return kf(
   end
   function kret(x) return k(x) end
   return foo(ktest)
end
```

Nota bene

```
function kt() return y(kret) end
```

All calls are tail calls

1970 Chris Wadsworth: Hey! Result of the Morris transformation is the *continuation*: the meaning of the rest of the program

Function calls are passed an extra argument: the continuation

Variables bound by continuations

Compiling with CPS

1977 Guy Steele: Hey we can compile with this!

Tail calls are literally `GOTO`, potentially passing values.

1978 Guy Steele: RABBIT Scheme compiler using CPS as IL

Rewrite so all calls are tail calls, compile as jumps

1984 David Kranz: ORBIT Scheme compiler using CPS, even for register allocation

What's missing?

1970 Fran Allen and John Cocke: Flow analysis

Both Turing award winners!

Range checking, GCSE, DCE, code motion, strength reduction, constant propagation, scheduling

Flow analysis for CPS

1984 Shivers: Whoops this is hard

Flow analysis in CPS: given $(f\ x)$, what values flow to f and x?

For data-flow analysis, you need control-flow analysis

For control-flow analysis, you need data-flow analysis

Solution 1: k-CFA

Solve both problems at once

1991 Shivers: k-CFA family of higher-order flow analysis

Based on CPS

Parameterized by precision

- 0-CFA: first order, quadratic...
- 1-CFA: second order, exponential!
- k-CFA: order k, exponential

2009 Van Horn: $k > 0$ intractable

Solution 2: Some conts are labels

Observation: Lambda terms in CPS are of three kinds

Procs

Entry points to functions of source program

```
function f(k)
    function ktest(val)
      function kt() return y(kret) end
      function kf() return z(kret) end
      if val then return kt() else return kf(
    end
    function kret(x) return k(x) end
    return foo(ktest)
end
```

Conts

Return points from calls; synthetic

```
function f(k)
   function ktest(val)
     function kt() return y(kret) end
     function kf() return z(kret) end
     if val then return kt() else return kf(
   end
   function kret(x) return k(x) end
   return foo(ktest)
end
```

Jumps

Jump targets; synthetic

```
function f(k)
   function ktest(val)
      function kt() return y(kret) end
      function kf() return z(kret) end
      if val then return kt() else return kf(
   end
   function kret(x) return k(x) end
   return foo(ktest)
end
```

Solution 2: Some conts are labels

1995 Kelsey: "In terms of compilation strategy, *conts* are return points, *jumps* can be compiled as `goto`s, and *procs* require a complete procedure-call mechanism."

Separate control and data flow

1992 Appel, "Compiling with Continuations" (ML)

What about SSA?

1986-1988 Rosen, Wegman, Ferrante, Cytron, Zadeck: “Binding, not assignment”

“The right number of names”

Better notation makes it easier to transform programs

Initial application of SSA was GVN

SSA and CPS

1995 Kelsey: “Making [continuation uses] syntactically distinct restricts how continuations are used and makes CPS and SSA entirely equivalent.”

SSA: Definitions must dominate uses

CPS embeds static proof of SSA condition: all uses must be in scope

1998 Appel: “SSA is Functional Programming”

Modern CPS

2007 Kennedy: Compiling with Continuations, Continued

Nested scope

Syntactic difference between continuations (control) and variables (data)

Why CPS in 2016?

SSA: How do I compile loops?

CPS: How do I compile functions?

“Get you a compiler that can do both”

Example: Contification

A function or clique of functions that always continues to the same label (calls the same continuation) can be integrated into the caller

Like inlining, widens first-order flow graph: a mother optimization

Unlike inlining, always a good idea: always a reduction

CPS facilitates contification

- Concept of continuation
- Globally unique labels and variable names
- Interprocedural scope
- Single term for program

Possible in SSA too of course

And yet

CPS: all uses must be in scope... but not all dominating definitions are in scope

Transformations can corrupt scope tree

```
function b0(k)
  function k1(v1) return k2() end
  function k2() return k(v1) end # XX
  k1(42)
end
```

1999 Fluet and Weeks: MLton switches to SSA

Alternate solution: CPS without nesting

Values in scope are values that dominate

Program is soup of continuations

“CPS soup”

CPS in Guile

```
(define-type Label Natural)

(struct Program
  ([entry : Label]
   [conts : (Map Label Cont)]))
```

Conts

```
(define-type Var Natural)
(define-type Vars (Listof Var))

(struct KEntry
  ([body : Label] [exit : Label]))
(struct KExpr
  ([vars : Vars] [k : Label] [exp : Exp]))
(struct KExit)

(define-type Cont (U KEntry KExpr KExit))
```

Exps

```
(define-type Op (U 'lookup 'add1 ...))

(struct Primcall ([op : Op] [args : Vars]))
(struct Branch ([kt : Label] [exp : Expr]))
(struct Call   ([proc : Var] [args : Vars]))
(struct Const  ([val : Literal]))
(struct Func   ([entry : Label]))
(struct Values ([args : Vars]))

(define-type Exp
 (U Primcall Branch Call Const Func Values))
```

See `language/cps.scm` for full details

```
;; (lambda () (if (foo) (y) #f))

(Map
 (k0 (KEntry k1 k10))
 (k1 (KExpr ()   k2  (Const 'foo)))
 (k2 (KExpr (v0) k3  (Primcall 'lookup (v0))
 (k3 (KExpr (v1) k4  (Call v1 ())))
 (k4 (KExpr (v2) k5  (Branch k8 (Values (v1)
 (k5 (KExpr ()   k6  (Const 'y)))
 (k6 (KExpr (v3) k7  (Primcall 'lookup (v3))
 (k7 (KExpr (v4) k10 (Call v4 ())))
 (k8 (KExpr ()   k9  (Const #f)))
 (k9 (KExpr (v5) k10 (Values (v5))))
 (k10 (KExit)))
```

Salient details

Variables available for use a flow property

Variables bound by `KExpr`; values given by predecessors

Expressions have labels and continue to other labels

Return by continuing to the label identifying function's `KExit`

Orders of CPS

Two phases in Guile

- Higher-order: Variables in “outer” functions may be referenced directly by “inner” functions; primitive support for recursive function binding forms
- First-order: Closure representations chosen, free variables (if any) accessed through closure

“[Interprocedural] binding is better than assignment”

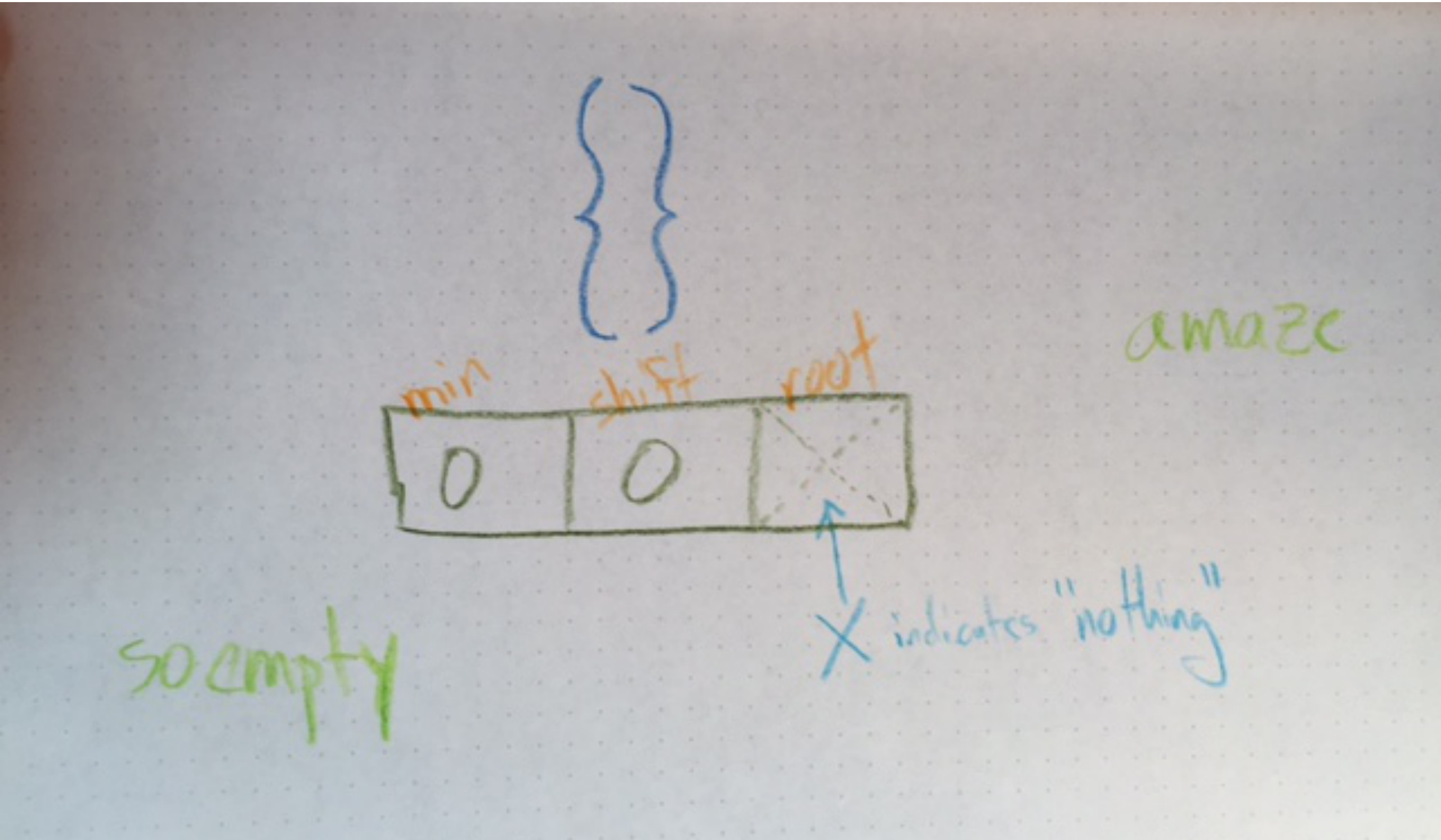
About those maps

```
(struct (v) IntMap
  ([min   : Natural]
   [shift : Natural]
   [root  : (U (Maybe v) (Branch v))]))
(define-type (Branch v)
  (U (Vectorof (Maybe Branch))
     (Vectorof (Maybe v))))
```

Shift 0 and root empty? `{}`

Shift 0? `{`*min*`: valueof(`*root*`)}`

Otherwise element *i* of *root*`[`*i*`]` is root for *min* +*i*`*2^(`*shift*`-5)`, at *shift*`-5`.


{ }
amaze
min
shift
root
0
0
X indicates "nothing"
so empty

aligned to 2^{shift}

$\{1 : a\}$

min	shift	root
1	0	a

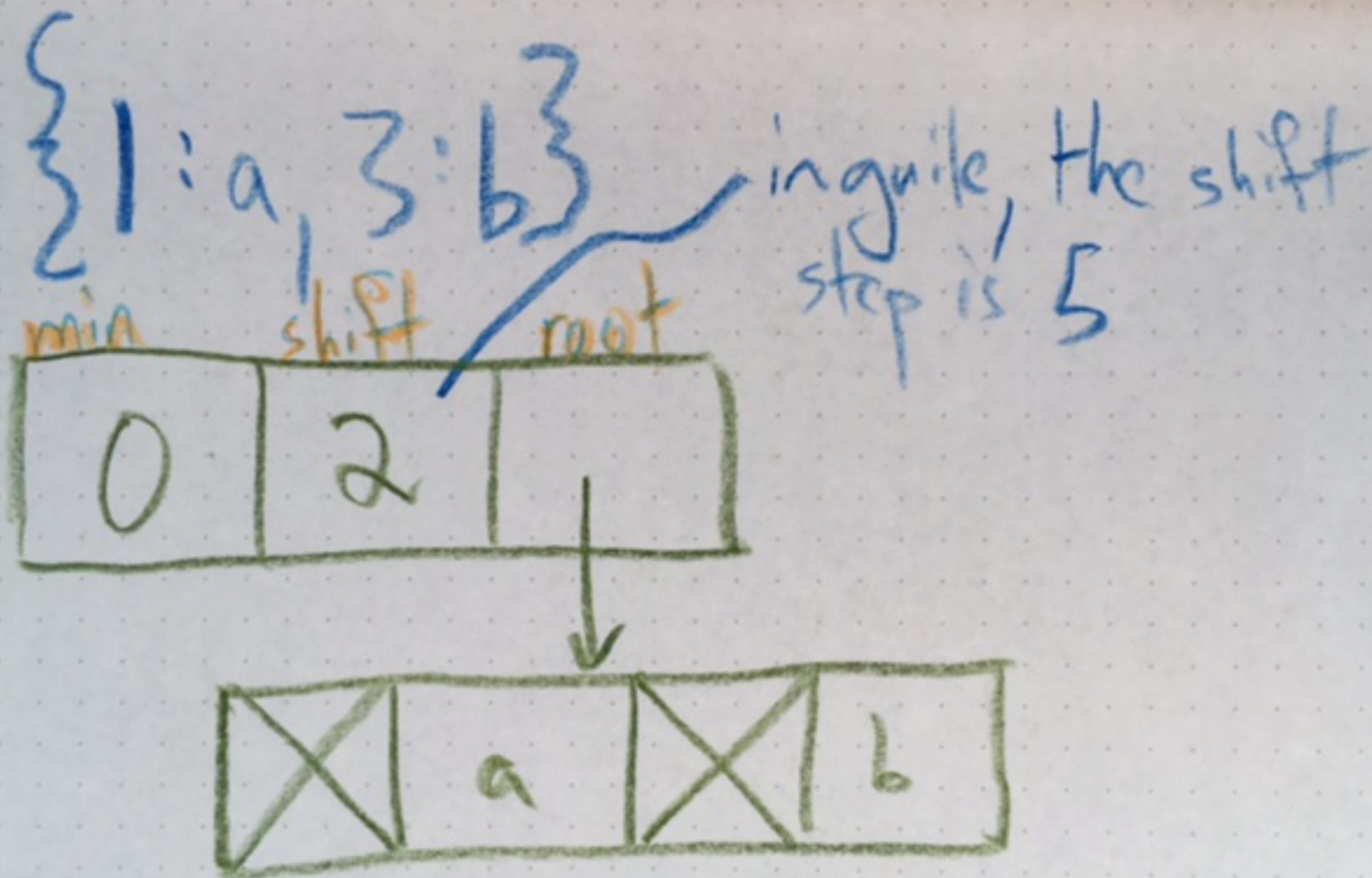
inclusive

not inclusive

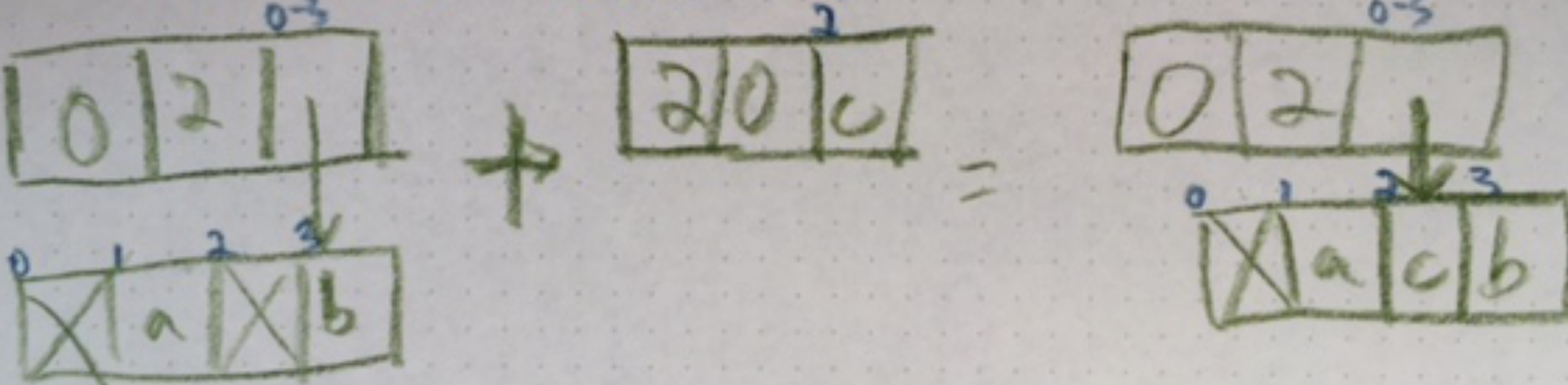
$[min, min + 2^{shift})$

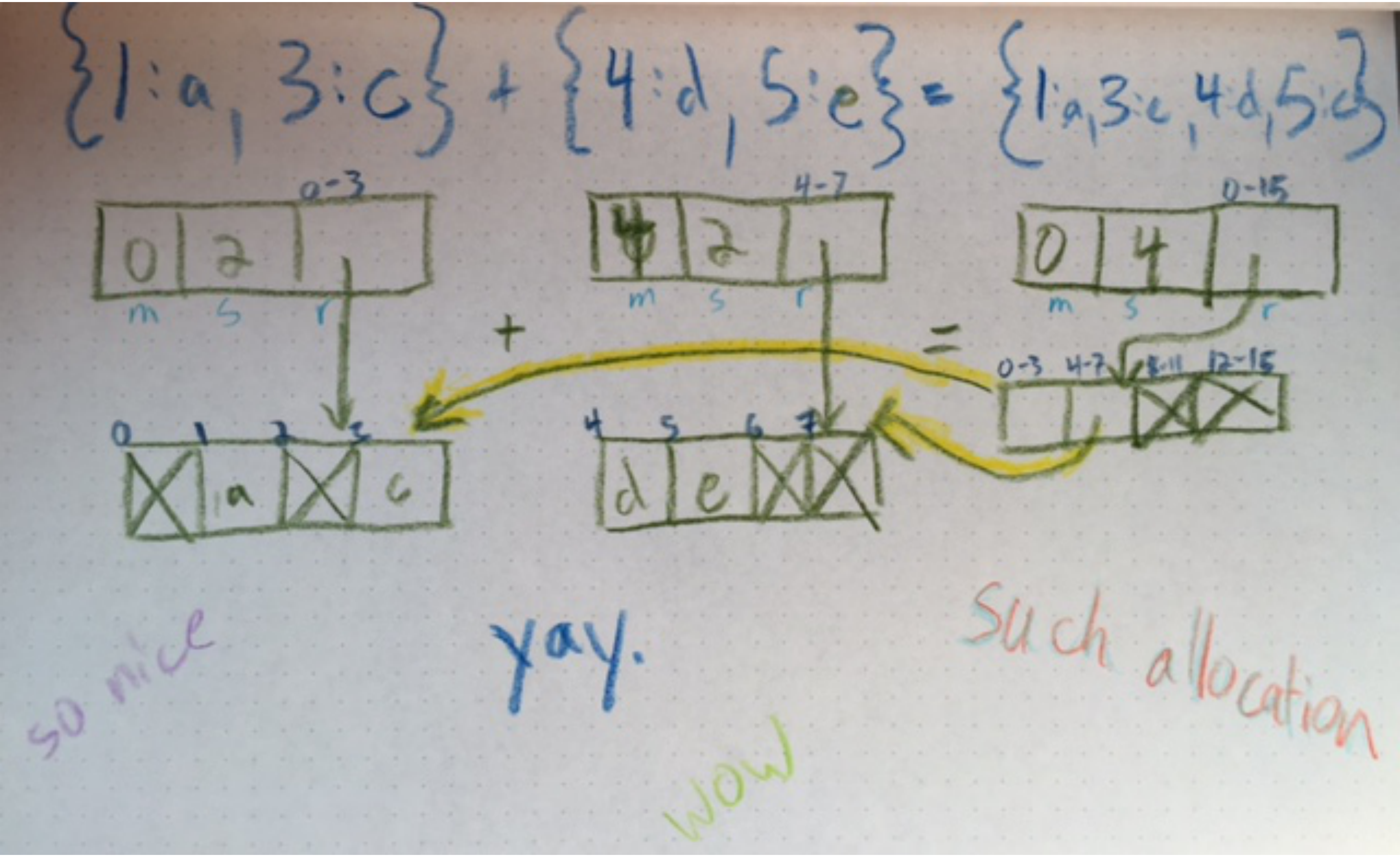
$[1, 1 + 2^{0})$

$[1, 2)$


{1: a, 3: b}
in guile, the shift step is 5
min
shift
root
0
2
a
b

$\{1:a, 3:b\} + \{2:c\} = \{1:a, 2:b, 3:b\}$





{1:a, 3:c} + {4:d, 5:e} = {1:a, 3:c, 4:d, 5:e}
0-3
0 2
m s r
4-7
4 2
m s r
0-15
0 4
m s r
+
=
0-3 4-7 8-11 12-15
0 1 2 3
a c
4 5 6 7
d e
so nice
yay.
such allocation
wow

Bagwell AMTs

Array Mapped Trie

Clojure-inspired data structures invented by Phil Bagwell

O(n log n) in size

Ref and update O(log n)

Visit-each near-linear

Unions and intersections very cheap

Clojure innovation

`clojure.org/transients`: Principled in-place mutation

```
(define (intmap-map proc map)
  (persistent-intmap
   (intmap-fold
     (lambda (k v out)
       (intmap-add! out k (proc k v)))
     map
     (transient-intmap empty-intmap))))
```

Still O(n log n) but significant constant factor savings

Intsets

“Which labels are in this function?”

```
(struct IntSet
  ([min   : Natural]
   [shift : Natural]
   [root  : (U Leaf Branch)]))
(define-type Leaf UInt32)
(define-type Branch
  (U (Vectorof (Maybe Branch))
     (Vectorof Leaf)))
```

Transient variants as well

Optimizing with persistent data structures

Example optimization: “Unboxing”

Objective: use specific limited-precision machine numbers instead of arbitrary-precision polymorphic numbers

```
function unbox_pass(conts):
  let out = conts
  for entry, body in conts.functions():
    let types = infer_types(conts, entry,
                            body)
    for label in body:
      match conts[label]:
        KExpr vars k (Primcall 'add1 (a)):
          if can_unbox?(label, k, a,
                        types, conts):
            out = unbox(label, vars, k, a,
                        out)
        _: pass
  return out
```

```
function can_unbox?(label, k, arg,
                    types, conts):
  match conts[k]:
    KExpr (result) _ _:
      let rtype, rmin, rmax =
        lookup_post_type(label, result)
      let atype, amin, amax =
        lookup_pre_type(label, a)
      return unboxable?(rtype, rmin, rmax)
         and unboxable?(atype, amin, amax)
```

```
function unbox(label, vars, k, arg, conts):
  let uarg, res = fresh_vars(conts, 2)
  let kbox, kop = fresh_labels(conts, 2)

  conts = conts.replace(label,
    KEntry vars kop (Primcall 'unbox (a)))

  conts = conts.add(kop,
    KEntry (ua) kbox (Primcall 'uadd1 (ua)))

  return conts.add(kbox,
    KEntry (res) k (Primcall 'box (res)))
```

Salient points

To get name of result(s), have to look at continuation

No easy way to get predecessors (without building predecessors map)

- No easy way to know if output var has other definitions

On the other hand... no easy way to write local-only passes

Backwards flow

```
y = x & 0xffffffff
```

We only need low 32 bits from x; can allow x to unbox...

...but can't reach through from & to x.

Solution: solve a flow problem (bits needed for each variable)

❧ Also works globally!

Whither yon basic block?

Not necessary; get in the way sometimes

Need globally unique names for terms anyway

Guile has terms that can bail out, unlike llvm; have to do big flow graph anyway

Odd: almost never need dominators! Full flow analysis instead.

Strengths

Simple – few moving parts

Immutability helps fit more of the problem into your head

Interprocedural bindings pre-closure-conversion easier to reason about than locations in global heap

Good space complexity for complicated flow analysis (type,range of all vars at all labels: $n \log n$)

Compared to SSA (1)

Just as rigid scheduling-wise (compare to sea-of-nodes)

Flow analysis over cont graph has more nodes than over basic block graph

Additional log n factor for most operations

Names as graph edges means lots of pointer chasing

Compared to SSA (2)

Sometimes have to renumber graph if pass wants specific ordering (usually topological)

Values that flow into phi vars have no names!

Lots of allocation (mitigate with zones?)

Always throwing away analysis

Summary

Better notation makes it easier to transform programs

If SSA + basic block graph works for you, great

If not, map to a notation that is more tractable for you, transform there, and come back

CPS name graph on persistent data structures seems to work for Guile; perhaps for you too?

Summary

Happy hacking!

`wingolog.org`

`@andywingo`

`wingo@igalia.com`