Writing Tucan

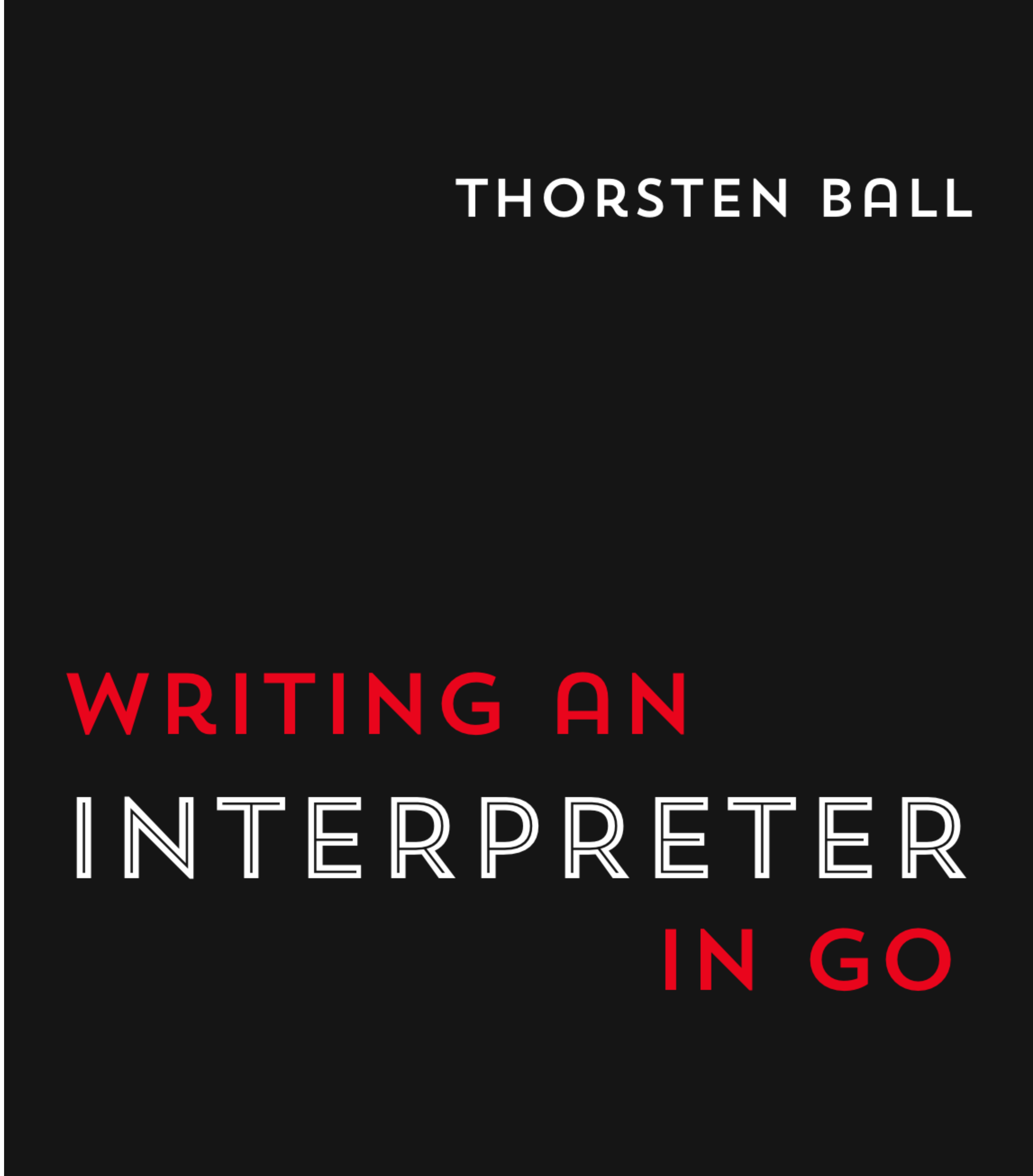
Writing an Optimizing Compiler in Rust

Thorsten Ball

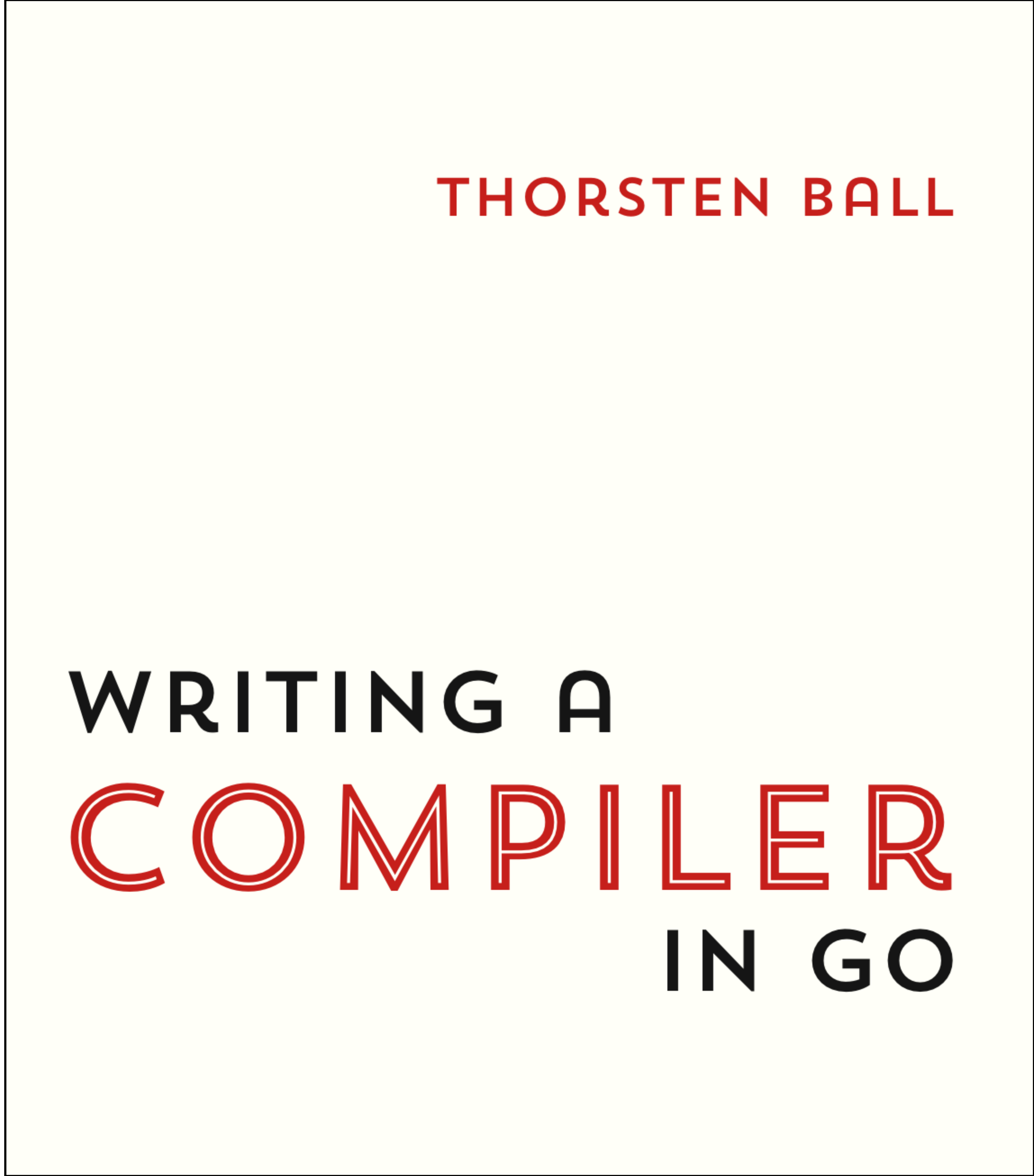
November 2024

thorstenball.com

2016



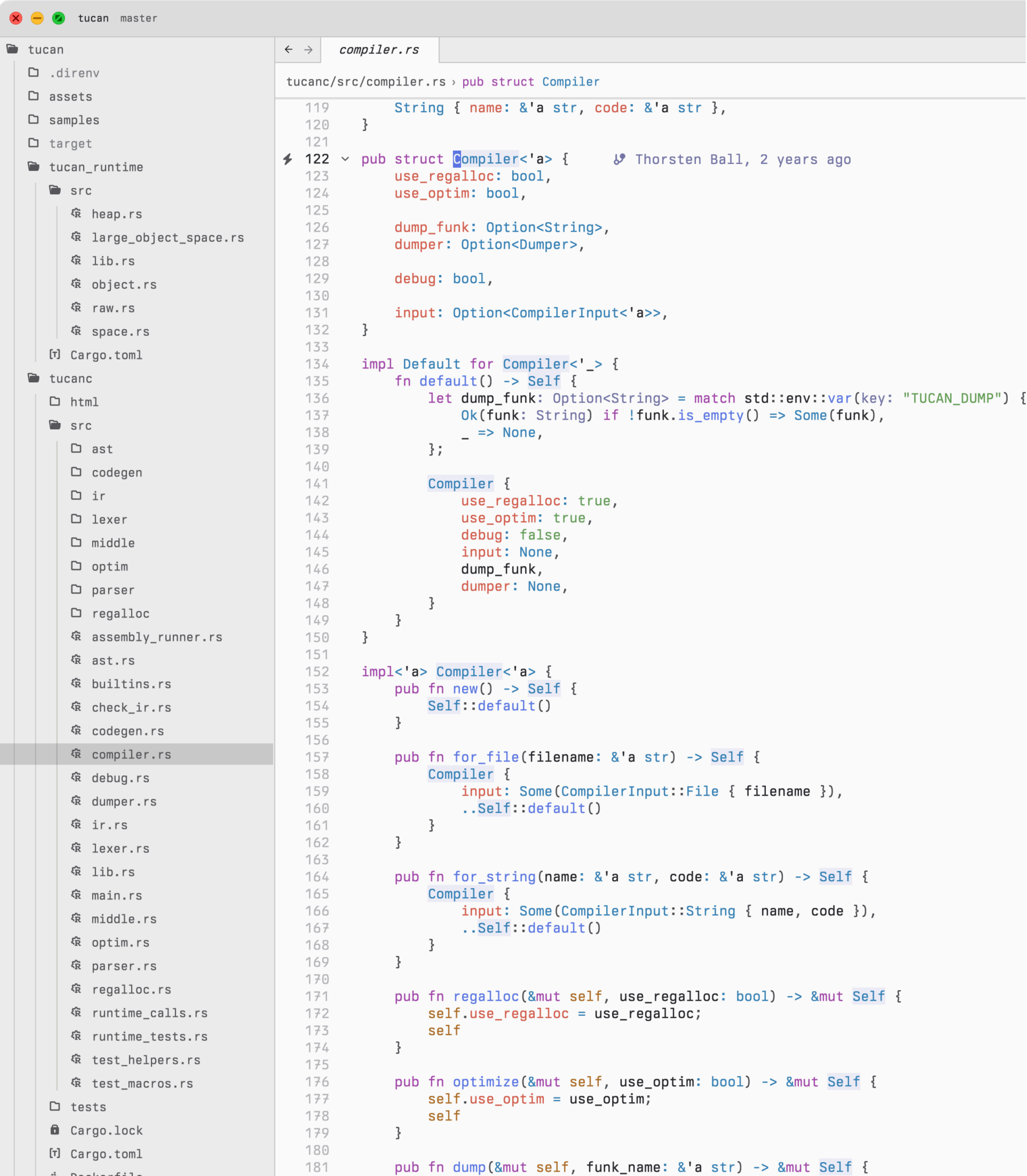
2018



2020

Tucan

- Optimizing compiler in Rust
- 18k lines of code, 0 third-party dependencies
- IR is a Control-Flow Graph in SSA form
- Optimizations
  - Dead Code Elimination
  - Sparse Conditional Constant Propagation
  - Dominator-based Value Numbering
  - Useless control-flow elimination
- Liveness analysis to compute live sets
- Linear scan register allocator
- x86-64 code generation
- Immix heap with GC



No:

- Language design
- Fancy type systems
- Novel implementation strategies
- Building the perfect system

Yes:

- Figuring out how an optimizing compiler works

```
funk my_function(a: int, b: int) → int {
    let c = a + b + 18 - 5 + 3 + 2;
    let d = 18 - 5 + 3;
    let e = a + b + 30;
    let f = 100 - a - b + 3 + 2;
    let g = c + d + e; // dead code
    let h = a + b + c + e + f;

    let k = 50;

    let result = 0;
    for (let i = 0; i < 10; i = i + 1) {
        result = result + 5;
        if (i < a) {
            if (k ≥ d) {
                double_print(i);
            } else {
                print_num(i);
            }
        } else {
            result = result + c;
        }
        for (let j = 5; j > 0; j = j - 1) {
            result = result - j;
        }
        print_num(result);
        print_string("-");
    }
    return result;
}
```

Lessons Learned

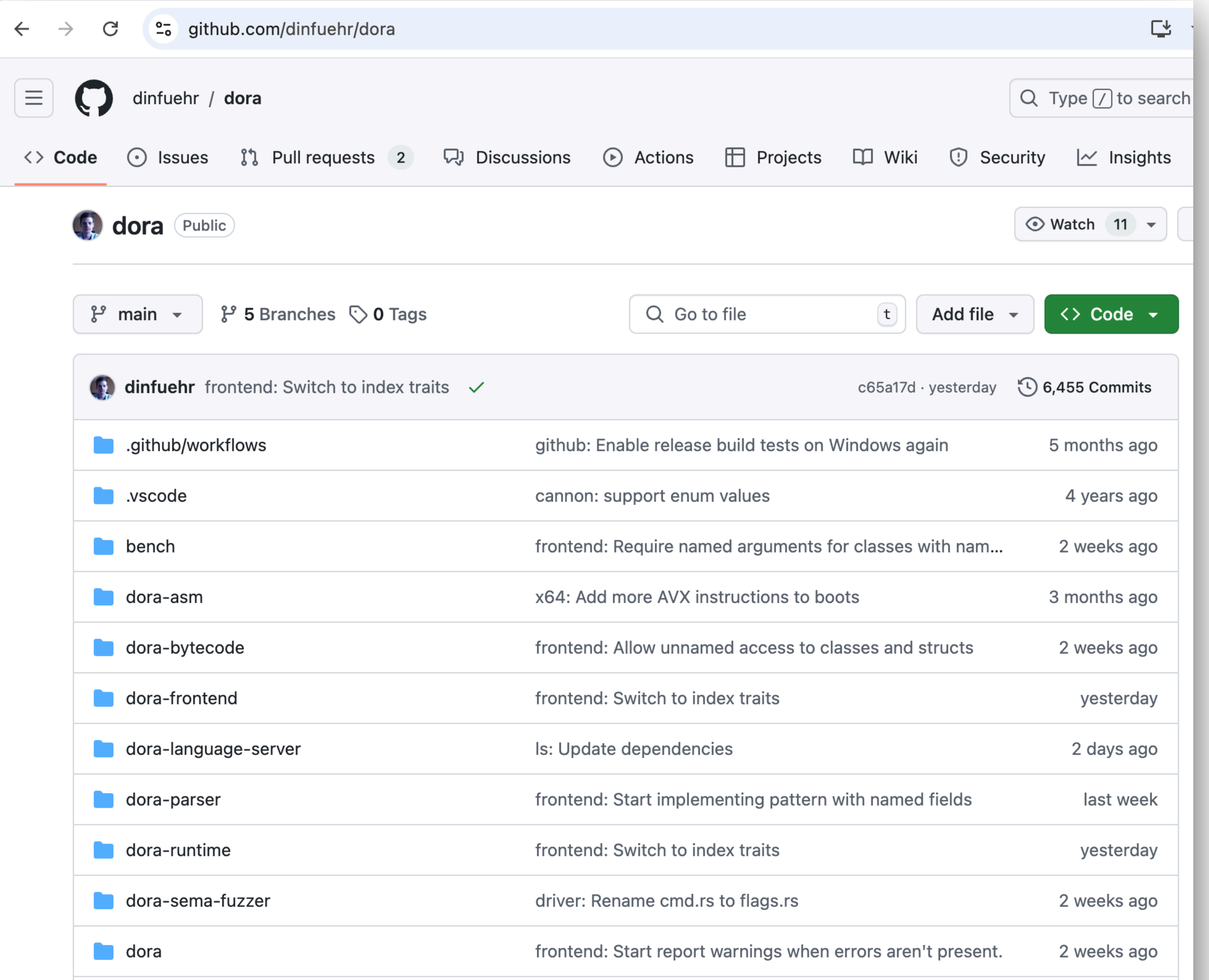
The bird’s name is Toucan

(But, hey: in German it’s “Tukan”)

Lexing & Parsing

Dora

Dominik Inführ's JIT compiler



Lesson:

Copy & Pasting & Deleting

… can kickstart projects

```
pub struct Parser<'a> {
    lexer: Lexer,
    token: Token,
    peek_token: Token,
    ast: &'a mut Ast,

    id_counter: RefCell<usize>,
    last_end: Option<u32>,
}

type ExprResult = Result<Box<Expr>, ParseErrorAndPos>;
type StmtResult = Result<Stmt, ParseErrorAndPos>;
type FunkResult = Result<Funk, ParseErrorAndPos>;

impl<'a> Parser<'a> {
    pub fn new(reader: Reader, ast: &'a mut Ast) -> Parser<'a> {
        let token: Token = Token::new(tok: TokenKind::Eof, pos: Position::r
        let peek_token: Token = Token::new(tok: TokenKind::Eof, pos: Positi
        let lexer: Lexer = Lexer::new(reader);

        Parser {
            lexer,
            token,
            peek_token,
            ast,
            id_counter: RefCell::new(1),
            last_end: Some(0),
        }
    }

    fn init(&mut self) -> Result<(), ParseErrorAndPos> {
        self.advance_token()?;
        self.advance_token()?;

        Ok(())
    }

    fn read_token(&mut self) -> Result<Token, ParseErrorAndPos> {
        self.last_end = if self.token.span.is_valid() {
            Some(self.token.span.end())
        } else {
            None
        };
        let peek_token: Token = self.lexer.read_token()?;
        let token: Token = mem::replace(dest: &mut self.peek_token, src: pe
        Ok(mem::replace(dest: &mut self.token, src: token))
    }
```

SSA

Static Single-Assignment

Original

```
a := b + c
b := c + 1
d := b + c
a := a + 1
e := a + b
```

SSA

$a_1 := b_1 + c_1$
$b_2 := c_1 + 1$
$d_1 := b_2 + c_1$
$a_2 := a_1 + 1$
$e_1 := a_2 + b_2$

Original

```
if B then
  a := b
else
  a := c
end
  … a …
```

SSA

if B then
  $a_1 := b$
else
  $a_2 := c$
End
$a_3 := \Phi(a_1, a_2)$
  … a_3 …

(From Markus Denker's Intro to SSA)

All the cool kids have it

- Rust
- LLVM
- Go
- LuaJIT
- PyPy
- WebKit

Should be easy, right?

Lesson:

SSA ain’t SSA

~/drive/notes/Tucan - Constructing SSA for Tucan from CFG.md

```markdown
# Constructing SSA for Tucan from CFG

## Current status

Okay, let's recap.

Right now we a program in the form of a control-flow graph (CFG) with each
instruction consisting of a target, and an instruction kind.

What we want is to turn this program into SSA form. SSA stands for Static Single
Assignment, another way to represent programs.

Why convert the program to SSA form? Because once a program is in SSA form it's
easier to run certain optimization passes over the program.

> If LLVM had not used the SSA form, we would need to run a separate data flow
> analysis to compute the use-def chains, which are mandatory for classical
> optimizations such as constant propagation and common subexpression elimination.

Example: https://hub.packtpub.com/introducing-llvm-intermediate-representation/

## Resources

<!-- Links up here because it's easier to add to the bottom of a section without having to move the links all the time -->

[ssabook]: http://ssabook.gforge.inria.fr/latest/book.pdf
[simple]: https://link.springer.com/content/pdf/10.1007/3-540-46423-9_8.pdf
[simpleandefficient]: https://pp.info.uni-karlsruhe.de/uploads/publikationen/braun13cc.pdf
[denkerpresentation]: https://marcusdenker.de/talks/08CC/08IntroSSA.pdf
[coirbuilder]: https://github.com/rsms/co/blob/master/src/ir/builder.ts#L1218
[firmgraphs]: https://pp.ipd.kit.edu/firm/GraphSnippets.html
[trufflegraphs]: https://chrisseaton.com/truffleruby/basic-truffle-graphs/
[graalgraphs]: https://chrisseaton.com/truffleruby/basic-graal-graphs/
[cliffclicktalk]: https://www.youtube.com/watch?v=98lt45Aj8mo
[gocompilerinternals]: https://eli.thegreenplace.net/2019/go-compiler-internals-adding-a-new-statement-to-go-part-2/
[llvmpresentation]: https://llvm.org/devmtg/2017-06/1-Davis-Chisnall-LLVM-2017.pdf
[simplefastdom]: https://www.cs.rice.edu/~keith/EMBED/dom.pdf
[waddledominator]: https://github.com/efritz/waddle/blob/master/src/main/scala/waddleo/dominators/DominatorUtil.scala#L92
[dominancefrontierslecture]: http://pages.cs.wisc.edu/~fischer/cs701.f05/lectures/Lecture22.pdf

### Books

Apparently Appel's book and the Cooper book contain algorithms for SSA.

### Papers

There's the [Static Single Assignment Book][ssabook].

Then there's the paper [Simple Generation of Static Single-Assignment Form][simple]:

> **Abstract**. The static single-assignment (SSA) form of a program pro-
> vides data flow information in a form which makes some compiler opti-
> mizations easy to perform. In this paper we present a new, simple method
> for converting to SSA form, which produces correct solutions for nonre-
> ducible control-flow graphs, and produces minimal solutions for reducible
> ones. Our timing results show that, despite its simplicity, our algorithm
> is competitive with more established techniques.

And the simple **and efficient** version: [Simple and Efficient Construction of Static Single Assignment Form][simpleandefficient

> **Abstract**. We present a simple SSA construction algorithm, which al-
> lows direct translation from an abstract syntax tree or bytecode into an
> SSA-based intermediate representation. The algorithm requires no prior
> analysis and ensures that even during construction the intermediate rep-
> resentation is in SSA form. This allows the application of SSA-based op-
> timizations during construction. After completion, the intermediate rep-
> resentation is in minimal and pruned SSA form. In spite of its simplicity,
> the runtime of our algorithm is on par with Cytron et al.’s algorithm.

This presentation also has a good overview of an SSA algorithm: [Markus Denker Presentation][denkerpresentation]

### Code

#### Go
```

Thorsten Ball @thorstenball · Dec 31, 2020

Thinking about SSA is melting my brain.

6 7

Slava Egorov
@mraleph

Just think about it as a data flow graph rather than anything else

3:54 PM · Dec 31, 2020

Rasmus Andersson
@rsms

You could simply use the memory address of a node to identify it, rather than a name or managed numeric identifier (since a value never changes.)

4:08 AM · Jan 1, 2021

Lesson:

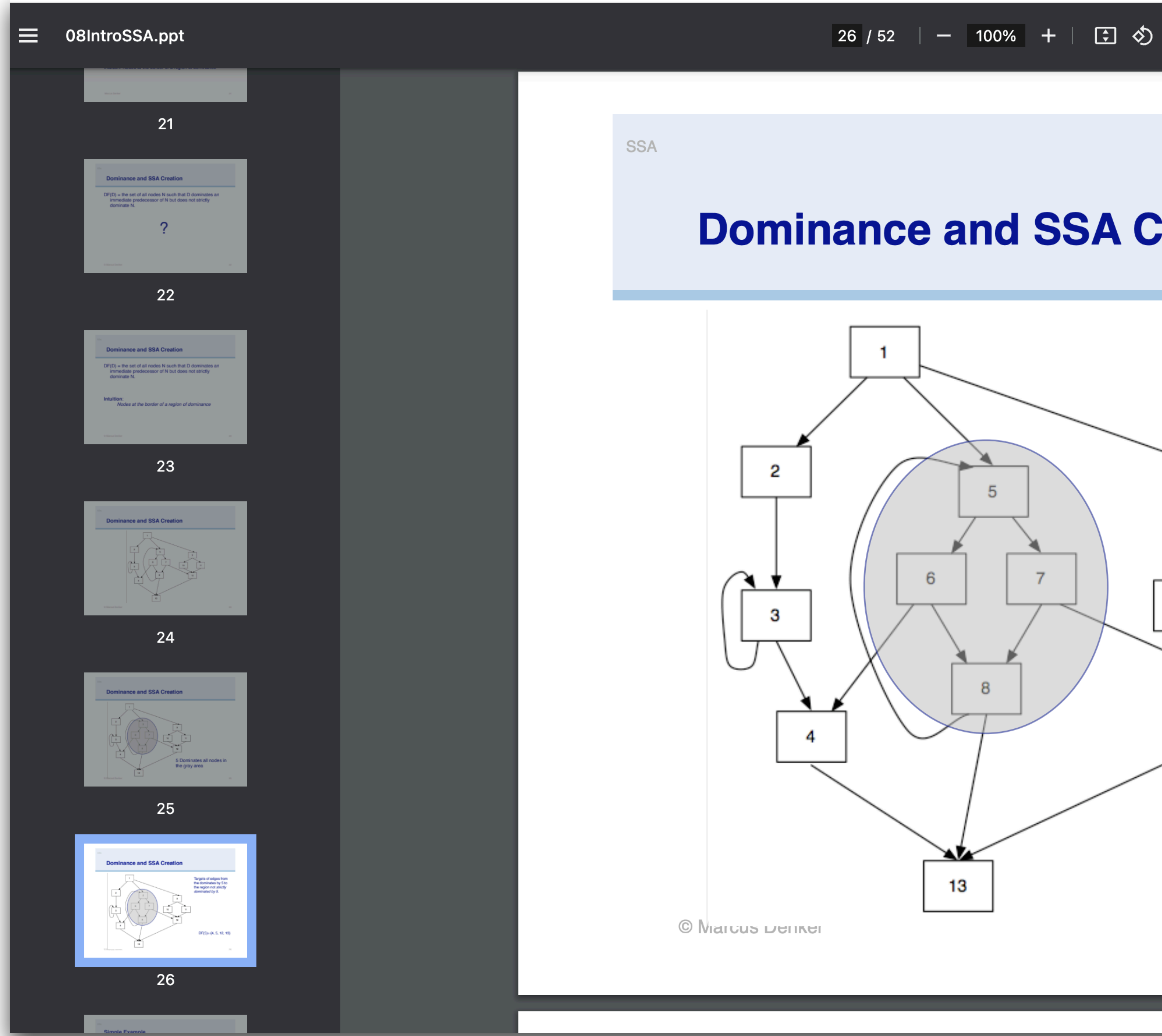
You have to read the code

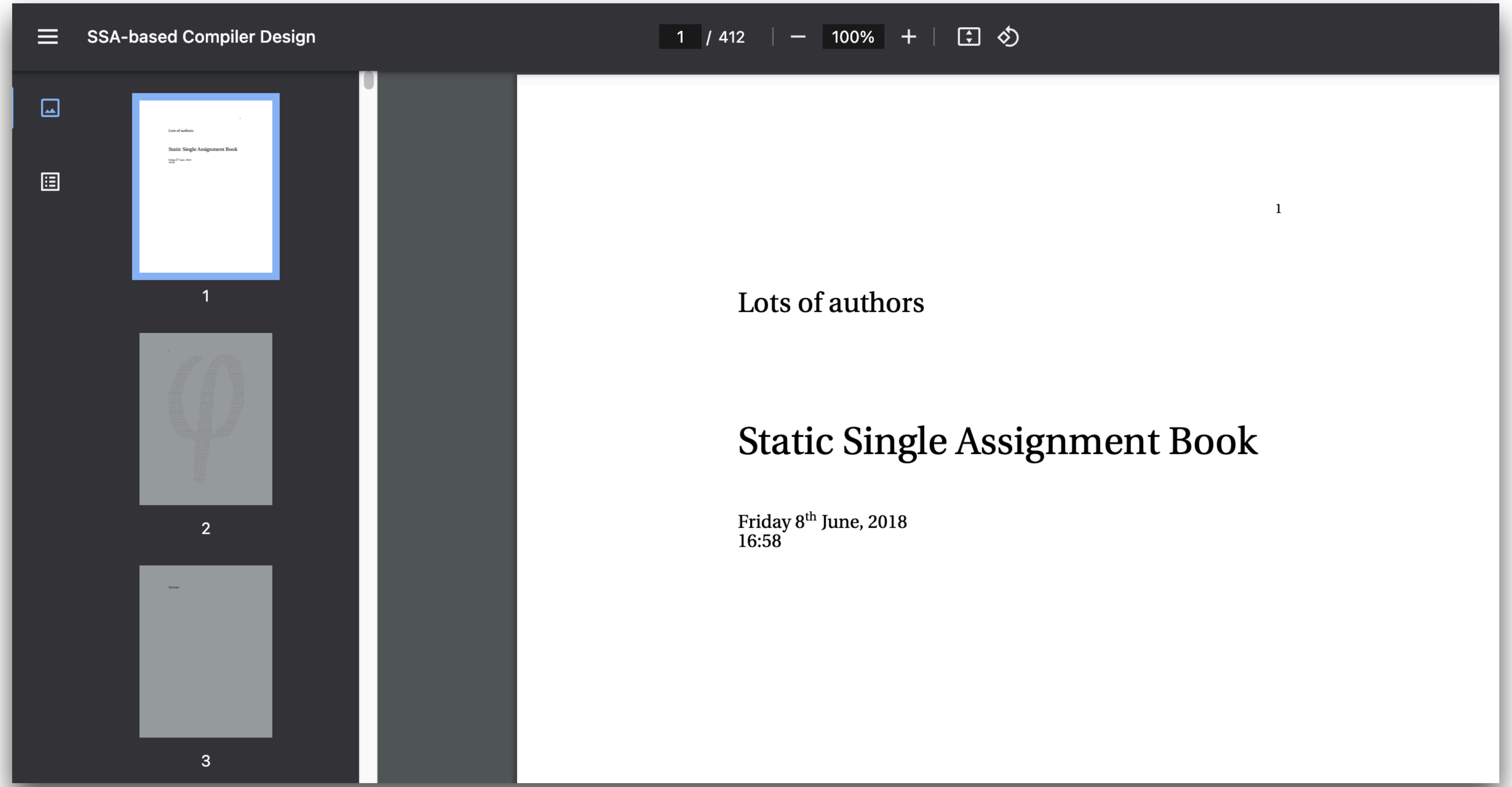
~/drive/notes/Tucan - Constructing SSA for Tucan from CFG.md

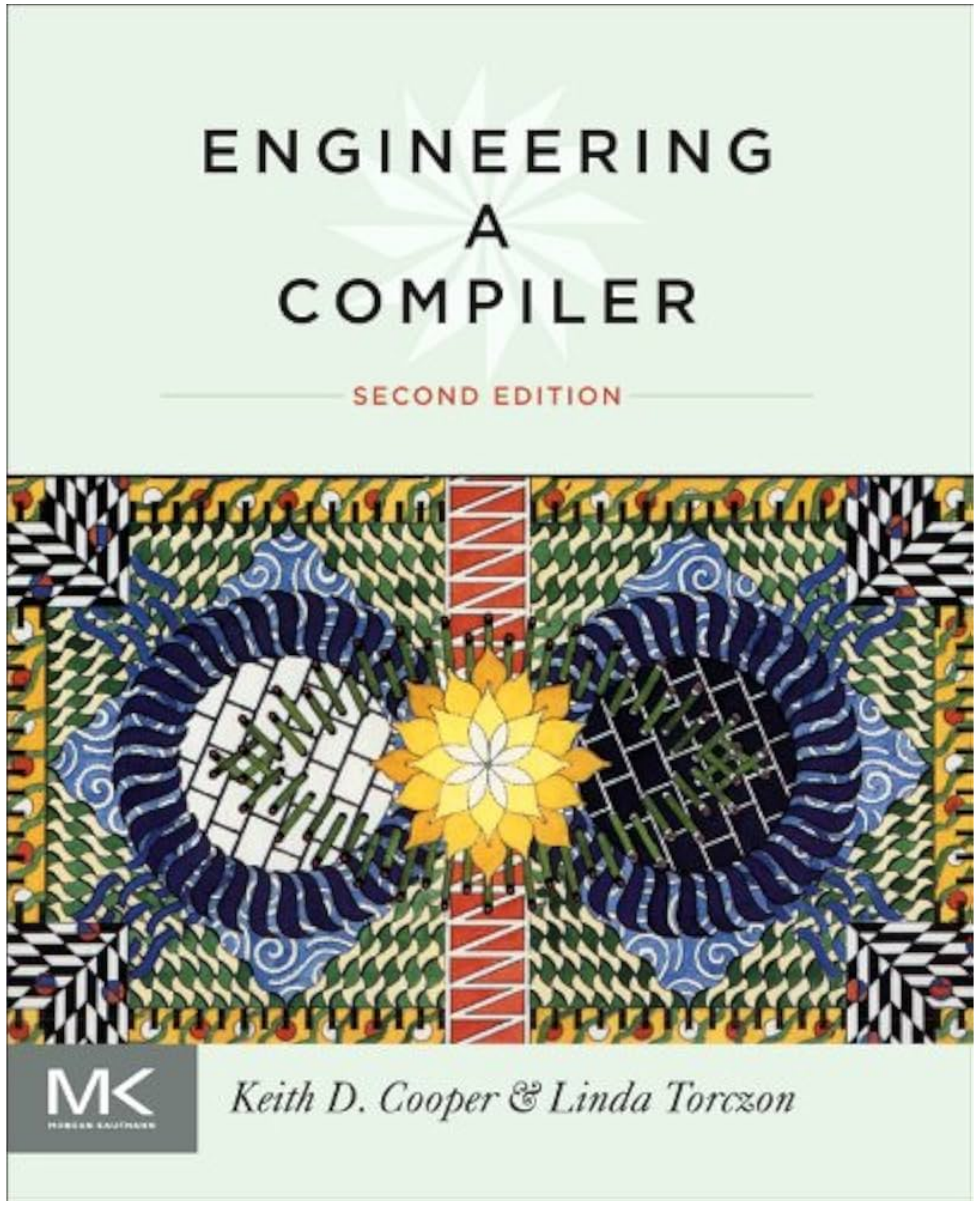
```markdown
### Code

#### Go

The **Go source code** also has conversion from AST to SSA and this is the en
https://sourcegraph.com/github.com/golang/go@master/-/blob/src/cmd/compile/in
go#L1106-1113

Here is the part of the code that handles assignment:
https://sourcegraph.com/github.com/golang/go@95ce805d14642a8e8e40fe1f8f50b9b5
b/src/cmd/compile/internal/gc/ssa.go#L1254

This is the most important file: `src/cmd/compile/internal/gc/ssa.go`

#### Co

@rsms' IR builder for his **Co** language creates a CFG in SSA form in this f
IRBuilder][coirbuilder] Pure gold!

#### BinaryAnalysisPlatform

Pass to transform to pruned SSA:
https://sourcegraph.com/github.com/BinaryAnalysisPlatform/bap/-/blob/lib/bap_
ssa.ml
And more information here:
https://sourcegraph.com/github.com/BinaryAnalysisPlatform/bap/-/blob/lib/bap/

Linked to by Rijnard: https://twitter.com/rvtond/status/1344008280171970560

#### irhydra

@mraleph's irhydra:
https://github.com/mraleph/irhydra/blob/master/saga/lib/src/flow/ssa.dart

[This comment][https://twitter.com/mraleph/status/1343907821495181313?s=20] b
interesting:

> Things really depend on how your IR looks like. For example in Dart VM each
> instruction is an malloced object so its identity (address) is it SSA name.
> the question of computing SSA form is just the question of placing phi's an
> computing their arguments

The `ssa.dart` file and the definitions in [`node.dart`]
(https://github.com/mraleph/irhydra/blob/master/saga/lib/src/flow/node.dart)
interesting.
```

Lesson:

You have to read the slides




SSA-based Compiler Design
1 / 412
100%
1
Lots of authors
Static Single Assignment Book
Friday 8th June, 2018
16:58
1
2
3


ENGINEERING
A
COMPILER
SECOND EDITION
MK
Keith D. Cooper & Linda Torczon

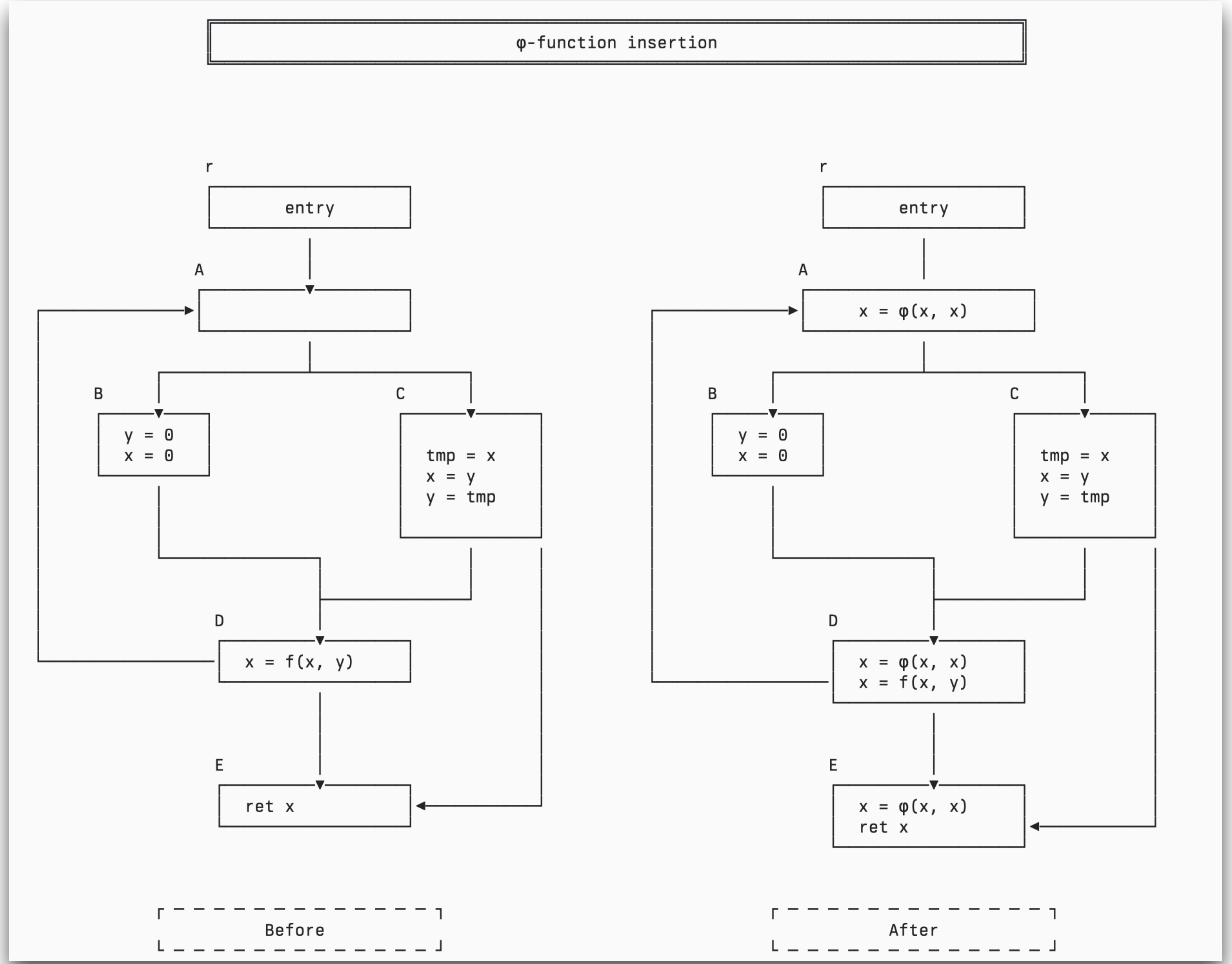
Lesson:

Papers aren’t meant to teach
Textbooks aren’t meant to teach

They are meant to share knowledge

Lesson:

ASCII graphs are awesome


φ-function insertion
r
entry
A
B
y = 0
x = 0
C
tmp = x
x = y
y = tmp
D
x = f(x, y)
E
ret x
Before
r
entry
A
x = φ(x, x)
B
y = 0
x = 0
C
tmp = x
x = y
y = tmp
D
x = φ(x, x)
x = f(x, y)
E
x = φ(x, x)
ret x
After

#[test]
fn diamond_shape() {
// b0 x = 11
// x = 22 x = 33
// b1 b2
// b3 y = x + 5
register_ids!(x, y, tmp);
let b0: BlockId = BlockId(0);
let f: &mut BlockFunk = &mut new_blo
let b1: BlockId = add_block(f, id: 1
let b2: BlockId = add_block(f, id: 2
terminate_if(funk: f, block_id: b0,
let b3_ins: Vec<Instruction> = vec![
let b3: BlockId = add_block(f, id: 3
terminate_goto(funk: f, block_id: b1
terminate_goto(funk: f, block_id: b2
// Phases 1 & 2: reset assignments,
analyse_construct_ssa!(f);
let b0_place: Place = first_target(f
#[test]
#[allow(clippy::many_single_char_names)]
fn complex_cfg_with_one_var() {
// b0
// b1
// b2 x = 2 x = 333 b3
// b4 x = x + 1
// b5 return x
register_ids!(x, b1_if_tmp, b3_if_tmp, b4
let b0: BlockId = BlockId(0);
let f: &mut BlockFunk = &mut new_block_fu
let b1: BlockId = add_block_empty(f, id:
let b2: BlockId = add_block(f, id: 2, ins
let b3: BlockId = add_block(f, id: 3, ins
f, id: 4, ins
f, id: 5, ins
k_id: b0, ta
id: b1, cond
k_id: b2, ta
id: b3, cond
id: b4, cond
ock_id: b5,
nments, anal
target(f, id
target(f, id
target(f, id
target(f, id
instruction
target(f, id
#[test]
#[allow(clippy::many_single_char_names)]
fn complex_cfg() {
// This is taken from "Engineering a compiler", p500 and
// First, the version without instructions:
// b0 b1 b2 b5 b6 b8 b7 b3 b4
// And here it is with instructions:
// b0 i = 1
// b1 a = 200 c = 300 a < c
// b5 a = 201 d = 501 a <= d
// b6 d = 502 b8 c = 302
// b2 b = 400 c = 301 d = 500
// b7 b = 401
// b3 y = a + b z = c + d i = i + 1 i <= 100
// b4 return
// Let's setup all the locations we need:
register_ids!(i, a, c, b, d, y, z, b1_tmp, b3_tmp, b5_tmp);
// Setup the function and create the blocks
let b0: BlockId = BlockId(0);
let f: &mut BlockFunk = &mut new_block_funk(id: 1, entry_block_id: b0, entry_block_ins: vec![mov!(i, 1)]);
#[test]
#[allow(clippy::many_single_char_names)]
fn complex_cfg_with_two_vars() {
// b0
// b1 a = 200 a < 5
// b5 a = 201 a < 50
// b6 b8
// b2 b = 400
// b3 b7 b = 401
// y = a + b
// b4 return
// Let's setup all the locations we need:
register_ids!(a, b, y, b1_tmp, b3_tmp, b5_tmp);
// Setup the function and create the blocks:
let b0: BlockId = BlockId(0);
let f: &mut BlockFunk = &mut new_block_funk_empty(id: 1, entry_block_id: b0);
let b1: BlockId = add_block(f, id: 1, ins: vec![mov!(a, 200)]);
let b2: BlockId = add_block(f, id: 2, ins: vec![mov!(b, 400)]);
let b3: BlockId = add_block(f, id: 3, ins: vec![bin_op!(y, BinOp::Add, a, b)]);
let b4: BlockId = add_block_empty(f, id: 4);
let b5: BlockId = add_block(f, id: 5, ins: vec![mov!(a, 201)]);
#[test]
#[allow(clippy::many_single_char_names)]
fn ssa_book_example_graph() {
// Example is adapted from "Static Single Assignment Book", Chapter 3.
// b0
// b1
// b2 y = 5 x = 8 tmp = x x = y y = tmp b3
// b4 x = x + y
// b5 return x
// Let's setup all the locations we need:
register_ids!(x, y, tmp, b1_if_tmp, b3_if_tmp, b4_if_tmp);

```rust
// Phase 1 of SSA construction: φ-function insertion
//
// Based on "3.1.2 φ-function insertion" in "Standard Construction and
// Destruction Algorithms" by J. Singer and F. Rastello in the SSA book.
pub fn insert_phi_functions(funk: &mut BlockFunk) {
    let mut phis: HashMap<BlockId, Vec<Instruction>> = HashMap::new();
    let assigns: HashMap<Place, HashSet<Var>> = funk.build_assigns();

    for (place: Place, defs: HashSet<Var>) in assigns {
        // Skipping the vars that are only written-to in a single block.
        if defs.len() == 1 {
            continue;
        }

        let mut done: HashSet<BlockId> = HashSet::new();
        let mut work_list: Vec<&Block> = Vec::new();

        let new_var: impl Fn(&BlockId) -> Var = |id: &BlockId| Var(*id, place);

        let mut def_blocks: HashSet<BlockId> = HashSet::new();
        for def_var: &Var in defs.iter() {
            work_list.push(&funk[def_var.0]);
            def_blocks.insert(def_var.0);
        }

        while let Some(block: &Block) = work_list.pop() {
            for df_block_id: BlockId in block.dom_frontier.iter().cloned() {
```

Code Generation

Relatively straightforward

- Emit x86 ASM
- Assemble with GCC

```rust
// movq
pub fn emit_movq_const(&mut self, constant: &ir::Constant, target: RegOrOffset) -> E
    match constant {
        ir::Constant::Int(i: &i64) => writeasm!(self, "\tmovq ${i}, {target}"),
        ir::Constant::Bool(true) => writeasm!(self, "\tmovq ${}, {}", &1, target),
        ir::Constant::Bool(false) => writeasm!(self, "\tmovq ${}, {}", &0, target),
        ir::Constant::String(s: &String) => unlowered_string!(s),
    }
}

pub fn emit_movq(&mut self, origin: RegOrOffset, target: RegOrOffset) -> EmitResult
    writeasm!(self, "\tmovq {origin}, {target}")
}

// subq/addq
pub fn emit_subq_ir(&mut self, integer: &u64, reg: Reg) -> EmitResult {
    writeasm!(self, "\tsubq ${integer}, {reg}")
}

pub fn emit_addq_ir(&mut self, integer: &u64, reg: Reg) -> EmitResult {
    writeasm!(self, "\taddq ${integer}, {reg}")
}

pub fn emit_addq(&mut self, origin: RegOrOffset, reg: impl Into<RegOrOffset>) -> Emi
    writeasm!(self, "\taddq {}, {}", origin, reg.into())
}

pub fn emit_subq(&mut self, origin: RegOrOffset, reg: impl Into<RegOrOffset>) -> Emi
    writeasm!(self, "\tsubq {}, {}", origin, reg.into())
}

// cmp, jump, etc.
pub fn emit_cmpq_to_reg(&mut self, origin: RegOrOffset, reg: Reg) -> EmitResult {
    writeasm!(self, "\tcmpq {origin}, {reg}")
}

pub fn emit_cmpq_const(&mut self, constant: &ir::Constant, target: RegOrOffset) -> E
    match constant {
        ir::Constant::String(s: &String) => unlowered_string!(s),
        ir::Constant::Bool(true) => writeasm!(self, "\tcmpq ${}, {}", &1, target),
        ir::Constant::Bool(false) => writeasm!(self, "\tcmpq ${}, {}", &0, target),
        ir::Constant::Int(i: &i64) => writeasm!(self, "\tcmpq ${i}, {target}"),
    }
}

pub fn emit_jmp(&mut self, label: &str) -> EmitResult {
    writeasm!(self, "\tjmp {label}")
}
```

Lesson:
Debuggability is precious

(A lesson one might have to learn multiple times)

Register Allocation

Or: My Darkest Hour

It's hard

```
# Register allocation for Tucan

## Resources

[linearscan]: https://link.springer.com/content/pdf/10.1007%
[craneliftregalloc]: https://github.com/bytecodealliance/was
[craneliftcorrectnessregalloc]: https://cfallin.org/blog/202
[craneliftcodegen]: https://blog.benj.me/2021/02/17/cranelif
[mikepallreverselinearscan]: http://lua-users.org/lists/lua-
[reverselinearscanblogpost]: http://brrt-to-the-future.blogs
[wimmerlinearscan]: http://citeseerx.ist.psu.edu/viewdoc/dow
[optimizedinterval]: https://www.usenix.org/legacy/events/ve

### Books

- Engineering a Compiler
- SSA Book (Seems like there's a newer version of the book h

### Papers

- [Linear Scan Register Allocation in the Context of SSA For
- [Linear Scan Register Allocation on SSA form][wimmerlinear
- [Optimized Interval Splitting in a Linear Scan Register Al

### Misc

- Cranelift's README on SSA-based regalloc: [craneliftregall
- Mike Pall on LuaJIT's "reverse linear scan" algorithm: [mi
- Blog post from one of the author's of MoarVM on "reverse l

### Slides

- https://ethz.ch/content/dam/ethz/special-interest/infk/ins
```

Thorsten Ball @thorstenball · Oct 31, 2021

Compilers folks: the computation of the LiveOut sets per block shown in the Linear-Scan-Regalloc-in-SSA paper (link.springer.com/content/pdf/10...) seems much simpler than the iterative fixed-point algorithms I've seen.

Is that only because SSA makes it simpler or am I missing something?

ranges [1,3[, [3,7[are merged into a single range [1,7[.

The algorithm BUILDINTERVALS() traverses the control flow graph in an arbitrary order, finds out which values are live at the end of every block, and computes the ranges for these values as described above.

```
BuildIntervals()
    for each block b do
        live ← {}
        for each successor s of b do
            live ← live ∪ s.live
            for each ϕ-function phi in s do
                live ← live – {phi} ∪ {phi.opd(b)}
        for each instruction i in live do ADDRANGE(i,
b, b.last.n+1)
        for all instructions i in b in reverse order
do
            live ← live – {i}
            for each operand opd of i do
                if opd ∉ live then
                    live ← live ∪ {opd}
                    ADDRANGE(opd, b, i.n)
```

Fig. 12 shows a sample program in source code and in intermediate representation with a ϕ-function for the value *d* and corresponding move instructions in the predecessor blocks. Fig. 13 shows the live intervals that are computed for this

3 1 13

Slava Egorov
@mraleph

Notice that it uses something called b.live (does not compute that!) - that's a live-in set for block b. In other words it expects that a liveness analysis has been run prior to constructing the intervals

---

Day after, 03 Nov 21, 6:30am.

The [paper version of the algorithm in the SSA book][computingliveness] has a clear list of requirements for the liveness analysis:

> Since our algorithm exploits advanced program properties some prerequisites
> have to be met by the input program and the compiler framework:
> * The CFG of the input program is available.
> * The program has to be in strict SSA form.
> * A loop-nesting forest of the CFG is available.

But they also seem to present *another* algorithm, one that doesn't need the loop-nesting forest:

> "Liveness Sets using Path Exploration"
> For SSA programs, another approach is possible that follows the classical definition of liveness:

> a variable is live at a program point p, if p belongs to a path of the CFG
> leading from a definition of that variable to one of its uses without passing
> through another definition of the same variable.

> Therefore, the live-range of a variable can be computed using a backward traversal starting
> on its uses and stopping when reaching its (unique) definition. For comparison,
> we designed optimized implementations of this path-exploration principle (see
> Section 5), for both SSA and non-SSA programs, and compared the efficiency
> of the resulting algorithms with our novel non-iterative data-flow algorithm.

And that algorithm doesn't require a loop-nesting forest to have been built:

```javascript
// Compute liveness sets by exploring paths from variable uses.
function Compute_LiveSets_SSA_ByUse(CFG) {
  for each basic block B in CFG do // Consider all blocks successively
    for each v ∈ PhiUses(B) do // Used in the φ of a successor block
      LiveOut(B) = LiveOut(B) ∪ {v}
      Up_and_Mark(B, v)
    for each v used in B (φ excluded) do // Traverse B to find all uses
      Up_and_Mark(B, v)
}
```

Liveness Analysis

```
// compute_loop_nesting_forest builds a loop-nesting forest (finding loops in a function and
// nesting them under each other).
//
// It's based on
//
// - Paul Havlak - Nesting of Reducible and Irreducible loops
//
// with the help of:
//
// - G. Ramalingam - Identifying Loops In Almost Linear Time
// - R. Endre Tarjan - Testing Flow Graph Reducibility
//
// Big difference to Havlak's algorithm is that we don't use the DFS' ranking to determine whether
// a block is an ancestor, but use dominators instead.
pub fn compute_loop_nesting_forests(funk: &BlockFunk) -> LoopForest {
    let (order, numbers) = compute_dfs_numbers(funk);

    let mut non_back_preds: Vec<HashSet<BlockId>> = Vec::with_capacity(order.len());
    let mut back_preds: Vec<HashSet<BlockId>> = Vec::with_capacity(order.len());

    let mut forest = LoopForest::new();
    let mut union_find = UnionFind::new(order.clone());

    let dominates = |a, b| funk[b].doms.contains(a);
```

regalloc: Just a 150 line function

Lesson:
I can do hard things?

Bonus Lesson

Thorsten Ball
to hanspeter.moessenboeck@jku.at

Nov 25, 2021 (3 years ago)

Sent · Reply to all · Actions

Hallo!

Ich baue privat grade einen Compiler in Rust und finde Ihr Paper "Linear Scan Register Allocation in the Context of SSA Form and Register Constraints" sehr gut und nützlich.

Ich habe aber eine Frage zu Section 4:

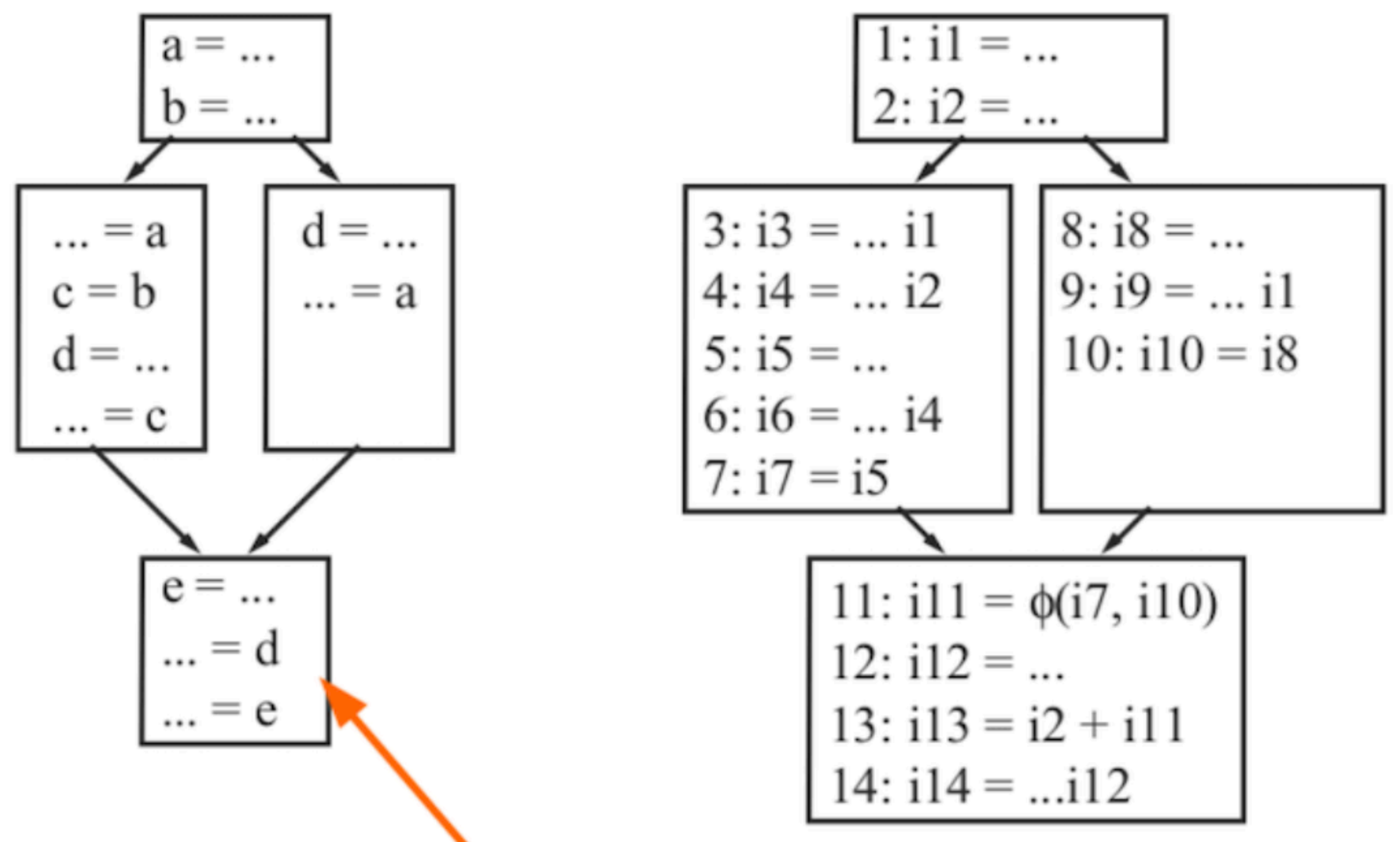


Fig. 12. Sample program in source code and in intermediate representation

This should be
... = b + d

right?

Thorsten Ball
to hanspeter.moessenboeck@jku.at ▾

Nov 25, 2021 (3 years ago)

Sent | Reply to all | Actions

Hallo!

Ich baue privat grade einen Compiler in Rust und finde Ihr Paper "Linear Scan Register Allocation in the Context of SSA Form and Register Constraints" sehr gut und nützlich.

Ich habe aber eine Frage zu Section 4:

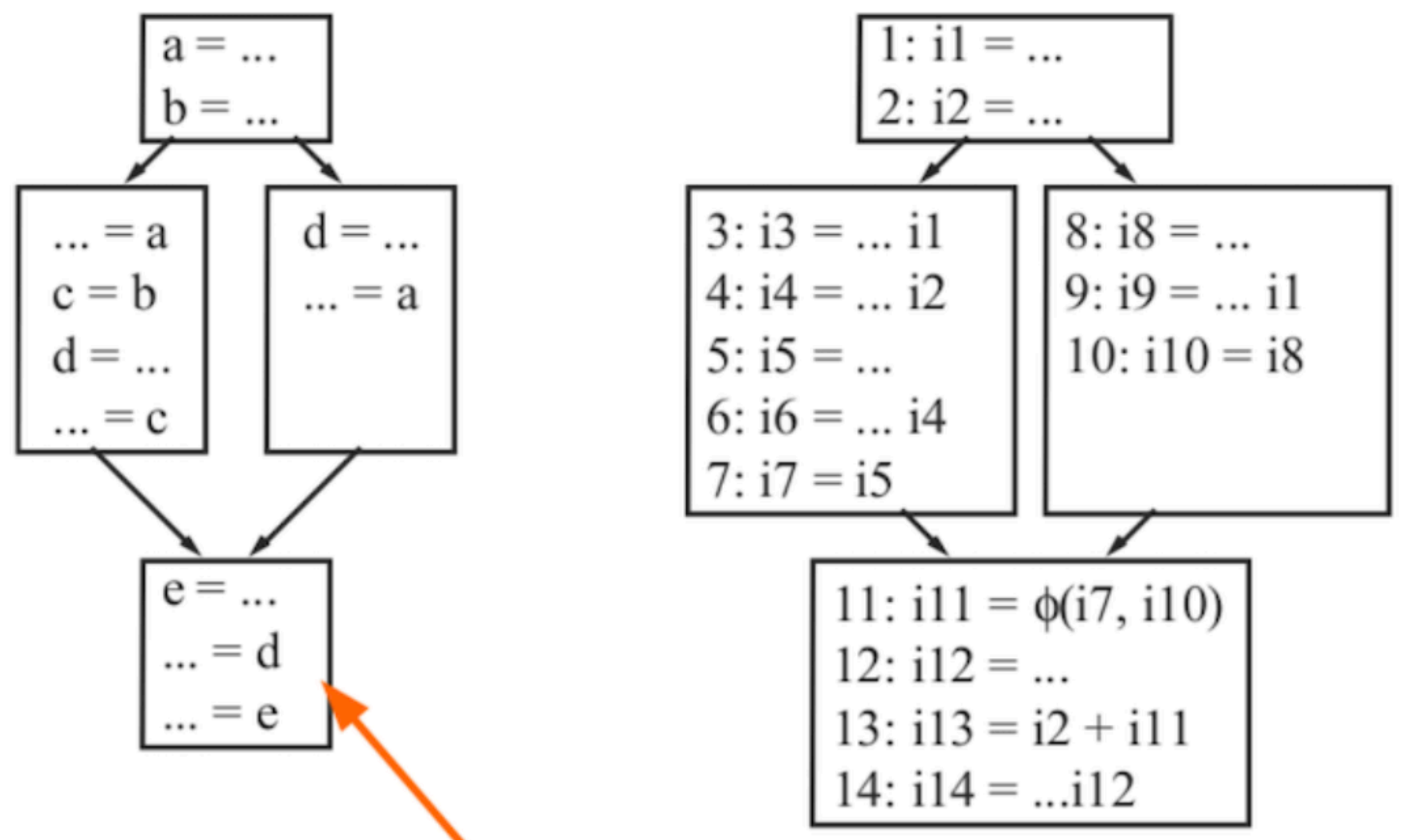


Fig. 12. Sample program in source code and in intermediate representation

This should be
... = b + d

right?

Peter Mössenböck
to Thorsten Ball ▾

Nov 25, 2021 (3 years ago)

Archive | Reply | Actions

Lieber Herr Ball,

Sie haben natürlich recht, und ich wundere mich, dass das noch niemandem aufgefallen ist.

Es muss entweder im linken Diagramm heißen
... = b + d
oder im rechten Diagramm
i13 = i11

Da aber in Fig.13 das Live-Intervall von b bis Instruktion 13 geht, ist die erste Interpretation die richtige.

Vielen Dank für den Hinweis.

Beste Grüße
Hanspeter Mössenböck

...

Lesson:
Even experts make mistakes

Optimizations

Optimizations

```rust
// Sparse Conditional Constant Propagation.
//
// This is based on "10.7.1 Combining Optimizations" in Cooper/Torczon - Engineering A Compiler,
// which is based on the "Sparse Simple Constant Propagation (SSCP)" algorithm presented earlier i
// Section 9.3.6

// "In SSCP [and SCCP], the algorithm initializes the value associated with each SSA name to
// ⊤, which indicates that the algorithm has no knowledge of the SSA name's
// value.
//
// If the algorithm subsequently discovers that SSA name x has the known constant value Ci, it
// models that knowledge by assigning Value(x) the semi-lattice element Ci.
//
// If it discovers that x has a changing value, it models that fact with the value ⊥."
#[derive(PartialEq, Eq, Clone, Debug)]
enum Value {
    Top, // ⊤
    Const(Constant),
    Bot, // ⊥
}

impl Value {
    fn is_bot(&self) -> bool {
        matches!(self, Value::Bot)
    }

    // Rules for Meet:
    // ⊤  ∧ x  = x    ∀ x
    // ⊥  ∧ x  = ⊥    ∀ x
    // Ci ∧ Cj = Ci   if Ci = Cj
    // Ci ∧ Cj = ⊥    if Ci != Cj
    fn meet(&self, other: &Self) -> Self {
        match (self, other) {
            (Value::Top, other: &Value) | (other: &Value, Value::Top) => other.clone(),
            (Value::Bot, _) | (_, Value::Bot) => Value::Bot,
            (Value::Const(i: &Constant), Value::Const(j: &Constant)) => {
                if i == j {
                    Value::Const(i.clone())
                } else {
                    Value::Bot
                }
            }
        }
    }
}
```

```rust
#[test]
fn handling_phi_functions_diamond_shape() {
    // This is taken from p8 of "Briggs, Cooper, Simpson - Value Numbering"
    //
    //                  BEFORE:                                          AFTER:
    //
    //
    //       b0   a_0 = param(0)                          b0   a_0 = 11
    //            b_0 = param(1)                               b_0 = 22
    //            c_0 = param(2)                               c_0 = 33
    //            d_0 = param(3)                               d_0 = 44
    //            e_0 = param(4)                               e_0 = 55
    //            f_0 = param(5)                               f_0 = 66
    //
    //            u_0 = a_0 + b_0                              u_0 = a_0 + b_0
    //            v_0 = c_0 + d_0                              v_0 = c_0 + d_0
    //            w_0 = e_0 + f_0                              w_0 = e_0 + f_0
    //
    //
    //                          u_1 = a_0 + b_0
    //   x_0 = c_0 + d_0        x_1 = e_0 + f_0
    //   y_0 = c_0 + d_0        y_1 = e_0 + f_0
    //
    //    b1                     b2                         b1                 b2
    //
    //
    //      b3   u_2 = φ(u_0, u_1)                        b3
    //           x_2 = φ(x_0, x_1)                           x_2 = φ(v_0, w_0)   <-- IMPORTANT
    //           y_2 = φ(y_0, y_1)
    //
    //           z_0 = u_2 + y_2                             z_0 = u_0 + x_2     <-- IMPORTANT
    //           u_3 = a_0 + b_0
    //
    //
    //
    versioned_registers![
        [a_0],
        [b_0],
        [c_0],
        [d_0],
        [e_0],
        [f_0],
        [u_0, u_1, u_2, u_3],
        [v_0],
        [w_0],
        [x_0, x_1, x_2],
        [y_0, y_1, y_2],
        [z_0]
    ];

    let b0_ins: Vec<Instruction> = vec![
        mov!(a_0, param!(0)),
        mov!(b_0, param!(1)),
        mov!(c_0, param!(2)),
        mov!(d_0, param!(3)),
        mov!(e_0, param!(4)),
        mov!(f_0, param!(5)),
```

Lesson:

Rust !❤️ Graph Manipulation

Uh, oh

~/drive/notes/Tucan - Building an optimizing compiler.md

```
# Tucan - Building an optimizing compiler

- [Compiler Resources](./Compiler\ Resources.md)
- [Tucan - Intermediate Representation (IR)](./Tucan\ -\ Intermediate\ Representation\ IR.md)
- [Tucan - Constructing SSA for Tucan from CFG](./Tucan\ -\ Constructing\ SSA\ for\ Tucan\ from\ CFG.md)
- [Tucan - Codegen for Tucan](./Tucan\ -\ Codegen\ for\ Tucan.md)
- [Tucan - Last expression statement is returned](./Tucan\ -\ Last\ expression\ statement\ is\ returned.md)
- [Tucan - Liveness Analysis](./Tucan\ -\ Liveness\ Analysis.md)
- [Tucan - Register allocation for Tucan](./Tucan\ -\ Register\ allocation\ for\ Tucan.md)
- [Tucan - Optimizations](./Tucan\ -\ Optimizations.md)
- [Tucan - Fixing regalloc liveness bug](./Tucan\ -\ Fixing\ regalloc\ liveness\ bug.md)
- [Tucan - Fixing phi operand bug](./Tucan\ -\ Fixing\ phi\ operand\ bug.md)
- [Tucan - Fixing another bug in register allocator](./Tucan\ -\ Fixing\ another\ bug\ in\ register\ allocator.md)
- [Tucan - Garbage Collection](./Tucan\ -\ Garbage\ Collection.md)

```

Thorsten Ball @thorstenball · Jun 12, 2022

good news: it's not in the optimization pass that I finished this week
bad news: found out my compiler is flaky

1 2 27

Thorsten Ball @thorstenball · Jun 12, 2022

good news: it's not in the optimization pass that I finished this week
bad news: found out my compiler is flaky

1 2 27

Thorsten Ball @thorstenball · Jun 14, 2022

oh no, it's in the register allocator

3 2 19

Thorsten Ball @thorstenball · Jun 12, 2022

good news: it's not in the optimization pass that I finished this week
bad news: found out my compiler is flaky

1 2 27

Thorsten Ball @thorstenball · Jun 14, 2022

oh no, it's in the register allocator

3 2 19

Ben L. Titzer @TitzerBL · Jun 14, 2022

The ninth circle of compiler hell has been reached

1 1 10

Time to bring out the big guns

Tree-sitter grammars

example.tuc

examples/example.tuc

```
funk hello_world() → bool {
  print_string("hello world!");
  return true;
}

// magic_funk is a magical function
funk magic_funk(a: int, b: int, c: int, d: bool, e: bool) → bool {
  // comment here
  let res = a + b + c;
  // comment there
  return res == d == e;
}

funk x(a: int, b: bool, c: string) → int {
  let a: int = 18;
  let b: bool = a == 10;
  let c: bool = a ≠ b;
  let d = true ≠ false;
  // comment
  let s = "foobar" == "foo bar";

  1 + 2;
  1 - 2;
  1 > 2;
  1 < 2;
  1 ⇒ 2;
```

example.tucir

examples/example.tucir

```
block_funk #1(node_id: #33, entry: #1)
block #1 (→ #6):
  %1_0 ← 5
  %2_0 ← 0
  terminator: goto(#6)

block #6 (← #1, #4) (→ #2, #3):
  %2_1 ← phi(var(#1, %2_0), var(#4, %2_2))
  %8_0 ← %2_1 < 10
  terminator: if(%8_0, (#3, #2))

block #2 (← #6):
  %18_0 ← true
  %19_0 ← false
  terminator: return(0)

block #3 (← #6) (→ #4, #5):
  %3_0 ← %2_1 < %1_0
  terminator: if(%3_0, (#5, #4))

block #4 (← #3, #5) (→ #6):
  %6_0 ← call print_num, args: (1)
  %7_0 ← %2_1 + 1
```

Tree-sitter grammars

crates/languages/src/rust/injections.scm

```
(macro_invocation
  (token_tree) @content
  (#set! "language" "rust"))

(macro_rule
  (token_tree) @content
  (#set! "language" "rust"))


;; Inject the tucan grammar into Rust strings that are bound to `let tucan =`
((let_declaration
   pattern: (identifier) @_new
   (#eq? @_new "tucan")
   value: [(raw_string_literal
             (string_content) @content)
           (string_literal
             (string_content) @content)]])
 (#set! "language" "tucan"))
```

Works in Zed and Neovim

```
#[test]
fn for_loop() {
    let tucan = r#"
        funk main() → int {
            let global_one = 0;
            let global_two = 99;

            for (let i = 0; i < 10; i = i + 1) {
              global_one = global_one + i;
              global_two = global_two + 0;
              print_num(i);
            }

            print_string("-");
            print_num(global_one);
            print_string("-");
            print_num(global_two);
            return 0;
        }"#;

    run("for_loop", tucan, 0, "0123456789-45-99");
}

#[test]
fn loop_containing_conditionals() {
    let tucan = r#"
        funk main() → int {
            let global = 0;
            for (let i = 0; i < 10; i = i + 1) {
                if (i < 5) {
                    global = global + i;
                } else {
                    global = global + i - 5;
                }
            }
            print_num(global);
```

Integration tests

```rust
#[test]
fn big_integration_boi() {
    let tucan: &str = r#"
        funk double_print(num: int) {
            let doubled = num;
            for (let i = num; i > 0; i = i - 1) {
                doubled = doubled + 1;
            }
            print_num(doubled);
        }

        funk my_function(a: int, b: int) -> int {
            let c = a + b + 18 - 5 + 3 + 2;
            let d = 18 - 5 + 3;
            let e = a + b + 30;
            let f = 100 - a - b + 3 + 2;
            let g = c + d + e; // dead code
            let h = a + b + c + e + f;

            let k = 50;

            let result = 0;
            for (let i = 0; i < 10; i = i + 1) {
                result = result + 5;
                if (i < a) {
                    if (k >= d) {
                        double_print(i);
                    } else {
                        print_num(i);
                    }
                } else {
                    result = result + c;
                }
                for (let j = 5; j > 0; j = j - 1) {
                    result = result - j;
                }
                print_num(result);
                print_string("-");
            }
            return result;
        }

        funk main() -> int {
            my_function(5, 10);
            my_function(50, 2);
            return 0;
        }"#;

    run(
        test_name: "big_integration_boi",
        code: tucan,
        exit_code: 0,
        output: "0-10-2-20-4-30-6-40-8-50--27--4-19-42-65-0-10-2-20-4-30-6-40-8-50-10-60-12-70-14-80-16-90-18-100-",
    );
} fn big_integration_boi
```

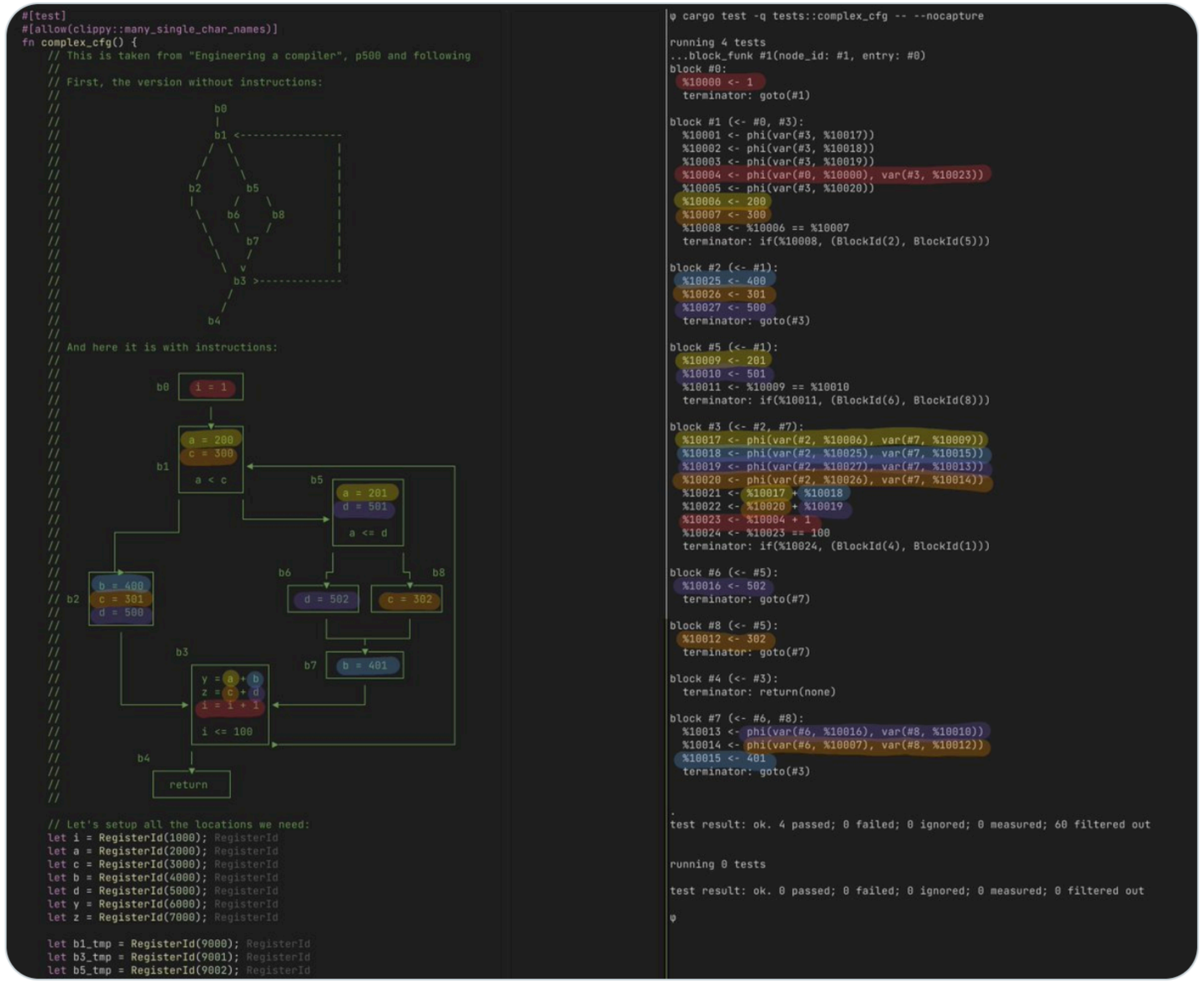
```rust
fn run(test_name: &str, code: &'static str, exit_code: i32, output: &'static str) {
    const TOTAL_RUNS: i32 = 12;

    let combinations = [(false, false), (true, true), (true, false), (false, true)];
    for i in 0..TOTAL_RUNS {
        let (optimize, regalloc) = combinations[i as usize % combinations.len()];

        print_test_line(test_name, i, TOTAL_RUNS, regalloc, optimize);

        let (asm, _) = Compiler::for_string(test_name, code)
            .optimize(optimize)
            .regalloc(regalloc)
            .compile()
            .expect("compiler failed");

        let asm_name = format!("integration-test-{test_name}-{i}");
        let result = AssemblyRunner::new_for_test(&asm_name, &asm)
            .run()
            .expect("failed to create assembly runner");

        assert_eq!(result.0, exit_code, "run #{i}: wrong exit code");
        assert_eq!(result.1, output, "run #{i}: wrong output");
    }
}
```

Thorsten Ball @thorstenball

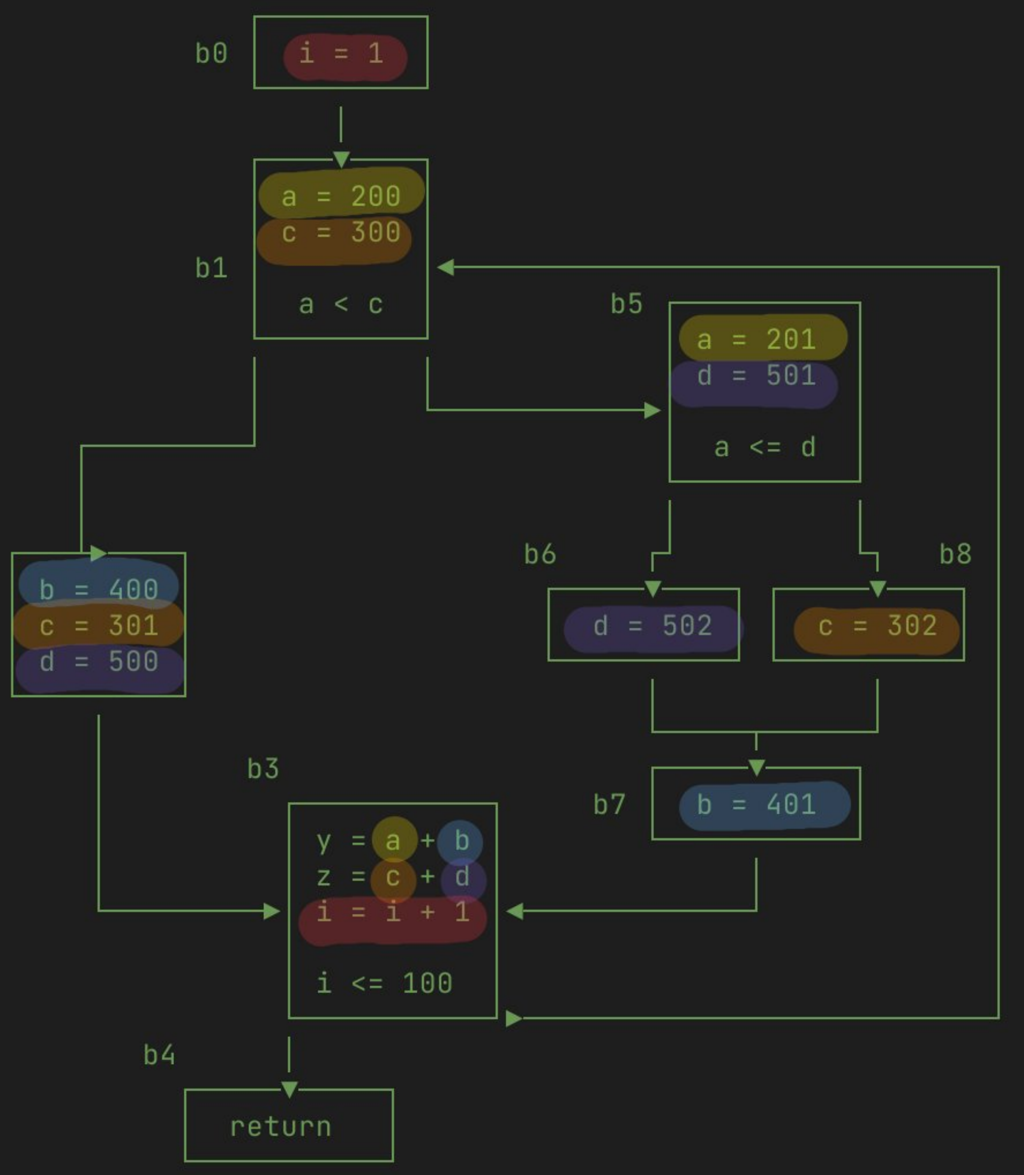
I'm 99.9% sure I've got it now.

Converting a CFG into SSA form. Static single assignment: every write is to a unique location.

On the left is the input to the test, on the right is the same CFG but in SSA form.



```
#[test]
#[allow(clippy::many_single_char_names)]
fn complex_cfg() {
    // This is taken from "Engineering a compiler", p500 and following
    //
    // First, the version without instructions:
    //
    //                          b0
    //                          |
    //                          b1 <----------------
    //                         /  \                |
    //                        /    \               |
    //                       /      \              |
    //                     b2        b5            |
    //                     |       /   \           |
    //                      \    b6     b8         |
    //                       \     \   /           |
    //                        \     b7             |
    //                         \    /              |
    //                          \  v               |
    //                           b3 >-------------
    //                          /
    //                         /
    //                       b4
    //
    // And here it is with instructions:
```




```
    // Let's setup all the locations we need:
    let i = RegisterId(1000); RegisterId
    let a = RegisterId(2000); RegisterId
    let c = RegisterId(3000); RegisterId
    let b = RegisterId(4000); RegisterId
    let d = RegisterId(5000); RegisterId
    let y = RegisterId(6000); RegisterId
    let z = RegisterId(7000); RegisterId

    let b1_tmp = RegisterId(9000); RegisterId
    let b3_tmp = RegisterId(9001); RegisterId
    let b5_tmp = RegisterId(9002); RegisterId
```

```
φ cargo test -q tests::complex_cfg -- --nocapture

running 4 tests
...block_funk #1(node_id: #1, entry: #0)
block #0:
  %10000 <- 1
  terminator: goto(#1)

block #1 (<- #0, #3):
  %10001 <- phi(var(#3, %10017))
  %10002 <- phi(var(#3, %10018))
  %10003 <- phi(var(#3, %10019))
  %10004 <- phi(var(#0, %10000), var(#3, %10023))
  %10005 <- phi(var(#3, %10020))
  %10006 <- 200
  %10007 <- 300
  %10008 <- %10006 == %10007
  terminator: if(%10008, (BlockId(2), BlockId(5)))

block #2 (<- #1):
  %10025 <- 400
  %10026 <- 301
  %10027 <- 500
  terminator: goto(#3)

block #5 (<- #1):
  %10009 <- 201
  %10010 <- 501
  %10011 <- %10009 == %10010
  terminator: if(%10011, (BlockId(6), BlockId(8)))

block #3 (<- #2, #7):
  %10017 <- phi(var(#2, %10006), var(#7, %10009))
  %10018 <- phi(var(#2, %10025), var(#7, %10015))
  %10019 <- phi(var(#2, %10027), var(#7, %10013))
  %10020 <- phi(var(#2, %10026), var(#7, %10014))
  %10021 <- %10017 + %10018
  %10022 <- %10020 + %10019
  %10023 <- %10004 + 1
  %10024 <- %10023 == 100
  terminator: if(%10024, (BlockId(4), BlockId(1)))

block #6 (<- #5):
  %10016 <- 502
  terminator: goto(#7)

block #8 (<- #5):
  %10012 <- 302
  terminator: goto(#7)

block #4 (<- #3):
  terminator: return(none)

block #7 (<- #6, #8):
  %10013 <- phi(var(#6, %10016), var(#8, %10010))
  %10014 <- phi(var(#6, %10007), var(#8, %10012))
  %10015 <- 401
  terminator: goto(#3)

.
test result: ok. 4 passed; 0 failed; 0 ignored; 0 measured; 60 filtered

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered o

φ
```

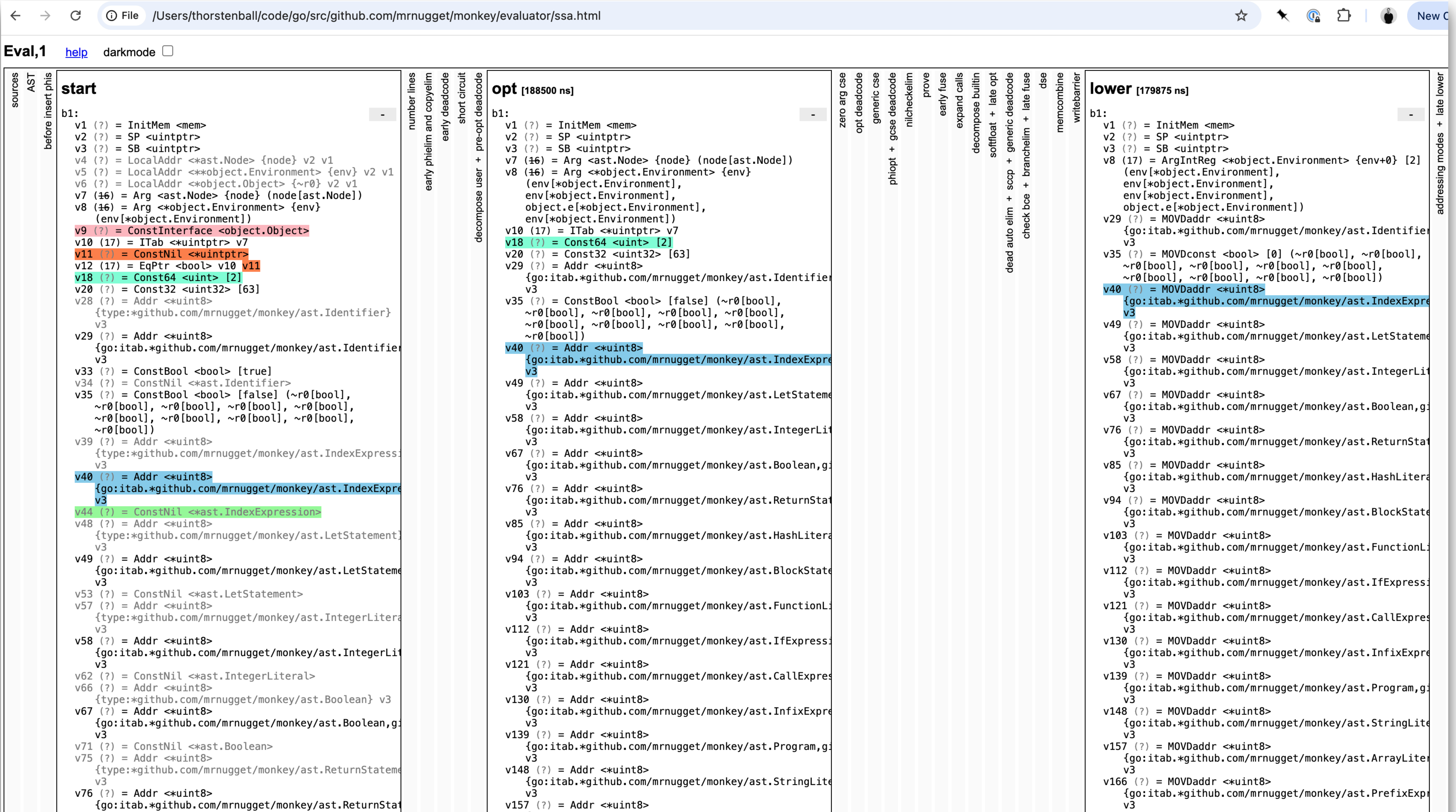
my_function

source

```
funk double_print(num: int) {
    let doubled = num;
    for (let i = num; i > 0; i = i - 1) {
        doubled = doubled + 1;
    }
    print_num(doubled);
}

funk my_function(a: int, b: int) → int {
    let c = a + b + 18 - 5 + 3 + 2;
    let d = 18 - 5 + 3;
    let e = a + b + 30;
    let f = 100 - a - b + 3 + 2;
    let g = c + d + e; // dead code
    let h = a + b + c + e + f;

    let k = 50;

    let result = 0;
    for (let i = 0; i < 10; i = i + 1) {
        result = result + 5;
        if (i < a) {
            if (k ≥ d) {
                double_print(i);
            } else {
                print_num(i);
            }
        } else {
            result = result + c;
        }
        for (let j = 5; j > 0; j = j - 1)
            result = result - j;
        }
        print_num(result);
        print_string("-");
    }
    return result;
}

funk main() → int {
    my_function(5, 10);
    my_function(50, 2);
    return 0;
}
```

pre-ssa | ssa | value_numbering | constant_propagation | unreachable_code_elimination | dead_code_elimination | clean | unreachable_code_elimination2 | regalloc | asm

(Stole it from Go)



Two lessons:

1. Keep thinking “how will I debug this?”
2. Invest in debug tooling

Runtime & GC

Immix

Immix - A Mark-Region Garbage Collector with Spac...

Immix: A Mark-Region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance *

Stephen M. Blackburn
Australian National University
Steve.Blackburn@anu.edu.au

Kathryn S. McKinley
The University of Texas at Austin
mckinley@cs.utexas.edu

Abstract

Programmers are increasingly choosing managed languages for modern applications, which tend to allocate many short-to-medium lived small objects. The garbage collector therefore directly determines program performance by making a classic space-time tradeoff that seeks to provide space efficiency, fast reclamation, and mutator performance. The three canonical tracing garbage collectors: semi-space, mark-sweep, and mark-compact each sacrifice one objective. This paper describes a collector family, called *mark-region*, and introduces *opportunistic* defragmentation, which mixes copying and marking in a single pass. Combining both, we implement *immix*, a novel high performance garbage collector that achieves all three performance objectives. The key insight is to allocate and reclaim memory in contiguous regions, at a coarse *block* grain when possible and otherwise in groups of finer grain *lines*. We show that immix outperforms existing canonical algorithms, improving total application performance by 7 to 25% on average across 20 benchmarks. As the mature space in a generational collector, immix matches or beats a highly tuned generational collector, e.g. it improves jbb2000 by 5%. These innovations and the identification of a new family of collectors open new opportunities for garbage collector design.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Memory management (garbage collection)

General Terms Algorithms, Experimentation, Languages, Performance, Measurement

Keywords Fragmentation, Free-List, Compact, Mark-Sweep, Semi-Space, Mark-Region, Immix, Sweep-To-Region, Sweep-To-Free-List

1. Introduction

Modern applications are increasingly written in managed languages and make conflicting demands on their underlying memory managers. For example, real-time applications demand pause-time guarantees, embedded systems demand space efficiency, and servers demand high throughput. In seeking to satisfy these demands, the literature includes reference counting collectors and three canonical tracing collectors: semi-space, mark-sweep, and mark-compact. These collectors are typically used as building blocks for more sophisticated algorithms. Since reference counting is incomplete, we omit it from further consideration here. Unfortunately, the tracing collectors each achieve only two of: *space efficiency*, *fast collection*, and *mutator performance* through contiguous allocation of contemporaneous objects.

Figure 1 starkly illustrates this dichotomy for full heap versions of mark-sweep (MS), semi-space (SS), and mark-compact (MC) implemented in MMTk [12], running on a Core 2 Duo. It plots the geometric mean of total time, collection time, mutator time, and mutator cache misses as a function of heap size, normalized to the best, for 20 DaCapo, SPECjvm98, and SPECjbb2000 benchmarks, and shows 99% confidence intervals. The crossing lines in Figure 1(a) illustrate the classic space-time trade-off at the heart of garbage collection. Mark-compact is uncompetitive in this setting due to its overwhelming collection costs. In smaller heap sizes, the space and collector efficiency of mark-sweep perform best since the overheads of garbage collection dominate total performance. Figures 1(c) and 1(d) show that the primary advantage for semi-space is 10% better mutator time compared with mark-sweep, due to better cache locality. Once the heap size is large enough, garbage collection time reduces, and the locality of the mutator dominates total performance so semi-space performs best.

To explain this tradeoff, we need to introduce and slightly expand memory management terminology. A tracing garbage collector performs *allocation* of new objects, *identification* of live objects, and *reclamation* of free memory. The canonical collectors all identify live objects the same way, by marking objects during a transitive closure over the object graph.

Reclamation strategy dictates allocation strategy, and the literature identifies just three strategies: (1) *sweep-to-free-list*, (2) *evacuation*, and (3) *compaction*. For example, mark-sweep collectors allocate from a free list, mark live objects, and then *sweep-to-free-list*

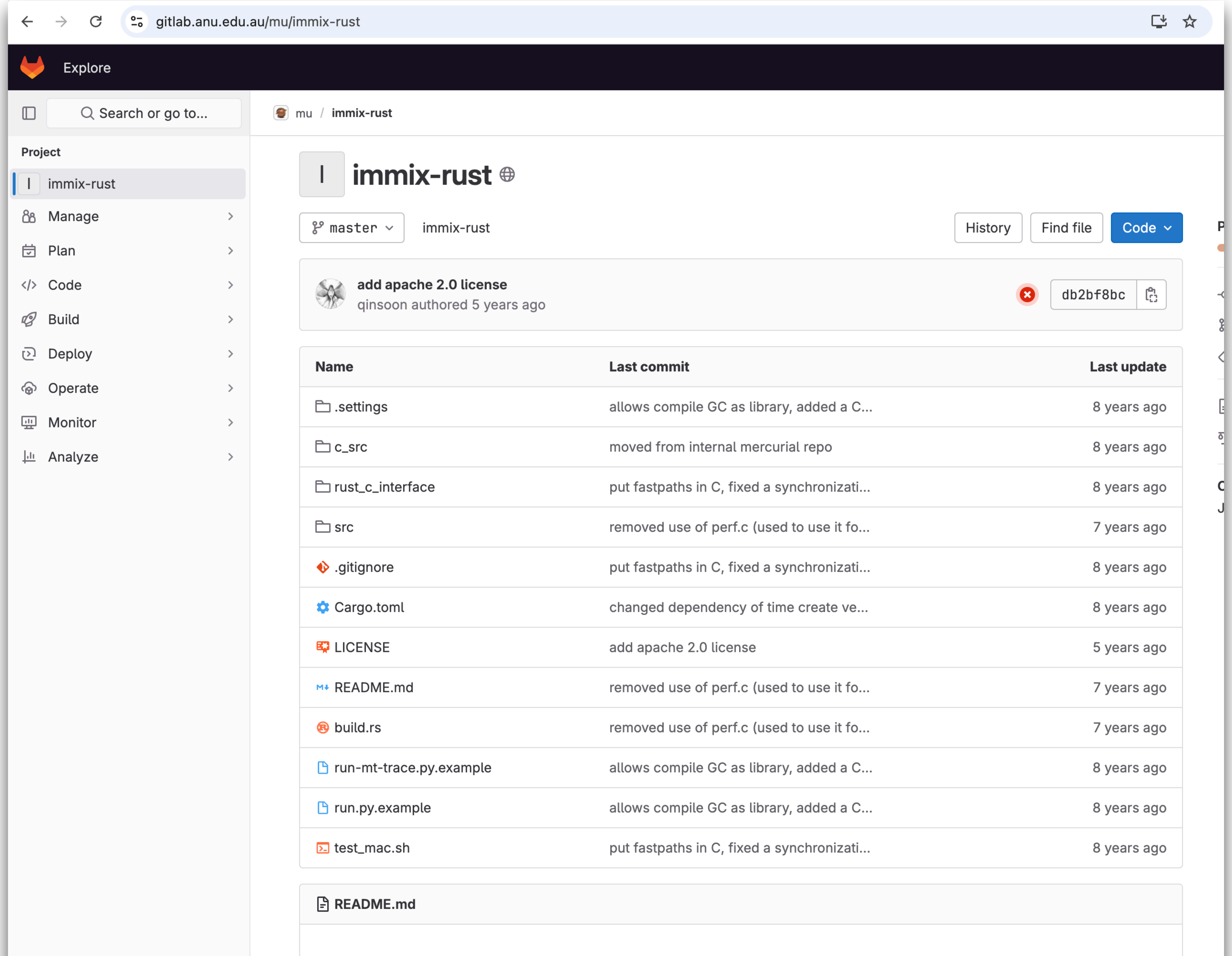
(a) Total Time (b) Garbage Collection Time

(c) Mutator Time (d) Mutator L2 Misses

Figure 1. Performance Tradeoffs For Canonical Collectors: Geometric Mean for 20 DaCapo and SPEC Benchmarks.

* This work is supported by ARC DP0666059, NSF CNS-0719966, NSF CCF-0429859, NSF EIA-0303609, DARPA F33615-03-C-4106, Microsoft, Intel, and IBM. Any opinions, findings and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

PLDI'08, June 7–13, 2008, Tucson, Arizona, USA.


gitlab.anu.edu.au/mu/immix-rust
Explore
Search or go to...
mu / immix-rust
Project
immix-rust
Manage
Plan
Code
Build
Deploy
Operate
Monitor
Analyze
immix-rust
master
immix-rust
History
Find file
Code
add apache 2.0 license
qinsoon authored 5 years ago
db2bf8bc
Name
Last commit
Last update
.settings
allows compile GC as library, added a C...
8 years ago
c_src
moved from internal mercurial repo
8 years ago
rust_c_interface
put fastpaths in C, fixed a synchronizati...
8 years ago
src
removed use of perf.c (used to use it fo...
7 years ago
.gitignore
put fastpaths in C, fixed a synchronizati...
8 years ago
Cargo.toml
changed dependency of time create ve...
8 years ago
LICENSE
add apache 2.0 license
5 years ago
README.md
removed use of perf.c (used to use it fo...
7 years ago
build.rs
removed use of perf.c (used to use it fo...
7 years ago
run-mt-trace.py.example
allows compile GC as library, added a C...
8 years ago
run.py.example
allows compile GC as library, added a C...
8 years ago
test_mac.sh
put fastpaths in C, fixed a synchronizati...
8 years ago
README.md

tucan_runtime/src/heap.rs

```rust
pub struct ImmixHeap {⋯
}

impl ImmixHeap {
    pub fn new() -> ImmixHeap {⋯
    }

    pub fn new_with_auto_gc(stack_bottom: *const *const c_void) -> ImmixHeap {⋯
    }

    pub fn enable_trace_gc(&mut self) {⋯
    }

    pub fn stats(&self) -> Stats {⋯
    }

    pub fn alloc<T>(&mut self, object: T) -> Result<RawPtr<T>, AllocError>
    where
        T: TucanObject,
    {
        let header_size: usize = size_of::<ObjectHeader>();
        let object_size: usize = size_of::<T>();
        let total_size: usize = header_size + object_size;

        let request: AllocRequest = AllocRequest::new(total_size)?;
        let space: *const u8 = self.find_space(request)?;

        self.allocated_objects_space += request.size;
        self.live_bytes += request.size;

        unsafe { write_header::<T>(space, size_bytes: object_size as u32, size_class: request.class) };

        let object_space: *const u8 = unsafe { space.add(count: header_size) };
        unsafe {
            std::ptr::write_bytes(dst: object_space as *mut T, val: 0, count: 1);
            std::ptr::write(dst: object_space as *mut T, src: object);
        }

        Ok(RawPtr::new(ptr: object_space as *const T))
    }

    pub fn alloc_array(&mut self, size_bytes: u32) -> Result<TucanByteArray, AllocError> {
        let header_size: usize = size_of::<ObjectHeader>();
        let total_size: usize = header_size + size_bytes as usize;

        let request: AllocRequest = AllocRequest::new(total_size)?;
        let space: *const u8 = self.find_space(request)?;

        self.allocated_arrays_space += request.size;
        self.live_bytes += request.size;

        unsafe { write_header::<TucanByteArray>(space, size_bytes, size_class: request.class) };

        let array_space: *const u8 = unsafe { space.add(count: header_size) };
        let array: &mut [u8] =
            unsafe { std::slice::from_raw_parts_mut(data: array_space as *mut u8, len: size_bytes as usize) };
        array.iter_mut().for_each(|b: &mut u8| *b = 0);

        Ok(RawPtr::new(ptr: array_space))
    }
```

Lesson:
Reference implementations are amazing

Thorsten Ball @thorstenball · Nov 20, 2022

Now if that's not a Bob Dylan line I don't know

encapsulation of `Address` and `ObjectReference` types, (*ii*) managing ownership of address blocks, (*iii*) managing global ownership of thread-local allocations, and (*iv*) utilizing Rust libraries to support efficient parallel collection.

4.1 Encapsulating Address Types

Memory managers manipulate raw memory, conjuring language-level objects from raw memory. Experience shows the importance of abstracting over both arbitary raw addresses and references to user-level objects [4, 11]. Such abstraction offers type safety and disambiguation with respect to implementation-language (Rust) references. Among the alternatives, raw pointers can be misleading and dereferencing an untyped arbitrary pointer may yield un-

3 · 19

Steve Blackburn (@steveblackburn@discuss.systems)
@stevemblackburn

🙄 My usual coauthor was not involved with that paper. No way that line would have slipped past her and her common subexpression elimination.

10:47 AM · Nov 20, 2022

Thorsten Ball @thorstenball · Dec 11, 2022

Hmmm, so is it possible to have an Immix heap without compaction?

I kinda want to avoid forwarding pointers as long as possible.

More reading to do...

2 6

goyox86 @goyox86 · Dec 11, 2022

I thought the idea of opportunistic defragmentation was kind of part of Immix

1

Thorsten Ball @thorstenball · Dec 11, 2022

Yeah, opportunistic defragmentation. I'm still trying to grok a lot of parts 🙂

1

Steve Blackburn (@steveblackburn@discuss.systems)
@stevemblackburn

Yes. You can treat opportunistic copying as an optimisation.

Start with no copying, the once you have that working, add in the copying.

We're doing this right now for our Julia and Ruby ports (where copying is a bit of a big deal). Non copying is a fine place to start.

Thorsten Ball
@thorstenball

love it

Laurence Tratt
@laurencetratt

My experience is similar. I wrote a simple testing library lang_tester to make this more palatable for different compilers. I have a barely-started successor to lang_tester to try and generalise things while still maintaining usability because it's such a powerful technique!

3:17 PM · Jun 14, 2022

1 4

Post your reply

Reply

CF Bolz-Tereick @cfbolz · Jun 14, 2022
It was probably one of you who pointed me to it, but I found the 'correctness' section on the recent Chris Fallin post very interesting: cfallin.org/blog/2022/06/0...

1 7

Laurence Tratt @laurencetratt · Jun 14, 2022
It's a very good post! My (admittedly somewhat limited) experience is that some components are much easier to fuzz than others though and I don't yet have a good intuition about when it's worth the effort or not.

1

CF Bolz-Tereick @cfbolz · Jun 14, 2022
I've never really worked on our (poor) register allocator but it's definitely had an enormously long bug tail. The fuzzer kept finding bugs after many days of running, and we keep finding the occasional bug in the wild too

1 3

Ben L. Titzer @TitzerBL · Feb 4, 2023
RegAlloc bugs are right up there with GC bugs as the things I most hate to spend time on.

3 55

Lesson:
Put stuff out into the world

The biggest lesson

Thank you