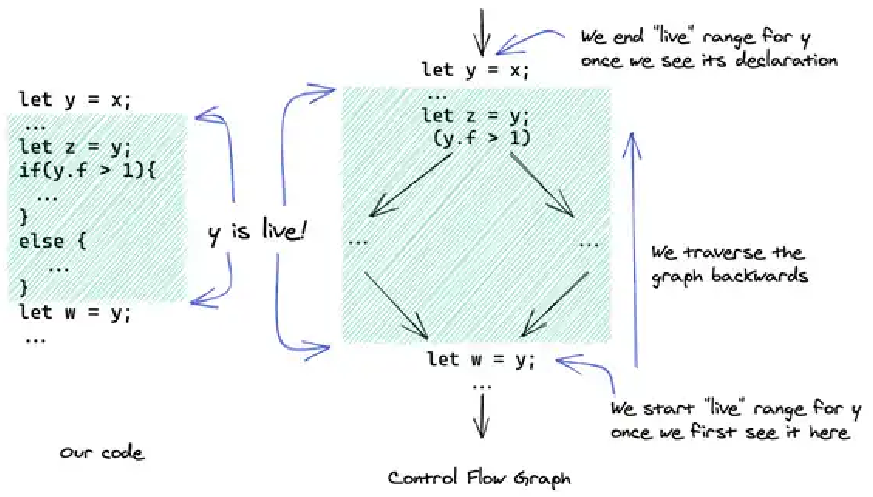
CREATING THE BOLT COMPILER: PART 5

A tutorial on liveness and alias dataflow analysis

JUNE 27, 2020   7 MIN READ

Last updated: December 20, 2020



SERIES: CREATING THE BOLT COMPILER

TABLE OF CONTENTS

Dataflow Analysis - the big picture

In the previous post in the series, we looked at how type-checking the core language worked. This analysis is *flow-insensitive* - it does not depend on the flow of the execution of the program. If an expression is of type `int` then it will have type `int` regardless of what was executed before or after it.

Some program properties do however depend on the execution path taken by a program. Dataflow analysis tracks how a particular value might propagate as the program executes.

For example, a value might *alias* - you could have multiple references pointing to that value. Whether two references `x` and `y` alias can't be determined by looking at their assignments in isolation - we need to track the execution to determine if they have the same value assigned to them.

alias_example

Copy

```
let x = someObject
...
let y = someObject // x and y alias as both point to same objec
```

Another example is **liveness analysis** - we say a value is *live* if some point later in the program it could be used, and *dead* otherwise.

liveness_example

Copy

```
let x = someValue // someValue is live
...
print(x) // as we use someValue here
x = someOtherValue // someOtherValue isn't used - so dead
```

Why do we care about alias analysis and liveness analysis? Well it turns out that Rust's *borrow-checker* uses a combination of these to determine when a reference is borrowed. Bolt has a *linear* capability in its type system which acts the same.

> Both Rust's borrow checker and Bolt's capabilities prevent data races - an explanation why is in my dissertation.

In a nutshell, Rust works on the principle of **ownership**: you either have one reference (owner) that can read/write (a *mutable* reference) or you can *borrow* (aka alias) the reference.

To enforce this we care about 2 things:

1. When is a reference borrowed? (alias analysis)
2. For how long is it borrowed? (liveness analysis)

Let's elaborate on that second point. When is a reference borrowed? The first version of Rust's borrow checker said this was whilst an alias was in scope.

borrow_example

Copy

```
let x = something // this is pseudocode not Rust
{
  let y = x
  ... // y will be used in future
  print(y)
  // no longer using y
  x = somethingElse
}
// y is out of scope
```

That means we cannot reassign `x` in the example until `y` is out of scope. This is a "lexical lifetime" - `y` 's borrow lifetime is determined by its lexical scope. So that means `x` can't be reassigned, since it is borrowed at that point. But `y` isn't being used so surely `x` isn't still borrowed?

That's the idea behind **non-lexical lifetimes** - the borrow ends when the value of `y` is dead - i.e. it is not being used.

In Bolt, we'll be tracking the lifetimes of references to *objects*.

Let's get implementing it!

Alias Analysis

The first step is to determine when two values alias. How do we know two references will point to the same object without actually executing the program?

We use **abstract interpretation**.

Abstract Interpretation

Abstract interpretation is the process of simulating the execution of the program but only storing the program properties we care about. So in our case, we don't care about what the actual result of the program execution is, we're just tracking the *set of aliases*.

> Implicit in this is a set of rules for how the program will execute, we call this its **operational semantics**. We will not go into details here, but it's things like:
>
> - Do we evaluate expressions left-to-right or right-to-left?
> - When calling a function, do we fully evaluate the argument expression and then call the function with the value of that argument, or do we just plug in the unevaluated argument expression directly into the function and only evaluate it at the point it is used in the function body? The former is called **call-by-value** and is used in most mainstream languages like Java and Python, the latter is called **call-by-name** and is used in Haskell and Lisp.
>
> There's a lot more to say about this - perhaps it's worthy of its own blog post? Send me a tweet if you would like one!

When do we have aliasing? When a new variable is declared or when it is reassigned. For simplicity (and also because we don't want the lifetime of the alias to be larger than the original value) we will not allow aliasing via reassigning an existing variable (e.g. `x := y`).

So we care about expressions of this form:

Copy

```
let x = e
```

The expression `e` can *reduce* to a value when executing e.g. `1+2` reduces to `3`.

If when executing `e` it reduces to a reference `y`, then the overall expression would look like:

Copy

```
let x = y
```

And so `x` would alias `y`. So let's write a function that, given an expression, will return the list of identifiers that it could possible reduce to. That'll tell us what `x` could possibly alias.

We'll store this helper function in an "environment" of helper functions used in the data-race type-checking stage of the Bolt compiler: `data_race_checker_env.mli`

The type signature of this OCaml function is as we'd expect:

[data_race_checker_env.mli]

Copy

```
val reduce_expr_to_obj_ids : expr -> identifier list
```

A reminder of the type of `expr` defined in the [previous post] - `loc` encodes the line number and position of the expression - used for error messages.

[typed_ast.mli]

Copy

```
type identifier =
  | Variable of type_expr * Var_name.t
  | ObjField of Class_name.t * Var_name.t * type_expr * Field_n
      (** class of the object, type of field *)
type expr =
  | Integer     of loc * int
  | Boolean     of loc * bool
  | Identifier  of loc * identifier
  | Constructor of loc * type_expr * Class_name.t * constructor
  | Let         of loc * type_expr * Var_name.t * expr
  | Assign      of loc * type_expr * identifier * expr
  | If          of loc * type_expr * expr * block_expr * block_
  ...

and block_expr =
  | Block of loc * type_expr * expr list
```

Starting off with the simple cases, it's clear an integer or a boolean value doesn't reduce to an identifier, and an identifier expression reduces to that identifier. A `new SomeClass()` constructor also doesn't reduce to an identifier.

data_race_checker_env.ml

Copy

```
let rec reduce_expr_to_obj_ids expr =
  match expr with
  | Integer _ | Boolean _ -> []
  | Identifier (_, id) -> [id]
  | Constructor (_, _, _, _) -> []
```

If we have a let expression or assigning to an identifier, then it reduces to that identifier (e.g. `let x = ___` reduces to `x`, and `x.f:= __` reduces to `x.f`):

data_race_checker_env.ml

Copy

```
...
  | Let (_, _, _, bound_expr) -> reduce_expr_to_obj_ids bound_e
  | Assign (_, _, _, assigned_expr) -> reduce_expr_to_obj_ids a
```

We can continue doing this for other cases. But what about an `if` statement? Does this expression reduce to `x` or `y`?

Copy

```
if (someCondition) {
  x
} else {
  y
}
```

In general, without actually executing the expression, we don't know. So we'll have to approximate.

Let's remind ourselves of our goal - to mark a value as borrowed/not linear (Rust / Bolt equiv. terminology) if it is aliased. We're trying to eliminate data races, so we have to be *conservative* - assume it might be aliased even if it isn't. The worst thing would be to let a data race slip through. Abstract interpretation *always* errs on the side of soundness.

So we'll *overapproximate* the list of possible identifiers the expression could reduce to - we don't know which branch so we'll count the identifiers from *both* branches.

data_race_checker_env.ml

Copy

```
...
  | If (_, _, _, then_expr, else_expr) ->
      let then_ids = reduce_block_expr_to_obj_ids then_expr in
      let else_ids = reduce_block_expr_to_obj_ids else_expr in
      then_ids @ else_ids
```

So in the example above, we'd return a list `[x, y]` .

Computing all aliases

So we know if we have an expression `let y = e` and `e` reduces to `x` , then we have `let y = x` and so `y` is an alias of `x` .

In the expression below, we would also run abstract interpretation (simple in this case) to find that `z` and `w` are aliases of `y` .

Copy

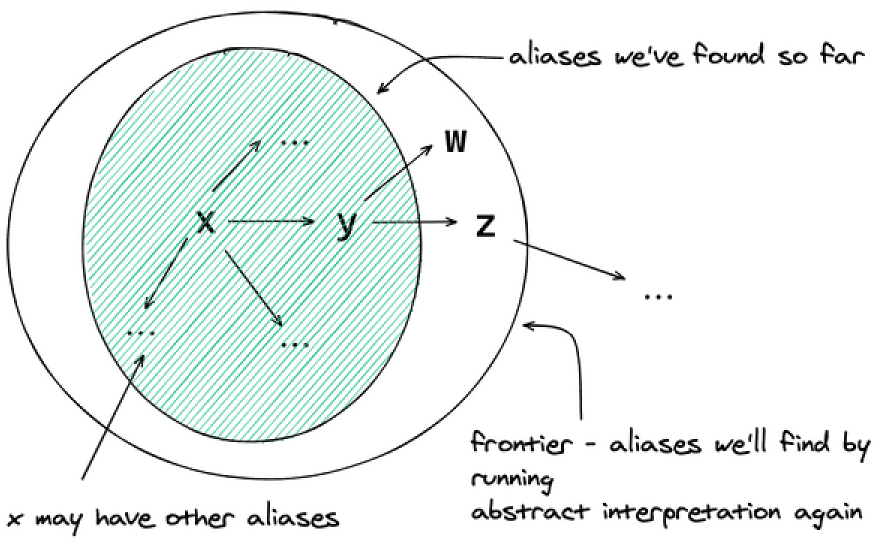
```
let y = x
...
let z = y
if(y.f > 1){
  ...
}
else {

}
...
```

```
let w = y
```

By transitivity, `z` and `w` must also be aliases of `x` .

I like to think of this like a graph where each edge is a direct/immediate alias. We can find all aliases, by repeatedly applying abstract interpretation in a “breadth-first search” style - each iteration we’re expanding the frontier. And each iteration we try to find aliases of the aliases we’ve found - so the first iteration we find aliases of `x` , then the next iteration we find aliases of `x` and `y` and so on…


aliases we've found so far
w
x
y
z
...
frontier - aliases we'll find by running abstract interpretation again
x may have other aliases

So we repeat, until we find no more aliases. The full code is linked in the repo, and includes an option of matching fields (i.e. should we consider aliases of `x.f` as well as `x` ?) We won’t in this case, but other aspects of Bolt’s data-race type-checker do use this.

But the beauty of functional programming is that the function says what we’re doing with no boilerplate:

data_race_checker_env.ml

Copy

```
let rec get_all_obj_aliases should_match_fields curr_aliases bl
    find_immediate_aliases_in_block_expr should_match_fields na
      block_expr
    |> fun updated_aliases ->
    if var_lists_are_equal updated_aliases curr_aliases
       then curr_aliases (* we're done! *)
    else get_all_obj_aliases should_match_fields updated_aliase
```

```
        (* repeat with an expanded frontier *)
```

Liveness Analysis

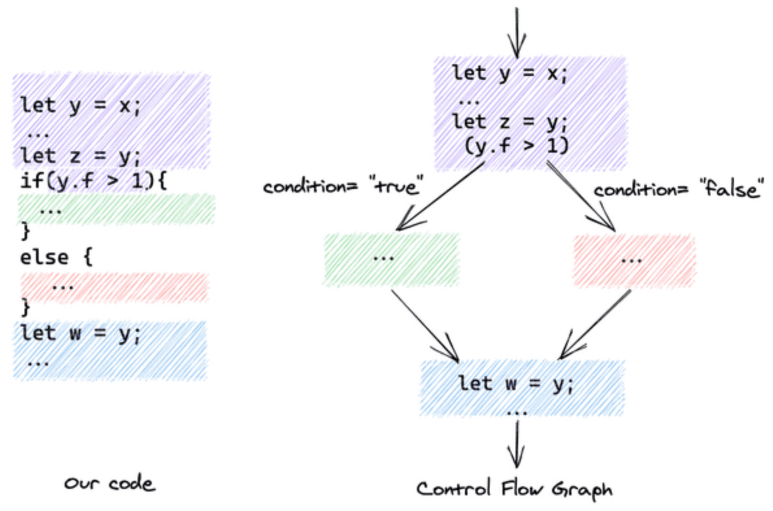
Alright, so we're halfway there - we've identified our aliases, now we need to find out when they're live. Remember a value is *live* if there's some program execution path in the future it will be used on.

Control Flow Graph

To do this, let's formalise the notion of "execution paths" in a program. We're going to be representing a program as a graph of instructions, with edges representing the steps in the execution. We call this the **Control Flow Graph** of the program.

However, if every statement represented a node on the graph, with edges between them, the graph would be incredibly large (a 100 line program would have 100 statements). We can be a bit cleverer and group together statements that will always execute right after each other.

If there's only **one path** from the start statement to the end statement in a group of statements, we can represent this as a **single node** in our control flow graph. We call this node a **basic block** of statements. Below you can see the lines of code highlighted in different colours to show which basic block they lie in.



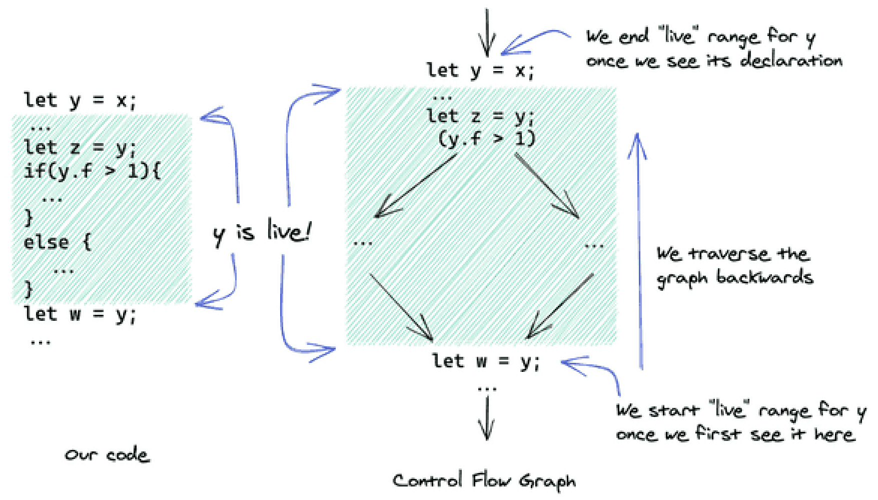
Note how the control flow graph shows two different execution paths, corresponding to the if-else branch taken.

In our graph, we can pick any statement in the program and ask, is the value of variable `y` live at this point?

A naive way of checking this is to traverse the graph forwards until you see a use of this value of `y` (so `y` is live) or until you reach the end (you therefore know `y` is dead). But this is hopelessly wasteful - for every statement we have to go forward and traverse the graph to check if the variable's are used.

We can look at it another way, if we traverse *backwards*, then the first use of `y` that we encounter is the last use if we traversed it going forwards. So as soon as we see `y` being used when we go backwards, we know that whilst `y` is defined, it is live.

A picture helps so let's look at our graph:


We end "live" range for y once we see its declaration
let y = x;
...
let z = y;
if(y.f > 1){
...
}
else {
...
}
let w = y;
...
y is live!
(y.f > 1)
We traverse the graph backwards
We start "live" range for y once we first see it here
Our code
Control Flow Graph

So to summarise, in liveness analysis we:

- Traverse the Control Flow Graph backwards
- Keep track of the set of live values
- If we see a value for the first time, we add it to our set
- We remove a value if we see its definition

Implementing Our Alias Liveness Checker

Now we have our theoretical understanding in place, let's implement the checker.

Whenever we encounter an identifier, we need to know what the original object reference was, the set of possible aliases, and separately those that are live.

type_alias_liveness.ml

Copy

```
let type_alias_liveness_identifier obj_name possible_aliases
  filter_linear_caps_fn live_aliases id =

  let id_name = get_identifier_name id in
```

Our function deals with three cases depending on the identifier name:

- It is the original object reference
- It is a possible alias
- It is another reference we don't care about

In the first case, we want to filter the linear capabilities (since the reference is not linear so can't use them) if there are live aliases. Along with the updated identifier, we also return the unchanged set of live aliases.

type_alias_liveness.ml

Copy

```
if id_name = obj_name then (* original object reference *)
    let maybe_updated_capabilities =
      update_capabilities_if_live_aliases filter_linear_caps_fn
      live_aliases (get_identifier_capabilities id) in
    (set_identifier_capabilities id maybe_updated_capabilities,
```

Otherwise, we need to check if the identifier is one of the possible aliases - if so we add it to the set of live aliases we are tracking. So we then return this updated set of live aliases along with the identifier.

type_alias_liveness.ml

Copy

```
else
  ( match
      List.find
        ~f:(fun poss_alias ->
              identifier_matches_var_name poss_alias id)
        possible_aliases
    with
  | Some alias -> alias :: live_aliases
```

```
      | None       -> live_aliases )
      |> fun updated_live_aliases -> (id, updated_live_aliases)
```

And the dual in our `type_alias_liveness_expr` function is when we see our `let x = e` expression. Here, remember we execute `e` first, and then assign it to `x`, so when traversing this backwards, we remove `x` from the set of live aliases first (since we've seen its definition), *then* we traverse the bound expression:

type_alias_liveness.ml

Copy

```
  | Let (loc, type_expr, var_name, bound_expr) ->
      (* remove this var from the set of live aliases *)
      type_alias_liveness_expr_rec
        (List.filter ~f:(fun name -> not (var_name = name)) liv
        bound_expr
      |> fun (updated_bound_expr, updated_live_aliases) ->
      (Let (loc, type_expr, var_name, updated_bound_expr), upda
```

One interesting case in our Control Flow Graph is the `if-else` split. We treat each branch independently of each other, since they are independent paths in our graph.

type_alias_liveness.ml

Copy

```
  | If (loc, type_expr, cond_expr, then_expr, else_expr) ->
      type_alias_liveness_block_expr_rec live_aliases then_expr
      |> fun (updated_then_expr, then_live_aliases) ->
      type_alias_liveness_block_expr_rec live_aliases else_expr
      |> fun (updated_else_expr, else_live_aliases) ->
```

How do we recombine the two branches? Well, the definition of liveness is if there is *some* path in which the value is used. Again, we don't know which branch is taken, because we can't execute the program, so we *over-approximate* - we assume both paths could have been taken - so *union* their sets of live aliases, when traversing the if-else condition expression:

type_alias_liveness.ml

Copy

```
  type_alias_liveness_expr_rec (then_live_aliases @ else_live_ali
        |> fun (updated_cond_expr, cond_live_aliases) ->
        ( If (loc, type_expr, updated_cond_expr, updated_then_exp
        , cond_live_aliases )
```

Another interesting case is when we have a while loop. We can't just traverse the loop once, since we might miss out on some values that could be used in a subsequent iteration and therefore are live. We don't know how many times we'll go round the loop so as ever we *over-approximate* - we'll keep going round the loop until the set of live aliases doesn't change (i.e. we've got all possible live aliases):

type_alias_liveness.ml

Copy

```
  and type_alias_liveness_loop_expr aliased_obj_name possible_ali
    filter_linear_caps_fn live_aliases loop_expr =
    type_alias_liveness_block_expr aliased_obj_name possible_alia
      live_aliases loop_expr
    |> fun (updated_loop_expr, updated_live_aliases) ->
    if var_lists_are_equal live_aliases updated_live_aliases then
       (* done! *)
      (updated_loop_expr, updated_live_aliases)
    else
    (* loop again! *)
      type_alias_liveness_loop_expr aliased_obj_name possible_ali
        filter_linear_caps_fn updated_live_aliases updated_loop_e
```

Where does this fit into Bolt?

Our liveness analysis fits into the overall type-checking for linear capabilities in Bolt's data-race type-checker:

type_linear_capabilities.ml

Copy

```
let type_linear_object_references obj_name obj_class class_defn
  let obj_aliases = ...
  ...
  |> fun updated_block_expr ->
  type_alias_liveness_block_expr obj_name obj_aliases
     filter_linear_caps_fn [] (* we start with empty set of liv
     updated_block_expr
  |> fun (typed_linear_obj_ref_block_expr, _) -> typed_linear_o
```

We'll talk more about the other aspects of data-race type-checking later, however the next stage of the tutorial is on *desugaring* - the process of taking our high-level language and simplifying it to a lower-level representation. This will set us up nicely for targeting LLVM later in the compiler series.

SERIES: CREATING THE BOLT COMPILER

← An accessible introduction to type theory and implementing a type-checker

Desugaring - taking our high-level language and simplifying it! →