Sequence 4.4 – Single static assignment

P. de Oliveira Castro S. Tardieu

Optimizations on the IR

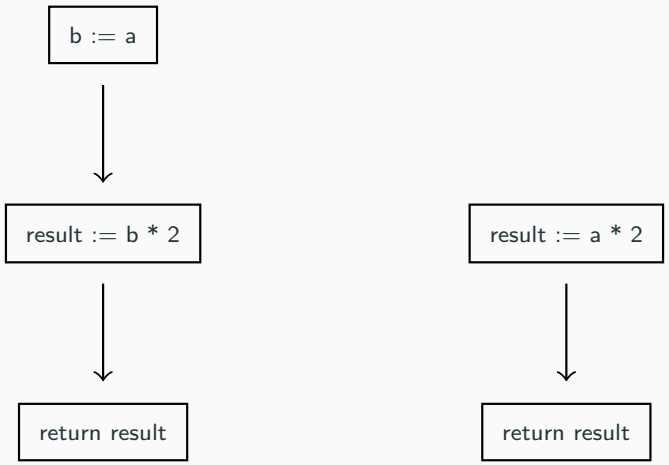
The intermediate representation (IR) is a good place to perform some source code language independent optimizations, such as:

- dead code elimination: some basic blocks are never reachable and can be removed;
- loop unfolding: when the number of iterations of a loop is known at compile time, the compiler may prefer to copy the body code rather than using branches;
- constant propagation: some variables always have the same value which can be used directly;
- variable fusion: when two variables have the same content at their point of use, one can be removed and the other always used.

Example: fusion of two variables

```
let function f(a: int): int =
      let var b := a in b * 2 end
in ... end
```

`f` can be represented as the leftmost chart below.

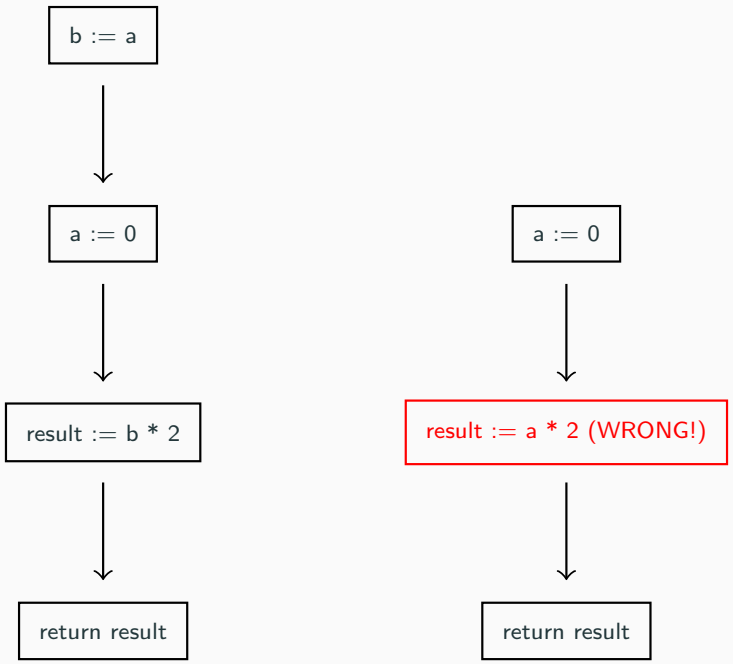


`a` is never used after `b := a`: it is safe to remove this copy, and to use `a` directly. This gives the rightmost chart above.

Blocked fusion

However, the fusion cannot happen as easily if the source variable is modified afterwards:

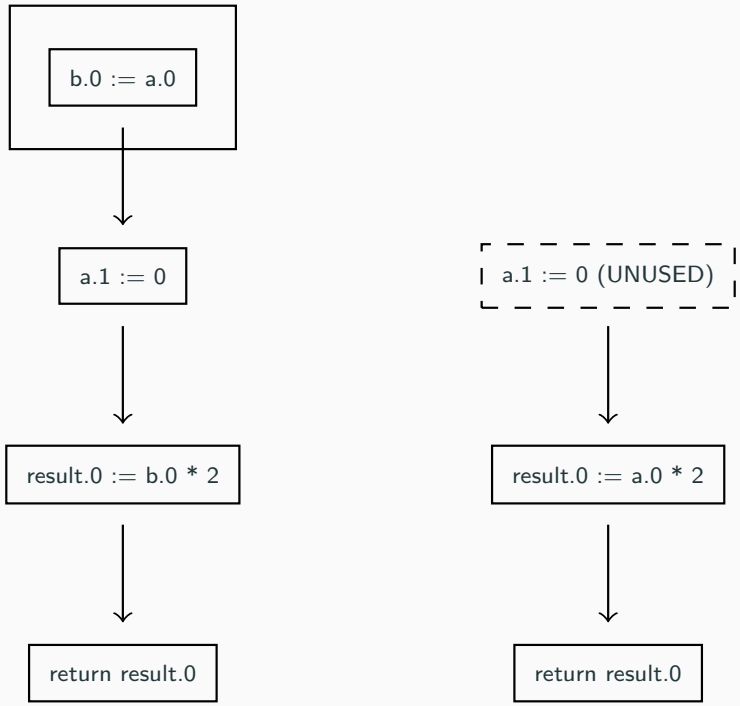
```
let function f(a: int): int =
      let var b := a in a := 0; b * 2 end
in ... end
```



Unique assignment

```
let function f(a: int): int =
      let var b := a in a := 0; b * 2 end
in ... end
```

Let's duplicate variables so that they are each assigned at one place in the code (`a.0`, `a.1`, etc.).



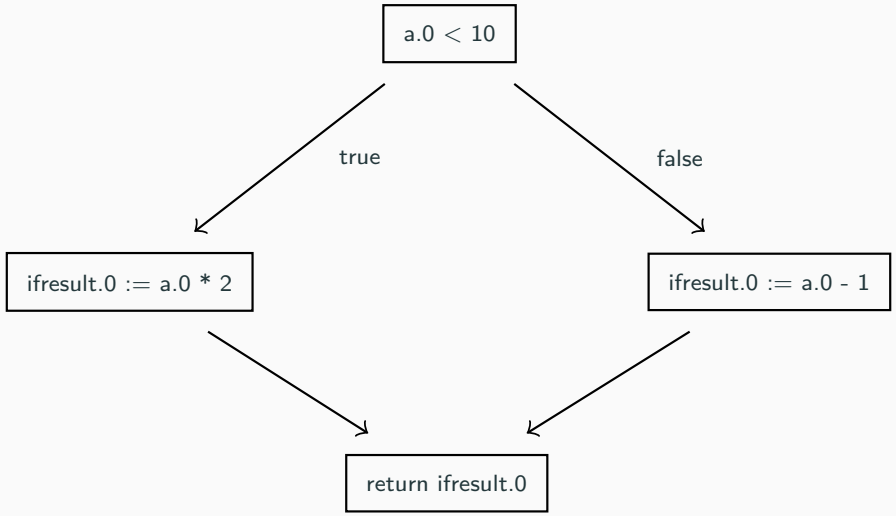
SSA

This technique is named *single static assignment* (or *SSA*):

- Every variable is statically assigned once.
- "Statically" is opposed to "dynamically": we are talking about source code (or IR code) assignments. If a block executes several times, such as a loop body, the variable instance will be assigned several times.
- To that purpose, every assignment creates a new variable with an incremented index number (`a.0`, `a.1`, etc.).
- Using SSA allows for many optimizations: variables fusion, constant propagation, commun subexpression elimination, etc.

But what about branches?

```
let function f(a: int): int =
      if a < 10 then a*2 else a-1
in ... end
```

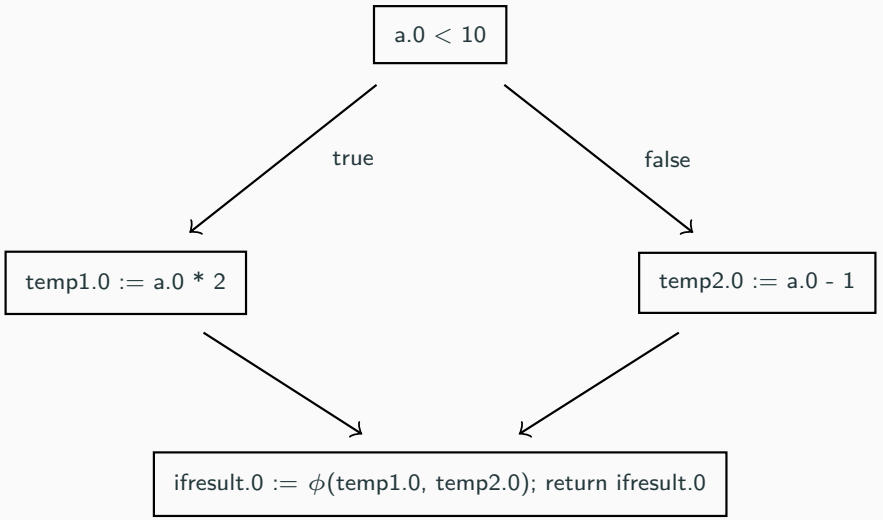


This is not a SSA form: `ifresult.0` is statically assigned at two places. ϕ functions *(phi)* come to the rescue.

ϕ functions

A ϕ function at the beginning of a block takes one of two values depending on whether the block gets entered by the first branch or by the second one.

```
let function f(a: int): int =
      if a < 10 then a*2 else a-1
in ... end
```



LLVM mem2reg to the rescue

The mem2reg (memory to register) optimization pass of LLVM will transform `alloca`/`load`/`store` manipulations into constructs with ϕ functions. The code generated initially by our compiler could be:

```
define i32 @f(i32) #0 {                    ; a is in %0
  %if_result = alloca i32
  %2 = icmp slt i32 %0, 10
  br i1 %2, label %if_then, label %if_else

if_then:                                    if_end:
  %3 = mul i32 %0, 2                          %5 = load i32, i32* %if_result
  store i32 %3, i32* %if_result               ret i32 %5
  br label %if_end                          }

if_else:
  %4 = sub i32 %0, 1
  store i32 %4, i32* %if_result
  br label %if_end
```

LLVM mem2reg to the rescue (cont'd)

After the mem2reg pass, the code becomes as follows. Note the introduction of a ϕ function (`phi`).

```
define i32 @f(i32) {
  %2 = icmp slt i32 %0, 10
  br i1 %2, label %if_then, label %if_else

if_then:
  %3 = mul i32 %0, 2
  br label %if_end

if_else:
  %4 = sub i32 %0, 1
  br label %if_end

if_end:
  %if_result.0 = phi i32 [ %3, %if_then ], [ %4, %if_else ]
  ret i32 %if_result.0 }
```

Conclusion

- SSA (single static assignment) ensures that every variable is assigned at one point only in the IR.
- Using SSA allows the compiler to perform many optimizations.
- When entering a block from several possible paths, a ϕ function will identify potential variables that should be fused together as one.
- The mem2reg optimization pass of LLVM will transform an `alloca`/`store`/`load` based IR into a SSA one.