Compilation (#6b) : SSA for Fun and Optimisations

Gabriel Radanne

Master 1, ENS de Lyon et Dpt Info, Lyon1

2021-2022

Our compiler so far

- Parsing and Typing
- Use of clever™ intermediate representations: CFG and SSA
- Register allocation and code emission

Today, we optimize!

Redundancy – Source of Optimization

Why are there redundancy in programs?

Programmer's convenience:

```
foo(x,y,z) = x * y + z
...
c = foo(a,1,b); // a + b
d = a + b
```

Higher level constructs:

```
x = a[i]; // x = *(a + i * 4)
...
a[i] = y // *(a + i * 4) = y
```

Simple dead code elimination

Dead code elimination: remove code that is never executed

```
a = 4; b = 10;
...
c = 2:
```

$\longmapsto$

```
...
c = 2:
```

Property

A variable is live at its definition if and only if its list of uses is not empty.

True in SSA because each variable has a single definition!

- A variable is **dead** if it has no uses.

Simple dead code elimination

```
while there is some variable v with no uses
  and the statement S that defines v has no other side effects
do:
 delete S
```

What happens when we delete an instruction?

How to implement this?

Simple dead code elimination

```
while there is some variable v with no uses
  and the statement S that defines v has no other side effects
do:
 delete S
```

What happens when we delete an instruction?

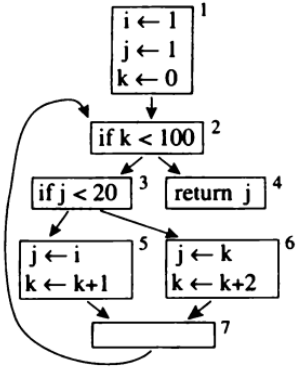
How to implement this?

- Maintain a worklist containing the variables to look at. Loop until it's empty.

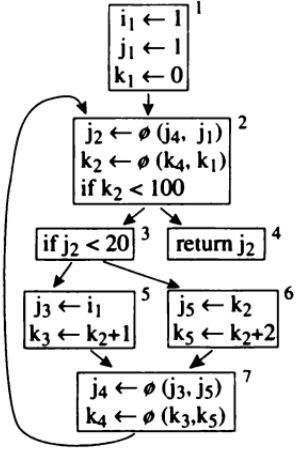
A running example

```
i ← 1
j ← 1
k ← 0
while k < 100
      if j < 20
         j ← i
         k ← k + 1
      else
         j ← k
         k ← k + 2
return j
```

Program example



CFG



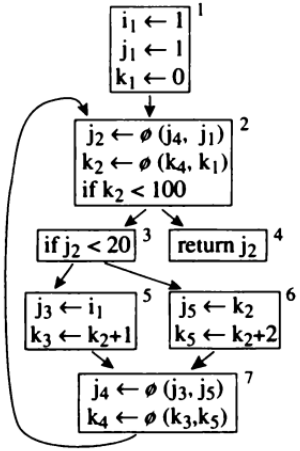
SSA

Simple constant propagation

We can propagate constants of single definitions:

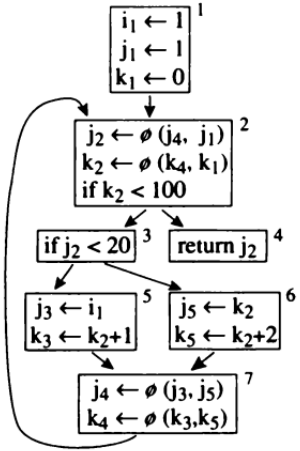
1. if $v \leftarrow c$, we can replace any use of v by c
2. if $v \leftarrow \phi(c, c, c)$, we can replace it by $v \leftarrow c$

- Again, using a worklist algorithm



The worklist algorithm

```
W = list of all statements in the SSA program
while W ≠ ∅:
  pop statement S ∈ W
  if S is v ← φ(c, ..., c):
    replace S by v ← c
  if S is v ← c:
    delete S from the program
    for each statement T using v:
      substitute v by c in T
      W = W ∪ {T}
```



Application of the worklist algorithm

This algorithm can be extended to also apply the following optimisations:

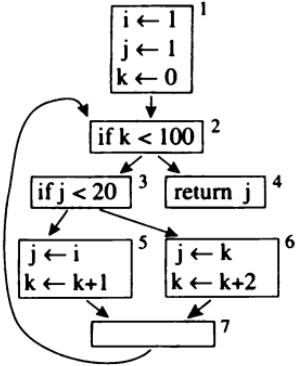
- **Copy propagation**: For each copy $x \leftarrow y$, replace x by y
- **Constant folding**: For each operation $x \leftarrow c_0 + c_1$, replaces $c_0 + c_1$ by its result
- **Constant conditions**: If the result of a jump is known, replace it by an absolute jump
- **Unreachable code**: If a block can't be accessed, remove it
- ...

Conditional Constant Propagation

What is the value of j ? Consider two cases:

- if $j = 1$ always
- if sometimes $j > 20$

▶ We need a more subtle analysis



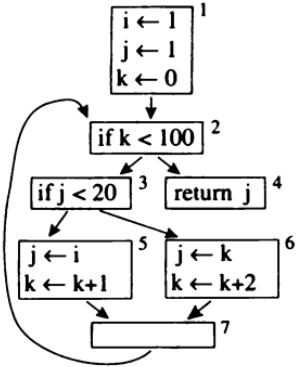
Conditional Constant Propagation

We want to keep track of values precisely!

We denote $\mathcal{V}[v]$ the value of v at a program point

- $\mathcal{V}[v] = \bot$ if we have no evidence that v is assigned
- $\mathcal{V}[v] = 4$ if we found evidence that v is assigned to 4
- $\mathcal{V}[v] = \top$ if we have found evidence that v is assigned to at least two different values

▶ This forms a **lattice**.

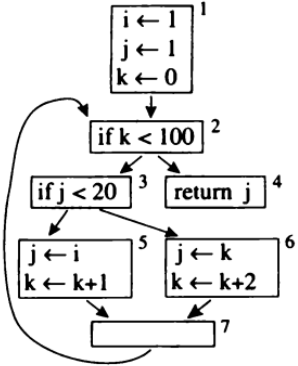


Conditional Constant Propagation

We also want to keep track of executability:

- $\mathcal{E}[B] = false$ if we have no evidence that B can ever be executed
- $\mathcal{E}[B] = true$ if we have evidence that B can be executed

Is computing $\mathcal{V}$ and $\mathcal{E}$ decidable?



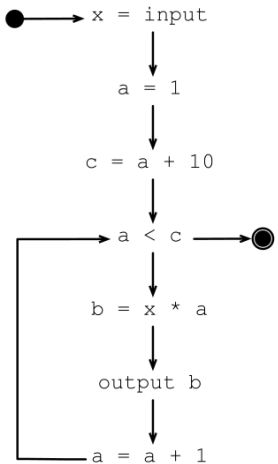
Conditional Constant Propagation

We will compute an over-approximation.

Let's try on a simpler non-SSA example.

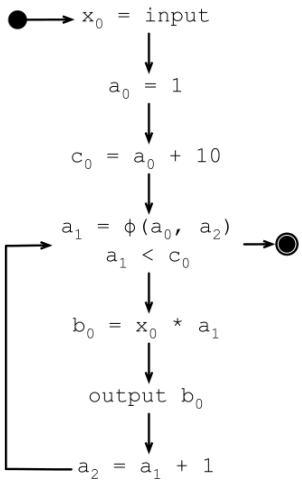
How would you quantify the space required (in term of variables and statements)?

Could we do better?



Sparse Conditional Constant Propagation

Let's try on the SSA version now.

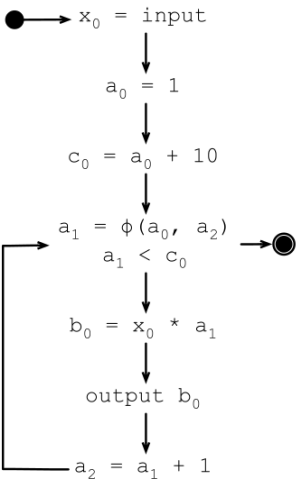


Sparse Conditional Constant Propagation

Let's try on the SSA version now.

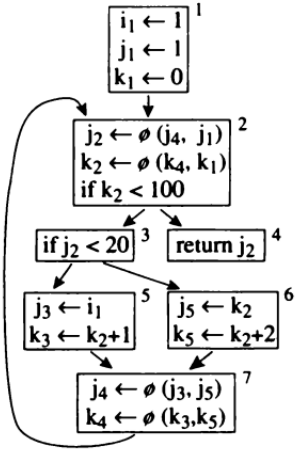
We can store $\mathcal{V}$ only once for each variable!

This is a **sparse** analysis.



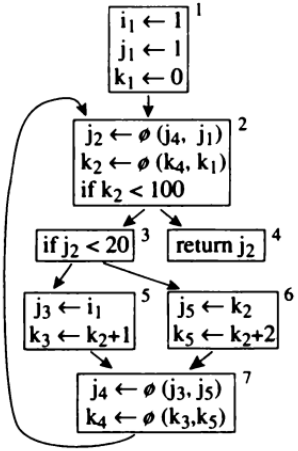
Sparse Conditional Constant Propagation – back to the big example

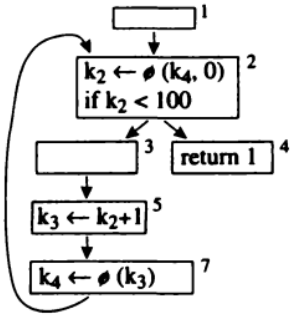
Compute $\mathcal{V}$ and $\mathcal{E}$



Sparse Conditional Constant Propagation – back to the big example

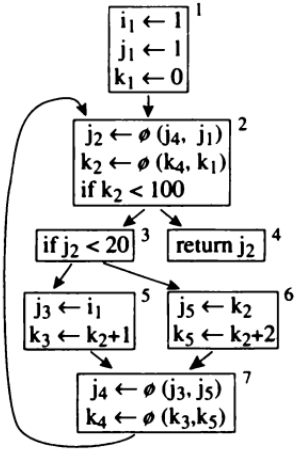
Compute $\mathcal{V}$ and $\mathcal{E}$

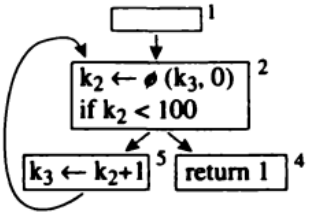




Sparse Conditional Constant Propagation – back to the big example

Compute $\mathcal{V}$ and $\mathcal{E}$





Abstract Interpretation

We computed $\mathcal{V}$ by walking through the program step by step.

We did a “simplified” execution of the program

- This is called **Abstract Interpretation**, an essential tool for program analysis

(see M2 course next year!)

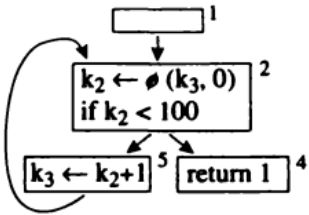
Dead code?

What about dead code elimination here?

- k_2 is used by k_3
- k_3 is used by k_2

Our previous dead code analysis pass doesn't work here.

- We need a new notion of dependency



Control dependencies

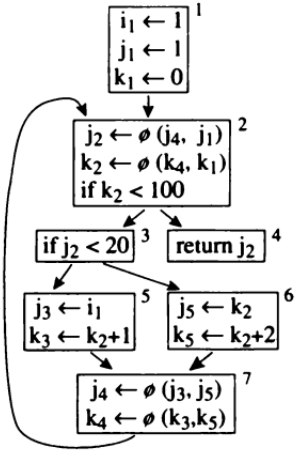
Control dependency

We say that block B is **control-dependent** on A if:

- from A We can branch to U and V
- A path $U \rightarrow \mathbf{exit}$ doesn't go through B
- All path $V \rightarrow \mathbf{exit}$ go through B

Control dependency Graph

The Control Dependency Graph has an edge from A to B if B is control dependent on A



Control dependencies

Control dependency

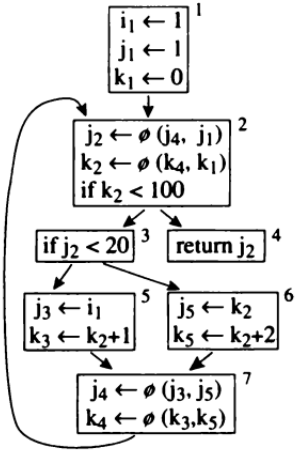
We say that block B is **control-dependent** on A if:

- from A We can branch to U and V
- A path $U \rightarrow \mathbf{exit}$ doesn't go through B
- All path $V \rightarrow \mathbf{exit}$ go through B

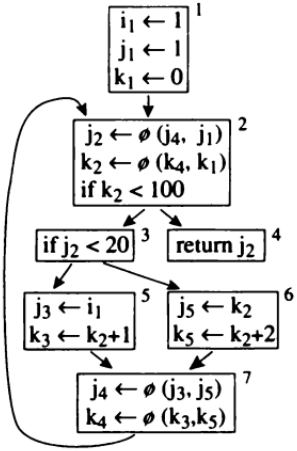
Control dependency Graph

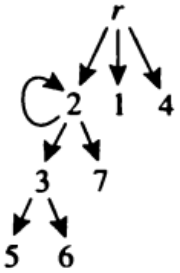
The Control Dependency Graph has an edge from A to B if B is control dependent on A

Computed using **post-dominators**!



Control dependencies





Aggressive code elimination

We can use the control dependency graph to perform aggressive code elimination

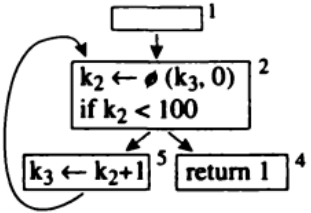
Similarly to the Conditional Constant Propagation, we assume a statement is dead until proven otherwise.

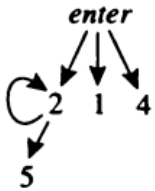
Aggressive code elimination – Algorithm

Mark live any statement which:

1. Performs side effects or returns
2. Defines a variable used in a live statement
3. Is a conditional branch upon which a statement is control-dependent

Then delete all unmarked statements





Uses of control dependency

Control dependency can also be used for parallelization

- If two statements are independent in the CDG, we can run them in parallel!

How to optimize loops?

Loops are where most of computation time is spent.

- Crucial to optimize them well

But loops are also much harder to optimize correctly!

Beware incorrect optimizations

Loop Invariant Code Motion

Extract instructions which are invariant of the loop variable

```
let a = ...;
let b = ...;
for (let i = 0; i < 100; ++i) {
   f(i, a * b);
}
```

$\longmapsto$

```
let a = ...;
let b = ...;
let c = a * b;
for (let i = 0; i < 100; ++i) {
   f(i, c);
}
```

How to obtain that information using previously-seen analysis?

Loop Unswitching

Variant of Loop Invariant Code motion to exchange tests and loops:

```
for (i = 0; i < 100; ++i) {
   if (c) {
// Loop-invariant value.
      f();
   } else {
      g();
   }
}
```

$\longmapsto$

```
if (c) {
   for (i = 0; i < 100; ++i) {
      f();
   }
} else {
   for (i = 0; i < 100; ++i) {
      g();
   }
}
```

Induction Variable Elimination

Induction Variable Elimination (also called “Strength reduction”) replaces iteration variables by simpler expressions

```
for (i = 0; i < 100; ++i) {
   a[i] = 0; // *(a + i * s)
}
```

$\longmapsto$

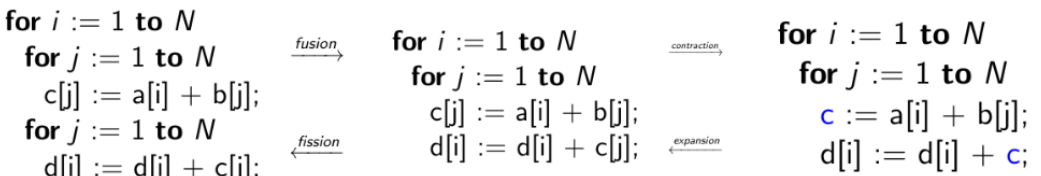
```
let a100 = a + 100 * s;
for (ai = a; ai < a100; ai += s) {
   *a_i = 0;
}
```

- We identify a simpler form for the affine expression $A * i + B$

Applies particularly to arrays.

Loop interchange, fusion, fission, . . .

Exchanging loops and cutting them into pieces



- Fit into a more general (and powerful) framework: the **Polyhedral Model**! (also see M2 course next year!)

Where are the loops?

By the way: how do we actually find the loops?

- Not so simple to identify on a CFG!

Where are the loops?

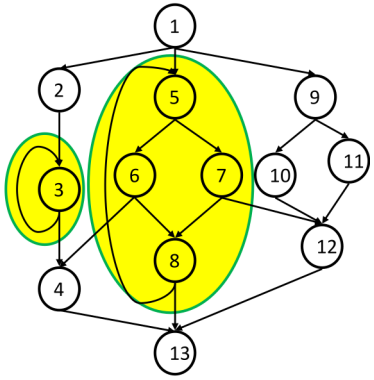
By the way: how do we actually find the loops?

- Not so simple to identify on a CFG!

A loop has a single entry point and contain a cycle

- A **header** dominates all nodes in the loop
- A **back edge** is an edge $t \rightarrow h$ whose head h dominates its tail t.

A loop is the smallest set of nodes containing a back edge and whose only entry point is the header.



Sources of inspiration used for these slides

The SSA book (Chapter 6 by Lennart Beringer)

SSA is functional programming (Andrew Appel — 1998)

A Correspondence between Continuation Passing Style (Richard Kelsey — 1995)

Looking through the SSA glass

Consider a simple couple of instructions:

```
x <- add y z
…
a <- lt x y
```

In normal CFG:

In SSA form:

Looking through the SSA glass

Consider a simple couple of instructions:

```
x <- add y z
…
a <- lt x y
```

In normal CFG:

- the variable `x` being assigned to is carefully distinguished from the expression to the right

In SSA form:

- names are globally unique. So assignments and def-sites of variables are in bijection

Looking through the SSA glass

Consider a simple couple of instructions:

```
x <- add y z
…
a <- lt x y
```

In normal CFG:

- the variable `x` being assigned to is carefully distinguished from the expression to the right
- instructions are computations to be processed:
  the meaning of their evaluation can be compromised at any time

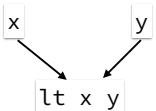
In SSA form:

- names are globally unique. So assignments and def-sites of variables are in bijection
- the instruction itself can be thought of as a static value

Looking through the SSA glass

Consider a simple couple of instructions:

```
x <- add y z
…
a <- lt x y
```



In normal CFG:

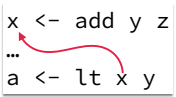
- the variable `x` being assigned to is carefully distinguished from the expression to the right
- instructions are computations to be processed:
  the meaning of their evaluation can be compromised at any time
- use sites can map to various def-sites

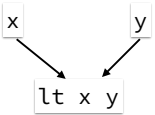
In SSA form:

- names are globally unique. So assignments and def-sites of variables are in bijection
- the instruction itself can be thought of as a static value
- use sites can be thought of as data flow graph edges

Looking through the SSA glass

Consider a simple couple of instructions:





In normal CFG:

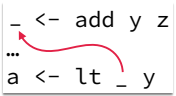
- the variable `x` being assigned to is carefully distinguished from the expression to the right
- instructions are computations to be processed:
  the meaning of their evaluation can be compromised at any time
- use sites can map to various def-sites

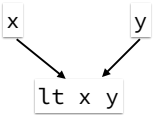
In SSA form:

- names are globally unique. So assignments and def-sites of variables are in bijection
- the instruction itself can be thought of as a static value
- use sites can be thought of as data flow graph edges

Looking through the SSA glass

Consider a simple couple of instructions:





In normal CFG:

- the variable `x` being assigned to is carefully distinguished from the expression to the right
- instructions are computations to be processed:
  the meaning of their evaluation can be compromised at any time
- use sites can map to various def-sites

In SSA form:

- names are globally unique. So assignments and def-sites of variables are in bijection
- the instruction itself can be thought of as a static value
- use sites can be thought of as data flow graph edges

Uses of variables can be represented as simple pointers to their defining instructions
(and LLVM do so to represent programs in memory!)

Binding in the functional world

The idea and application of the SSA form stems from the imperative world.
But in the mean time, the functional world has been doing their own compilation!

The job is of course seemingly quite distinct. We are now fundamentally playing with:

- (Mutually recursive) functions `fun f(x,y) = e`
- let-binding constructs `let x = e1 in e2`

Binding in the functional world

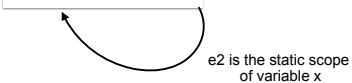
The idea and application of the SSA form stems from the imperative world.
But in the mean time, the functional world has been doing their own compilation!

The job is of course seemingly quite distinct. We are now fundamentally playing with:

- (Mutually recursive) functions
- let-binding constructs

```
fun f(x,y) = e
```

```
let x = e1 in e2
```



And when it comes to the central purpose of all this story, variables, we rely on a powerful idea: static scopes!

Binding in the functional world

The idea and application of the SSA form stems from the imperative world.
But in the mean time, the functional world has been doing their own compilation!

The job is of course seemingly quite distinct. We are now fundamentally playing with:

- (Mutually recursive) functions

```
fun f(x,y) = e
```

- let-binding constructs

```
let v = 3 in
  let y = (let v = 2 * v in 4 * v)
  in y * v + z
```

And when it comes to the central purpose of all this story, variables, we rely on a powerful idea: static scopes!

Binding in the functional world

The idea and application of the SSA form stems from the imperative world.
But in the mean time, the functional world has been doing their own compilation!

The job is of course seemingly quite distinct. We are now fundamentally playing with:

- (Mutually recursive) functions

```
fun f(x,y) = e
```

- let-binding constructs

```
let v = 3 in
  let y = (let v = 2 * v in 4 * v)
  in y * v + z
```

And when it comes to the central purpose of all this story, variables, we rely on a powerful idea: static scopes!

Binding in the functional world

The idea and application of the SSA form stems from the imperative world.
But in the mean time, the functional world has been doing their own compilation!

The job is of course seemingly quite distinct. We are now fundamentally playing with:

- (Mutually recursive) functions

```
fun f(x,y) = e
```

- let-binding constructs

```
let v = 3 in
  let y = (let v = 2 * v in 4 * v)
  in y * v + z
```

And when it comes to the central purpose of all this story, variables, we rely on a powerful idea: static scopes!

Binding in the functional world

The idea and application of the SSA form stems from the imperative world.
But in the mean time, the functional world has been doing their own compilation!

The job is of course seemingly quite distinct. We are now fundamentally playing with:

- (Mutually recursive) functions
- let-binding constructs

```
fun f(x,y) = e
```

```
let v = 3 in
  let y = (let v = 2 * v in 4 * v)
  in y * v + z
```

And when it comes to the central purpose of all this story, variables, we rely on a powerful idea: static scopes!

Binding in the functional world

The idea and application of the SSA form stems from the imperative world.
But in the mean time, the functional world has been doing their own compilation!

The job is of course seemingly quite distinct. We are now fundamentally playing with:

- (Mutually recursive) functions
- let-binding constructs

```
fun f(x,y) = e
```

```
let v = 3 in
  let y = (let v = 2 * v in 4 * v)
  in y * v + z
```

And when it comes to the central purpose of all this story, variables, we rely on a powerful idea: static scopes!

Binding in the functional world

The idea and application of the SSA form stems from the imperative world.
But in the mean time, the functional world has been doing their own compilation!

The job is of course seemingly quite distinct. We are now fundamentally playing with:

- (Mutually recursive) functions
- let-binding constructs

```
fun f(x,y) = e
```

```
let v = 3 in
  let y = (let v = 2 * v in 4 * v)
  in y * v + z
```

And when it comes to the central purpose of all this story, variables, we rely on a powerful idea: static scopes!

Unicity of names is unnecessary, but can be enforced by alpha renaming

Binding in the functional world

```
let v = 3 in
  let y = (let v = 2 * v in 4 * v)
  in y * v + z
```

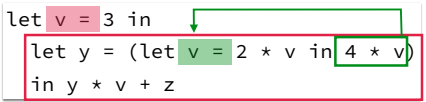
Binding in the functional world

```
let v = 3 in
  let y = (let v = 2 * v in 4 * v)
  in y * v + z
```

Binding shadows: enforces that each use-site maps to a unique def-site

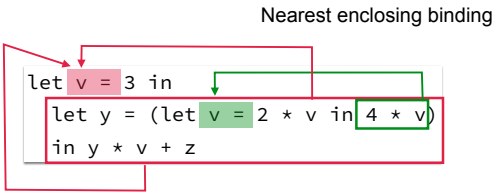
Binding in the functional world

Nearest enclosing binding



Binding shadows: enforces that each use-site maps to a unique def-site

Binding in the functional world



Binding shadows: enforces that each use-site maps to a unique def-site

Binding in the functional world

```
let v = 3 in
  let y = (let v = 2 * v in 4 * v)
  in y * v + z
```

Binding shadows: enforces that each use-site maps to a unique def-site

Well scoped: the only fresh variables are formal arguments

Binding in the functional world

```
fun f(z) =
  let v = 3 in
    let y = (let v = 2 * v in 4 * v)
    in y * v + z
```

Binding shadows: enforces that each use-site maps to a unique def-site

Well scoped: the only fresh variables are formal arguments

Binding in the functional world

```
fun f(z) =
  let v = 3 in
    let y = (let v = 2 * v in 4 * v)
    in y * v + z
```

Binding shadows: enforces that each use-site maps to a unique def-site

Well scoped: the only fresh variables are formal arguments

Each use of a variable is dominated by its unique definition!

Binding in the functional world

```
fun f(z) =
  let v = 3 in
    let y = (let v = 2 * v in 4 * v)
    in y * v + z
```

Binding shadows: enforces that each use-site maps to a unique def-site

Well scoped: the only fresh variables are formal arguments

Each use of a variable is dominated by its unique definition!

Uniqueness by scope and not by name: referential transparency!

Referential transparency: compositional equational reasoning!

Control flow in the functional world

CPS style: where an imperative compiler would carry on a return address, the functional one calls a continuation

```
let v = 3 in
  let y = (let v = 2 * v in 4 * v)
  in k(y * v + z)
```

Control flow in the functional world

CPS style: where an imperative compiler would carry on a return address, the functional one calls a continuation

```
let k = λx . 2 * x in
```

```
let v = 3 in
  let y = (let v = 2 * v in 4 * v)
  in k(y * v + z)
```

Control flow in the functional world

CPS style: where an imperative compiler would carry on a return address, the functional one calls a continuation

```
fun f(k) =
```

```
let v = 3 in
  let y = (let v = 2 * v in 4 * v)
  in k(y * v + z)
```

```
in let k = λx . 2 * x in f(k)
```

Control flow in the functional world

```
fun f(y,k) =
  let x = 4 in
  let k' = λz. k(z*x)
  if y > 0
    then let z = y * 2 in k'(z)
    else let z = 3 in k'(z)
```

Control flow in the functional world

CPS style: where an imperative compiler would carry on a return address, the functional one calls a continuation

```
fun f(y,k) =
  let x = 4 in
  let k' = λz. k(z*x)
  if y > 0
    then let z = y * 2 in k'(z)
    else let z = 3 in k'(z)
```

Control flow in the functional world

CPS style: where an imperative compiler would carry on a return address, the functional one calls a continuation

parameterized by a continuation

```
fun f(y,k) =
  let x = 4 in
  let k' = λz. k(z*x)
  if y > 0
    then let z = y * 2 in k'(z)
    else let z = 3 in k'(z)
```

Control flow in the functional world

CPS style: where an imperative compiler would carry on a return address, the functional one calls a continuation

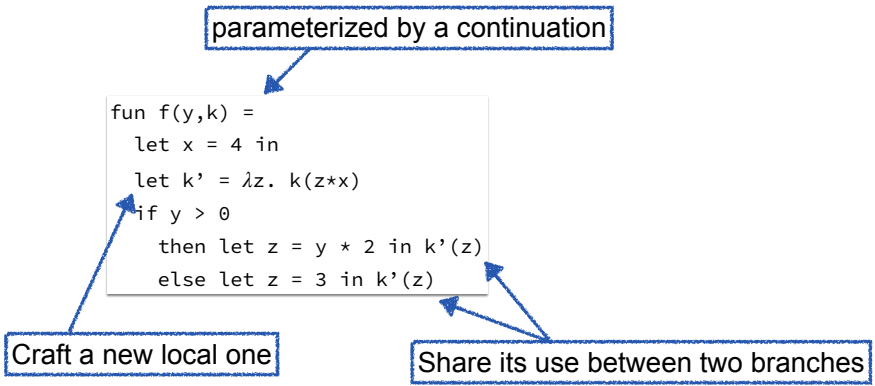
parameterized by a continuation

```
fun f(y,k) =
  let x = 4 in
  let k' = λz. k(z*x)
  if y > 0
    then let z = y * 2 in k'(z)
    else let z = 3 in k'(z)
```

Craft a new local one

Control flow in the functional world

CPS style: where an imperative compiler would carry on a return address, the functional one calls a continuation



Control flow in the functional world

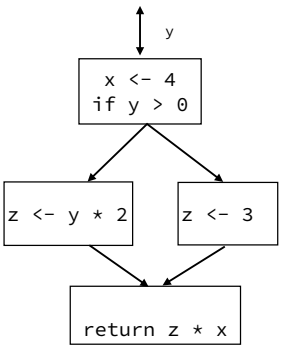
CPS style: where an imperative compiler would carry on a return address, the functional one calls a continuation

parameterized by a continuation

```
fun f(y,k) =
  let x = 4 in
  let k' = λz. k(z*x)
  if y > 0
    then let z = y * 2 in k'(z)
    else let z = 3 in k'(z)
```

Craft a new local one

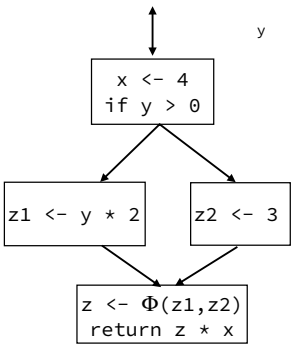
Share its use between two branches



Control flow in the functional world

CPS style: where an imperative compiler would carry on a return address, the functional one calls a continuation

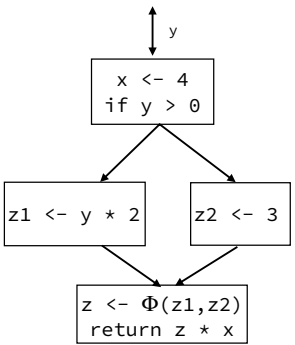
```
fun f(y,k) =
  let x = 4 in
  let k' = λz. k(z*x)
  if y > 0
    then let z = y * 2 in k'(z)
    else let z = 3 in k'(z)
```



Control flow in the functional world

CPS style: where an imperative compiler would carry on a return address,
the functional one calls a continuation

```
fun f(y,k) =
  let x = 4 in
  let k' = λz. k(z*x)
  if y > 0
    then let z1 = y * 2 in k'(z1)
    else let z2 = 3 in k'(z2)
```



continuation <-> phi-node

calls to the continuation <-> arguments to the phi-node

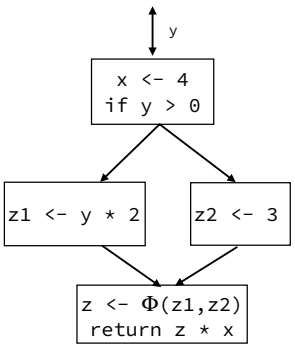
scope of a let-bind <-> dominance region of an assignment

Control flow in the functional world

CPS style: where an imperative compiler would carry on a return address,
the functional one calls a continuation

```
fun f(y,k) =
  let x = 4 in
  let k' = λz. k(z*x)
  if y > 0
    then let z1 = y * 2 in k'(z1)
    else let z2 = 3 in k'(z2)
```

z <- Φ(z1,z2)



continuation <-> phi-node

calls to the continuation <-> arguments to the phi-node

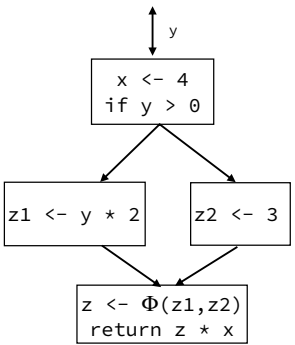
scope of a let-bind <-> dominance region of an assignment

Control flow in the functional world

CPS style: where an imperative compiler would carry on a return address,
the functional one calls a continuation

```
fun f(y,k) =
  let x = 4 in
  let k' = λz. k(z*x)
  if y > 0
    then let z1 = y * 2 in k'(z1)
    else let z2 = 3 in k'(z2)
```

```
z <- Φ(z1,z2)
```



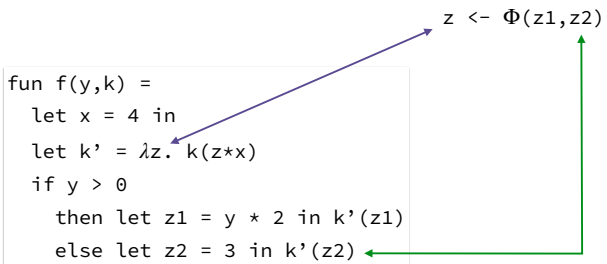
continuation <-> phi-node

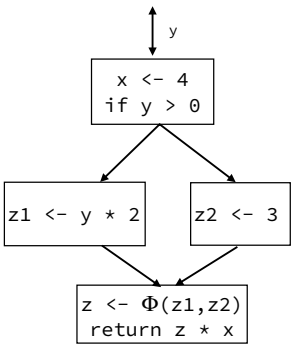
calls to the continuation <-> arguments to the phi-node

scope of a let-bind <-> dominance region of an assignment

Control flow in the functional world

CPS style: where an imperative compiler would carry on a return address,
the functional one calls a continuation





continuation <-> phi-node

calls to the continuation <-> arguments to the phi-node

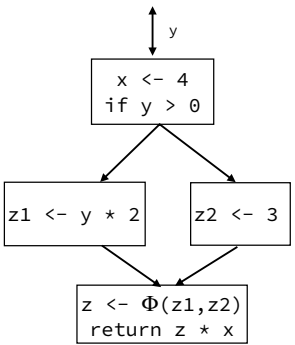
scope of a let-bind <-> dominance region of an assignment

Control flow in the functional world

CPS style: where an imperative compiler would carry on a return address,
the functional one calls a continuation

```
fun f(y,k) =
  let x = 4 in
  let k' = λz. k(z*x)
  if y > 0
    then let z1 = y * 2 in k'(z1)
    else let z2 = 3 in k'(z2)
```

z <- Φ(z1,z2)

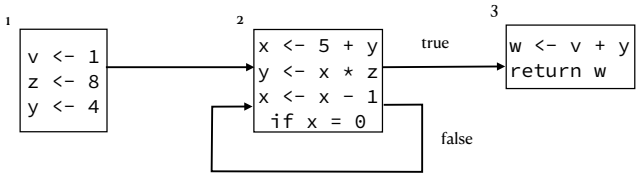


continuation <-> phi-node

calls to the continuation <-> arguments to the phi-node

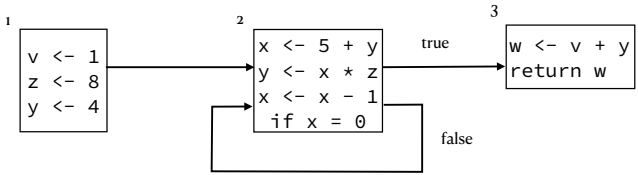
scope of a let-bind <-> dominance region of an assignment

SSA: functional construction



We are going to turn the CFG above in functional style, but via its functional representation

SSA: functional construction



We are going to turn the CFG above in functional style, but via its functional representation

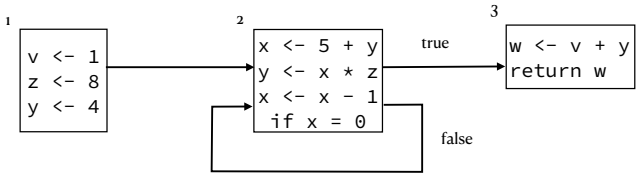
CPS-style

```
fun f(y,k) =
  let x = 4 in
  let k' = λz. k(z*x)
  if y > 0
    then let z = y * 2 in k'(z)
    else let z = 3 in k'(z)
```

(let-normal) direct-style (i.e. with tail recursive calls)

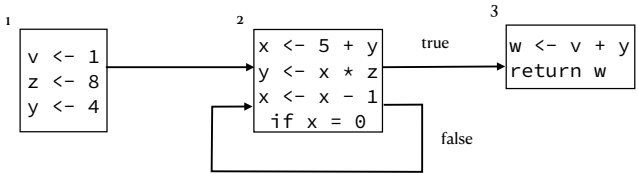
```
fun f(y) =
  let x = 4 in
  fun g(z) = z*x
  if y > 0
    then fun h1() = let z = y * 2 in g(z)
         in h1()
    else fun h2() = let z = 3 in g(z)
         in h2()
```

SSA: functional construction



We are going to turn the CFG above in functional style, but via its functional representation

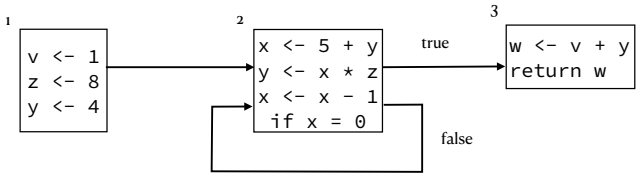
SSA: functional construction



We are going to turn the CFG above in functional style, but via its functional representation

Liveness analysis + one mutually recursive function per block

SSA: functional construction

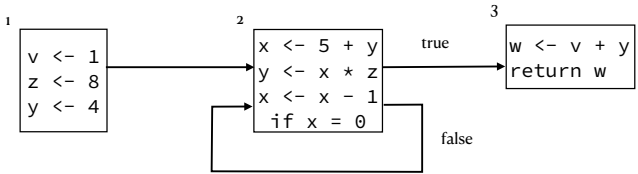


We are going to turn the CFG above in functional style, but via its functional representation

Liveness analysis + one mutually recursive function per block

```
fun f1()      = let v = 1, z = 8, y = 4
                in f2(v,z,y)
```

SSA: functional construction

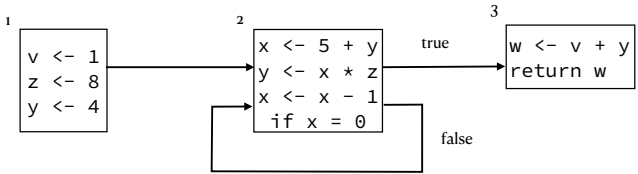


We are going to turn the CFG above in functional style, but via its functional representation

Liveness analysis + one mutually recursive function per block

```
fun f1()      = let v = 1, z = 8, y = 4
                in f2(v,z,y)
fun f2(v,z,y) = let x = 5 + y, y = x * z, x = x - 1
                in if x = 0
                   then f3(y,v)
                   else f2(v,z,y)
```

SSA: functional construction



We are going to turn the CFG above in functional style, but via its functional representation

Liveness analysis + one mutually recursive function per block

```
fun f1()      = let v = 1, z = 8, y = 4
                in f2(v,z,y)
fun f2(v,z,y) = let x = 5 + y, y = x * z, x = x - 1
                in if x = 0
                   then f3(y,v)
                   else f2(v,z,y)
fun f3(y,v)   = let w = v + y
                in w
```

SSA: functional construction

- All functions declarations are closed
- Unique definition-site per use is satisfied
- In a let binding `let x = e1 in e2`, the subterm `e2` corresponds to the successor in the control flow of the assignment to `x`

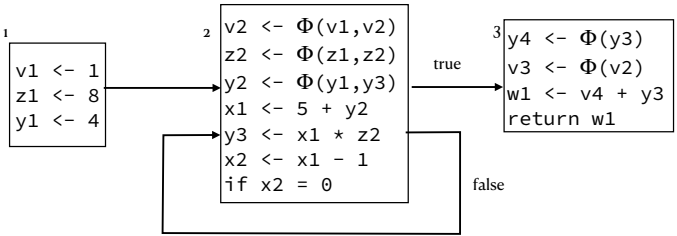
```
fun f1()      = let v = 1, z = 8, y = 4
                in f2(v,z,y)
fun f2(v,z,y) = let x = 5 + y, y = x * z, x = x - 1
                in if x = 0
                   then f3(y,v)
                   else f2(v,z,y)
fun f3(y,v)   = let w = v + y
                in w
```

SSA: functional construction

- All functions declarations are closed
- Unique definition-site per use is satisfied
- In a let binding `let x = e1 in e2`, the subterm `e2` corresponds to the successor in the control flow of the assignment to `x`

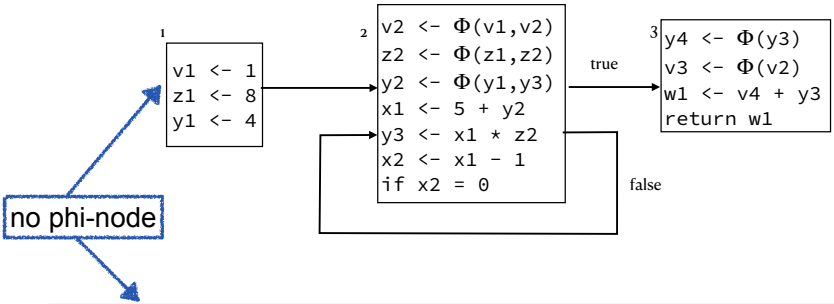
```
fun f1()         = let v1 = 1, z1 = 8, y1 = 4
                   in f2(v1,z1,y1)
fun f2(v2,z2,y2) = let x1 = 5 + y2, y3 = x1 * z2, x2 = x1 - 1
                   in if x2 = 0
                      then f3(y3,v2)
                      else f2(v2,z2,y3)
fun f3(y4,v3)    = let w1 = v4 + y3
                   in w1
```

SSA: functional construction



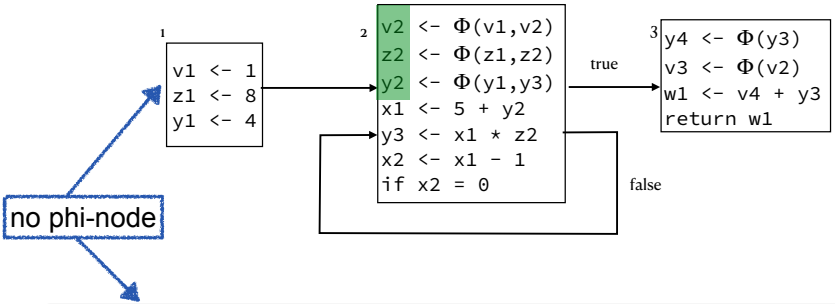

```
fun f1()          = let v1 = 1, z1 = 8, y1 = 4
                    in f2(v1,z1,y1)
fun f2(v2,z2,y2) = let x1 = 5 + y2, y3 = x1 * z2, x2 = x1 - 1
                    in if x2 = 0
                       then f3(y3,v2)
                       else f2(v2,z2,y3)
fun f3(y4,v3)     = let w1 = v4 + y3
                    in w1
```

SSA: functional construction



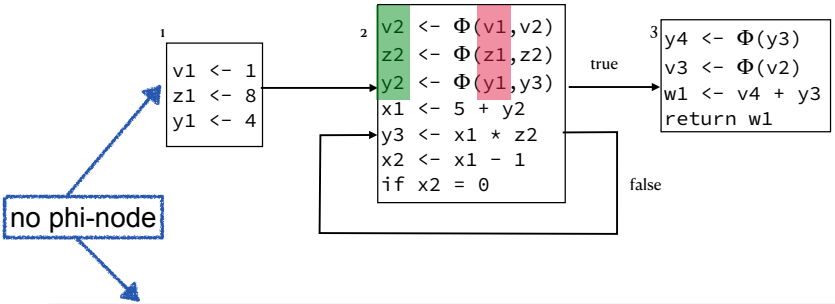

```
fun f1()          = let v1 = 1, z1 = 8, y1 = 4
                    in f2(v1,z1,y1)
fun f2(v2,z2,y2) = let x1 = 5 + y2, y3 = x1 * z2, x2 = x1 - 1
                    in if x2 = 0
                       then f3(y3,v2)
                       else f2(v2,z2,y3)
fun f3(y4,v3)     = let w1 = v4 + y3
                    in w1
```

SSA: functional construction



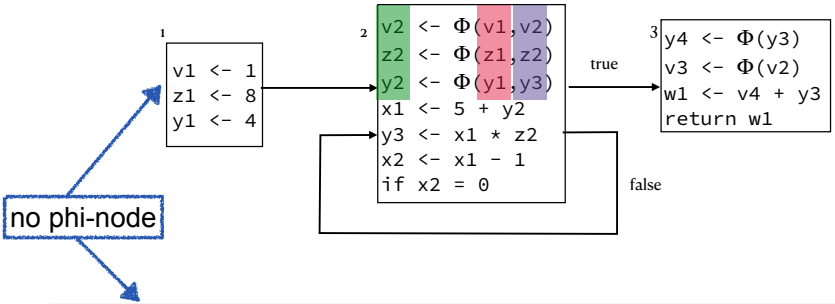

```
fun f1()          = let v1 = 1, z1 = 8, y1 = 4
                    in f2(v1,z1,y1)
fun f2(v2,z2,y2)  = let x1 = 5 + y2, y3 = x1 * z2, x2 = x1 - 1
                    in if x2 = 0
                       then f3(y3,v2)
                       else f2(v2,z2,y3)
fun f3(y4,v3)     = let w1 = v4 + y3
                    in w1
```

SSA: functional construction



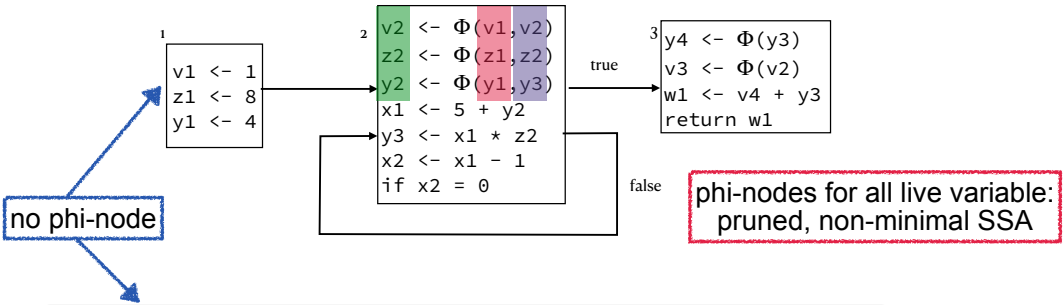

```
fun f1()          = let v1 = 1, z1 = 8, y1 = 4
                    in f2(v1,z1,y1)
fun f2(v2,z2,y2)  = let x1 = 5 + y2, y3 = x1 * z2, x2 = x1 - 1
                    in if x2 = 0
                       then f3(y3,v2)
                       else f2(v2,z2,y3)
fun f3(y4,v3)     = let w1 = v4 + y3
                    in w1
```

SSA: functional construction




```
fun f1()          = let v1 = 1, z1 = 8, y1 = 4
                    in f2(v1,z1,y1)
fun f2(v2,z2,y2)  = let x1 = 5 + y2, y3 = x1 * z2, x2 = x1 - 1
                    in if x2 = 0
                       then f3(y3,v2)
                       else f2(v2,z2,y3)
fun f3(y4,v3)     = let w1 = v4 + y3
                    in w1
```

SSA: functional construction




```
fun f1()          = let v1 = 1, z1 = 8, y1 = 4
                    in f2(v1,z1,y1)
fun f2(v2,z2,y2)  = let x1 = 5 + y2, y3 = x1 * z2, x2 = x1 - 1
                    in if x2 = 0
                       then f3(y3,v2)
                       else f2(v2,z2,y3)
fun f3(y4,v3)     = let w1 = v4 + y3
                    in w1
```

Block sinking

```
fun f1()          = let v1 = 1, z1 = 8, y1 = 4
                    in f2(v1,z1,y1)
fun f2(v2,z2,y2) = let x1 = 5 + y2, y3 = x1 * z2, x2 = x1 - 1
                    in if x2 = 0
                       then f3(y3,v2)
                       else f2(v2,z2,y3)
fun f3(y4,v3)     = let w1 = v4 + y3
                    in w1
```

Block sinking

```
fun f1()          = let v1 = 1, z1 = 8, y1 = 4
                    in f2(v1,z1,y1)
fun f2(v2,z2,y2) = let x1 = 5 + y2, y3 = x1 * z2, x2 = x1 - 1
                    in if x2 = 0
                       then f3(y3,v2)
                       else f2(v2,z2,y3)
fun f3(y4,v3)     = let w1 = v4 + y3
                    in w1
```

```
fun f1() =
   let v = 1, z = 8, y = 4
   in fun f2(v,z,y) =
      let x = 5 + y, y = x * z, x = x - 1
      in if x = 0
         then fun f3(y,v) = let w = y + v in w
              in f3(y,v)
         else f2(v,z,y)
   in f2(v,z,y)
in f1()
```

Block sinking

```
fun f1()          = let v1 = 1, z1 = 8, y1 = 4
                    in f2(v1,z1,y1)
fun f2(v2,z2,y2)  = let x1 = 5 + y2, y3 = x1 * z2, x2 = x1 - 1
                    in if x2 = 0
                       then f3(y3,v2)
                       else f2(v2,z2,y3)
fun f3(y4,v3)     = let w1 = v4 + y3
                    in w1
```

```
fun f1() =
   let v = 1, z = 8, y = 4
   in fun f2(v,z,y) =
      let x = 5 + y, y = x * z, x = x - 1
      in if x = 0
         then fun f3(y,v) = let w = y + v in w
              in f3(y,v)
         else f2(v,z,y)
   in f2(v,z,y)
in f1()
```

Block sinking

```
fun f1()          = let v1 = 1, z1 = 8, y1 = 4
                    in f2(v1,z1,y1)
fun f2(v2,z2,y2)  = let x1 = 5 + y2, y3 = x1 * z2, x2 = x1 - 1
                    in if x2 = 0
                       then f3(y3,v2)
                       else f2(v2,z2,y3)
fun f3(y4,v3)     = let w1 = v4 + y3
                    in w1
```

Essentially: based on the DT of the call graph

The dominance relationship becomes apparent in the scoping

```
fun f1() =
   let v = 1, z = 8, y = 4
   in fun f2(v,z,y) =
      let x = 5 + y, y = x * z, x = x - 1
      in if x = 0
         then fun f3(y,v) = let w = y + v in w
              in f3(y,v)
         else f2(v,z,y)
   in f2(v,z,y)
in f1()
```

Parameter dropping

```
fun f1() =
   let v = 1, z = 8, y = 4
   in fun f2(v,z,y) =
      let x = 5 + y, y = x * z, x = x - 1
      in if x = 0
         then fun f3(y,v) = let w = y + v in w
              in f3(y,v)
         else f2(v,z,y)
   in f2(v,z,y)
in f1()
```

We drop the formal parameters that can be statically ruled out as semantically irrelevant

Parameter dropping

```
fun f1() =
   let v = 1, z = 8, y = 4
   in fun f2(v,z,y) =
      let x = 5 + y, y = x * z, x = x - 1
      in if x = 0
         then fun f3(y,v) = let w = y + v in w
              in f3(y,v)
         else f2(v,z,y)
   in f2(v,z,y)
in f1()
```

We drop the formal parameters that can be statically ruled out as semantically irrelevant

Parameter dropping

```
fun f1() =
   let v = 1, z = 8, y = 4
   in fun f2(v,z,y) =
      let x = 5 + y, y = x * z, x = x - 1
      in if x = 0
         then fun f3() = let w = y + v in w
              in f3()
         else f2(v,z,y)
   in f2(v,z,y)
in f1()
```

We drop the formal parameters that can be statically ruled out as semantically irrelevant

Parameter dropping

```
fun f1() =
   let v = 1, z = 8, y = 4
   in fun f2(v,z,y) =
      let x = 5 + y, y = x * z, x = x - 1
      in if x = 0
         then fun f3() = let w = y + v in w
              in f3()
         else f2(v,z,y)
   in f2(v,z,y)
in f1()
```

We drop the formal parameters that can be statically ruled out as semantically irrelevant

Parameter dropping

```
fun f1() =
   let v = 1, z = 8, y = 4
   in fun f2(y) =
      let x = 5 + y, y = x * z, x = x - 1
      in if x = 0
         then fun f3() = let w = y + v in w
              in f3()
         else f2(y)
   in f2(y)
in f1()
```

We drop the formal parameters that can be statically ruled out as semantically irrelevant

Minimal SSA form

```
fun f1() =
   let v = 1, z = 8, y = 4
   in fun f2(y) =
      let x = 5 + y, y = x * z, x = x - 1
      in if x = 0
         then fun f3() = let w = y + v in w
              in f3()
         else f2(y)
   in f2(y)
in f1()
```