PROGRAM SLICING

PROGRAM ANALYSIS AND OPTIMIZATION – DCC888

Fernando Magno Quintão Pereira

fernando@dcc.ufmg.br

The material in these slides have been taken from a number of papers, including Frank Tip's "A Survey of Program Slicing Techniques", 1994.

Definitions♤

- A *program slice* is the part of a program that (potentially) affects the values computed at some point of interest.
  - The point of interest is usually called a *slice criterion*. In our case, a slice criterion will be a program variable.
- The task of finding the program slice with respect to variable v is called *program slicing*.
- According to Weiser♧, a slice S with respect to variable v is a reduced, executable program, obtained from an original program P by removing statements that do not influence the computation of v.

♤: A Survey of Program Slicing Techniques, Frank Tip, 1994

♧: Program slices: formal, psychological, and practical investigations of an automatic program abstraction method, 1979

Definitions♤

- A *program slice* is the part of a program that (potentially) affects the values computed at some point of interest.
  - The point of interest is usually called a *slice criterion*. In our case, a slice criterion will be a program variable.
- The task of finding the program slice with respect to variable v is called *program slicing*.
- According to **Weiser**♧, a slice S with respect to variable v is a reduced, executable program, obtained from an original program P by removing statements that do not influence the computation of v.

What are program slices good for?

♤: A Survey of Program Slicing Techniques, Frank Tip, 1994

♧: Program slices: formal, psychological, and practical investigations of an automatic program abstraction method, 1979

Applications of Program Slicing: Debugging

The main use of program slicing is as an aid to debugging tools. Slicing lets the developer abstract away details of the buggy program which are not relevant to detect inconsistent behavior.

```
int divMod (int a, int b, int* m) {
   int quot = 0;
   while (a > b) {
      a -= b;
      quot++;
   }
   *m = a;
   return quot;
}
```

```
int divMod (int a, int b, int* m) {
   int quot = 0;
   while (a > b) {
      a -= b;
      quot++;
   }
   *m = a;
   return quot;
}
```

For instance, if we want to find a problem with the statement *m = a, in the function above, we do not need to consider the operations marked in gray on the right. The resulting program is simpler.

Other Applications: Information Flow Analyses

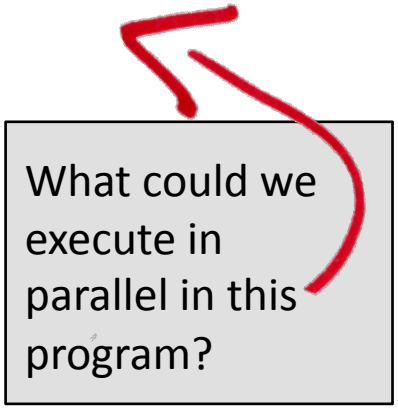
- Program slicing is very useful, because it lays at the foundations of most implementations of information flow analyses.
  - Information disclosure analyses: can sensitive data reach a public output channel?
    - Compute a **backward** slice from the output channel, and see if this slice touches operations that produce sensitive data.
  - Tainted flow analyses: can public input data reach a sensitive operation?
    - Compute a **forward** slice from the public input channel, and see if this slice touches a sensitive operation.

One more cool application: parallelization

- Which statements in a program can be executed in parallel?
  - Program slicing helps us to answer this question.

```
void min_max(int* a, int N, int* min, int* max) {
  int min_aux = MAX_INT;
  int max_aux = MIN_INT;
  for (int i = 0; i < N; i++) {
    if (a[i] < min_aux) {
      min_aux = a[i];
    }
    if (a[i] > max_aux) {
      max_aux = a[i];
    }
  }
  *min = min_aux;
  *max = max_aux;
}
```

```
void min_max(int* a, int N, int* min, int* max) {
  int min_aux = MAX_INT;
  int max_aux = MIN_INT;
  for (int i = 0; i < N; i++) {
    if (a[i] < min_aux) {
      min_aux = a[i];
    }
    if (a[i] > max_aux) {
      max_aux = a[i];
    }
  }
  *min = min_aux;
  *max = max_aux;
}
```



  - Independent slices can be executed in parallel.

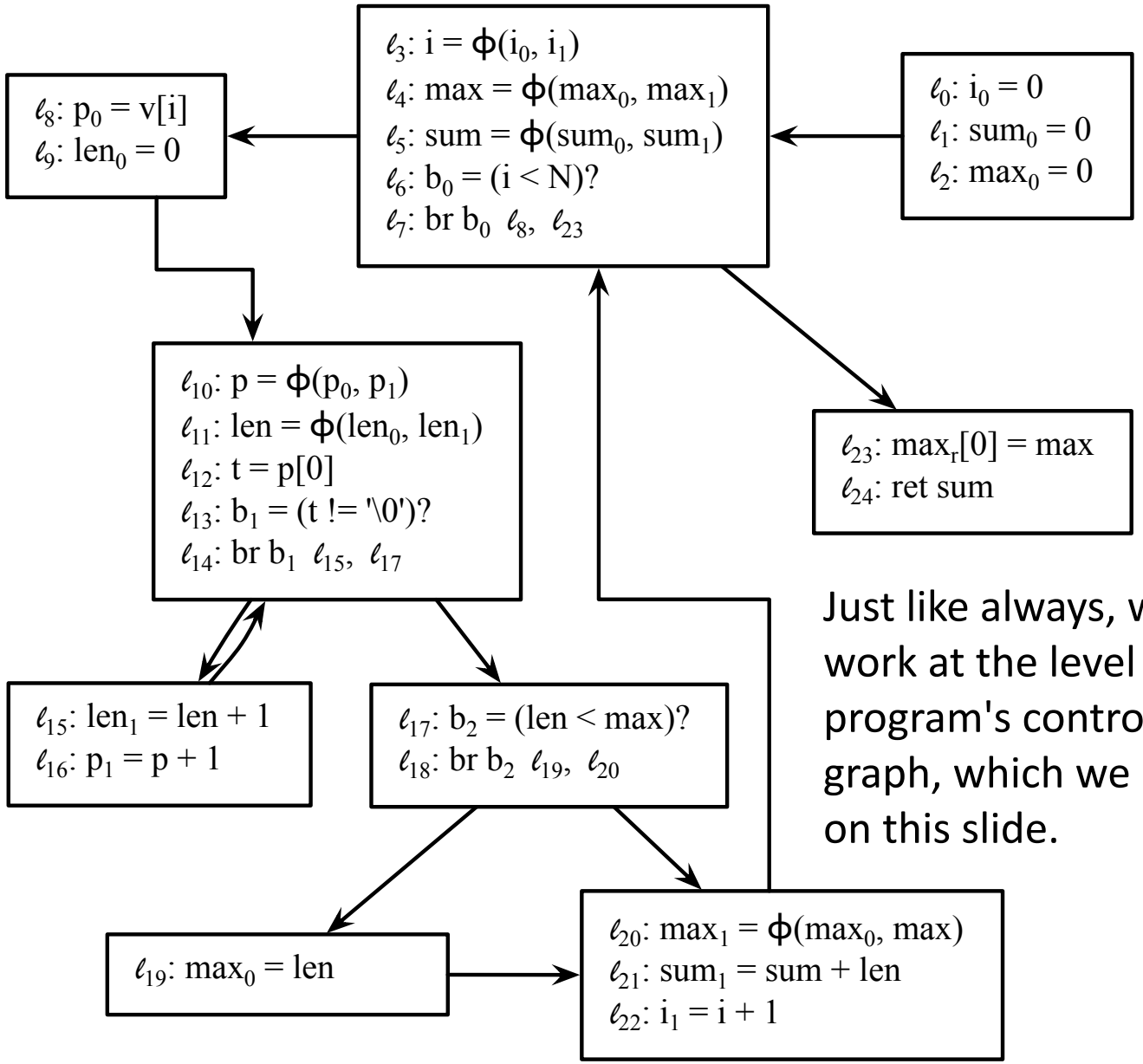
Warm up Example

```
int sstrlen(char** v, int N, int* maxr) {
  int sum = 0;
  int max = 0;
  int i;
  char* p;
  for (i = 0; i < N; i++) {
    int len = 0;
    for (p = v[i]; *p != '\0'; p++) {
      len++;
    }
    if (len > max) {
      max = len;
    }
    sum += len;
  }
  *maxr = max;
  return sum;
}
```

We shall illustrate how to compute slices in this program. What the program does is immaterial.

Does the computation of **maxr** depend on the computation of **sum** in any way?

Control Flow Graph



Just like always, we will work at the level of the program's control flow graph, which we can see on this slide.

PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DCC-UFMG

Dependences

- The backward slice with respect to a variable v contains all the instructions that v *depends* on, directly or indirectly.
- We recognize two types of dependences:
  - *Data dependence*: variable v depends on variable u in program P if P contains an instruction that reads u and defines v, e.g., v = u + 1;
  - *Control dependences*: we say that variable v depends on a variable p, if p is used as the predicate of a branch that determines the value that v is assigned, e.g., if p then v = 0 else v = 1.
- Transitivity: if v depends on u, which depends on t, then v depends on t.

(Control) Dependences

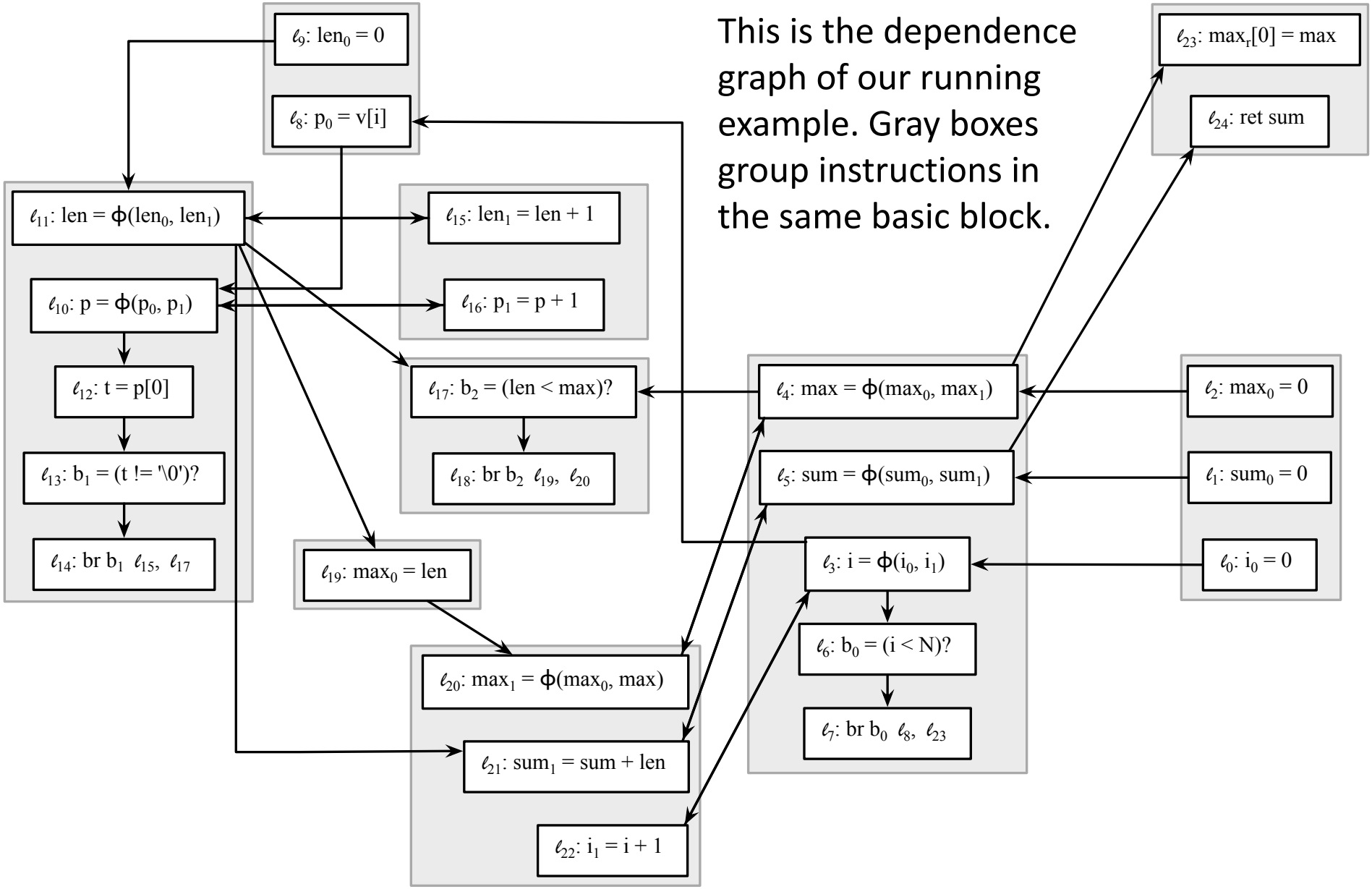
- The backward slice with respect to a variable v contains all the instructions that v *depends* on, directly or indirectly.
- We recognize two types of dependences:
  - *Data dependence*: variable v depends on variable u in program P if P contains an instruction that reads u and defines v, e.g., v = u + 1;
  - *Control dependences*: we say that variable v depends on a variable p, if p is used as the predicate of a branch that determines the value that v is assigned, e.g., if p then v = 0 else v = 1.
- Transitivity: if v depends on u, which depends on t, then v depends on t.

Data Dependence Graph

- We can represent the dependences in a program as a directed graph, which we shall call the *Dependence Graph*.
  - The dependence graph has a vertex for every program variable.
  - The dependence graph has an edge from variable u to variable v if, and only if, v depends on u.

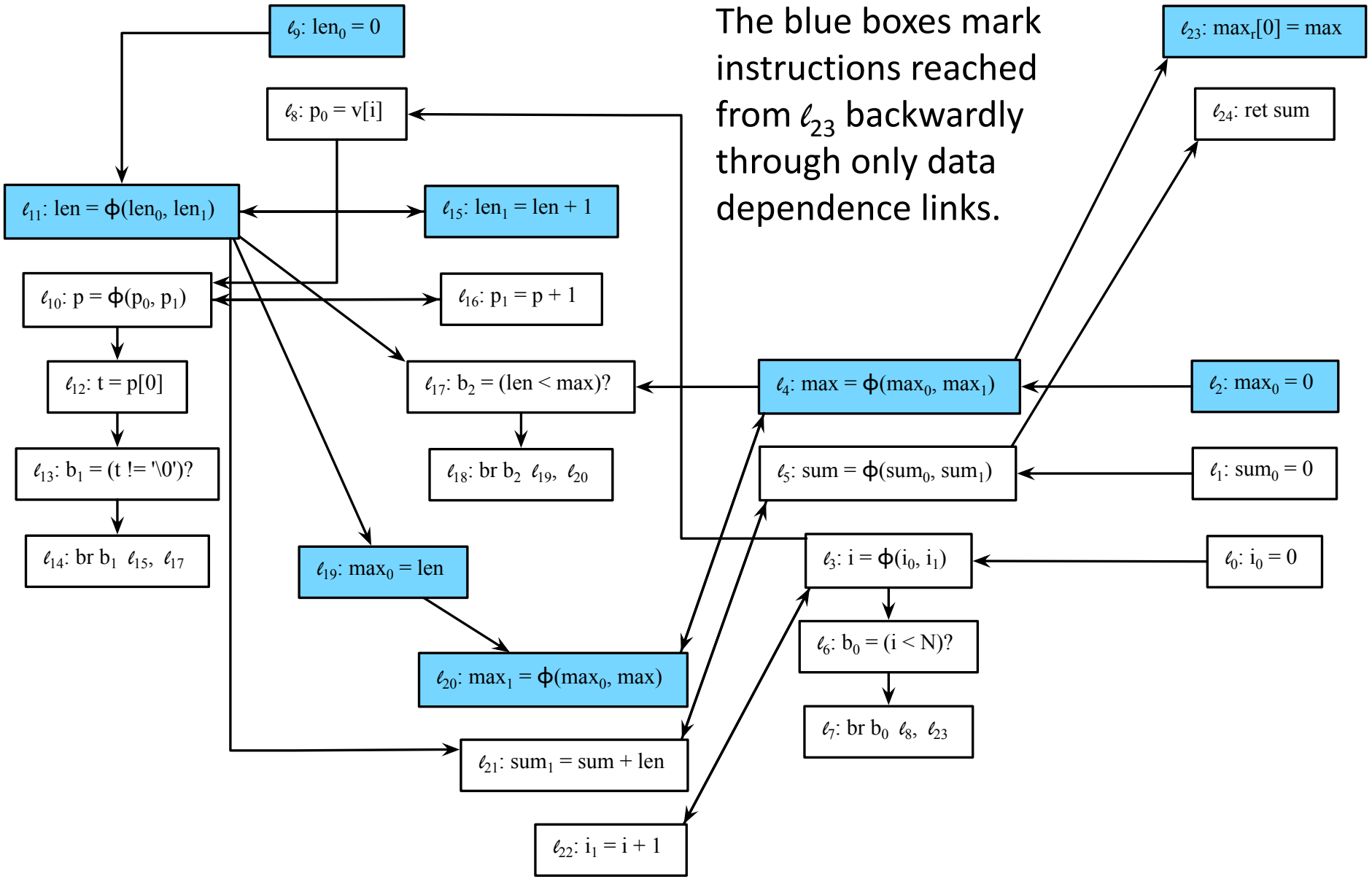
1) Can the dependence graph contain a cycle?
2) Can this graph contain a quadratic number of edges? In other words, can it be dense?
3) Can it have self-loops?
4) What about double edges? Can we have multiple edges between two nodes?

Data Dependence Graph



This is the dependence graph of our running example. Gray boxes group instructions in the same basic block.

Backward slice of ℓ_{23}: $max_r[0] = max$ (Data Deps Only)



The blue boxes mark instructions reached from ℓ_{23} backwardly through only data dependence links.

Only Data Dependences are not Enough

```
int sstrlen(char** v, int N, int* maxr) {
  int sum = 0;
  int max = 0;
  int i;
  char* p;
  for (i = 0; i < N; i++) {
    int len = 0;
    for (p = v[i]; *p != '\0'; p++) {
      len++;
    }
    if (len > max) {
      max = len;
    }
    sum += len;
  }
  *maxr = max;
  return sum;
}
```

If we consider only data dependences, we will miss all the control flow in the program.

Is the slice that we obtain with only data dependences executable?

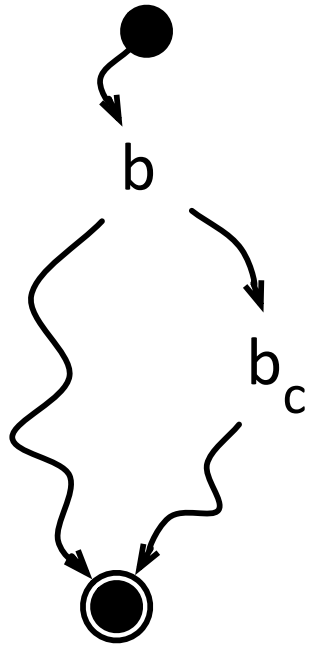
Only Data Dependences are not Enough

```
int sstrlen(char** v, int N, int* maxr) {
  int sum = 0;
  int max = 0;
  int i;
  char* p;
  for (i = 0; i < N; i++) {
    int len = 0;
    for (p = v[i]; *p != '\0'; p++) {
      len++;
    }
    if (len > max) {
      max = len;
    }
    sum += len;
  }
  *maxr = max;
  return sum;
}
```

The slice that we obtain with data dependences is executable, because all the uses of a variable v have been defined before v is assigned a value. However, it does not compute correctly the variable of interest, because we are missing all the control flow in the program.

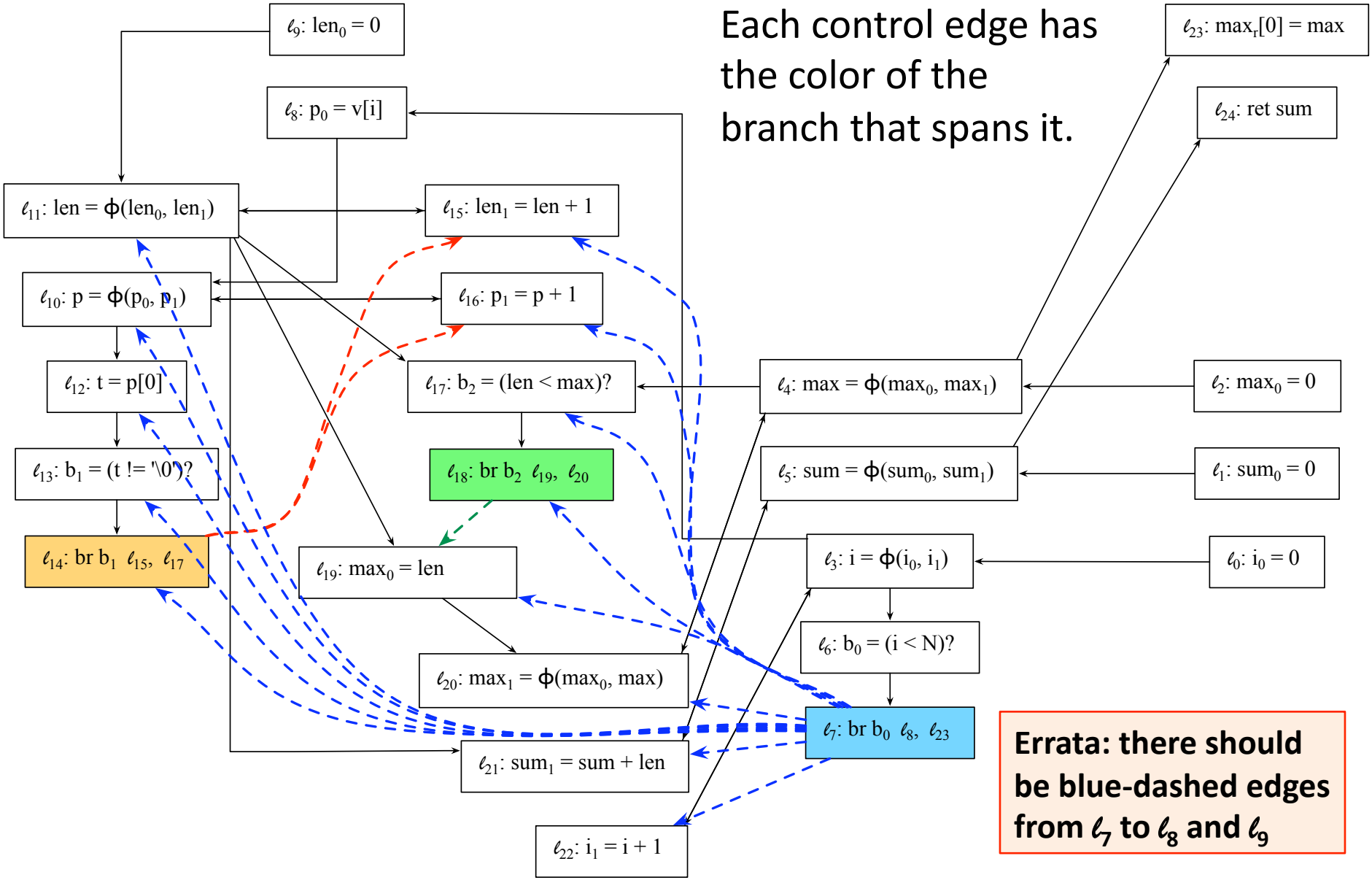
Control Dependences

- Control dependences are usually defined in terms of post-dominance.
- Recap: a node b in the CFG is post-dominated by a node b_p if all paths from b to END$^{\heartsuit}$ pass through b_p.
- A node b_c is control dependent on a node b if:
  - there exists a path P from b to b_c, such that b_c post-dominates every node in this path, excluding b and b_c themselves.
  - b is not post-dominated by b_c.



$^{\heartsuit}$: We assume that every control flow graph has two special nodes: START which dominates every other node, and is dominated by none, and END, which post-dominates every other node, and is post-dominated by none.

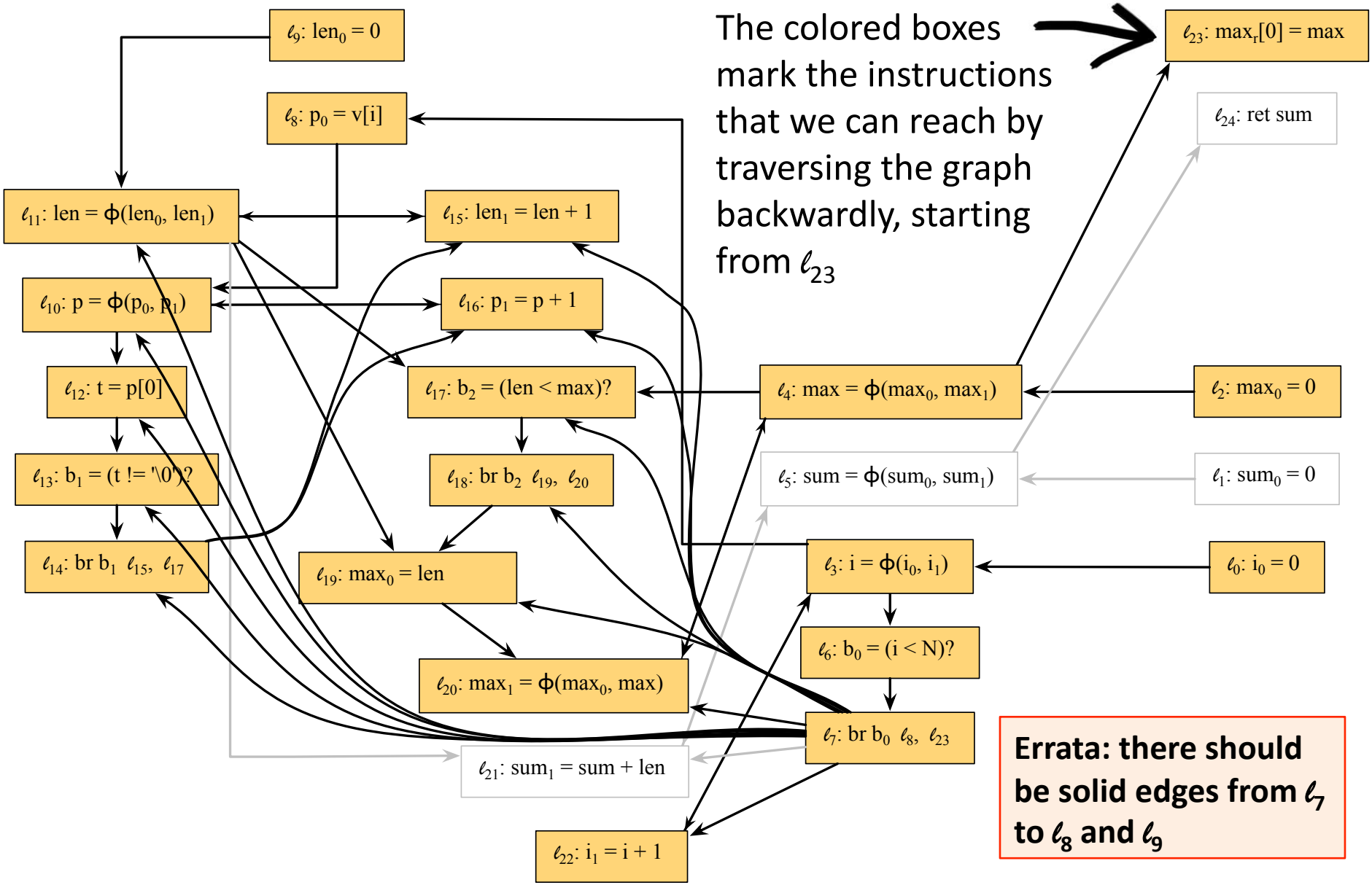
Control Dependences



Each control edge has the color of the branch that spans it.

Errata: there should be blue-dashed edges from ℓ_7 to ℓ_8 and ℓ_9

Backward slice of ℓ_{23}: $max_r[0] = max$ (Full Deps)



The colored boxes mark the instructions that we can reach by traversing the graph backwardly, starting from ℓ_{23}

Errata: there should be solid edges from ℓ_7 to ℓ_8 and ℓ_9

Executable Slice

```
int sstrlen(char** v, int N, int* maxr) {
  int sum = 0;
  int max = 0;
  int i;
  char* p;
  for (i = 0; i < N; i++) {
    int len = 0;
    for (p = v[i]; *p != '\0'; p++) {
      len++;
    }
    if (len > max) {
      max = len;
    }
    sum += len;
  }
  *maxr = max;
  return sum;
}
```

This program is almost executable: given that we have removed its return statement, we would have to convert the function to void, to get a valid C program. If we do this, then this code correctly computes the value of `maxr`.

Fernando Magno Quintão Pereira

fernando@dcc.ufmg.br

lac.dcc.ufmg.br

Compilers Laboratory

Department of Computer Science
Universidade Federal de Minas Gerais
Federal University of Minas Gerais, Brazil

Dynamic and Forward Slices

fernando@dcc.ufmg.br

DCC 888

Quick Detour: Dynamic Slicing

- So far we have been talking only about *static* slicing.
  - Our slices hold for any possible input of the program.
- There exist also the notion of a *dynamic slice* of a program.
  - Given a statement *s*, located at a program point ℓ, a dynamic slice contains all the statements that have influenced the execution of *s* for a particular input of the program.

1) What are dynamic slices good for?
2) What are the advantages of dynamic slices over static slices?
3) What are the advantages of static over dynamic slices?

Dynamic Slicing: Example

```
int sstrlen(char** v, int N, int* maxr) {
  int sum = 0;
  int max = 0;
  int i;
  char* p;
  for (i = 0; i < N; i++) {
    int len = 0;
    for (p = v[i]; *p != '\0'; p++) {
      len++;
    }
    if (len > max) {
      max = len;
    }
    sum += len;
  }
  *maxr = max;
  return sum;
}
```

What is the dynamic slice of `*maxr = max`, given an execution in which `N = 1`, and `v[0] == '\0'`?

Dynamic Slicing: Example

```
int sstrlen(char** v, int N, int* maxr) {
  int sum = 0;
  int max = 0;
  int i;
  char* p;
  for (i = 0; i < N; i++) {
    int len = 0;
    for (p = v[i]; *p != '\0'; p++) {
      len++;
    }
    if (len > max) {
      max = len;
    }
    sum += len;
  }
  *maxr = max;
  return sum;
}
```

Remember:
`N == 1`
`v[0] == '\0'`

What is different from the static slice that we had seen before?

CFGGrind

- There are several tools that produce dynamic slices. For an example, check: https://github.com/rimsa/CFGgrind
  - Based on Valgrind

Second Quick Detour: Forward Slicing

- In this class, we shall deal only with backward slices:
  - Which statements *affect* the slicing criterion?
- There exist also the notion of a forward slice:
  - Which statements *are affected* by the slicing criterion?

Second Quick Detour: Forward Slicing

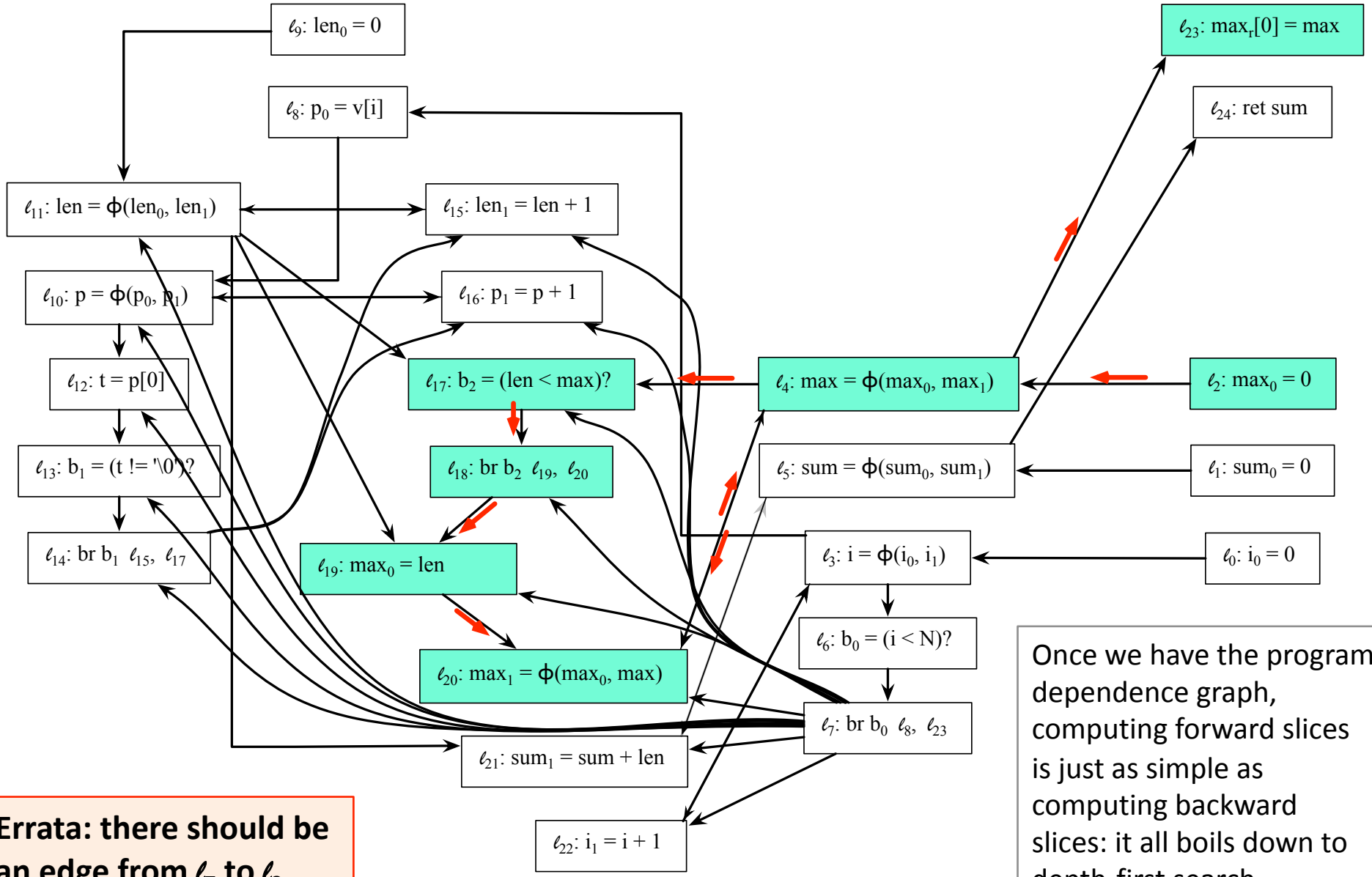
- In this class, we shall deal only with backward slices:
  - Which statements *affect* the slicing criterion?
- There exist also the notion of a forward slice:
  - Which statements *are affected* by the slicing criterion?

```
int sstrlen(char** v, int N, int* maxr) {
  int sum = 0;
  int max = 0;
  int i;
  char* p;
  for (i = 0; i < N; i++) {
    int len = 0;
    for (p = v[i]; *p != '\0'; p++) {
      len++;
    }
    if (len > max) {
      max = len;
    }
    sum += len;
  }
  *maxr = max;
  return sum;
}
```

1) What is a forward slice good for?

2) What is the forward slice with respect to "int max = 0" in the program on the left?

Computing a Forward Slice



Errata: there should be an edge from ℓ_7 to ℓ_9

Once we have the program dependence graph, computing forward slices is just as simple as computing backward slices: it all boils down to depth-first search.

Forward Slicing: Example

```
int sstrlen(char** v, int N, int* maxr) {
  int sum = 0;
  int max = 0;
  int i;
  char* p;
  for (i = 0; i < N; i++) {
    int len = 0;
    for (p = v[i]; *p != '\0'; p++) {
      len++;
    }
    if (len > max) {
      max = len;
    }
    sum += len;
  }
  *maxr = max;
  return sum;
}
```

Is this slice executable?

How to build a solution to the tainted flow analysis (with only two states: tainted and clean) using forward and backward slices?

Forward Slicing might be Non-executable

```
int sstrlen(char** v, int N, int* maxr) {
  int sum = 0;
  int max = 0;
  int i;
  char* p;
  for (i = 0; i < N; i++) {
    int len = 0;
    for (p = v[i]; *p != '\0'; p++) {
      len++;
    }
    if (len > max) {
      max = len;
    }
    sum += len;
  }
  *maxr = max;
  return sum;
}
```

Is this slice executable?

Not really: it makes reference to a certain variable **len**, which is not defined in the slice itself. This is a problem with purely forward slices: they might yield code that is non-executable.

Fernando Magno Quintão Pereira

fernando@dcc.ufmg.br

lac.dcc.ufmg.br

Compilers Laboratory

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSIDADE FEDERAL DE MINAS GERAIS
FEDERAL UNIVERSITY OF MINAS GERAIS, BRAZIL

BUILDING SLICES IN HAMMOCK GRAPHS

fernando@dcc.ufmg.br

DCC 888

Hammock Graphs

- Let G be a control flow graph for program P. A hammock H is an induced subgraph of G with a distinguished node V in H called the entry node and a distinguished node W not in H called the exit node such that:
  - All edges from (G - H) to H go to V.
  - All edges from H to (G - H) go to W.

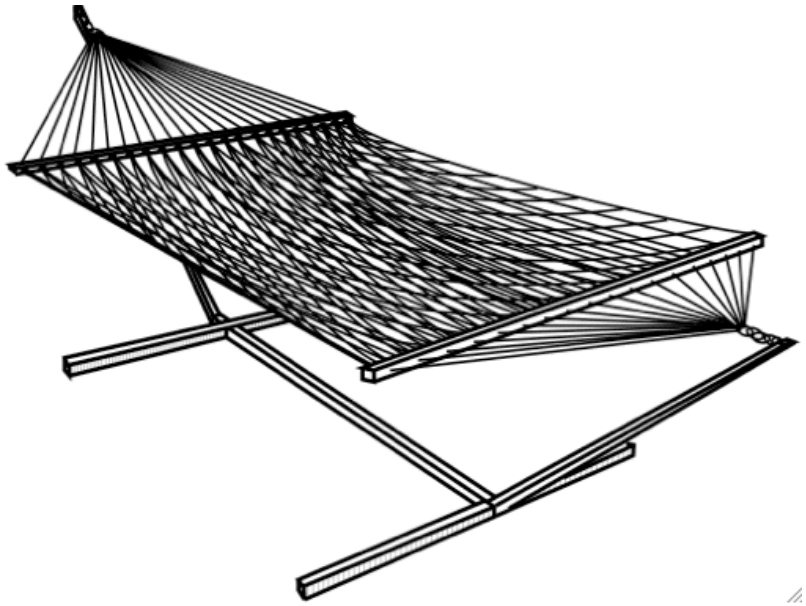
Why do hammock graphs have this name?

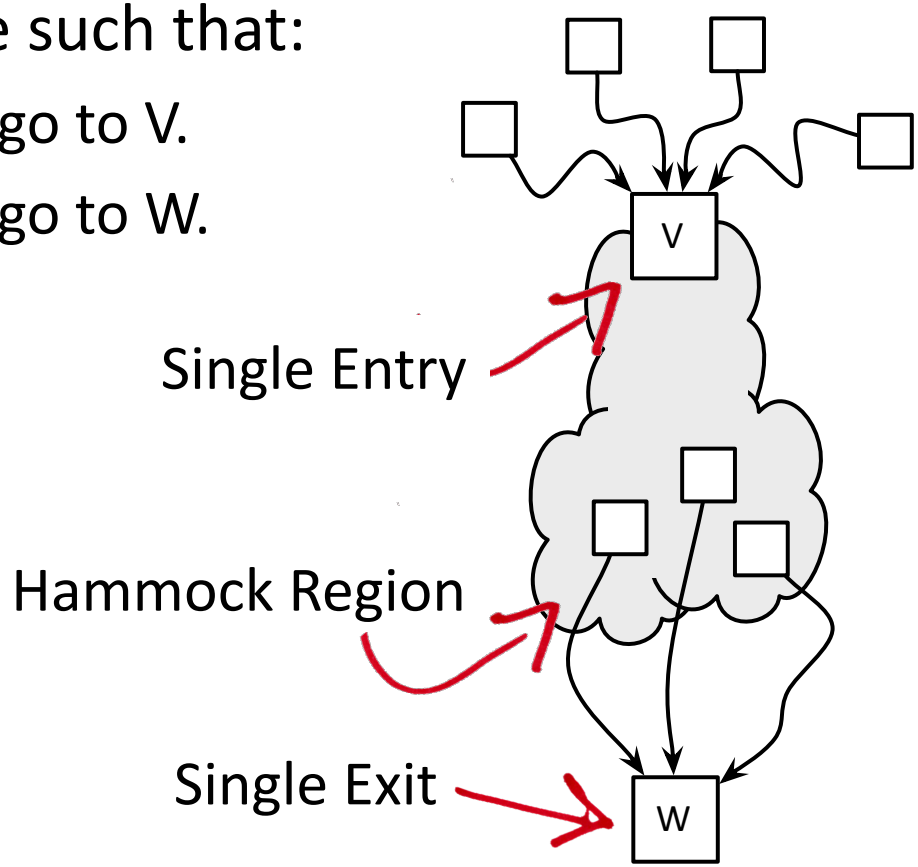
Theorem 1 (*Ferrante et al.*): Let G be a control flow graph for program P. Let H be a hammock of G. If X is in H and Y is in (G – H), then Y is not control dependent on X[◇].

◇: For a proof, see The program dependence graph and its use in optimization, 1987

Hammock Graphs

- Let G be a control flow graph for program P. A hammock H is an induced subgraph of G with a distinguished node V in H called the entry node and a distinguished node W not in H called the exit node such that:
  - All edges from (G - H) to H go to V.
  - All edges from H to (G - H) go to W.



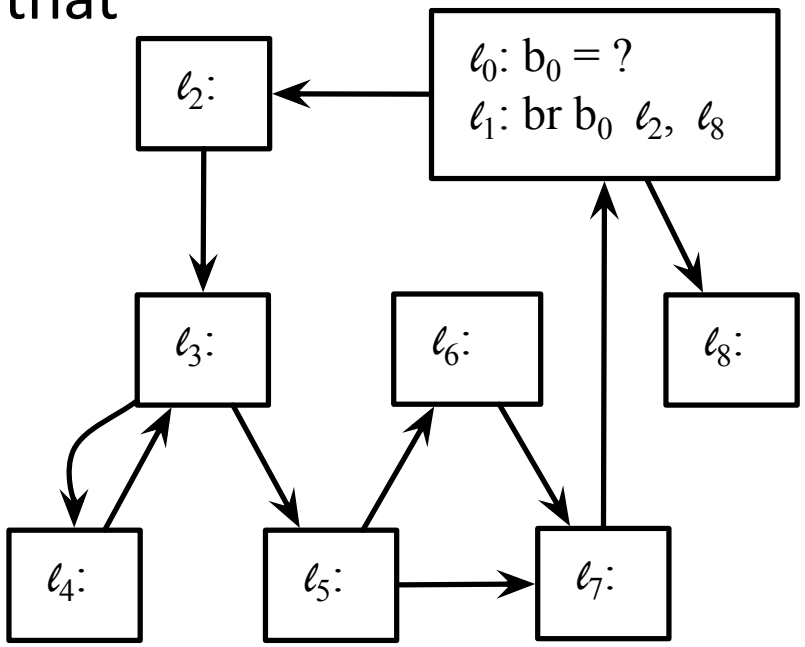


Finding Hammock Regions♤

- The hammock region of a branch gives us the set of statements that are control dependent on this branch.
- Let p be a predicate that governs the outcome of a branch, e.g., ℓ: brz p ℓ_x, ℓ_y. Let ℓ_p be the label that post-dominates ℓ. The hammock region of p is the set of statements in paths from ℓ to ℓ_p that do not contain ℓ_p.

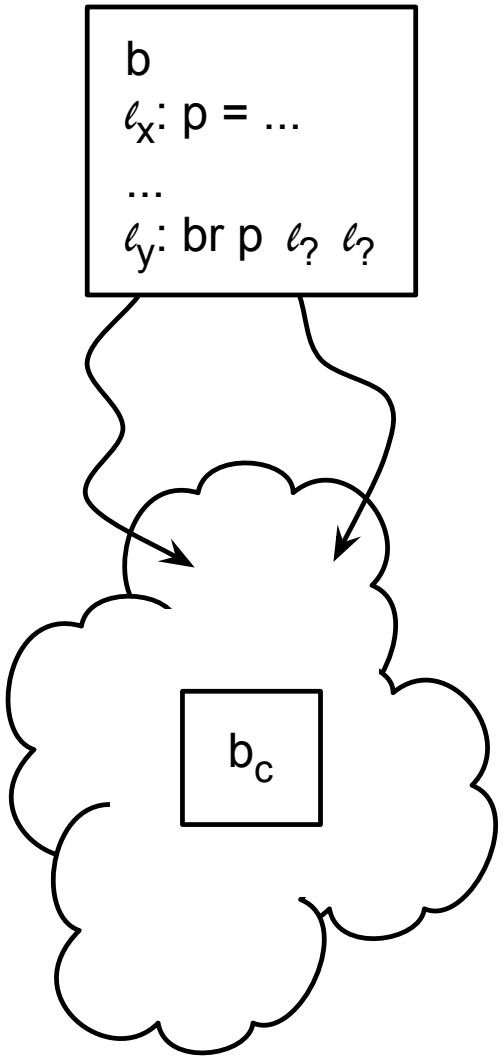
1) If we enumerate the hammock regions, we can easily find the control deps. Why?

2) What is the hammock region of b_0?



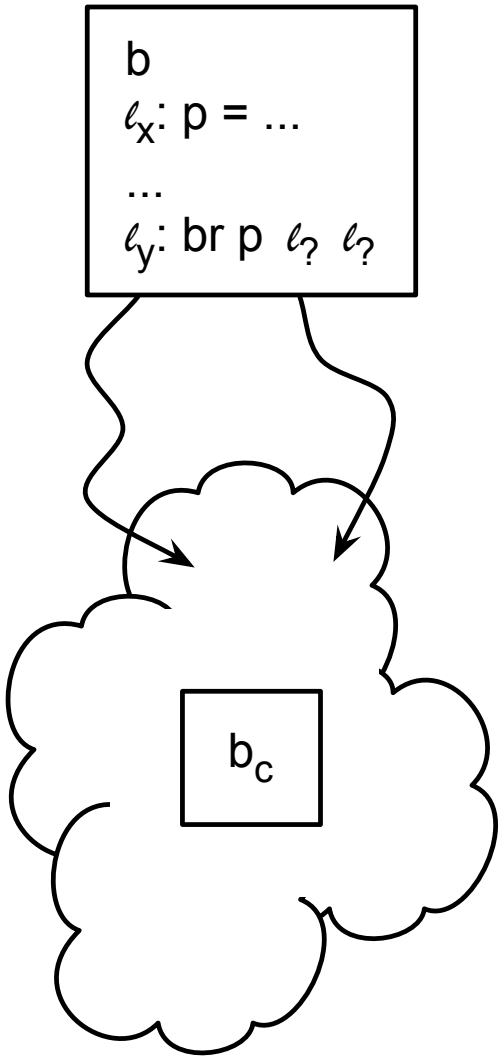
♤: Naturally, we assume that we are talking about hammock graphs only.

Some Nomenclature



The predicate p controls the branch at ℓ_y

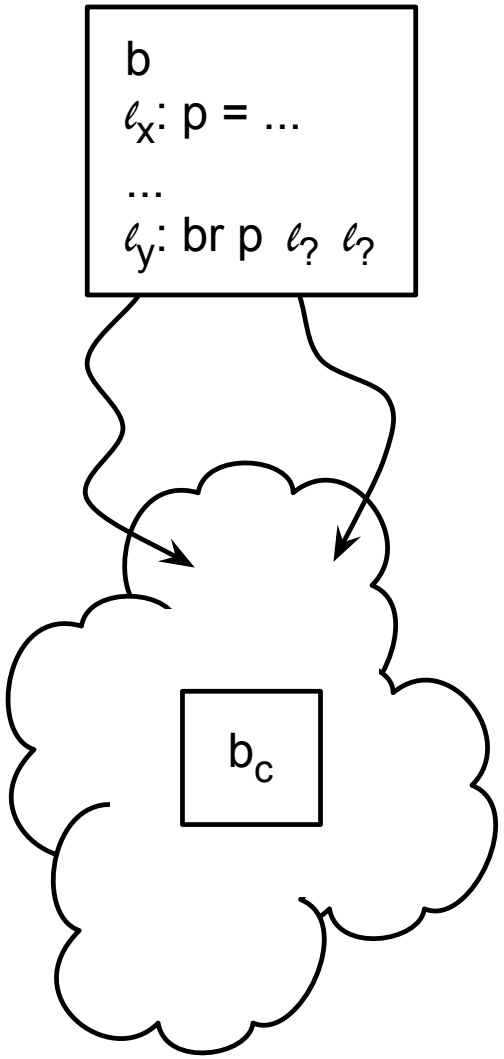
Some Nomenclature



The predicate p controls the branch at ℓ_y

Influence region of p

Some Nomenclature



The predicate p controls the branch at ℓ_y

Influence region of p

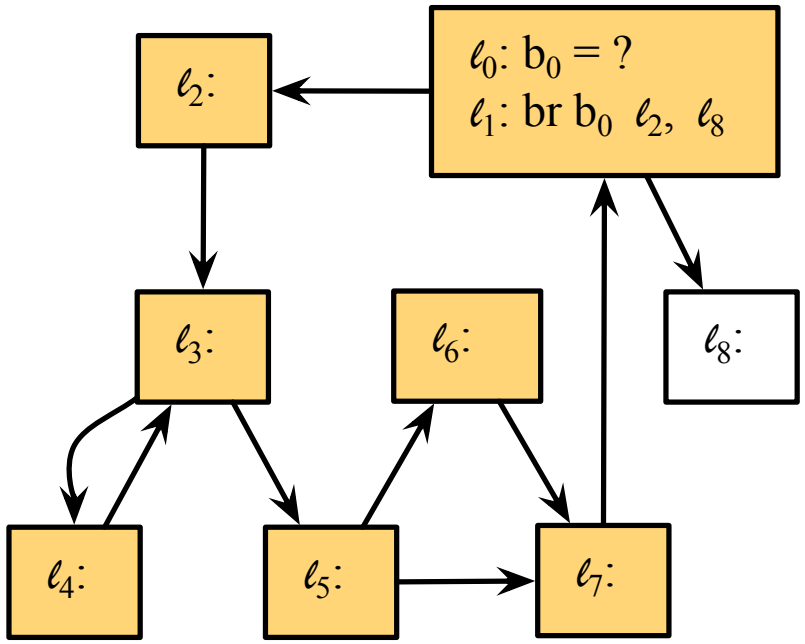
b_c is controled by p

Finding Hammock Regions

- Let p be a predicate that governs the outcome of a branch, e.g., ℓ: brz p ℓ_x, ℓ_y. Let ℓ_p be the label that post-dominates ℓ. The hammock region of p is the set of statements in paths from ℓ to ℓ_p that do not contain ℓ_p.

The colored nodes mark the hammock region of b_0. We find this region by traversing the control flow graph, and stopping the search whenever we hit ℓ_8, the post-dominator of ℓ_1, the branch where b_0 is used.

If we were to find every hammock region in the CFG through this method, what would be the final complexity?

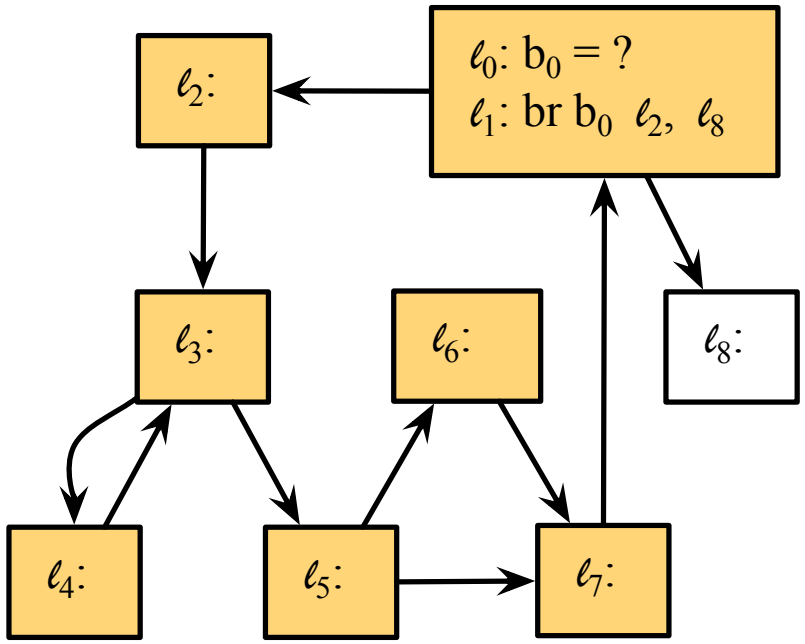


Finding Hammock Regions

- Let p be a predicate that governs the outcome of a branch, e.g., ℓ: brz p ℓ_x, ℓ_y. Let ℓ_p be the label that post-dominates ℓ. The hammock region of p is the set of statements in paths from ℓ to ℓ_p that do not contain ℓ_p.

This method is rather costly, because we end up visiting the same node several times, as it might be part of different hammock regions. In the end, we will inspect edges O(V×E) times. That is, for each node in the graph (O(V)), we may traverse the entire graph (O(E)).

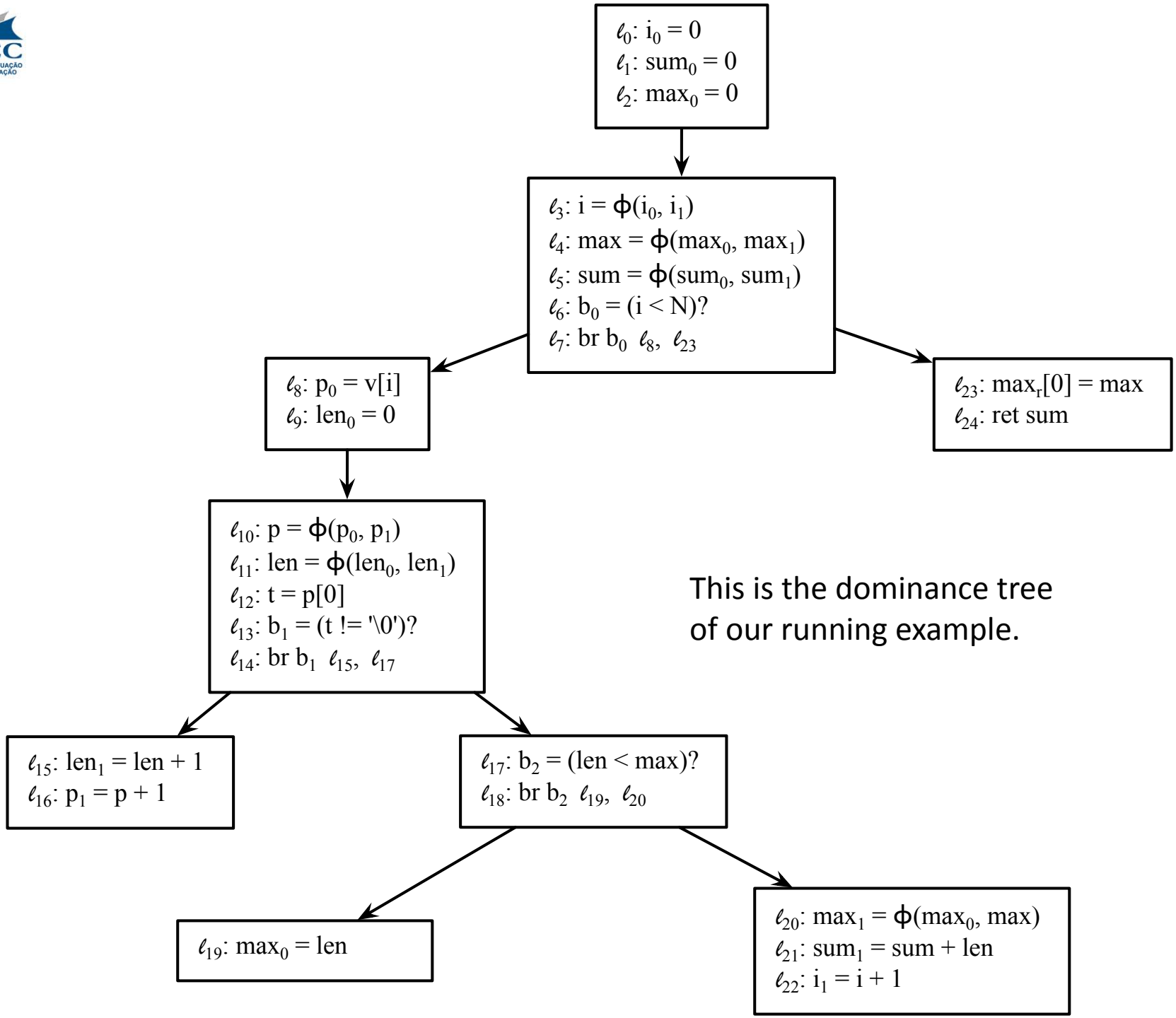
Fortunately, we do not need to mark every hammock region to find out control dependences.



Dominance Tree

- We will rely on the hammock structure to find control dependences in the program.
- To this end, we will traverse the program's dominance (or dominator) tree.
- Quick recap:
  - Node v dominates node u if, and only if, every path from START to u goes across v.
  - Node v is the immediate dominator of u if, and only if, v dominates u, and, for any node v' that also dominates u, either v' dominates v, or v = v'.
- The immediate dominator of a node is unique, and this relation is total and does not admit cycles. Thus, it determines a tree out of the nodes in the program's CFG.

Dominance Tree



This is the dominance tree of our running example.

Dominance Tree and Hammock Regions

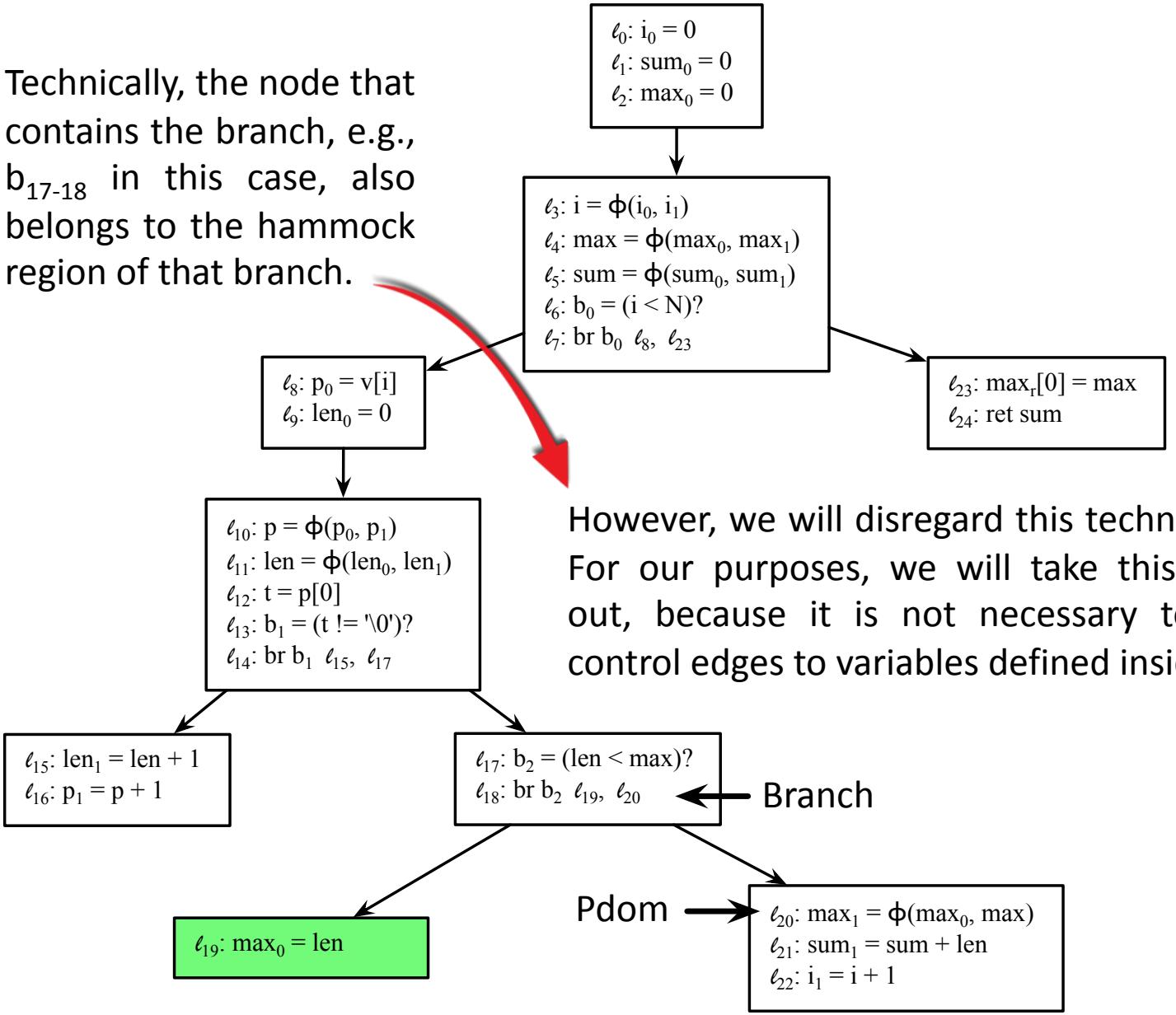
- As we will prove later on, all the nodes that belong into a hammock region of a predicate p lay under the node that contains p in the dominance tree.
- We shall capitalize on this fact, to add control flow edges to our graph.

We will use hammock graphs, plus dominance trees, to find a quick way to determine control dependences.

And we will try to add as few as possible control dependence edges to our graph.

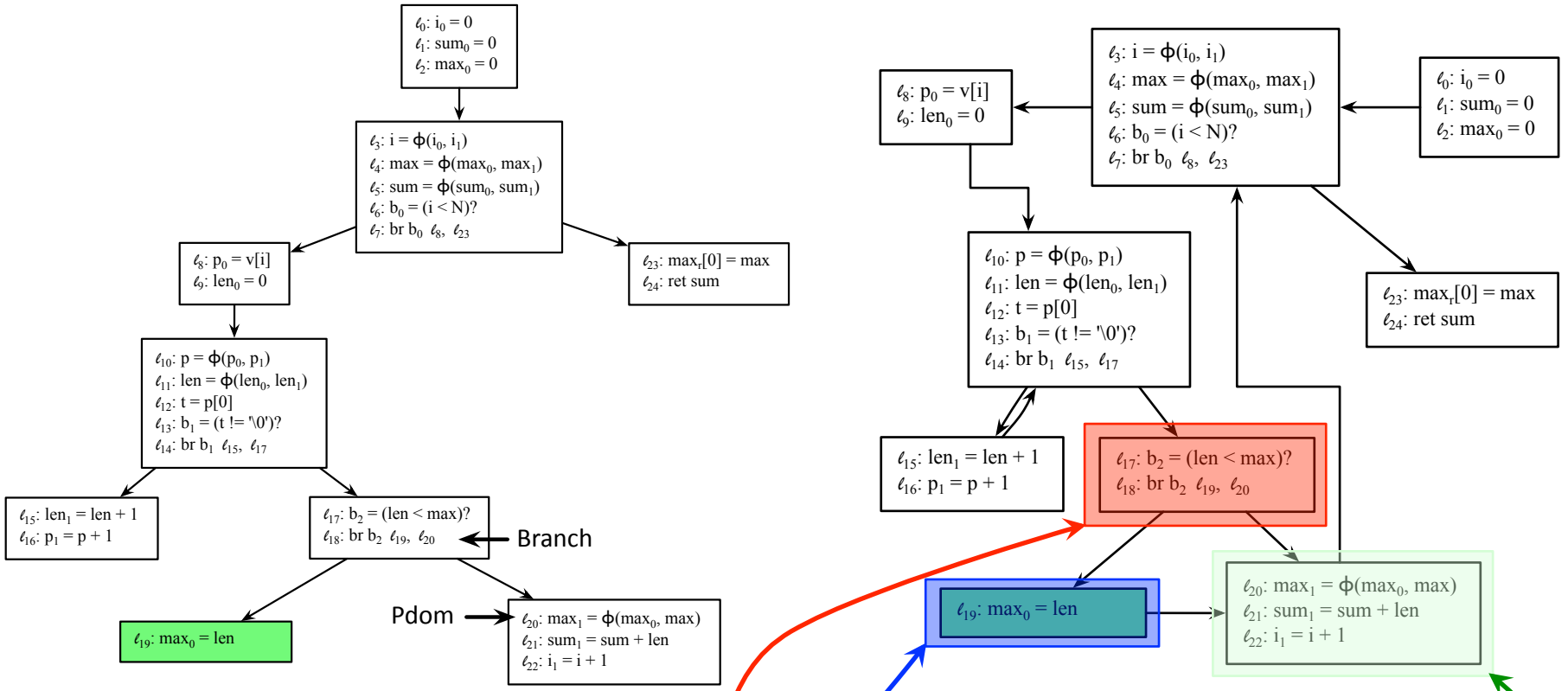
Hammock Region of br b_2 ℓ_{19}, ℓ_{20}

Technically, the node that contains the branch, e.g., $b_{17\text{-}18}$ in this case, also belongs to the hammock region of that branch.



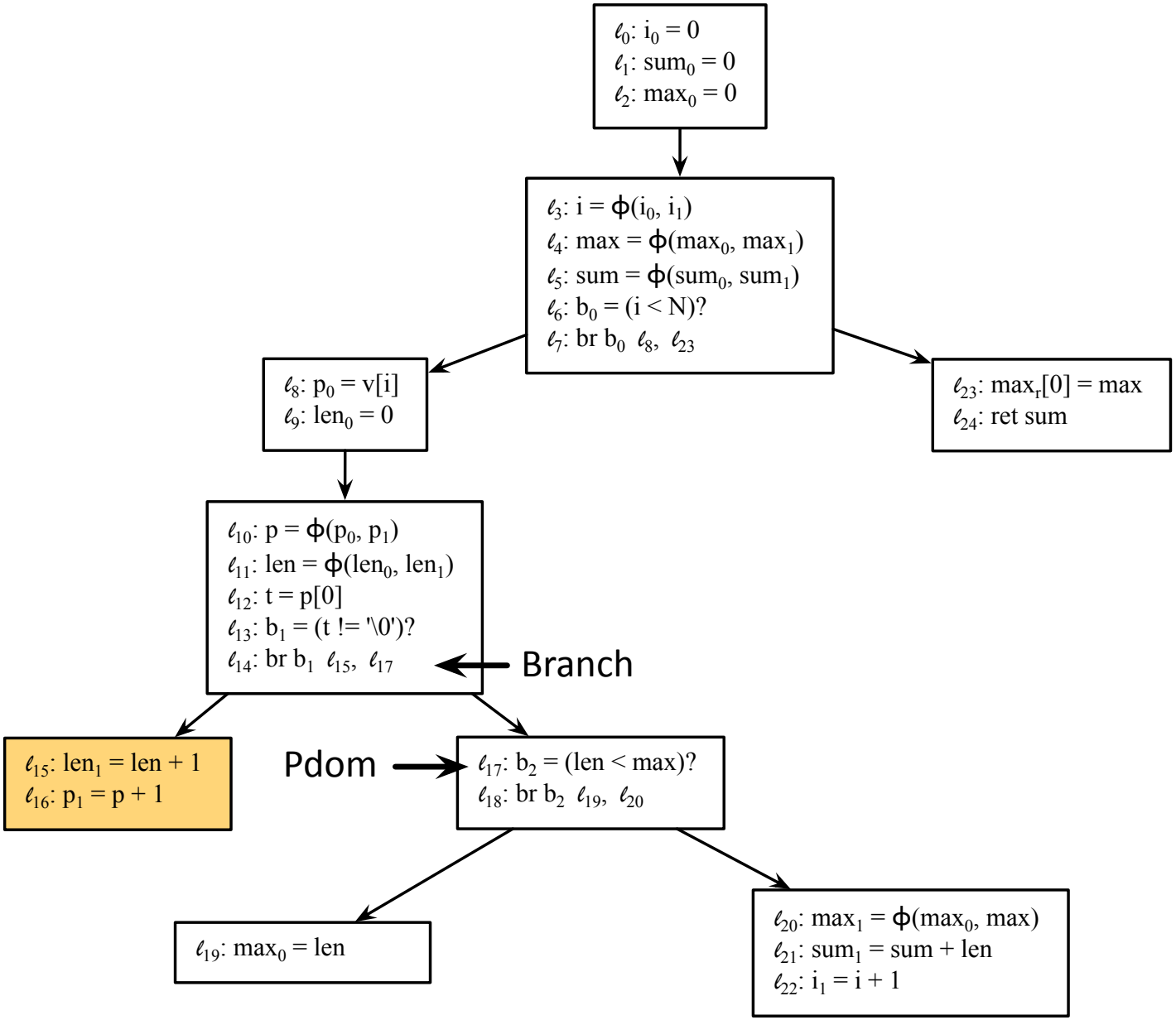
However, we will disregard this technicality. For our purposes, we will take this node out, because it is not necessary to add control edges to variables defined inside it.

Hammock Region of br b_2 ℓ_{19}, ℓ_{20}

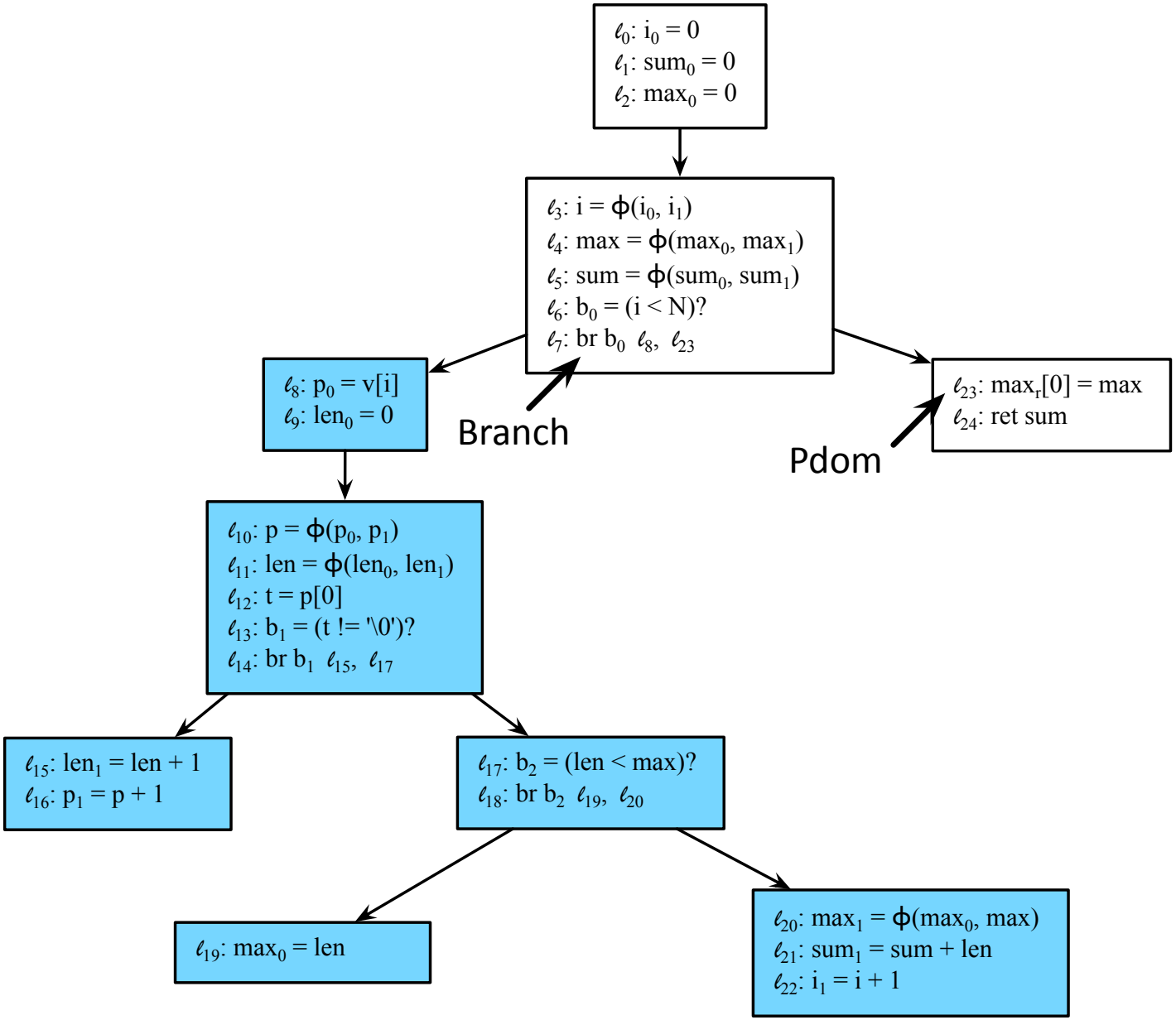


This block controls this **one**, because once we visit **it**, **its** branch decides if we visit **this** block or not. On the other hand, **this** block does not control **this** other one, because we must visit **it** independently on the result of the **branch**.

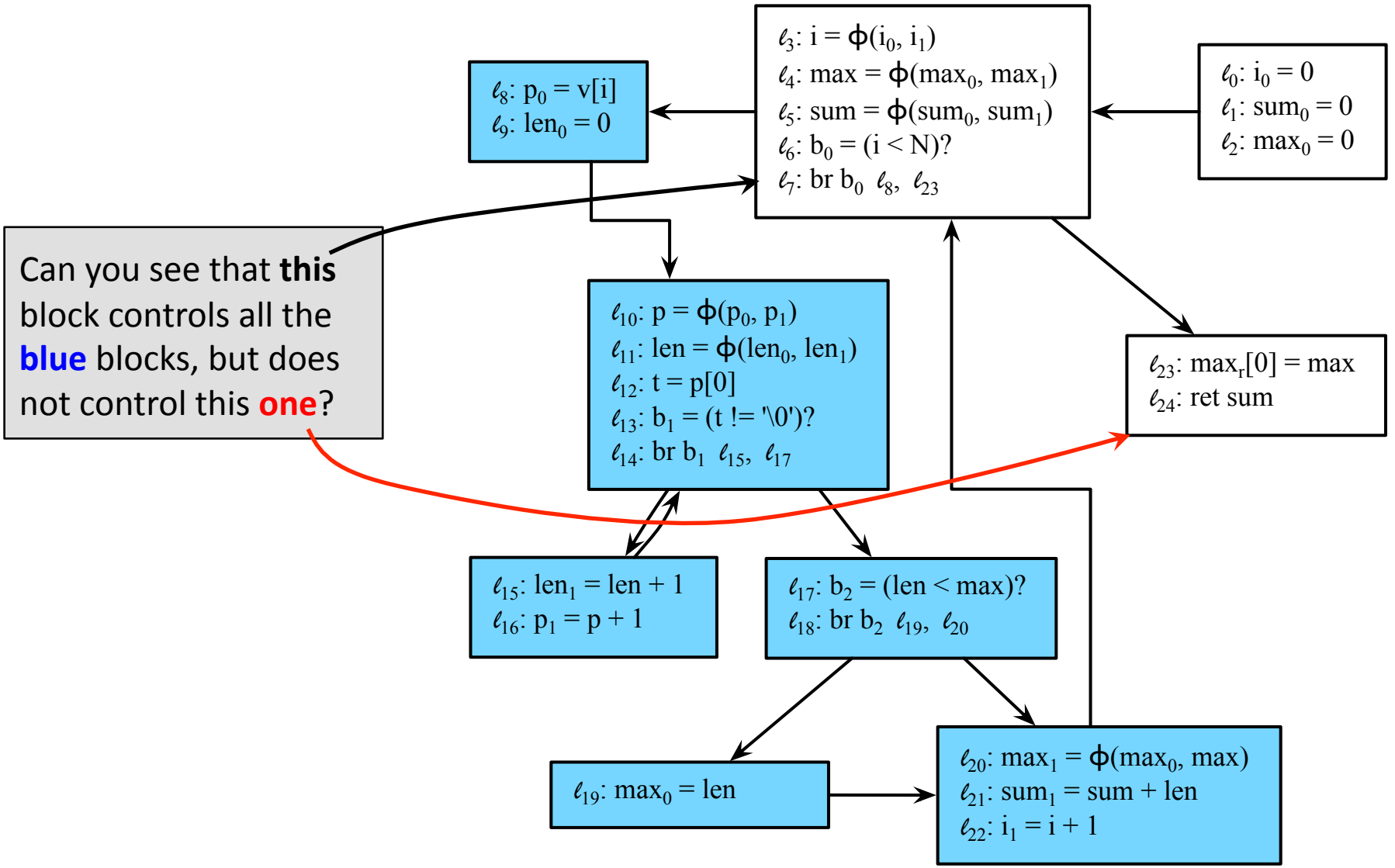
Hammock Region of br b_1 ℓ_{15}, ℓ_{17}



Hammock Region of br b_0 ℓ_8, ℓ_{23}



Hammock Region of br b_0 ℓ_8, ℓ_{23}



Fernando Magno Quintão Pereira

fernando@dcc.ufmg.br

lac.dcc.ufmg.br

Compilers Laboratory

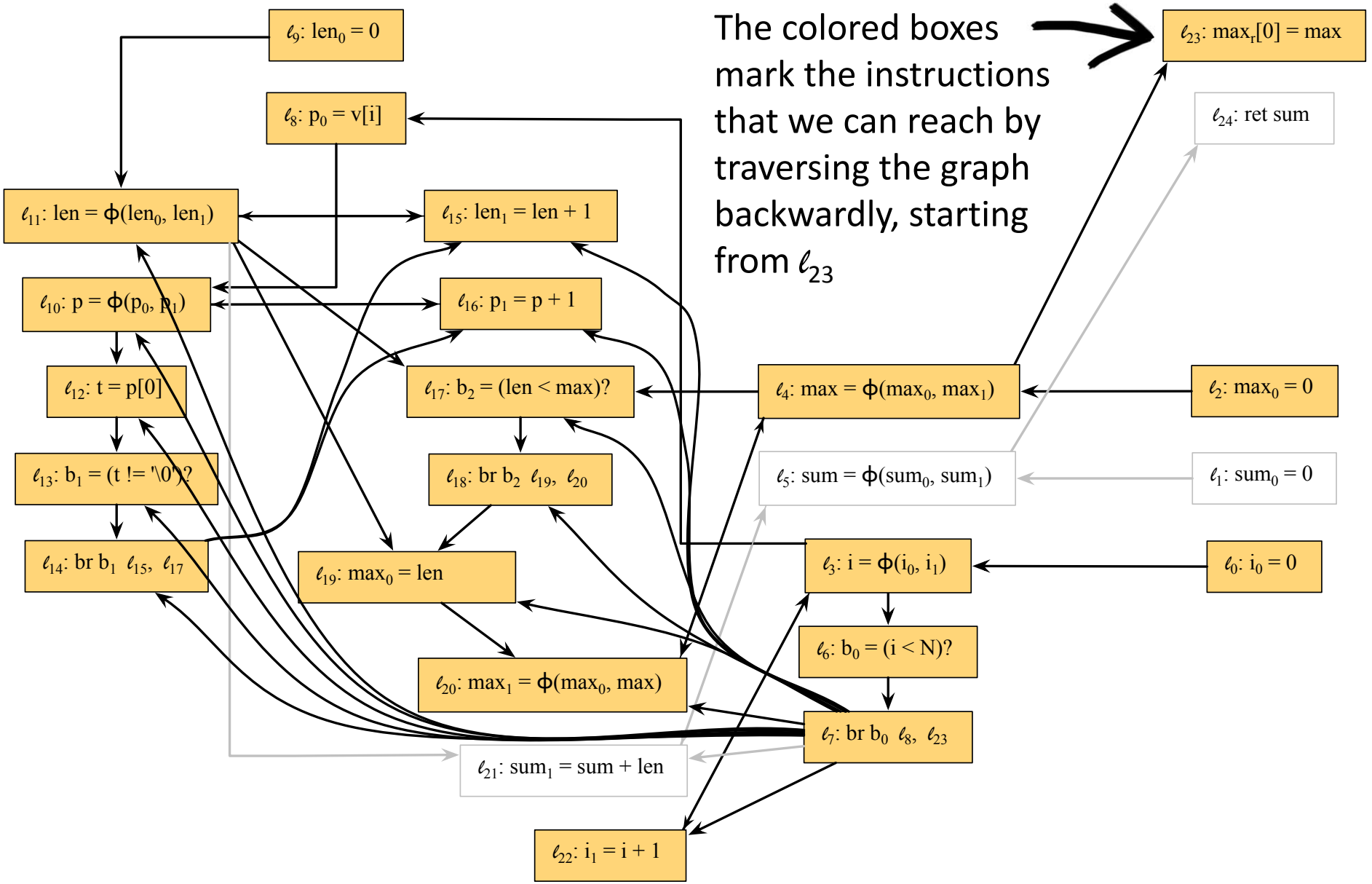
DEPARTMENT OF COMPUTER SCIENCE
UNIVERSIDADE FEDERAL DE MINAS GERAIS
FEDERAL UNIVERSITY OF MINAS GERAIS, BRAZIL

DETECTING CONTROL DEPENDENCIES

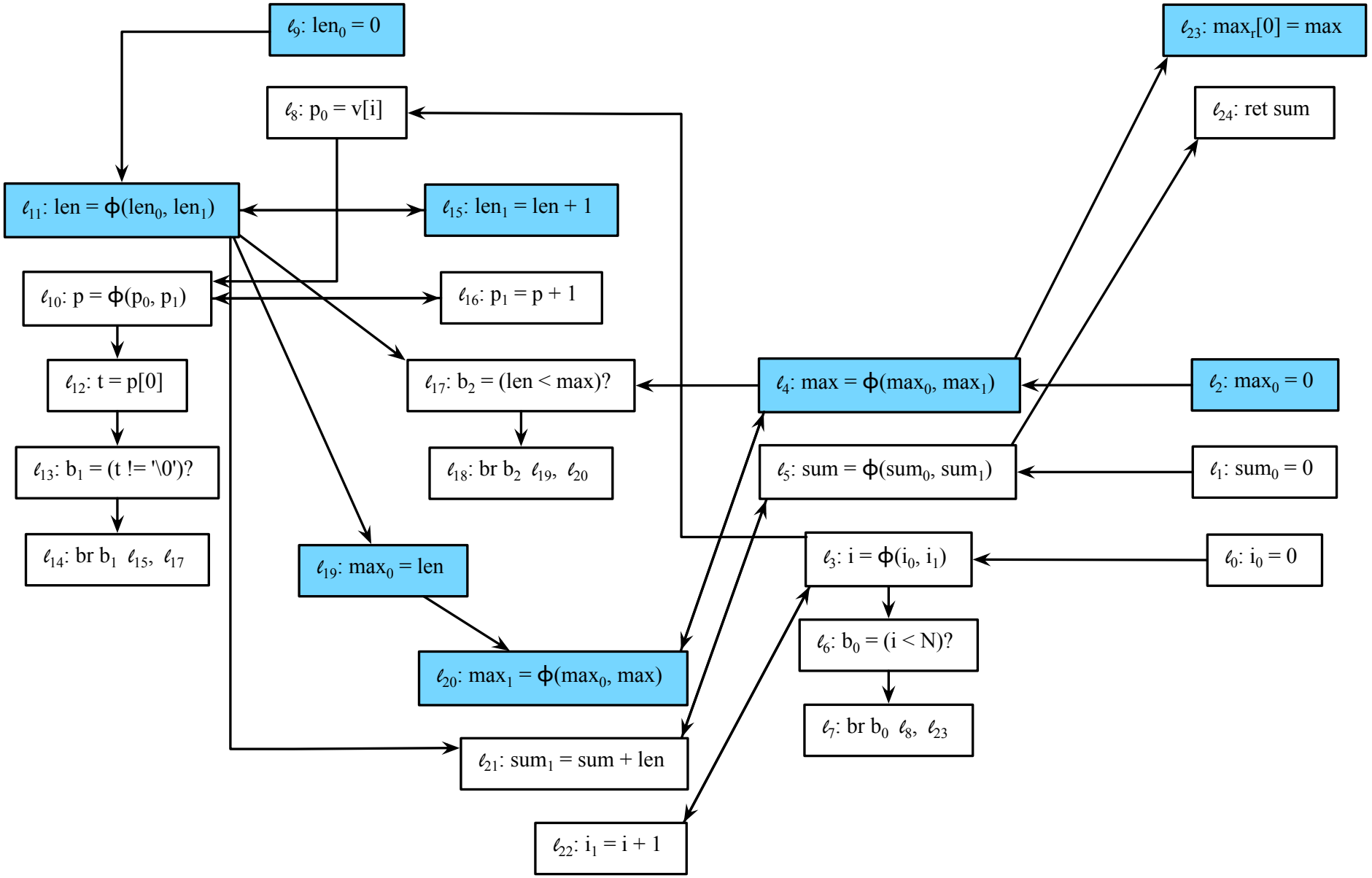
fernando@dcc.ufmg.br

DCC 888

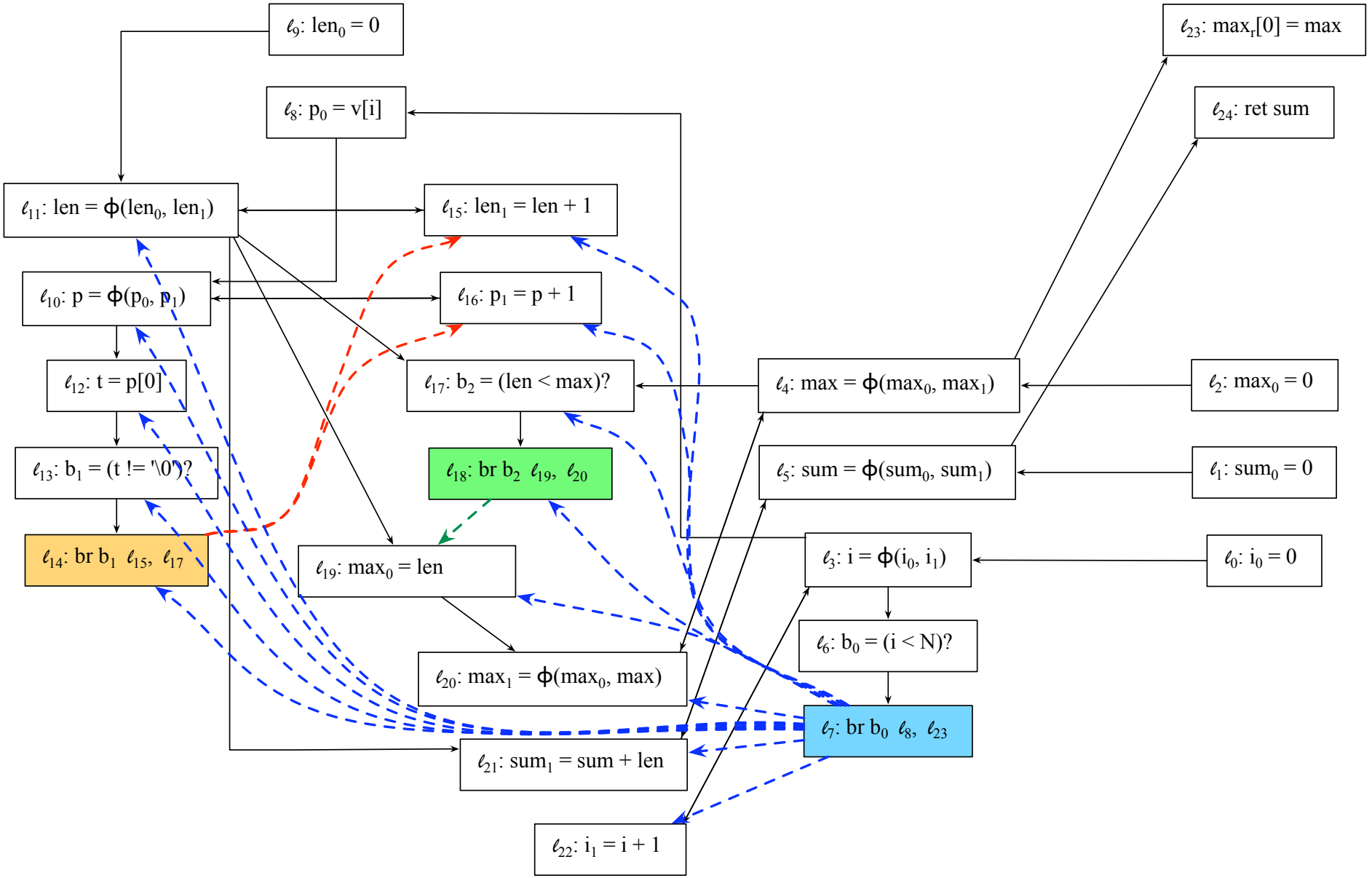
Backward slice of ℓ_{23}: $max_r[0] = max$ (Full Deps)



Data Dependences



Control Dependencies



A Functional Algorithm to Create Control Edges

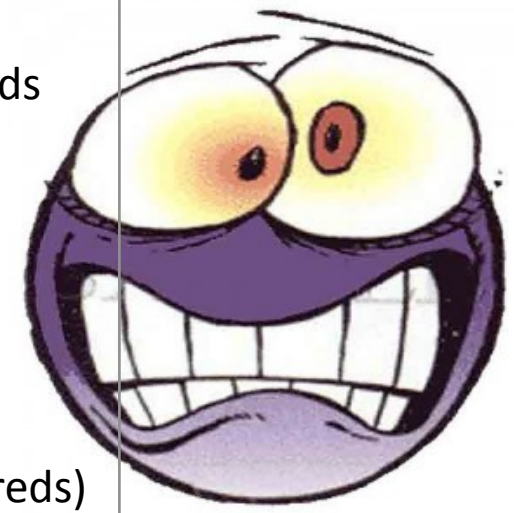
```
fun vchildren [] _ = []
   | vchildren (n::ns) preds = vnode n preds @ vchildren ns preds
and vnode n preds =
   let
      val (basic_block, ns) = inspect n
   in
      if n is the immediate post dominator of (top preds)
      then vnode n (pop preds)
      else link basic_block (top preds) @ vchildren ns (push n preds)
   end
```

inspect *separates a node n, of the dominator tree, into a basic block and a set of children ns, which are the next nodes in the dominator tree.*

A bit of SML syntax: the empty brackets, e.g., [], denote an empty list. Double colons, ::, are list constructor. For instance, (h::t) is a list with head element h, and tail t. The at symbol, @, is list concatenation. The underline, _, represents any pattern. The keywords "**fun** f … **and** g …" are used to create mutually recursive functions: f may call g, and g may call f.

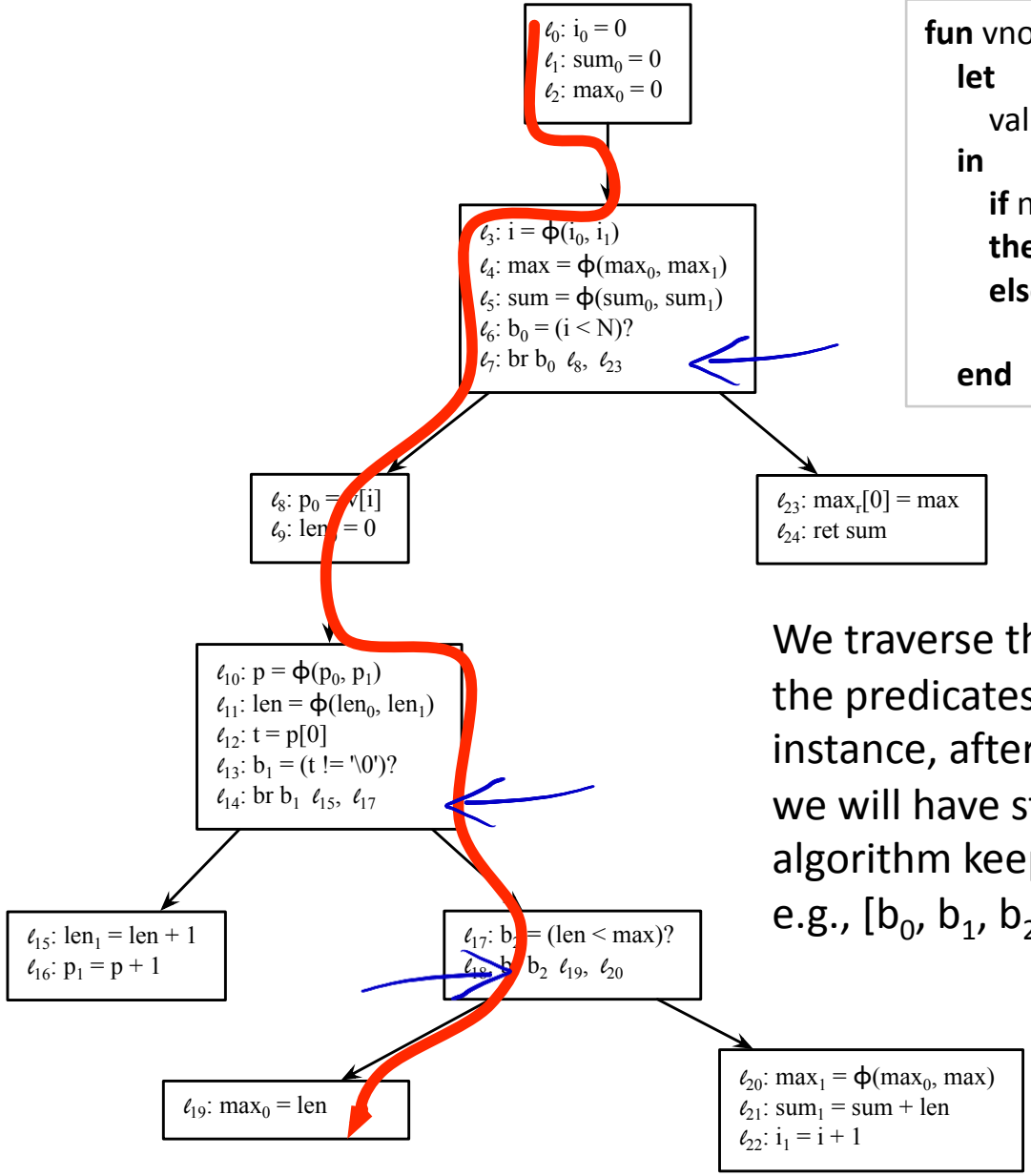
It is not that complicated…

```
fun vchildren [] _ = []
   | vchildren (n::ns) preds = vnode n preds @ vchildren ns preds
and vnode n preds =
   let
      val (basic_block, ns) = inspect n
   in
      if n is the immediate post dominator of (top preds)
      then vnode n (pop preds)
      else link basic_block (top preds) @ vchildren ns (push n preds)
   end
```



We will be going down the dominance tree of the program, stacking predicates. Whenever we find a node that terminates with a branch, we stack that predicate up. Whenever we visit an instruction i, we create an edge between the predicate on the top of the stack, and i. It is that simple! Function vchildren visits the children of a node in the dominance tree. Function vnode visits the node itself.

Stack of Predicates

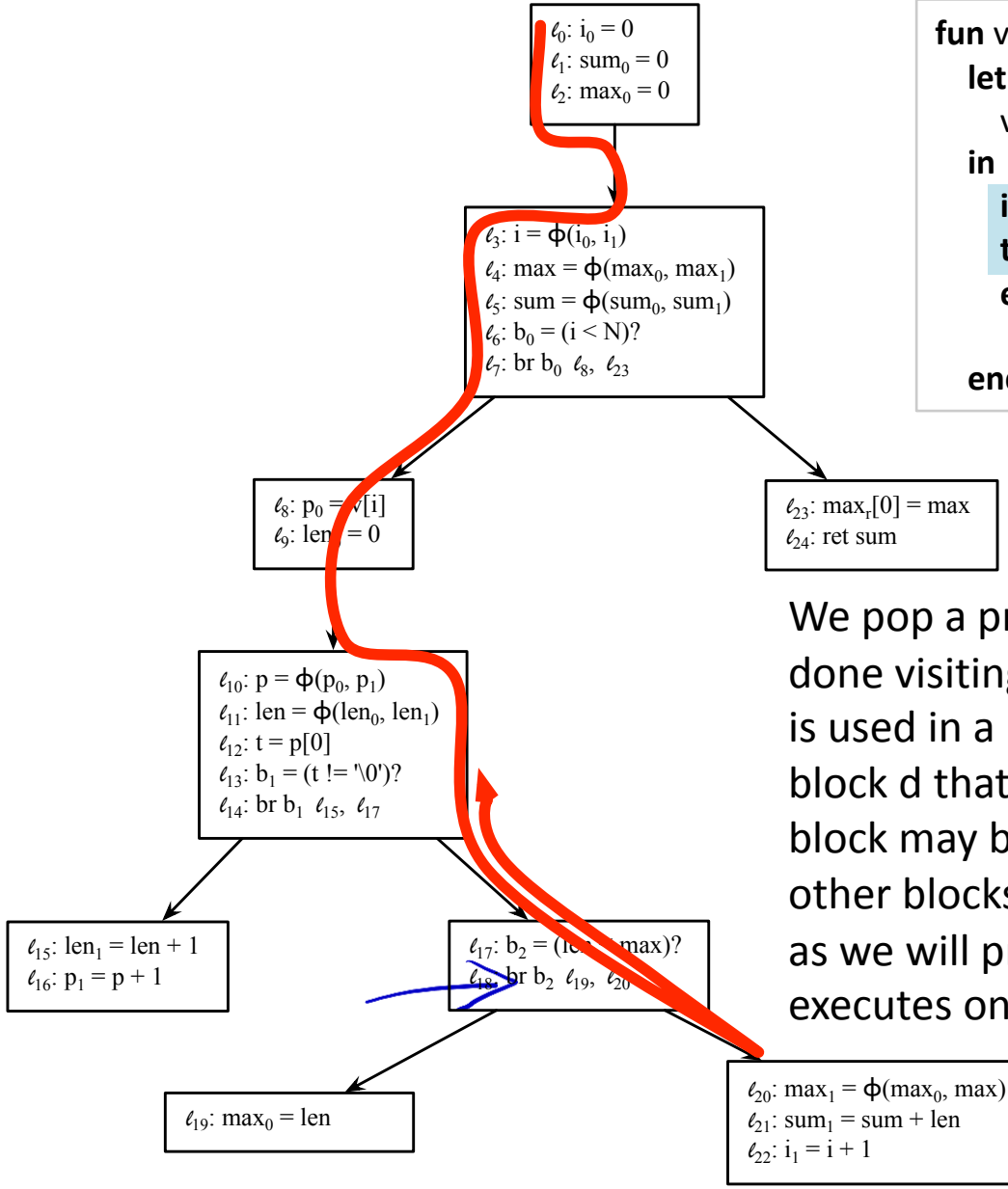



```
fun vnode n preds =
  let
    val (basic_block, ns) = inspect n
  in
    if n is the immediate post dominator of (top preds)
    then vnode n (pop preds)
    else link basic_block (top preds) @
         vchildren ns (push n preds)
  end
```

We traverse the dominator tree, stacking the predicates that we find on the way. For instance, after walking along the red path, we will have stacked b_0, b_1 and b_2. Our algorithm keeps a list with such predicates, e.g., $[b_0, b_1, b_2]$.

```
fun push n preds =
  if n terminates with branch on p
  then (p :: preds)
  else preds
```

Stack of Predicates




```
fun vnode n preds =
  let
    val (basic_block, ns) = inspect n
  in
    if n is the immediate post dominator of (top preds)
    then vnode n (pop preds)
    else link basic_block (top preds) @
         vchildren ns (push n preds)
  end
```

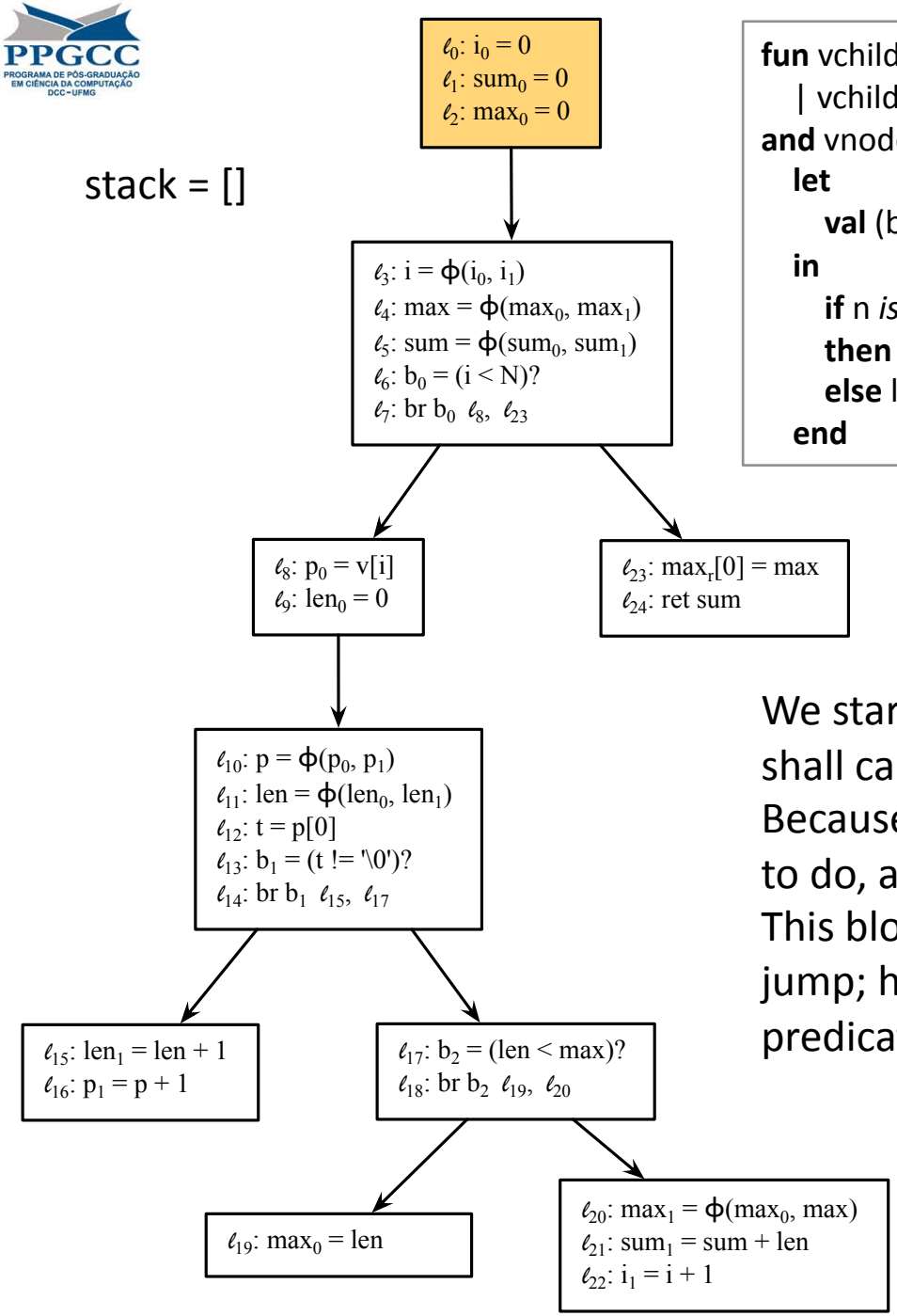
We pop a predicate p from the stack if (i) we are done visiting every child of b, the block where p is used in a branch, or (ii) we reach the basic block d that post-dominates b. Notice that a block may be the post-dominator of several other blocks that contain predicates; however, as we will prove later on, **this** piece of code executes only once.

Functional Algorithm

```
fun vchildren [] _ = []
   | vchildren (n::ns) preds = vnode n preds @ vchildren ns preds
and vnode n preds =
   let
      val (basic_block, ns) = inspect n
   in
      if n is the immediate post dominator of (top preds)
      then vnode n (pop preds)
      else link basic_block (top preds) @ vchildren ns (push n preds)
   end
```

This is a functional algorithm; thus, every child node is visited with the same stack. In other words, changes on the stack, like these **pushes** and **pops**, do not reflect on the stack that was passed originally to every child by the vchildren call. Therefore, we do not need an explicit pop operation once we are done visiting all the children of a node, because these children have all been visited with the same stack.

stack = []

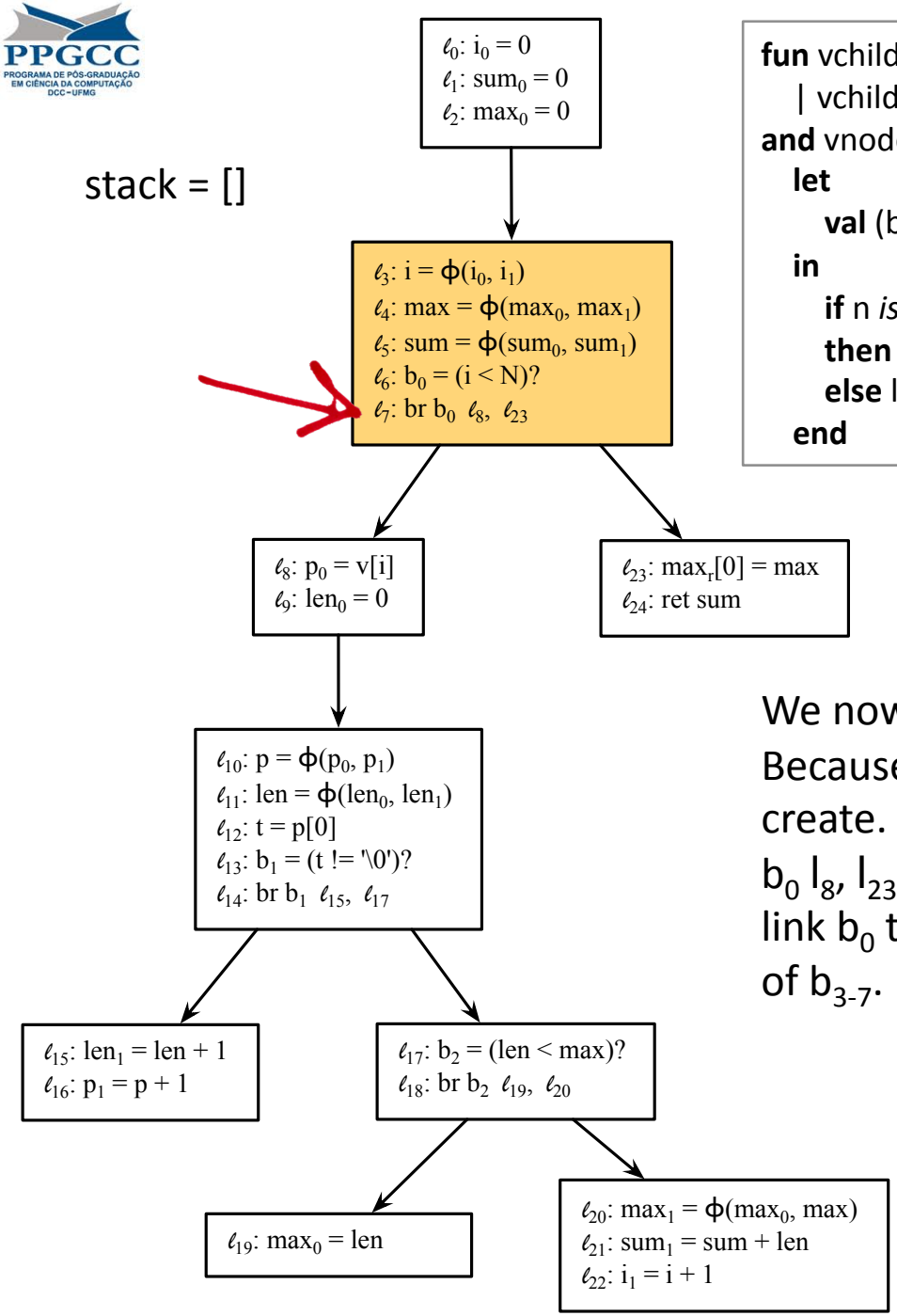



```
fun vchildren [] _ = []
   | vchildren (n::ns) preds = vnode n preds @ vchildren ns preds
and vnode n preds =
  let
     val (basic_block, ns) = inspect n
  in
     if n is the immediate post dominator of (top preds)
     then vnode n (pop preds)
     else link basic_block (top preds) @ vchildren ns (push n preds)
  end
```

We start visiting the first basic block, which we shall call $b_{0\text{-}2}$, with an empty stack of predicates. Because the stack is empty, there is nothing much to do, and we move on to visit the children of b. This block, $b_{0\text{-}2}$, terminates with an unconditional jump; hence, we do not have to stack up a new predicate, given that there is none.

```
fun push n preds =
  if n terminates with branch on p
  then (p :: preds)
  else preds
```

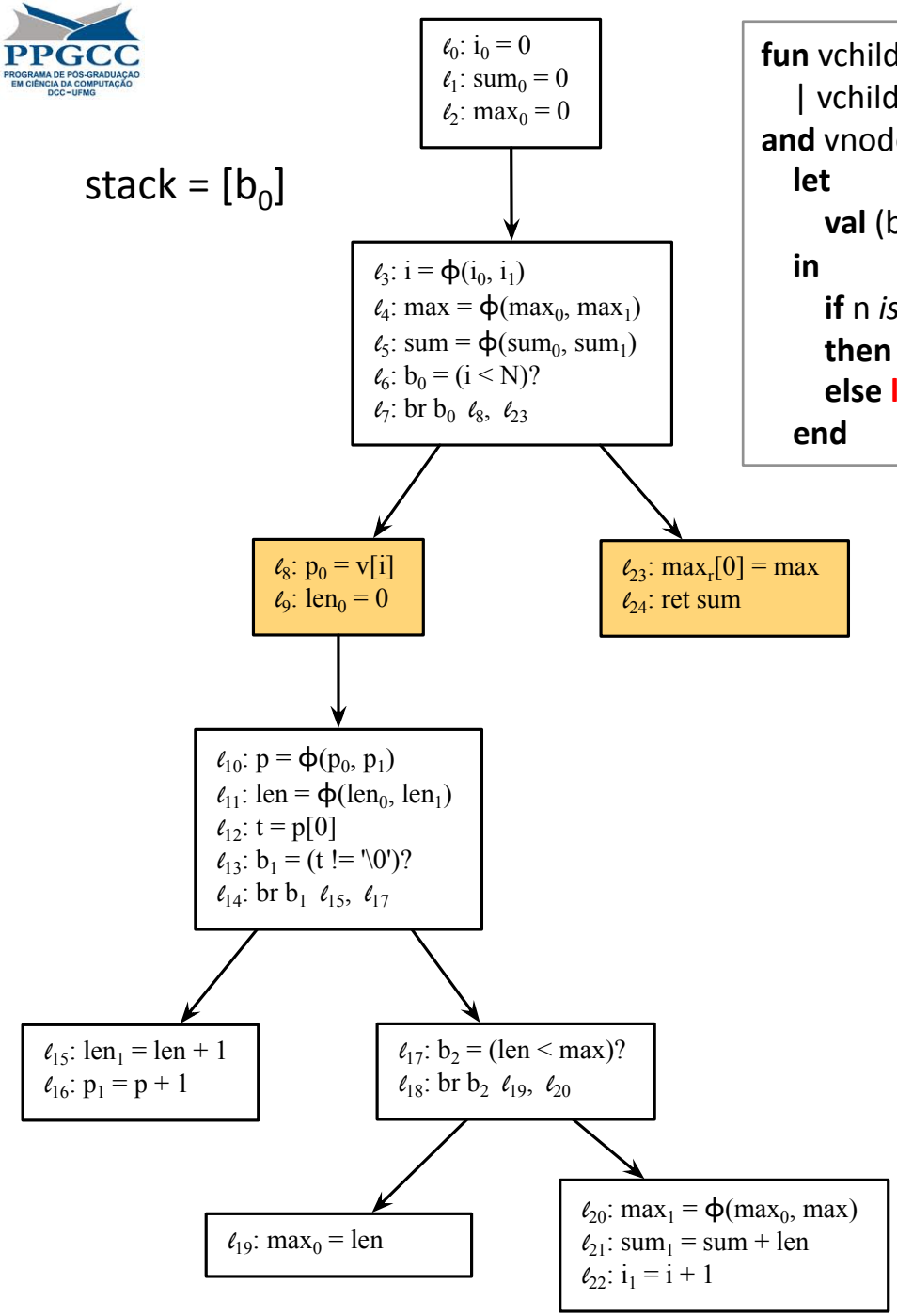
stack = []




```
fun vchildren [] _ = []
   | vchildren (n::ns) preds = vnode n preds @ vchildren ns preds
and vnode n preds =
  let
     val (basic_block, ns) = inspect n
  in
     if n is the immediate post dominator of (top preds)
     then vnode n (pop preds)
     else link basic_block (top preds) @ vchildren ns (push n preds)
  end
```

We now visit $b_{3\text{-}7}$, again, with an empty stack. Because the stack is empty, there are no links to create. But this block terminates with a branch br b_0 l_8, l_{23}. Thus, we push b_0 up. In this way, we will link b_0 to the instructions defined in the children of $b_{3\text{-}7}$.

```
fun push n preds =
  if n terminates with branch on p
  then (p :: preds)
  else preds
```




```
fun vchildren [] _ = []
   | vchildren (n::ns) preds = vnode n preds @ vchildren ns preds
and vnode n preds =
  let
     val (basic_block, ns) = inspect n
  in
     if n is the immediate post dominator of (top preds)
     then vnode n (pop preds)
     else link basic_block (top preds) @ vchildren ns (push n preds)
  end
```

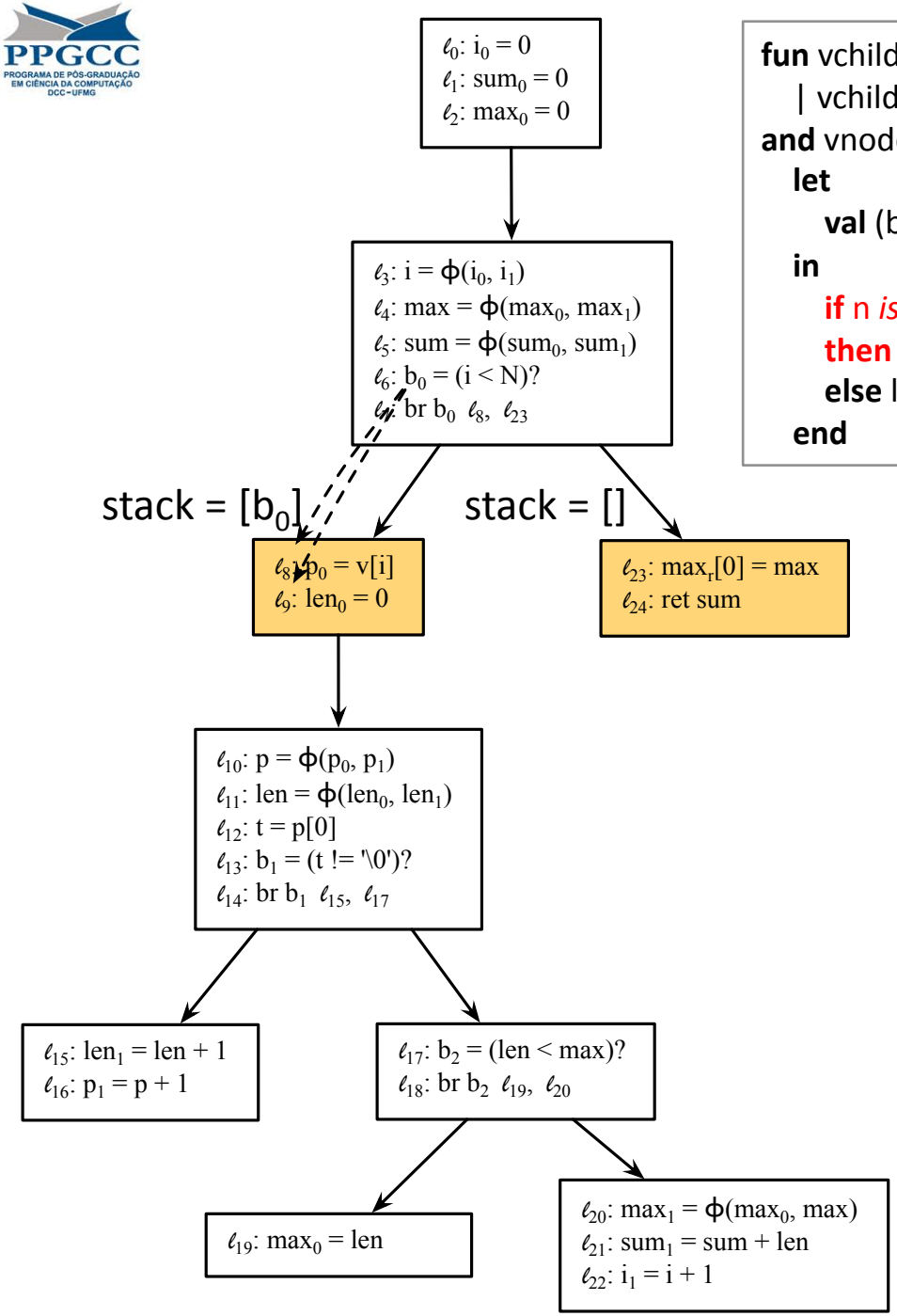
We are now visiting the children of $b_{3\text{-}7}$. The order in which we visit these children is not important. Both are being visited with the stack $[b_0]$. The important event that takes place is the linking of nodes. We have not shown this code so far; hence, it is given below:

```
fun  link [] _ = []
     | link _ NULL = []
     | link (a = b :: ins) p = (p, a) :: link ins p
     | link (a = b+c :: ins) p = (p, a) :: link ins p
     | link (a =φ(b,c) :: ins) p = (p, a) :: link ins p
     | link _ _ = []
```

Link

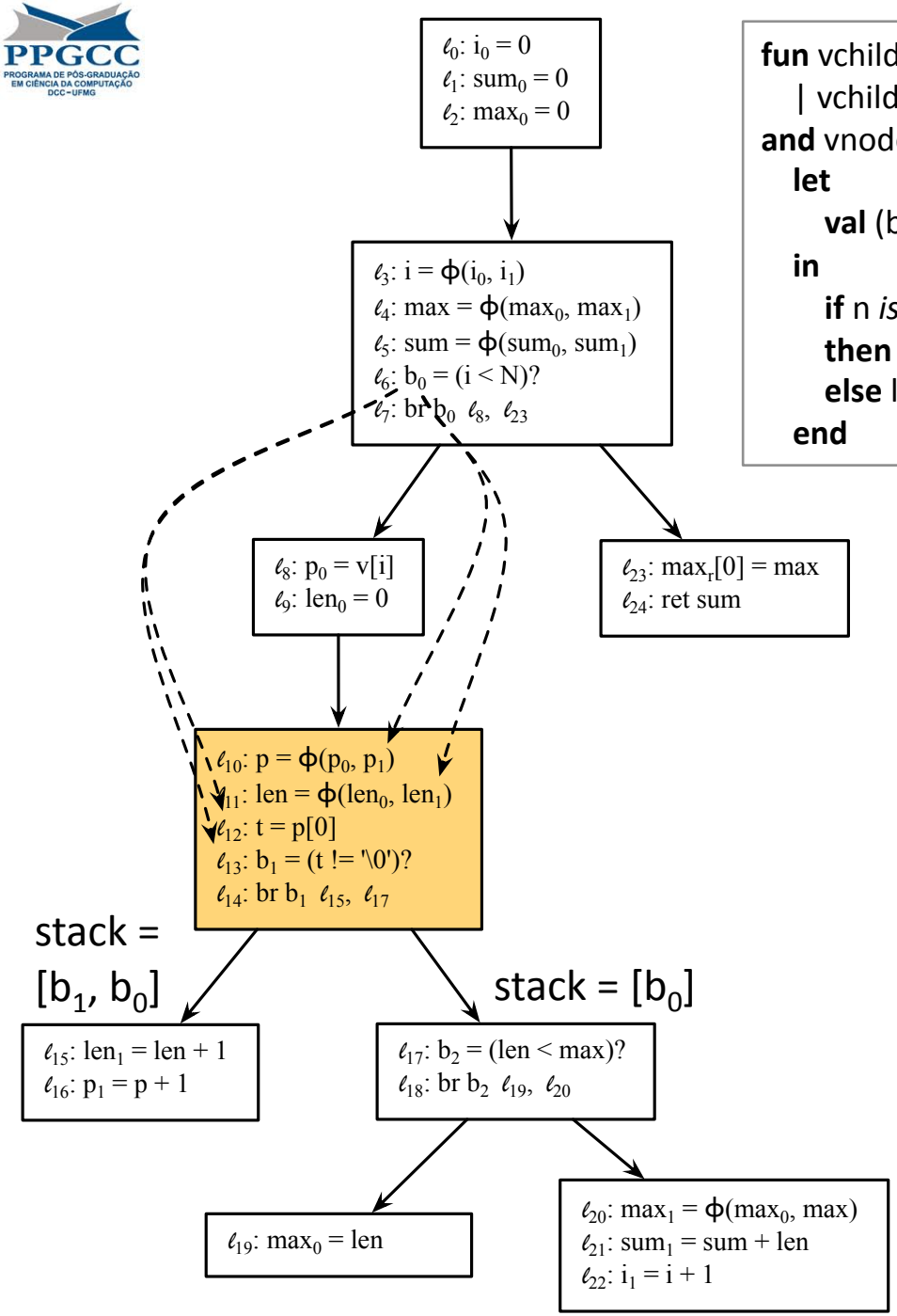
```
fun vchildren [] _ = []
   | vchildren (n::ns) preds = vnode n preds @ vchildren ns preds
and vnode n preds =
   let
      val (basic_block, ns) = inspect n
   in
      if n is the immediate post dominator of (top preds)
      then vnode n (pop preds)
      else link basic_block (top preds) @ vchildren ns (push n preds)
   end

fun  link [] _ = []
    | link _ NULL = []
    | link (a = b :: ins) p = (p, a) :: link ins p
    | link (a = b+c :: ins) p = (p, a) :: link ins p
    | link (a =φ(b,c) :: ins) p = (p, a) :: link ins p
    | link _ _ = []
```

Each basic block is a sequence of instructions, that terminates either with a RET, a BRANCH or a JUMP. We call these instructions *terminators*. Instructions that define new variables, e.g., **MOV**, **ADD** and **PHI** add new links to our dependence graph. The **other** instructions do not. The algorithm terminates if we are **done with this list** or reach a terminator. If we receive a **null** label (when the predicate stack is empty), we do not even start.



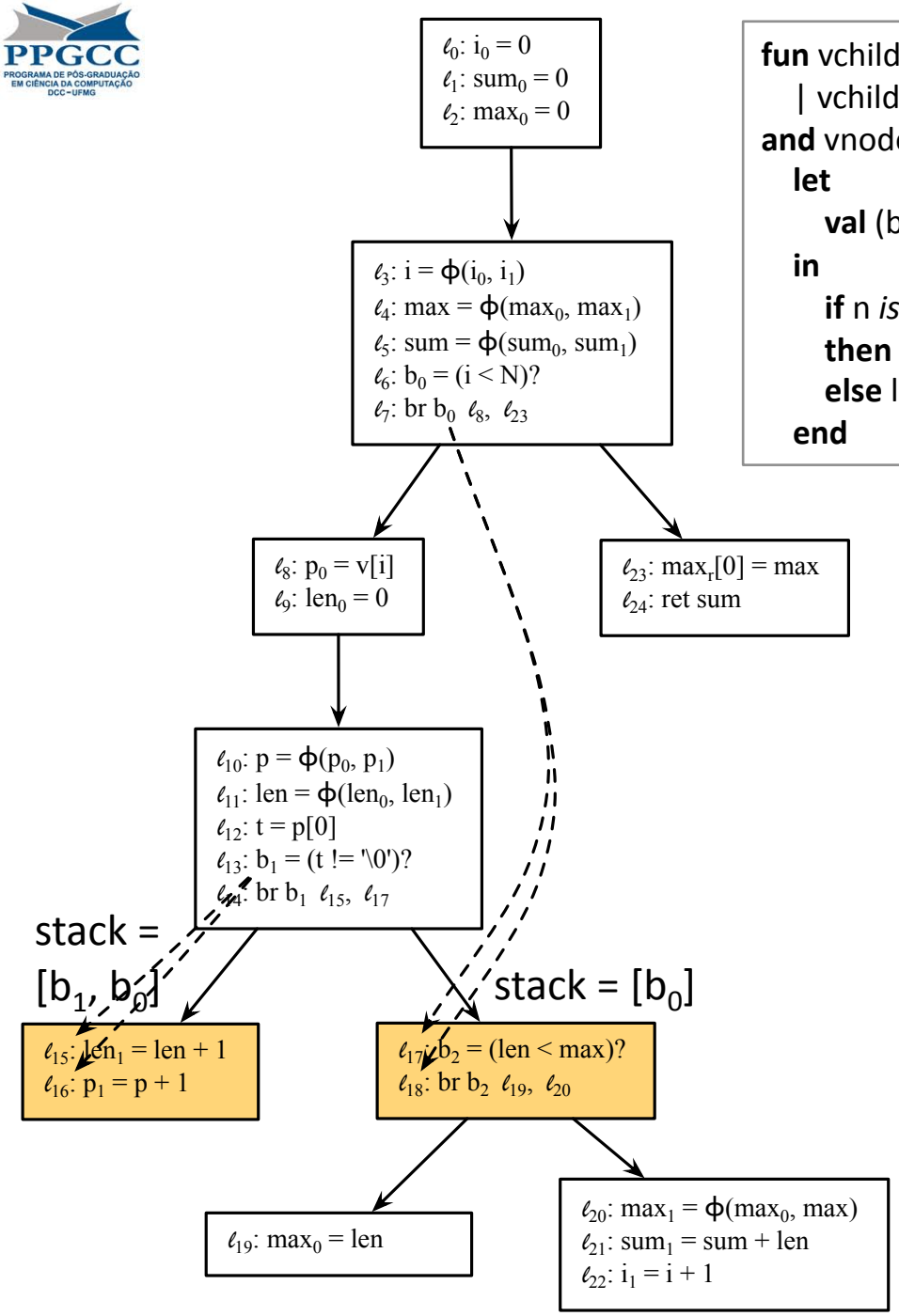

```
fun vchildren [] _ = []
   | vchildren (n::ns) preds = vnode n preds @ vchildren ns preds
and vnode n preds =
   let
      val (basic_block, ns) = inspect n
   in
      if n is the immediate post dominator of (top preds)
      then vnode n (pop preds)
      else link basic_block (top preds) @ vchildren ns (push n preds)
   end
```

In this case, we will create edges (b_0, p_0) and (b_0, len_0). However, we will not create the edge (b_0, max_r). This happens because $b_{23\text{-}24}$ is the post-dominator of $b_{3\text{-}7}$, the block where b_0 is defined. When we visit $b_{23\text{-}24}$, coming out of $b_{3\text{-}7}$, we must pop b_0 out of the stack. There is a cool result that we can prove about hammock graphs: if b dominates its exit block b_p, then b is the immediate dominator of b_p. Thus, if b dominates b_p, there will be a direct link between b and b_p in the tree. In other words, **this** code happens only once during a traversal of the dominance tree.



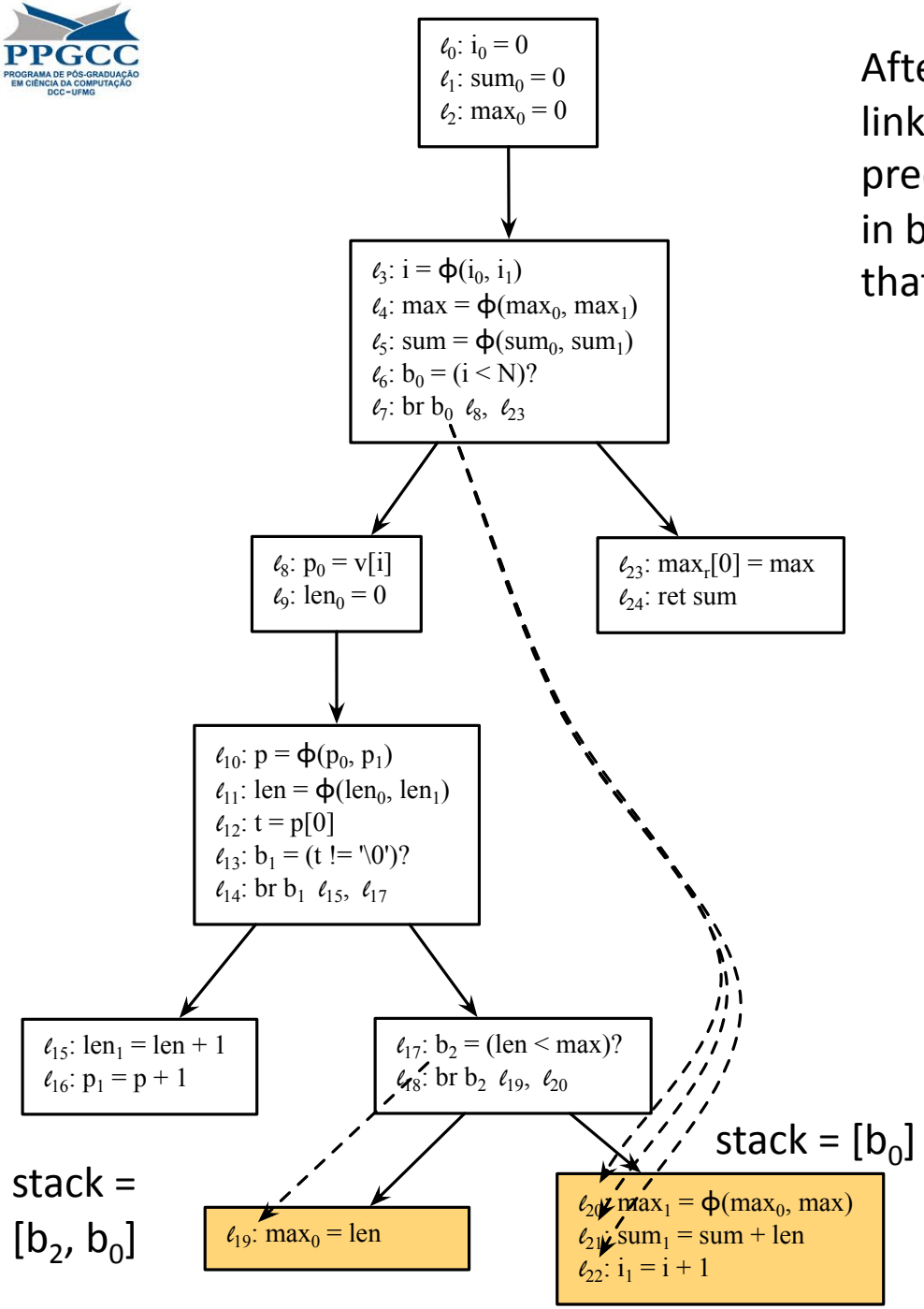

```
fun vchildren [] _ = []
   | vchildren (n::ns) preds = vnode n preds @ vchildren ns preds
and vnode n preds =
   let
      val (basic_block, ns) = inspect n
   in
      if n is the immediate post dominator of (top preds)
      then vnode n (pop preds)
      else link basic_block (top preds) @ vchildren ns (push n preds)
   end
```

Once we visit b_{10_14}, there are two things that we must do. First, we must link every instruction in this block (at least those that define variables) to b_0, which is the predicate currently at the top of the stack. Second, we must update the stack to visit b_{15_16}. Notice that this updating happens only to visit b_{15_16}. We visit b_{17_18} with the old stack, i.e., $[b_0]$, because this block post-dominates b_{10_14}.

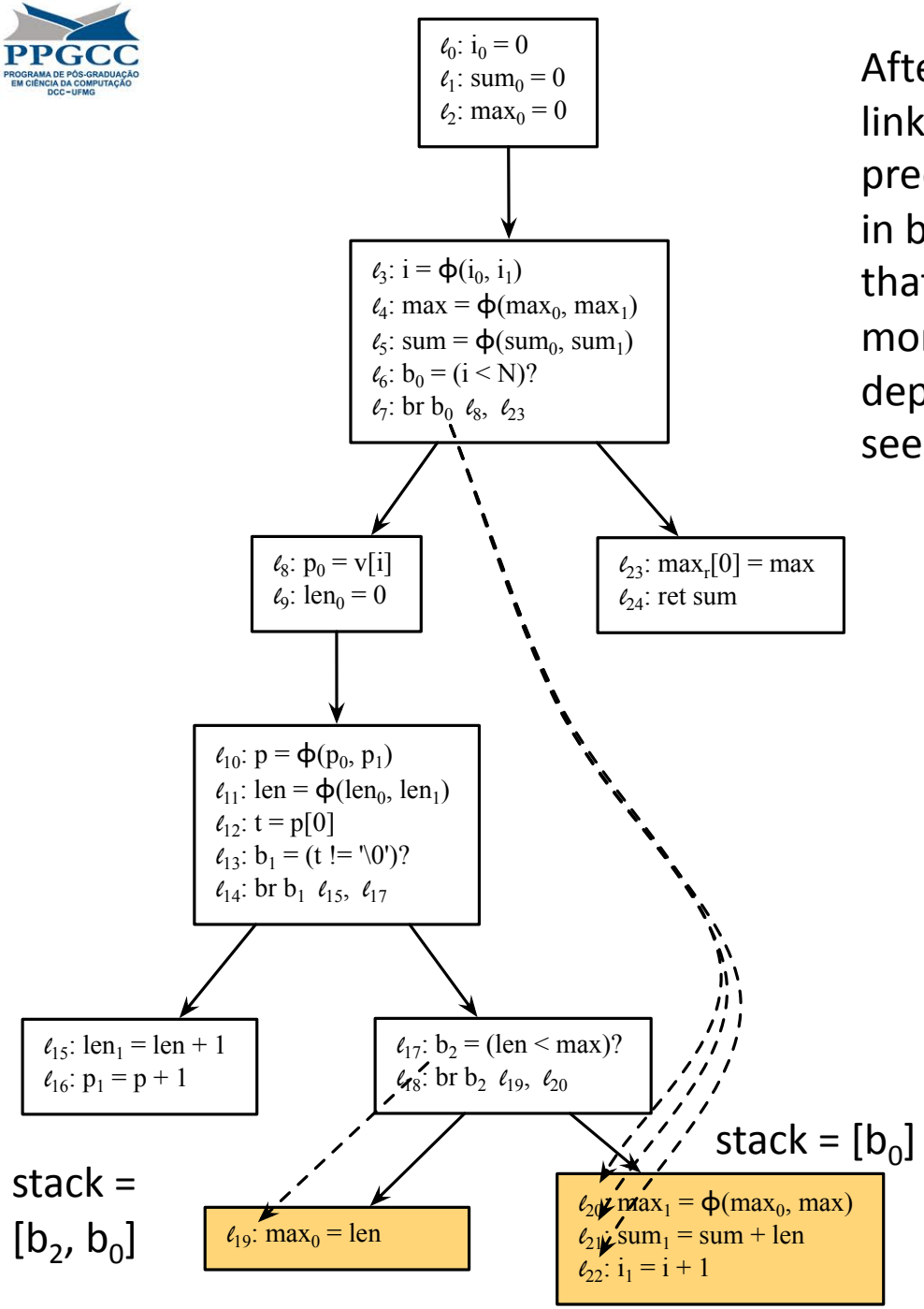



```
fun vchildren [] _ = []
   | vchildren (n::ns) preds = vnode n preds @ vchildren ns preds
and vnode n preds =
   let
      val (basic_block, ns) = inspect n
   in
      if n is the immediate post dominator of (top preds)
      then vnode n (pop preds)
      else link basic_block (top preds) @ vchildren ns (push n preds)
   end
```

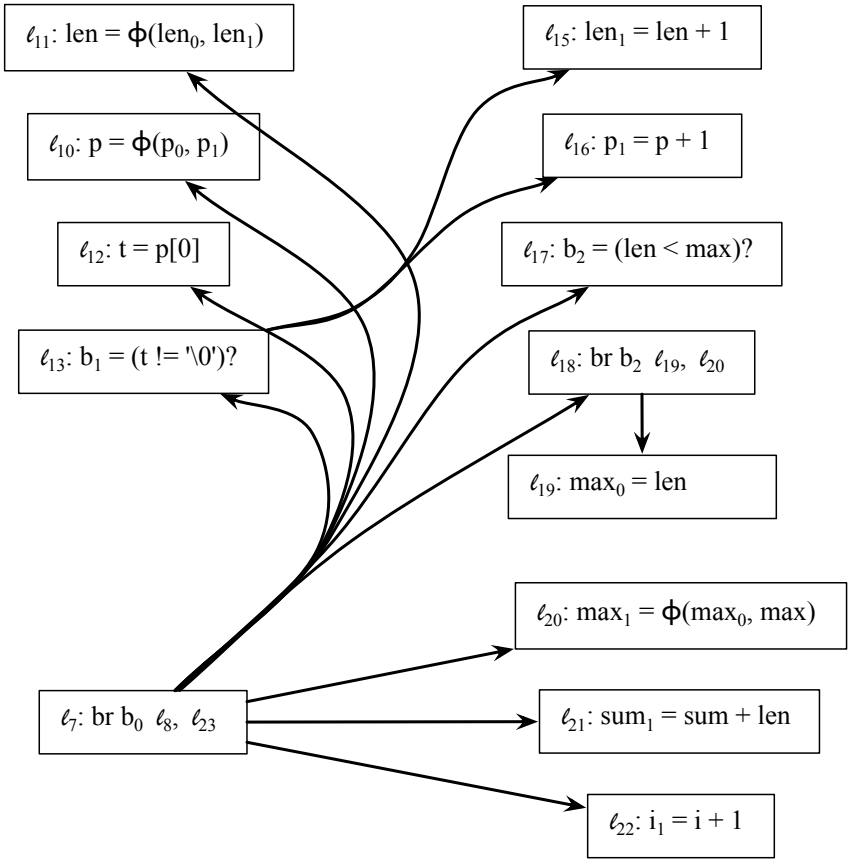
We link every node of $b_{15\text{-}16}$ with b_1, because this is the predicate on the top of the stack once $b_{15\text{-}16}$ is visited. From $b_{15\text{-}16}$ we are done on this side of the tree, because this block has no children. Once we visit b_{17_18} we link every instruction in this block to b_0, and not b_1, because b_0 is the node on the top of the stack when b_{17_18} is visited. We do not have to stack b_1 up to visit b_{17_18}, because this block post-dominates b_{10_14}.



After visiting $b_{19\text{-}19}$ and $b_{20\text{-}22}$ we are done. We link instructions in $b_{19\text{-}19}$ to b_2, the top predicate on the stack, and we link instructions in $b_{20\text{-}22}$ to b_0, the predicate on the stack when that block is visited.



After visiting $b_{19\text{-}19}$ and $b_{20\text{-}22}$ we are done. We link instructions in $b_{19\text{-}19}$ to b_2, the top predicate on the stack, and we link instructions in $b_{20\text{-}22}$ to b_0, the predicate on the stack when that block is visited. Given that we have no more blocks to visit, we are done! The control dependence edges that we have created can be seen below:



Fernando Magno Quintão Pereira

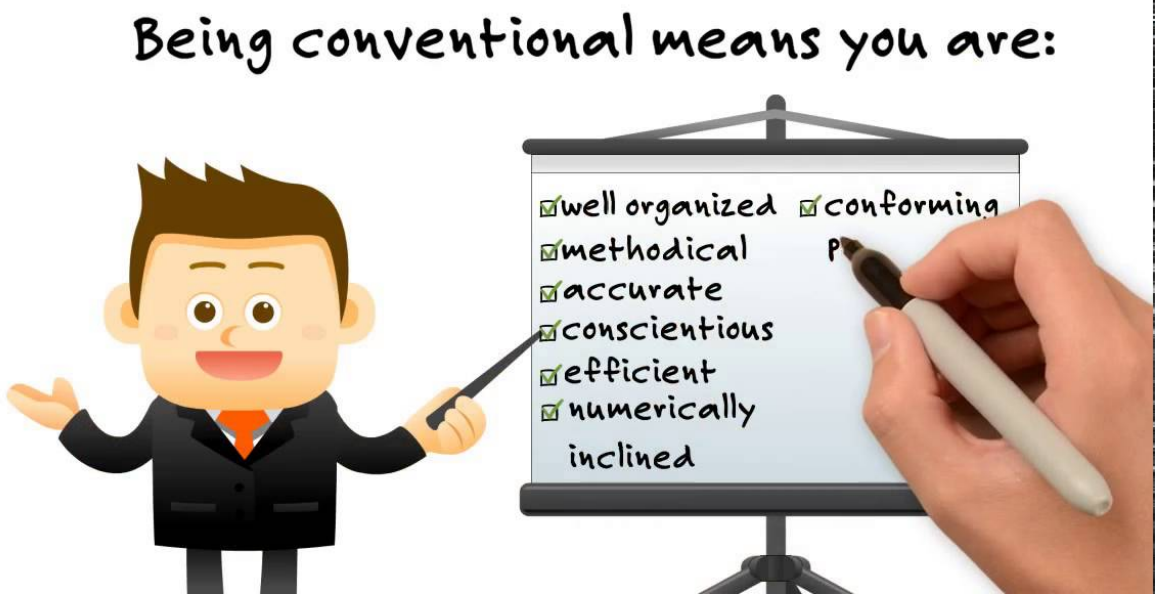
fernando@dcc.ufmg.br

lac.dcc.ufmg.br

Compilers Laboratory

Department of Computer Science
Universidade Federal de Minas Gerais
Federal University of Minas Gerais, Brazil

Conventional SSA-Form



fernando@dcc.ufmg.br

Non-Conventional SSA-Form

- Two variables v_1 and v_2 are *related* by a phi-function if:
  - They appear in the same phi-function, e.g:
    - v_1 = phi(…, v_2, …)
    - v = phi(…, v_1, …, v_2, …)
  - There exists a third variable v, such that v_1 and v_2 are related to v, e.g., v = phi(…, v_1, …), v' = phi(…, v, …, v_2, …)

Non-Conventional SSA-Form

- Two variables v_1 and v_2 are *related* by a phi-function if:
  - They appear in the same phi-function, e.g:
    - v_1 = phi(…, v_2, …)
    - v = phi(…, v_1, …, v_2, …)
  - There exists a third variable v, such that v_1 and v_2 are related to v, e.g., v = phi(…, v_1, …), v' = phi(…, v, …, v_2, …)
- A program is in *Conventional Static Single Assignment* (CSSA) form if, and only if, all the related variables have non-overlapping ranges.

Non-Conventional SSA-Form

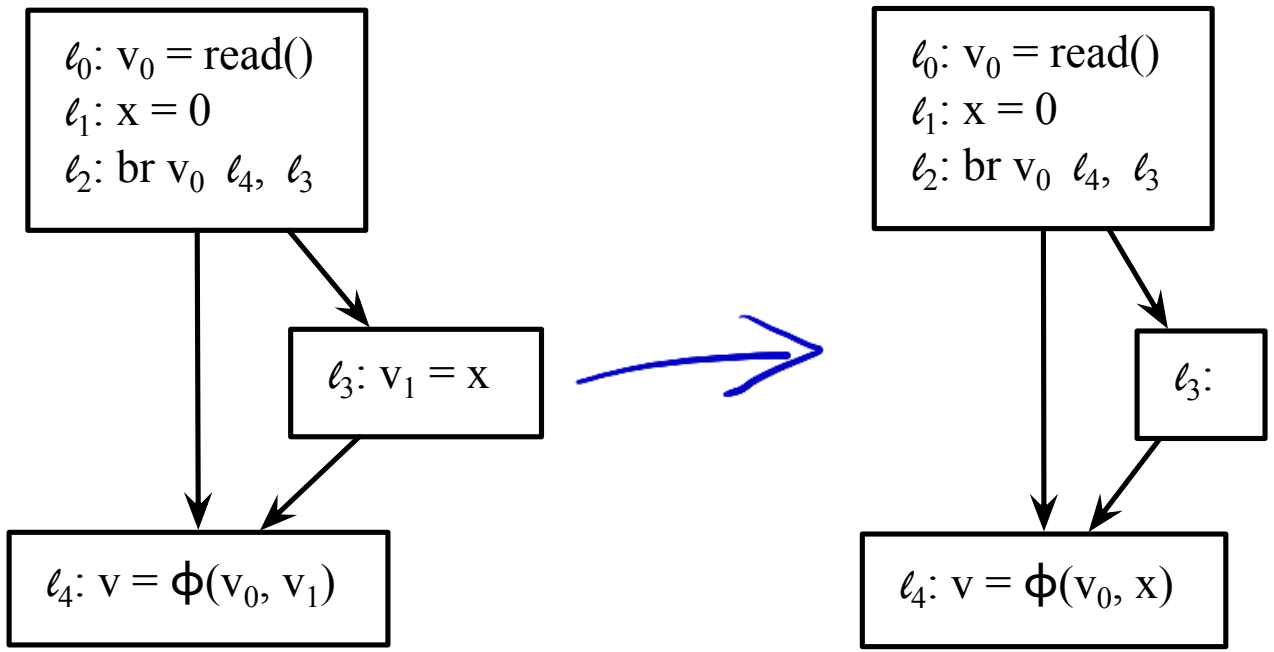
- Two variables v_1 and v_2 are *related* by a phi-function if:
  - They appear in the same phi-function, e.g:
    - v_1 = phi(..., v_2, ...)
    - v = phi(..., v_1, ..., v_2, ...)
  - There exists a third variable v, such that v_1 and v_2 are related to v, e.g., v = phi(..., v_1, ...), v' = phi(..., v, ..., v_2, ...)
- A program is in *Conventional Static Single Assignment* (CSSA) form if, and only if, all the related variables have non-overlapping ranges.

Why is **this** true?

Right after we convert a program to the Static Single Assignment form, this program is in conventional SSA form. However, code optimizations, such as copy propagation, may break the CSSA core property.

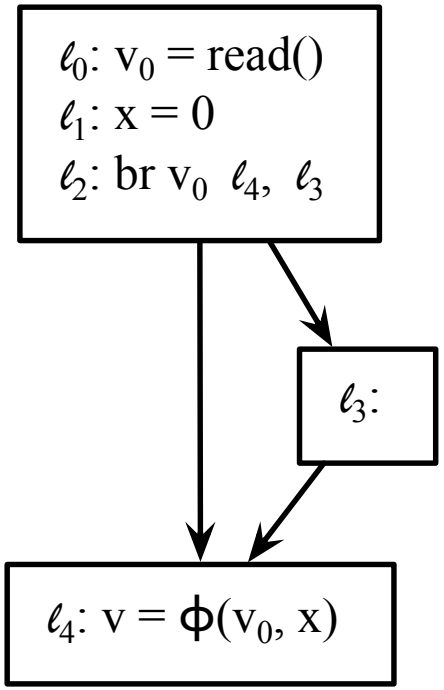
Example of Non-CSSA form Program

The program on the left is in CSSA format, because all the phi-related variables, i.e., v_0, v_1 and v, have disjoint live ranges. We obtain the program on the right after applying copy propagation on the original program. This new program is not in CSSA form, because phi-related variables v_0 and x are simultaneously alive at the end of block $b_{0\text{-}2}$.



Non-CSSA form is a pain in the neck

Several algorithms that work on SSA form programs are simpler if they run on CSSA code. If the CSSA property is absent, then a number of well-known problems surface, such as the **lost copy**♢, and the **spare register**♡. Our algorithm also suffers a bit once this property is not present. For instance, if we had to slice the program on the right with regard to l_4, then we would miss the control dependence edge between predicate v_0 and the phi-function that defines v. In other words, the branch on v_0 controls which assignment the phi-function at ℓ_4 performs. However, we do not add edges between $b_{0\text{-}2}$ and $b_{4\text{-}4}$, because the latter block post-dominates the former.



♢: Practical Improvements to the Construction and Destruction of Static Single Assignment Form, 1998
♡: SSA Elimination after Register Allocation, 2009

Handling Non-CSSA form programs

We can handle non-CSSA form programs in two ways:

1. Convert the program to CSSA form.

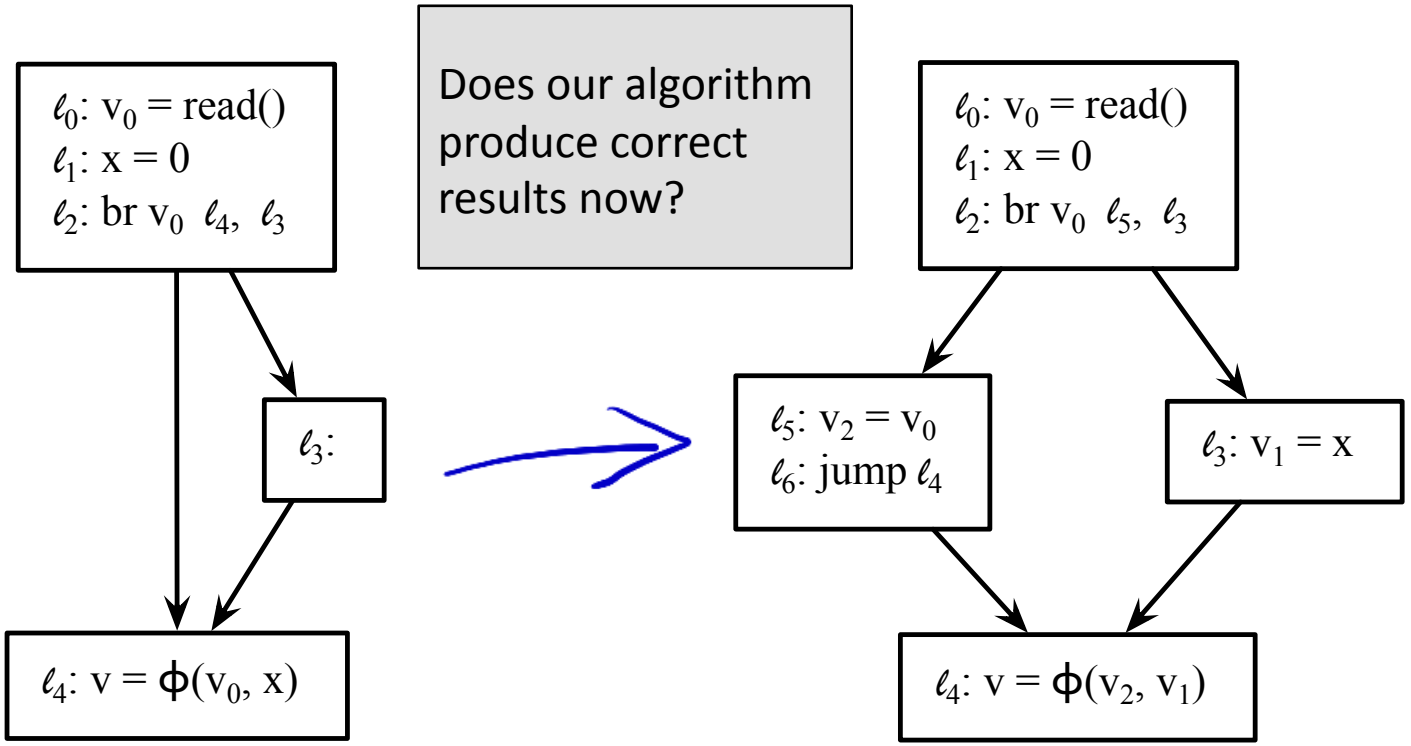
How do we convert an SSA-form program into CSSA format?

2. Modify the algorithm to add control dependence edges from a node to the phi-functions on its immediate post-dominator.

Is it necessary to add edges whenever there is a phi-function at the post-dominator of the predicate?

CSSA Conversion

It is fairly easy to convert a program to CSSA format. We must (i) break critical edges, and (ii) split live ranges right before the phi-function, at the predecessors of the node where the phi-function is located♧.



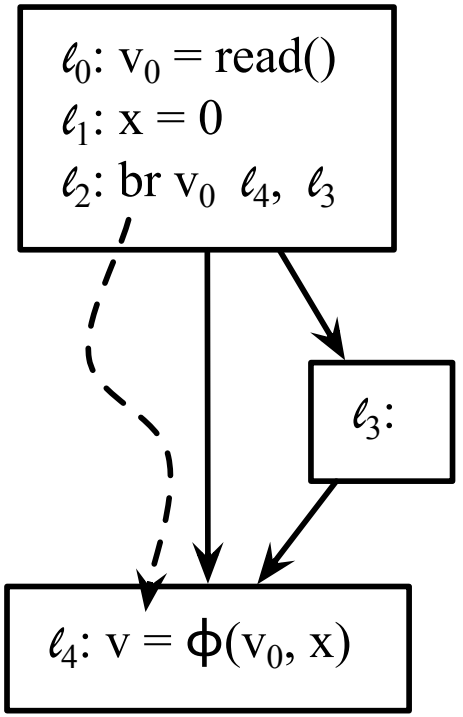
♧: Translating out of Static Single Assignment Form, 1999

Extending the Algorithm

We can also solve our problem by extending the algorithm to handle non-CSSA form programs. For instance, we can run the step below independently from the main algorithm:

For each predicate p, defined at block b, with post-dominator b', do: for each phi-function v = phi(..., v_1, ..., v_2, ...) at b', if v_1 and v_2 are different, and at least one of them reach the phi on an edge controlled by p, then add an edge from p to the phi-function.

Can you think on a situation in which no control edges would be created between p and b'?

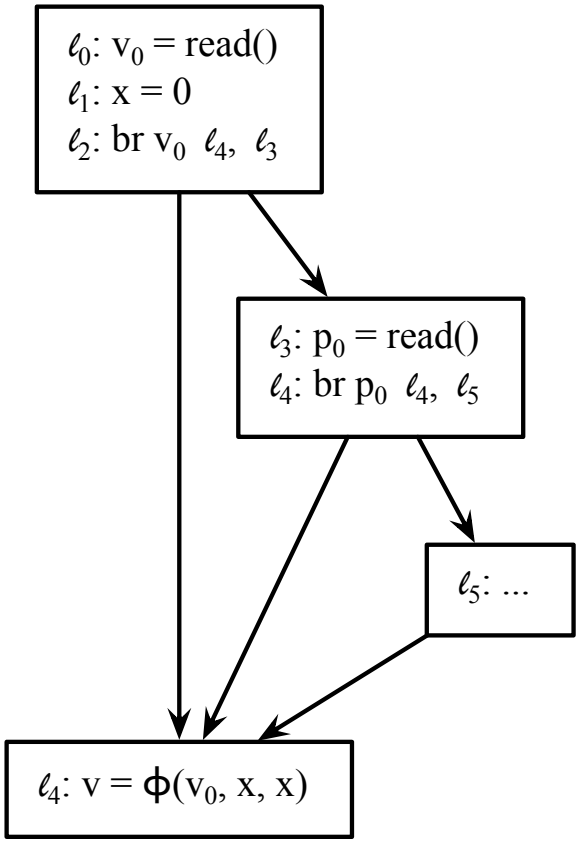


Extending the Algorithm

We can also solve our problem by extending the algorithm to handle non-CSSA form programs. For instance, we can run the step below independently from the main algorithm:

For each predicate p, defined at block b, with post-dominator b', do: for each phi-function v = phi(..., v_1, ..., v_2, ...) at b', if v_1 and v_2 are different, and at least one of them reach the phi on an edge controlled by p, then add an edge from p to the phi-function.

The extra edge would not be necessary between p_0 and ℓ_4, in the program on the right, because, independently on which path we take after the branch that p_0 controls, we will always dump the same value, x, in the phi-function.



A Bit of History

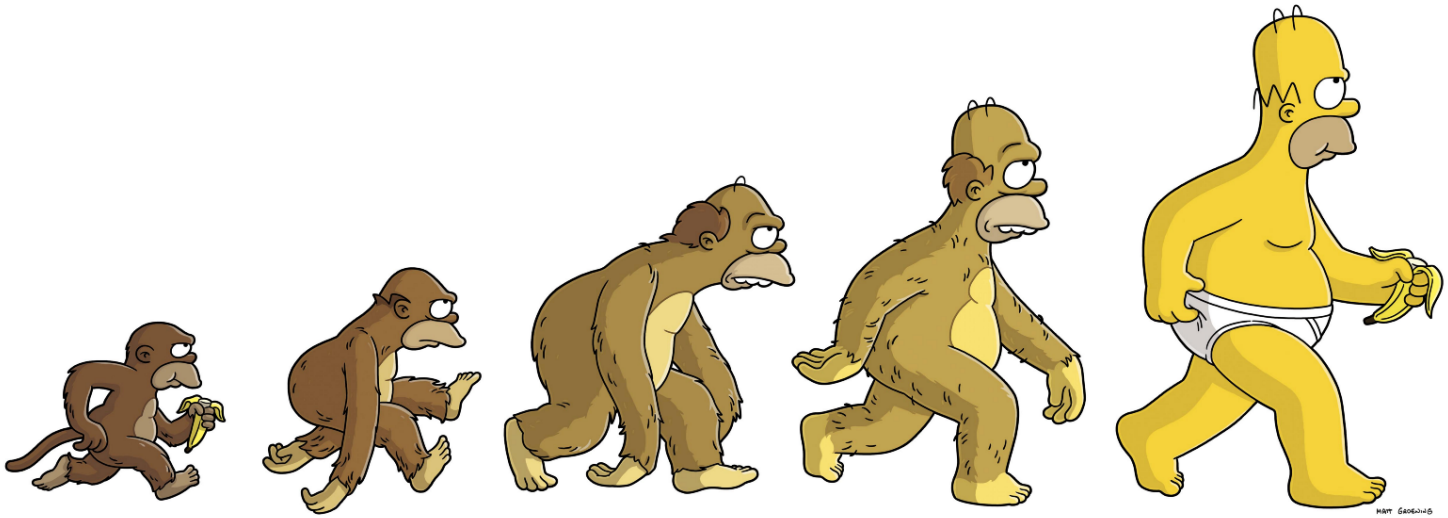
- The idea of program slicing seems to have appeared in a work by Weiser.
- The algorithm discussed in this presentation was taken from a paper by Rodrigues *et al*.
- The notion of a dependence graph seems to be due to Ottenstein[2].
- Control dependences were discussed in a work by Ferrante *et al*.

- Weiser, M. Program slices: formal, psychological, and practical investigations of an automatic program abstraction method, PhD thesis, University of Michigan, (1979)
- Rodrigues. B, Aranha, D. Pereira, F. A Sparse representation of implicit flows with applications to side-channel detection, CC, 110-120 (2016).
- Ottenstein, K., Ottenstein, L., The program dependence graph in a software development environment, SIGPLAN Notices 19(5):177-184, (1984)
- Ferrante, J., Ottenstein, K, Warren, J., The program dependence graph and its use in optimization. TOPLAS, 9(3):319-349, 1987

Compilers Laboratory

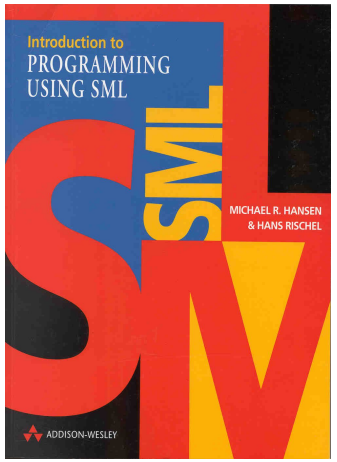
DEPARTMENT OF COMPUTER SCIENCE
UNIVERSIDADE FEDERAL DE MINAS GERAIS
FEDERAL UNIVERSITY OF MINAS GERAIS, BRAZIL

PROTOTYPING IN SML



An SML Prototype

- It is often useful to prototype the algorithms, as we try them.
  - Usually we may find problems in our design, by just trying it out on a few examples.



- When prototyping, a few hints are handy:
  - Choose a simple language. Often C is too low-level for a prototype.
    - Declarative programming is good here: prolog, SML, and even python.
  - Set aside all the unnecessary complexity.
    - You only need a proof of concept, not the final implementation, which will come in heavier artillery, such as C or, more usually, C++.

A Simple Instruction Set

```
datatype Instruction =
    UNY of string * string
  | BIN of string * string * string
  | PHI of string * string * string
  | BRZ of string * Instruction list * Instruction list
  | JMP of Instruction list
  | RET;
```

We shall consider six types of instructions: unary operations such as (a = b), binary operations, such as (a = b + c), phi-functions, branches, unconditional jumps, and return statements. We could, in principle, do without binary operations or jumps, but this would restrict a bit the expressivity of our examples. *There is always a tradeoff between complexity and expressivity*. We will not worry about the semantics of these instructions. We do not need this semantics. We only need the syntactical structure of programs.

Representing Dominance Trees

```
datatype DomTree =
      BRANCH of Instruction list * string * DomTree list
    | JUMP of Instruction list * DomTree list
```

We will represent dominance trees as collections of two types of nodes, branches and unconditional jumps. We distinguish them because we need to treat blocks that terminate in branches on a special way: they will give us new predicates to push up on the stack.

Below, in tiny fonts, we have the dominance tree that represents our running example.

val t19_19 = JUMP ([UNY ("max0", "len")], []);

val t20_22 = JUMP ([PHI ("max1", "max0", "max"), BIN ("sum1", "sum", "len")], []);

val t17_18 = BRANCH ([BIN ("b2", "len", "max")], "b2", [**t19_19**, **t20_22**]);

val t15_16 = JUMP ([BIN ("len1", "len", "1"), BIN ("p1", "p", "1")], []);

val t10_14 = BRANCH ([PHI ("p", "p0", "p1"), PHI("len", "len0", "len1"), BIN ("t", "p", "0"), BIN ("b1", "t", "EOS")], "b1", [**t15_16, t17_18**]);

val t8_9 = JUMP ([BIN ("p0", "v", "i"), UNY ("len0", "0")], [**t10_14**]);

val t23_24 = JUMP ([BIN ("maxr", "0", "max"), UNY ("sumr", "sum")], []);

val t3_7 = BRANCH ([PHI ("i", "i0", "i1"), PHI ("max", "max0", "max1"), PHI ("sum", "sum0", "sum1"), BIN ("b0", "i", "N")], "b0", [**t8_9, t23_24**]);

val t0_2 = JUMP ([UNY ("i0", "0"), UNY ("sum0", "0"), UNY ("max0", "0")], [**t3_7**]);

The Core Algorithm

```
fun vchildren [] _ _ = []
  | vchildren (n::ns) preds pdom =
      vnode n preds pdom @ vchildren ns preds pdom
and vnode n preds pdom =
      let
         val (bb, ns) = inspect n
         fun top nil = "" | top (h::t) = h
         fun pop nil = nil | pop (h::t) = t
      in
         if is_immediate_post_dom pdom n (top preds)
         then vnode n (pop preds) pdom
         else link bb (top preds) @ vchildren ns (push n preds) pdom
      end
```

This algorithm is very similar to that one which we have shown in pseudo-code. That is the beauty of prototyping on a high-level language. We must implement a few **auxiliary functions**, though.

Creating Edges – the link function

```
fun link [] _ = []
  | link _ "" = []
  | link ((UNY (a, _)) :: insts) label = (label, a) :: link insts label
  | link ((BIN (a, _, _)) :: insts) label = (label, a) :: link insts label
  | link ((PHI (a, _, _)) :: insts) label = (label, a) :: link insts label
  | link _ _ = []
```

We will be returning a list of edges, e.g., [("b0","p0"),("b1","max0"), …]. Link is the function in charge of creating the edges. It receives a list of instructions, and returns a list of edges. These edges are pairs of strings, like (source, destination). We use the empty string, e.g., "", to denote the initial label, which we pass on to the algorithm when it starts traversing the dominance tree. We only have to create edges when we visit either a unary operation, a binary operation or a phi-function. The other instructions to not ask for edges. Furthermore, once we find them, we can safely terminate the linking routine, as any of these other instructions (BRZ, JMP, RET) terminates a basic block.

Manipulating nodes of the dominance tree

```
fun push (BRANCH (_, p, _)) predicates = (p :: predicates)
  | push (JUMP (_, _)) predicates = predicates
```

Function push adds new labels onto the stack of predicates. It distinguish between the two kinds of nodes of the dominance tree, because only BRANCH nodes requires us to push new labels.

```
fun inspect (BRANCH (basicBlock, _, children)) = (basicBlock, children)
  | inspect (JUMP (basicBlock, children)) = (basicBlock, children)
```

Function inspect deconstructs a node n of the dominance tree into a pair formed by a basic block and the children of n. The basic block is a list of instructions, and children is a list of new dominance trees♡.

♡ datatype DomTree = BRANCH of Instruction list * string * DomTree list | JUMP of Instruction list * DomTree list

Testing for Post-Dominance

```
fun is_immediate_post_dom [] _ _ = false
  | is_immediate_post_dom ((nx, lbx) :: rest) n lb =
      if lb = lbx andalso n = nx
      then true
      else is_immediate_post_dom rest n lb
```

Our post-dominance test is a *stub*, i.e., it already receives a list of pair of predicates and their post-dominators, and check if a given predicate, and a given block, constitute a pair in this list. An example of such an input list is given below:

```
val pdom_ex = [(t23_24, "b0"), (t17_18, "b1"), (t20_22, "b2")]
```

Once we have all this code put together, for instance, in a file called build.sml, it is very easy to test it:

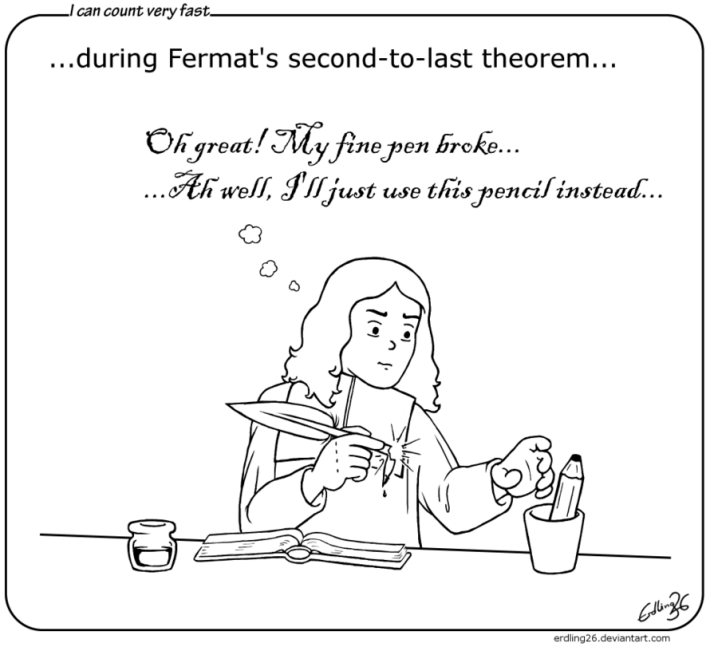
```
$> sml
- use "build.sml";
- val ans = vnode t0_2 nil pdom_ex;
. . .
```

Compilers Laboratory

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSIDADE FEDERAL DE MINAS GERAIS
FEDERAL UNIVERSITY OF MINAS GERAIS, BRAZIL

CORRECTNESS

DCC 888



Proving Correctness

We have designed an algorithm that finds control dependences between nodes. We want to prove a few properties about this algorithm.

1. We create a control edge (p, b) if, and only if, b is in the hammock region controlled by p.
2. We create at most one control edge reaching any node.

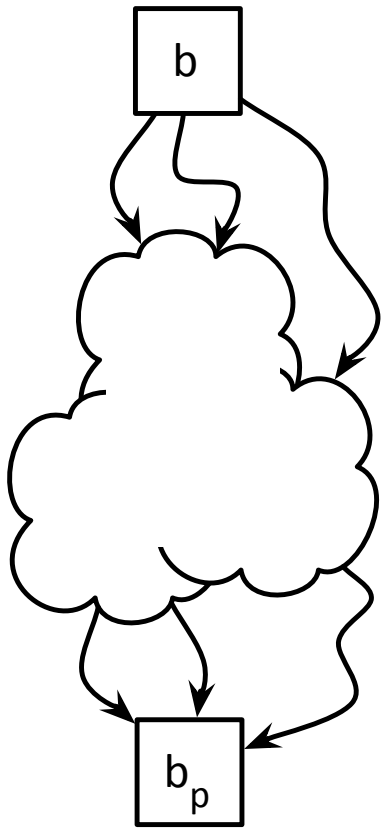
```
fun vchildren [] _ = []
   | vchildren (n::ns) preds = vnode n preds @ vchildren ns preds
and vnode n preds =
   let
      val (basic_block, ns) = inspect n
   in
      if n is the immediate post dominator of (top preds)
      then vnode n (pop preds)
      else link basic_block (top preds) @ vchildren ns (push n preds)
   end
```

a) Does (1) ensure correctness?

b) Does (2) have anything to do with correctness?

Dominance and Immediate Dominance

Lemma 1: Let b be an entry point of a minimal region in a hammock graph, and let b_p be the corresponding exit region. If b dominates b_p, then b_p is a child of b in the dominator tree◇.



1) Can b dominate b_p, without b_p being a child of b in the dom-tree?

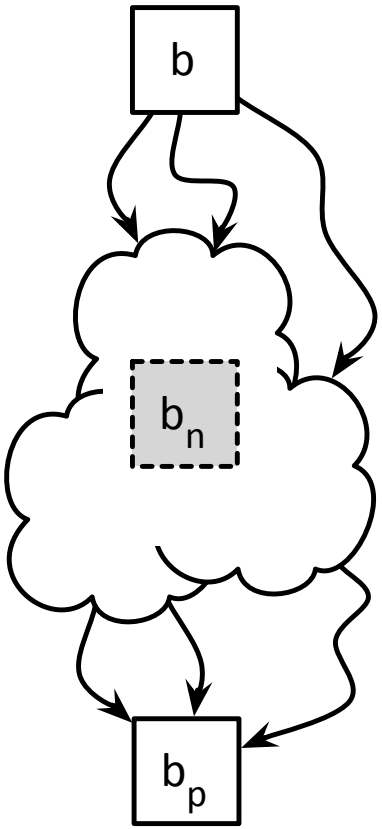
2) Does this look like something we prove by which technique? E.g.: Induction, reduction to absurd, pigeon whole principle, etc.

◇: Another way to put it: b is the immediate dominator of b_p.

Dominance and Immediate Dominance

Lemma 1: Let b be an entry point of a minimal region in a hammock graph, and let b_p be the corresponding exit region. If b dominates b_p, then b_p is a child of b in the dominator tree.

Proof: assume that there exists another node b_n in the hammock region that dominates b_p. Then b_n is the entry point of a region (b_n, b_p), and (b, b_p) is not minimal.



Towards Correctness (0)

Corollary 1: if b ends with a branch on predicate p, then the algorithm pops this predicate before visiting the post-dominator of b.

Proof: by Lemma 1, if b dominates its postdominator, then it does it strictly, and **this** step happens.

```
fun vnode n preds =
  let
    val (basic_block, ns) = inspect n
  in
    if n is the immediate post dominator of (top preds)
    then vnode n (pop preds)
    else link basic_block (top preds) @ vchildren ns (push n preds)
  end
```

Can we create a control edge between a predicate in block b, and any instruction at b_p, the post-dom of b?

Towards Correctness (1)

Lemma A: We can only create a control edge from a block n, to another block n' if n dominates n'.

Write your proof here

```
fun vchildren [] _ = []
   | vchildren (n::ns) preds = vnode n preds @ vchildren ns preds
and vnode n preds =
   let
      val (basic_block, ns) = inspect n
   in
      if n is the immediate post dominator of (top preds)
      then vnode n (pop preds)
      else link basic_block (top preds) @ vchildren ns (push n preds)
   end
```

Given a node n, inspect n returns the instructions of n, e.g., basic_block, plus the children of n, e.g., ns

Towards Correctness (1)

Lemma A: We can only create a control edge from a block n, to another block n' if n dominates n'.

Proof: we are traversing the dominator tree of the program, because ns is the set of nodes that are immediately dominated by n. Once we push n onto the stack, we can only visit children of n.

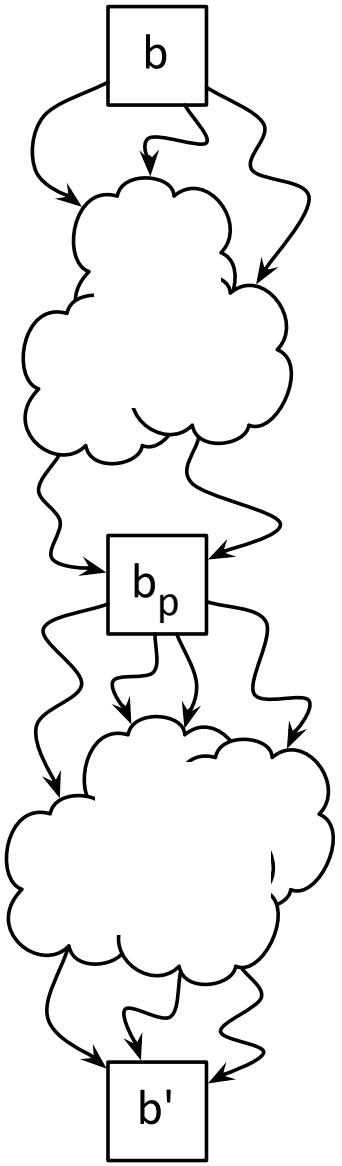
```
fun vchildren [] _ = []
   | vchildren (n::ns) preds = vnode n preds @ vchildren ns preds
and vnode n preds =
   let
      val (basic_block, ns) = inspect n
   in
      if n is the immediate post dominator of (top preds)
      then vnode n (pop preds)
      else link basic_block (top preds) @ vchildren ns (push n preds)
   end
```

Given a node n, inspect n returns the instructions of n, e.g., basic_block, plus the children of n, e.g., ns

Transitive Dominance

Lemma 2: If b, b' and b_p are blocks such that: (i) b dominates b'; (ii) b' post-dominates b; (iii) b_p is the immediate post-dominator of b; then either b_p = b' or b_p dominates b'.

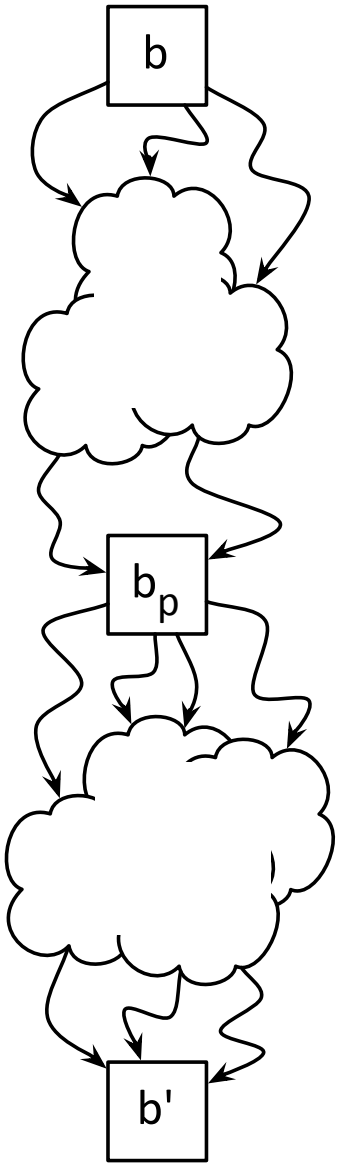
Write your proof here



Transitive Dominance

Lemma 2: If b, b' and b_p are blocks such that: (i) b dominates b'; (ii) b' post-dominates b; (iii) b_p is the immediate post-dominator of b; then either b_p = b' or b_p dominates b'.

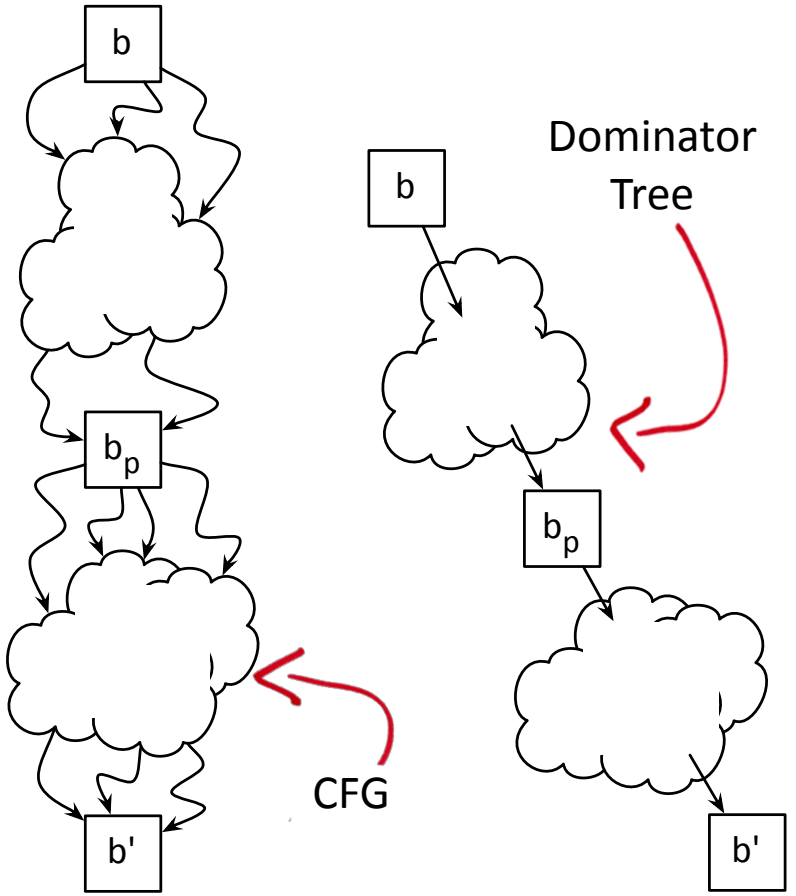
Proof: let's assume $b_p \neq b'$. If b_p does not dominate b', then there exists a path from START to b' that does not go across b_p. If this path does not touch b, then we contradict (i). If it does, then there exists a path from b to b', and from there to END, due to (ii), that does not go across b_p (remember: $b_p \neq b'$). Hence, we contradict (iii)



Towards Correctness (2)

Lemma 3: if b' post-dominates a node b, then b' is never visited with a stack that contains b.

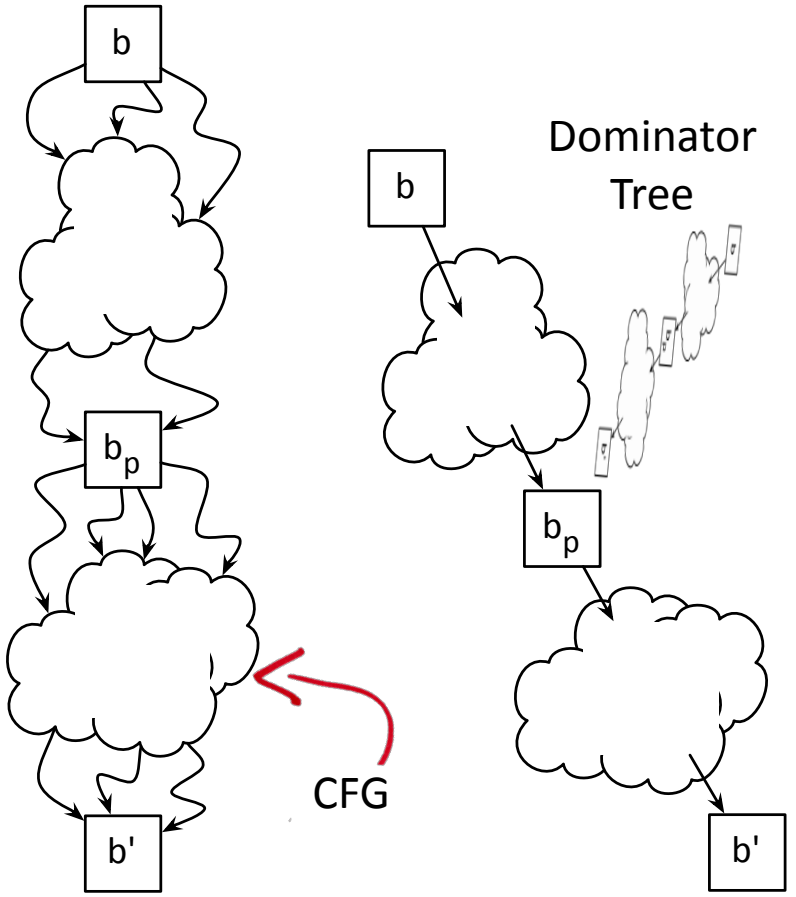
Write your proof here



Towards Correctness (2)

Lemma 3: if b' post-dominates a node b, then b' is never visited with a stack that contains b.

Proof: if b' is the immediate post-dominator of b, then we use corollary 1, or else we have two cases to analyze. If b does not dominate b', then when b' is visited, the stack will not contain p due to Lemma A, because b' is not a child of b in the dominator tree. If b dominates b', then b_p, the immediate post-dominator of b, dominates b' as well, due to Lemma 2. The predicate p will be popped once we visit b_p from b, due to corollary 1.



Towards Correctness (3)

Lemma 4: if a node b_c is in the influence region controlled by a node b, then b_c is visited with a stack that contains b.

Write your proof here

```
fun vnode n preds =
  let
    val (basic_block, ns) = inspect n
  in
    if n is the immediate post dominator of (top preds)
    then vnode n (pop preds)
    else link basic_block (top preds) @ vchildren ns (push n preds)
  end
```

Towards Correctness (3)

Lemma 4: if a node b_c is in the hammock region controlled by a node b, then b_c is visited with a stack that contains b.

Proof: If b_c is in the hammock region of b, then b dominates b_c, and b_c is under b in the dominance tree. Thus, we push in the predicate p of b before moving on to b_c. The predicate is only removed once we visit the post-dom of b, but this node is already outside b's hammock region.

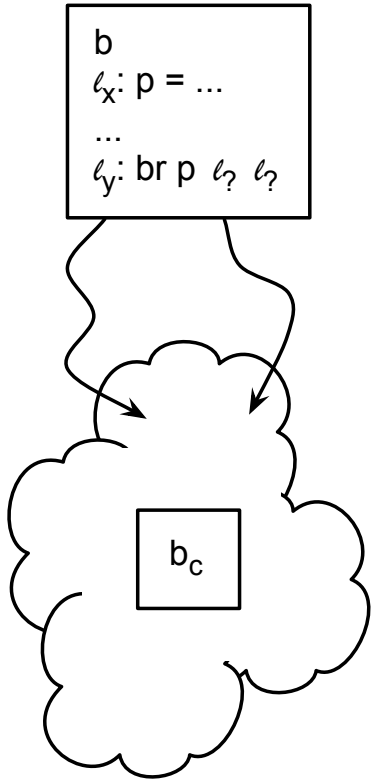
```
fun vnode n preds =
  let
    val (basic_block, ns) = inspect n
  in
    if n is the immediate post dominator of (top preds)
    then vnode n (pop preds)
    else link basic_block (top preds) @ vchildren ns (push n preds)
  end
```

Towards Correctness (4)

Theorem 1: we visit a node b_c with a stack containing a predicate p if, and only if, b_c is in the hammock region of a node b which defines p, and uses it as a predicate.

We assume that a predicate is always defined in the block where it is used. We can ensure this assumption by copying each predicate to a new variable right before the branch that it controls.

Write your proof here



Towards Correctness (4)

Theorem 1: we visit a node b_c with a stack containing a predicate p if, and only if, b_c is in the hammock region of a node b which defines p, and uses it as a predicate.

We assume that a predicate is always defined in the block where it is used. We can ensure this assumption by copying each predicate to a new variable right before the branch that it controls.

Proof (sufficiency): Lemma 4

Proof (necessity): If b_c is not in the influence region of b, then either (i) b does not dominate b_c, or (ii) b is post-dominated by a node b_p, and b_p dominates b_c. If we have (i), then the algorithm does not push p before going to b_c, because b_c is not a child of b in the dominator tree. If we have (ii), then we use Lemma 3.

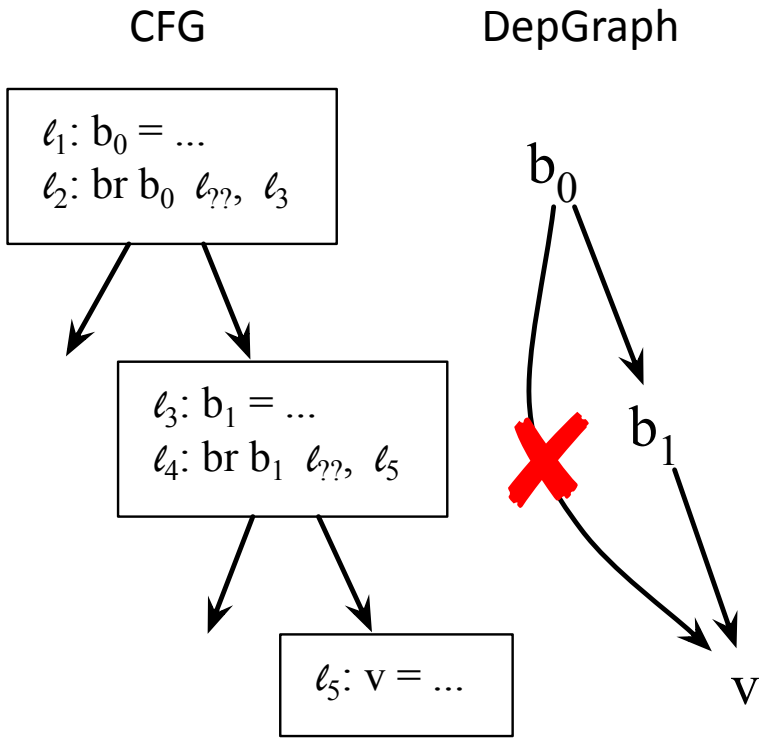
Towards Correctness (5)

Lemma 5: A node has at most one incoming control edge.

Proof: we link a node with the predicate on the top of the stack. A node is only linked once.

Corollary 3: The algorithm creates O(N) control edges, where N is the total of nodes in the control flow graph.

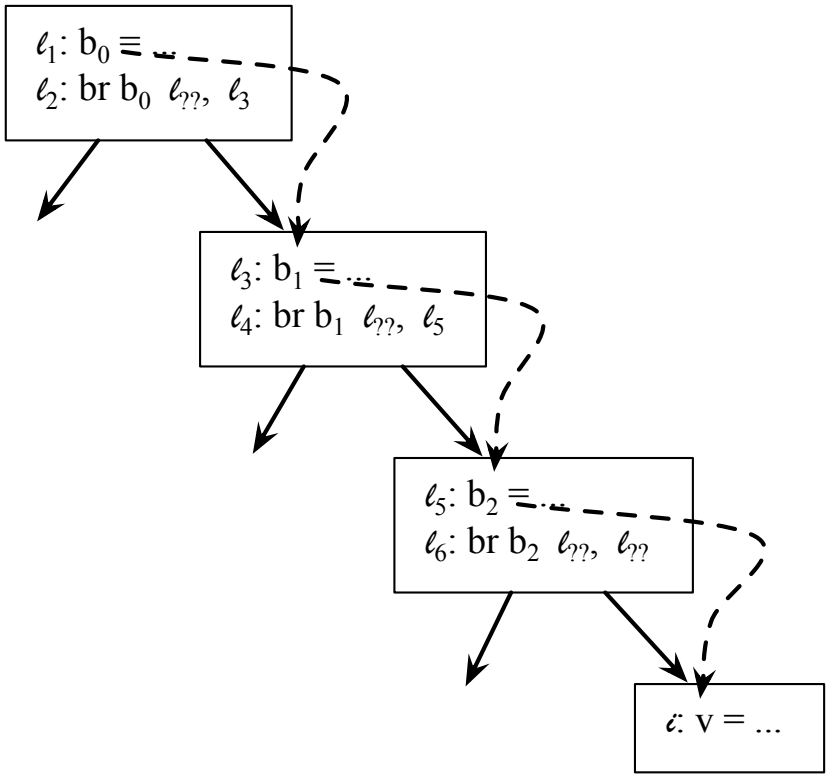
Proof: from Lemma 5, we can have at most N – 1 control edges in the dependence graph.



Transitive Dependences

Theorem 2: if p is a predicate that controls a node b_c, then there exists a chain of control dependence edges linking p to any instruction i in b_c.

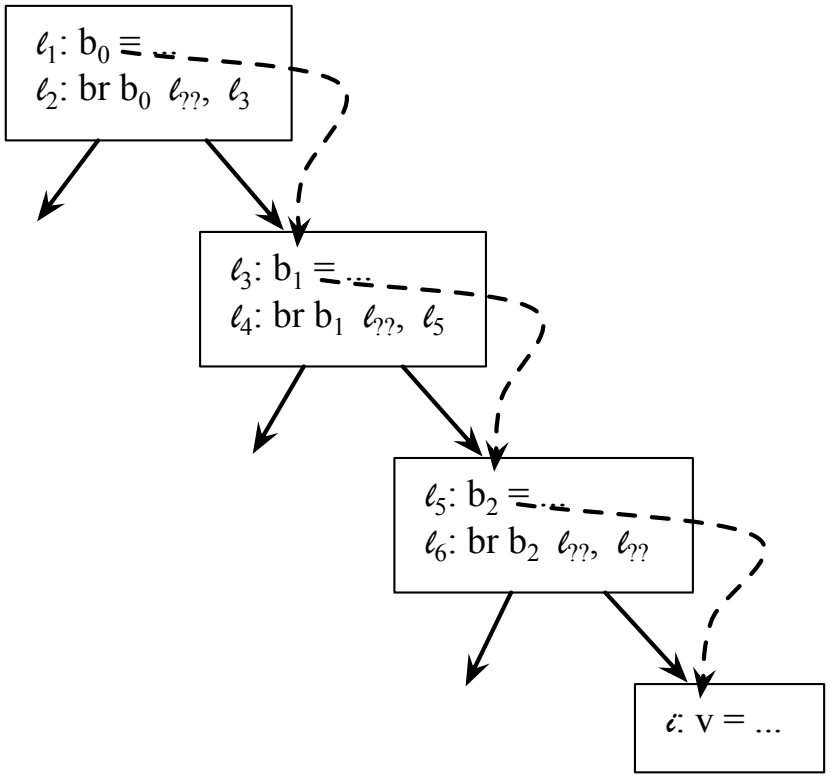
Write your proof here



Transitive Dependences

Theorem 2: if p is a predicate that controls a node b_c, then there exists a chain of control dependence edges linking p to any instruction i in b_c.

Proof: Let b be the block where p is defined. If there exist no other branch between b and b_c in the dominator three, then p will be on the stack until when b_c is visited by Theorem 1. Otherwise, let b' be the first block that contains a branch between b and b_c. Let p' be the predicate of b'. There will be an edge between p and the instruction that defines p', because p is on the top of the stack when b' is visited (Theorem 1). We repeat this reasoning **inductively** for p' and i.



A Bit of History

- The idea of program slicing seems to have appeared in a work by Weiser.
- The algorithm discussed in this presentation was taken from a paper by Rodrigues *et al*.
- The notion of a dependence graph seems to be due to Ottenstein[2].
- Control dependences were discussed in a work by Ferrante *et al*.

- Weiser, M. Program slices: formal, psychological, and practical investigations of an automatic program abstraction method, PhD thesis, University of Michigan, (1979)
- Rodrigues. B, Aranha, D. Pereira, F. A Sparse representation of implicit flows with applications to side-channel detection, CC, 110-120 (2016).
- Ottenstein, K., Ottenstein, L., The program dependence graph in a software development environment, SIGPLAN Notices 19(5):177-184, (1984)
- Ferrante, J., Ottenstein, K, Warren, J., The program dependence graph and its use in optimization. TOPLAS, 9(3):319-349, 1987