From Assembly Language To Lambda Calculus

or

How to Trick a FORTRAN Hacker Into Writing a Functional Program

Patryk Zadarnowski

University of New South Wales
Sydney

MOTIVATION

① Data-flow analysis

② Simplify compiler design

③ Communication between functional and imperative research communities

④ Common backend for FORTRAN and Haskell

⑤ Formal reasoning about program transformation

⑥ Search for better data structures!

THE TARGET LANGUAGE

Administrative Normal Form (ANF):

Restricted form of a direct-style lambda calculus:

- ➜ no nested `let`'s
- ➜ no nested function applications
- ➜ no anonymous lambda expressions

Administrative Normal Form

Syntax:

$e \; ::=$

Administrative Normal Form

$e ::= \texttt{let}\ \bar{x} = \bar{v}\ \texttt{in}\ e$

Syntax:

- copy

Administrative Normal Form

$$
\begin{aligned}
e \; ::= \; & \texttt{let}\ \bar{x} = \bar{v}\ \texttt{in}\ e \quad | \\
& \texttt{let}\ \bar{x} = v(\bar{v})\ \texttt{in}\ e
\end{aligned}
$$

Syntax:

- copy
- calls

Administrative Normal Form

$$
\begin{aligned}
e \;::=\; & \texttt{let}\ \bar{x} = \bar{v}\ \texttt{in}\ e \quad | \\
& \texttt{let}\ \bar{x} = v(\bar{v})\ \texttt{in}\ e \quad | \\
& \bar{v}
\end{aligned}
$$

Syntax:

- copy
- calls
- returns

Administrative Normal Form

$$
\begin{aligned}
e ::= \; & \texttt{let}\ \bar{x} = \bar{v}\ \texttt{in}\ e \quad | \\
& \texttt{let}\ \bar{x} = v(\bar{v})\ \texttt{in}\ e \quad | \\
& \bar{v} \quad | \\
& v(\bar{v})
\end{aligned}
$$

Syntax:

- copy
- calls
- returns
- jumps

Administrative Normal Form

$$
\begin{aligned}
e \;::=\;& \texttt{let}\ \bar{x} = \bar{v}\ \texttt{in}\ e \quad | \\
& \texttt{let}\ \bar{x} = v(\bar{v})\ \texttt{in}\ e \quad | \\
& \bar{v} \quad | \\
& v(\bar{v}) \quad | \\
& \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2
\end{aligned}
$$

Syntax:

- copy
- calls
- returns
- jumps
- branches

Administrative Normal Form

$$
\begin{aligned}
e \; ::= \;& \texttt{let}\ \bar{x} = \bar{v}\ \texttt{in}\ e \quad | \\
& \texttt{let}\ \bar{x} = v(\bar{v})\ \texttt{in}\ e \quad | \\
& \bar{v} \quad | \\
& v(\bar{v}) \quad | \\
& \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \quad | \\
& \texttt{letrec}\ \bar{f}\ \texttt{in}\ e \\
f \; ::= \;& x(\bar{x}) = e
\end{aligned}
$$

Syntax:

- copy
- calls
- returns
- jumps
- branches
- code labels

Administrative Normal Form

$$
\begin{array}{lcl}
e & ::= & \texttt{let}\ \bar{x} = \bar{v}\ \texttt{in}\ e \quad | \\
 & & \texttt{let}\ \bar{x} = v(\bar{v})\ \texttt{in}\ e \quad | \\
 & & \bar{v} \quad | \\
 & & v(\bar{v}) \quad | \\
 & & \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \quad | \\
 & & \texttt{letrec}\ \bar{f}\ \texttt{in}\ e \\
f & ::= & x(\bar{x}) = e \\
v & ::= & x \quad | \quad c
\end{array}
$$

Syntax:

- copy
- calls
- returns
- jumps
- branches
- code labels

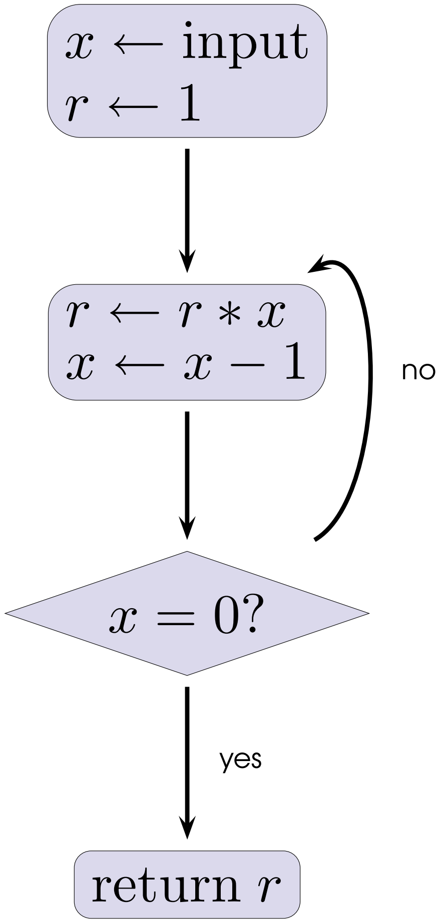
The Source Language

Control Flow Graph:

Symbolic (pre-register allocation) assembly language (three address code) organized into a control flow graph:

- ➜ Program organized as a collection of procedures (and an entry point)
- ➜ Each procedure organized as a collection of basic blocks (and an entry point)
- ➜ Each basic block consists of a sequence of assignments followed by a control transfer (`jmp` or `ret`)
- ➜ Data passed to basic blocks in preset named locations (variables or registers)

Control Flow Graphs



Basic blocks of a procedure form a directed rooted graph with nodes representing basic blocks and edges representing control transfer between basic blocks.

THE CONTROL FLOW SYNTAX

$s \quad ::=$

Syntax:

THE CONTROL FLOW SYNTAX

$s \ ::= \ e; \ \bar{p}$

Syntax:

- program

THE CONTROL FLOW SYNTAX

$$
\begin{aligned}
s &::= e;\ \bar{p} \\
p &::= \texttt{proc}\ x(\bar{x})\ \{\ e;\ \bar{b}\ \}
\end{aligned}
$$

Syntax:

- program
- procedures

The Control Flow Syntax

$$
\begin{aligned}
s &::= e;\ \bar{p} \\
p &::= \texttt{proc}\ x(\bar{x})\ \{\ e;\ \bar{b}\ \} \\
b &::= x\colon e
\end{aligned}
$$

Syntax:

- program
- procedures
- basic blocks

THE CONTROL FLOW SYNTAX

$$
\begin{array}{lcl}
s & ::= & e;\ \bar{p} \\
p & ::= & \texttt{proc}\ x(\bar{x})\ \{\ e;\ \bar{b}\ \} \\
b & ::= & x\colon e \\
e & ::= & \bar{x} = \texttt{call}\ x(\bar{v});\ e \quad | \\
 & & \bar{x} = \texttt{mov}\ \bar{v};\ e \quad | \\
 & & \texttt{jmp}\ x; \quad | \\
 & & \texttt{ret}\ \bar{v}; \quad | \\
 & & \texttt{br}\ v, x;\ e
\end{array}
$$

Syntax:

- program
- procedures
- basic blocks
- expressions

The Control Flow Syntax

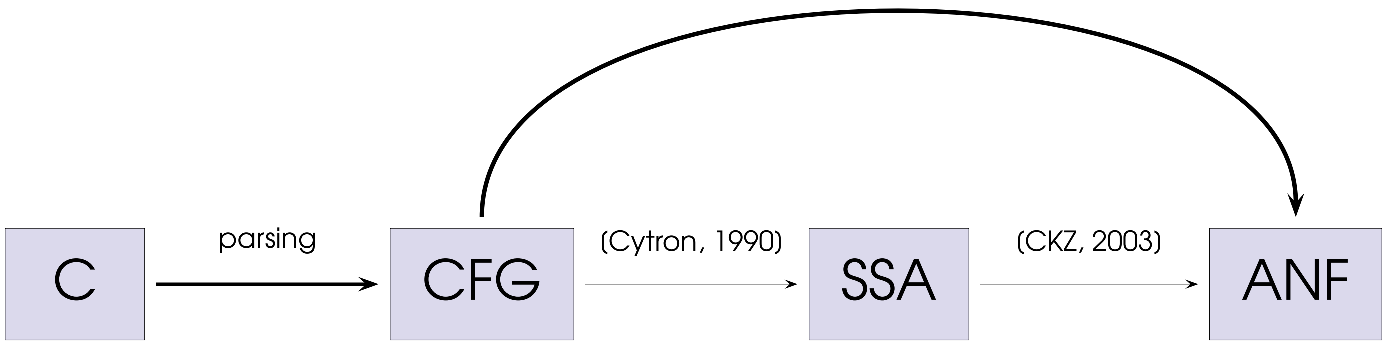
$$
\begin{aligned}
s &::= e;\ \bar{p} \\
p &::= \texttt{proc}\ x(\bar{x})\ \{\ e;\ \bar{b}\ \} \\
b &::= x\colon e \\
e &::= \bar{x} = \texttt{call}\ x(\bar{v});\ e \quad | \\
&\quad\ \ \bar{x} = \texttt{mov}\ \bar{v};\ e \quad | \\
&\quad\ \ \texttt{jmp}\ x; \quad | \\
&\quad\ \ \texttt{ret}\ \bar{v}; \quad | \\
&\quad\ \ \texttt{br}\ v, x;\ e \\
v &::= x \quad | \quad c
\end{aligned}
$$

Syntax:

- program
- procedures
- basic blocks
- expressions

Translation — Overview

TRANSLATION — OVERVIEW



- Direct
- Formal
- Efficient
- Incremental

Translation — Part 1

➜ Assignments converted to a chain of `let` bindings using name hiding to avoid renaming during translation:

$$\begin{array}{lcl} y = \texttt{add}\ x, 1; & & \texttt{let}\ y = \texttt{add}(x, 1)\ \texttt{in} \\ y = \texttt{mul}\ y, 2; & \longrightarrow & \texttt{let}\ y = \texttt{mul}(y, 2)\ \texttt{in} \\ \texttt{ret}\ y; & & y \end{array}$$

➜ Each basic block converted to a separate function.

➜ Formal parameters to block functions obtained from the list of variables occurring free in the block (including any blocks reachable from it.)

➜ Arguments to jumps syntatically identical to the parameter list.

Translation — Part 1

➜ Assignments converted to a chain of `let` bindings using name hiding to avoid renaming during translation:

$$\begin{array}{lcl} y = \texttt{add}\ x, 1; & & \texttt{let}\ y = \texttt{add}(x, 1)\ \texttt{in} \\ y = \texttt{mul}\ y, 2; & \longrightarrow & \texttt{let}\ y = \texttt{mul}(y, 2)\ \texttt{in} \\ \texttt{ret}\ y; & & y \end{array}$$

➜ Each basic block converted to a separate function.

➜ Formal parameters to block functions obtained from the list of variables occurring free in the block (including any blocks reachable from it.)

➜ Arguments to jumps syntatically identical to the parameter list.

☑ Simple and fast

☑ Results in (almost) lambda-lifted program

✗ Very long parameter lists

Translation — Part 2

Minimizing Parameter Lists:

➜ Nest function definitions!

- Variables defined in the environment do not need to be passed as a parameter.

➜ Function f defined within g iff all control-flow paths to f lead through g.

➜ This is known as the *dominance property.* (Lowry, 1969)

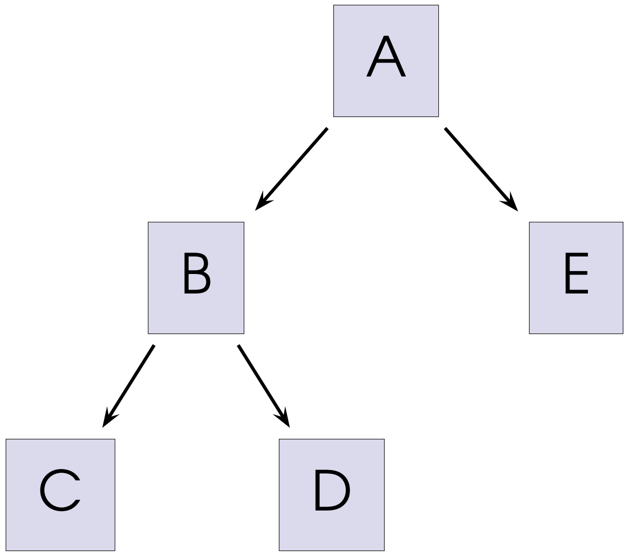
➜ Can be formalized as:

$$\vdash \texttt{start} \geq x$$

$$\forall z.z \rightarrow y \implies z \geq x \vdash x \geq y$$

Dominator Trees for ANF — 1

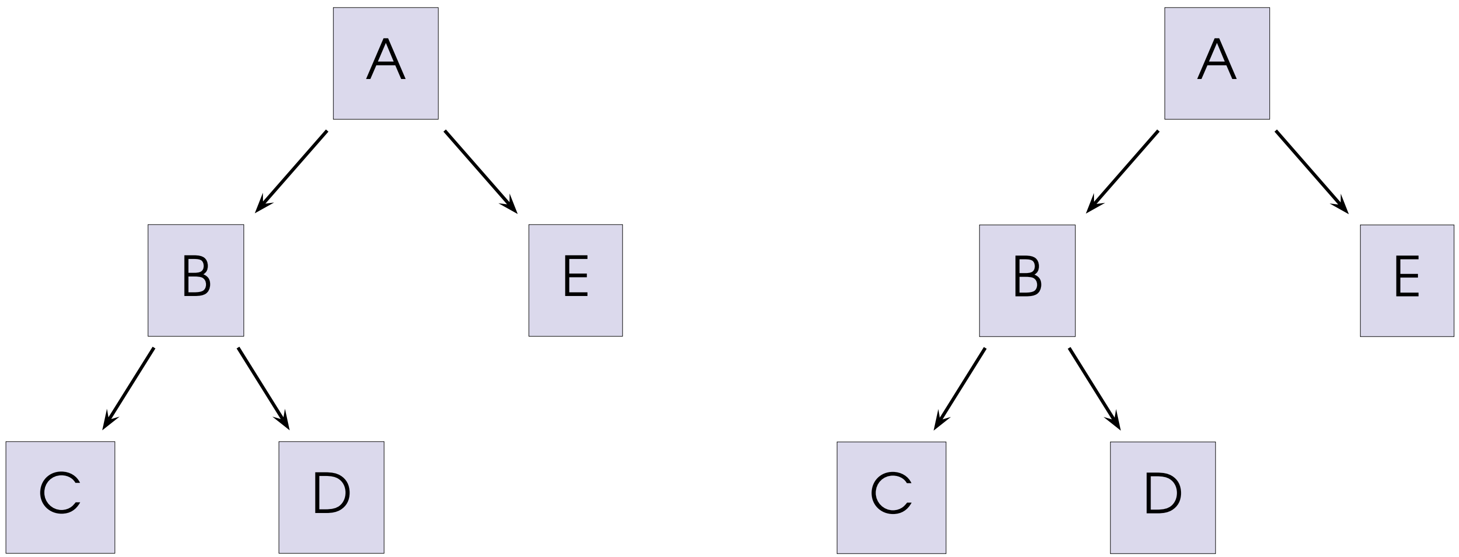
Tree Programs:



CFG

Dominator Trees for ANF — 1

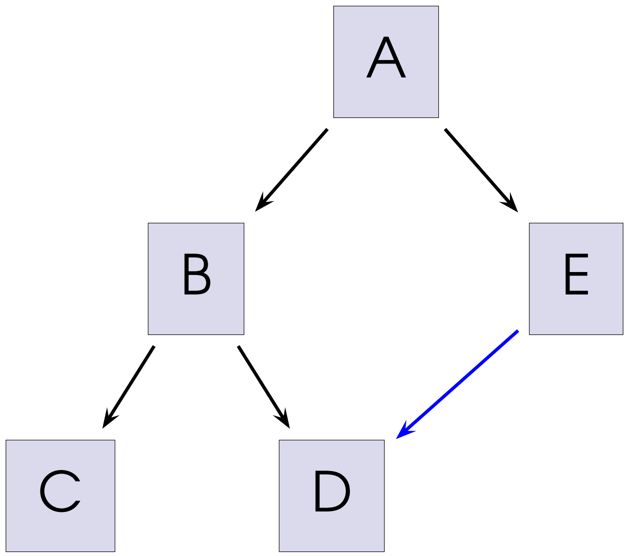
Tree Programs:



CFG DT

DOMINATOR TREES FOR ANF — 2

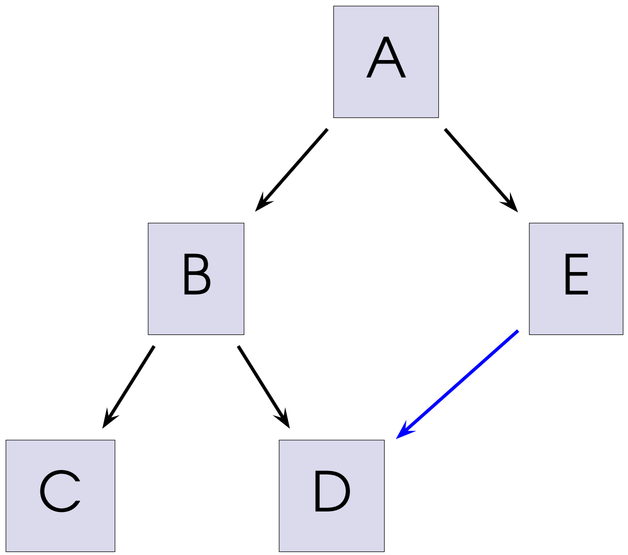
Cross-jumps:



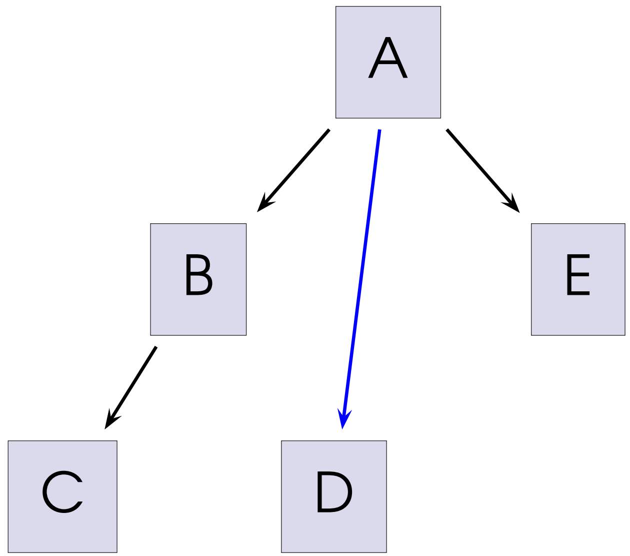
CFG

Dominator Trees for ANF — 2

Cross-jumps:



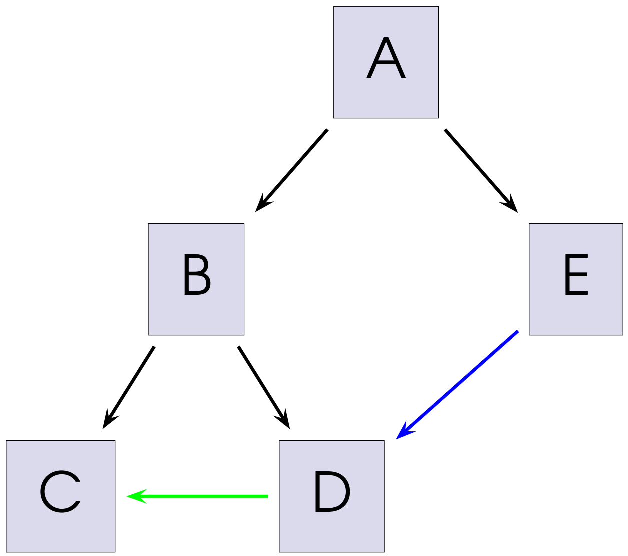
CFG



DT

Dominator Trees for ANF — 3

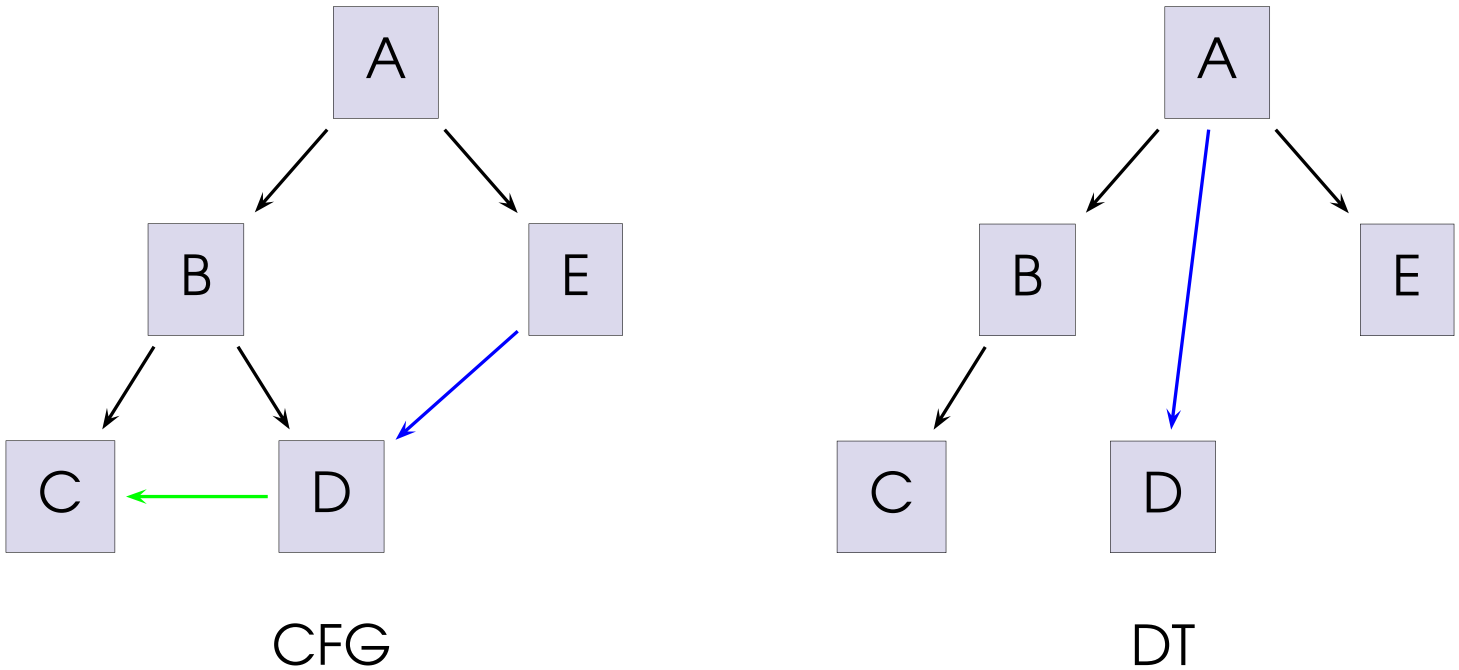
Sibling edges:



CFG

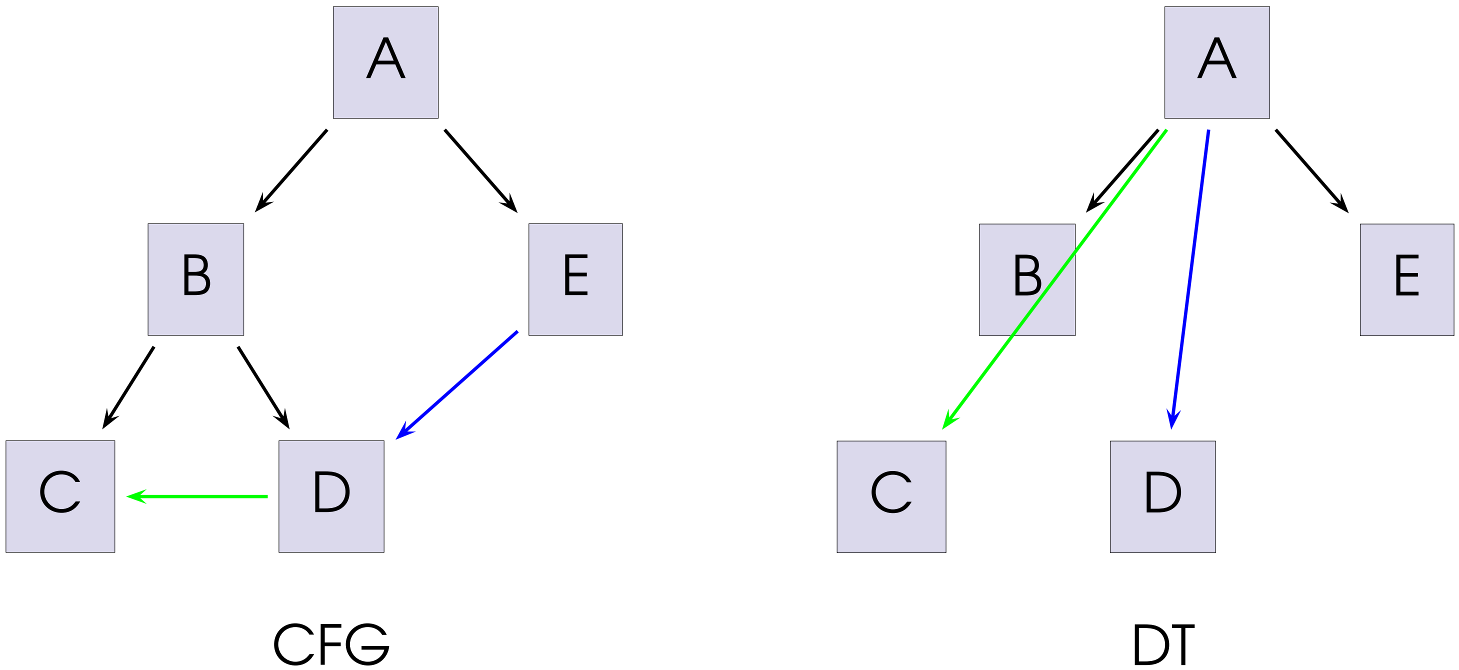
Dominator Trees for ANF — 3

Sibling edges:



DOMINATOR TREES FOR ANF — 3

Sibling edges:



Implementation

```
insert dc dt (src, dst)
  | processed dt dst = lift dc dt (src, dst)
  | otherwise = process dst (dc ∪ dst) (dt ∪ (src, dst))

lift dc dt (src, dst) =
  | (parent dt dst) ∈ dc = dt
  | otherwise = lift dc (lift' dt (src, dst)) (src, dst)

lift' dt src dst =
  foldS lift" (dt ∪ (parent dt (parent dt dst)), dst) cfg dst

lift" dt sib =
  | parent dt sib == parent dt dst = lift' dt sib
  | otherwise = dt
```

Evaluation

- ➜ Simple: 15 lines of Haskell code.
- ➜ Fast: $O(n^2)$.
- ➜ Incremental: the dominator tree always valid for the subset of the CFG explored so far.
- ➜ The traversal order doesn't affect the output
- ➜ . . . but it affects complexity!

USING THE INFORMATION

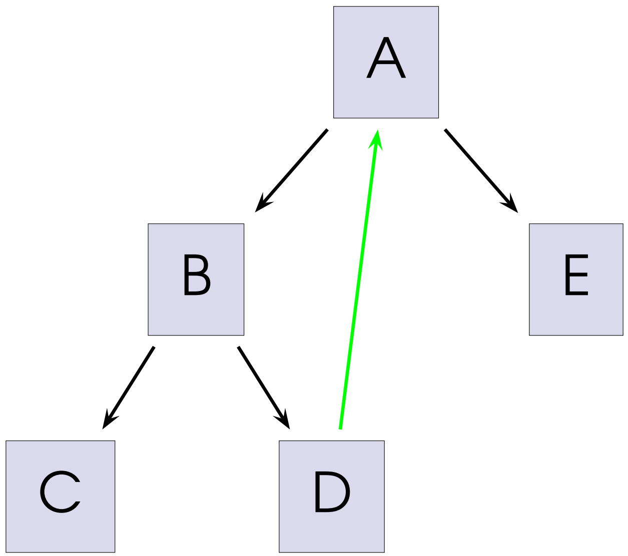
① Traverse the CFG constructing the dominator tree for the procedure.

② Compute the list of formal parameters for each function.

③ Traverse the DT constructing an ANF function for each block x:

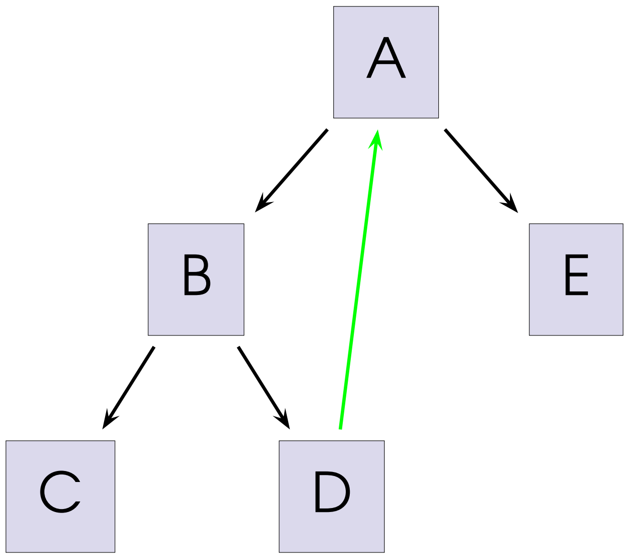
- For leaf blocks, simply translate the expression as above.
- Otherwise, create a `letrec` defining functions for all blocks dominated by x, with the translated expression in the `letrec` body.

Parameter Computation



DT

Parameter Computation



DT

- Parameters of A include all variables redefined on the path from A to D (along the DT spine).
- In general, parameters to x include all variables redefined along the dominator chain to x from the common dominator of each caller and x.
- Can this information be maintained while computing the DT?

Conclusions

- Dominators: can be formalized.
- The SSA form: reduntant.
- Lambda calculus: usable as a low-level representation.
- GHC: doesn't need an imperative back-end.

Future Work:

- Implementation in FunCC.
- One pass translation from a parse tree.
- Improvements?