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!
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**

\[ e ::= \text{let } \bar{x} = \bar{v} \text{ in } e \]

**Syntax:**

- **copy**
**Administrative Normal Form**

\[ e ::= \text{let } \bar{x} = \bar{v} \text{ in } e \mid \text{let } \bar{x} = v(\bar{v}) \text{ in } e \]

**Syntax:**
- copy
- calls
**Administrative Normal Form**

\[ e ::= \text{let } \bar{x} = \bar{v} \text{ in } e \mid \text{let } \bar{x} = v(\bar{v}) \text{ in } e \mid \bar{v} \]

**Syntax:**
- copy
- calls
- returns
ADMINISTRATIVE NORMAL FORM

Syntax:

- copy
- calls
- returns
- jumps
**Administrative Normal Form**

\[
e ::= \text{let } \bar{x} = \bar{v} \text{ in } e \mid \\
\text{let } \bar{x} = v(\bar{v}) \text{ in } e \mid \\
\bar{v} \mid \\
v(\bar{v}) \mid \\
\text{if } v \text{ then } e_1 \text{ else } e_2
\]

**Syntax:**

- copy
- calls
- returns
- jumps
- branches
**Administrative Normal Form**

\[
e ::= \text{let } \bar{x} = \bar{v} \text{ in } e \mid \text{let } \bar{x} = v(\bar{v}) \text{ in } e \mid \bar{v} \mid v(\bar{v}) \mid \text{if } v \text{ then } e_1 \text{ else } e_2 \mid \text{letrec } \bar{f} \text{ in } e
\]

\[
f ::= x(\bar{x}) = e
\]

**Syntax:**
- copy
- calls
- returns
- jumps
- branches
- code labels
**Administrative Normal Form**

\[ e ::= \text{let } \bar{x} = \bar{v} \text{ in } e \mid \text{let } \bar{x} = v(\bar{v}) \text{ in } e \mid \bar{v} \mid v(\bar{v}) \mid \text{if } v \text{ then } e_1 \text{ else } e_2 \mid \text{letrec } \bar{f} \text{ in } e \]

\[ f ::= x(\bar{x}) = e \]

\[ v ::= x \mid c \]

**Syntax:**

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

```
\[
x \leftarrow \text{input} \\
r \leftarrow 1
\]
```

```
\[
r \leftarrow r \times x \\
x \leftarrow x - 1
\]
```

```
\[
x = 0? \\
\]
```

```
\[
\text{yes} \\
\text{return } r
\]
```

```
\[
\text{no}
\]
```
THE CONTROL FLOW SYNTAX

Syntax:

s ::=
THE CONTROL FLOW SYNTAX

Syntax:

• program

\[ s ::= e; \overline{p} \]
THE CONTROL FLOW SYNTAX

Syntax:

- program
- procedures

\[ s ::= e; \bar{p} \]
\[ p ::= \text{proc } x(x) \{ e; \bar{b} \} \]
Syntax:

- program
- procedures
- basic blocks

The Control Flow Syntax

\[
\begin{align*}
s & ::= e; \overline{p} \\
p & ::= \text{proc } x(\overline{x}) \{ e; \overline{b} \} \\
b & ::= x: e
\end{align*}
\]
The Control Flow Syntax

Syntax:

- program
- procedures
- basic blocks
- expressions
**The Control Flow Syntax**

```
\[ 
\begin{align*}
  s &::= e; \bar{p} \\
  p &::= \text{proc } x(\bar{x}) \{ e; \bar{b} \} \\
  b &::= x: e \\
  e &::= \bar{x} = \text{call } x(\bar{v}); e \mid \\
  &\quad \bar{x} = \text{mov } \bar{v}; e \mid \\
  &\quad \text{jmp } x; \mid \\
  &\quad \text{ret } \bar{v}; \mid \\
  &\quad \text{br } v, x; e \\
  v &::= x \mid c
\end{align*} \]
```

**Syntax:**
- program
- procedures
- basic blocks
- expressions
TRANSLATION — OVERVIEW

C → CFG → SSA → ANF

parsing (Cytron, 1990) (CKZ, 2003)
**Translation — Overview**

- Direct
- Formal
- Efficient
- Incremental

Diagram:

```
C --> parsing --> CFG --> SSA --> ANF
(Cytron, 1990)   (CKZ, 2003)
```
Translation — Part 1

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

  \[
  \begin{align*}
  y &= \text{add } x, 1; \\
  y &= \text{mul } y, 2; \\
  \text{ret } y;
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{let } y &= \text{add}(x, 1) \quad \text{in} \\
  \text{let } y &= \text{mul}(y, 2) \quad \text{in} \\
  y
  \end{align*}
  \]

- 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 syntactically identical to the parameter list.
Translations: Part 1

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

  \[ y = \text{add} \, x, 1; \quad \text{let } y = \text{add}(x, 1) \text{ in} \]

  \[ y = \text{mul} \, y, 2; \quad \rightarrow \quad \text{let } y = \text{mul}(y, 2) \text{ in} \]

  \[ \text{ret } y; \quad \text{in} \]

- 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 syntactically identical to the parameter list.

- Simple and fast
- Results in (almost) lambda-lifted program
- Very long parameter lists
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:

\[
\vdash \text{start} \geq x \\
\forall z. z \rightarrow y \implies z \geq x \vdash x \geq y
\]
Dominator Trees for ANF — 1

Tree Programs:

```
        A
       / \  
      B   E
     /     |
    C       D
```

CFG
DOMINATOR TREES FOR ANF — 1

Tree Programs:

```
A
  |
  V
B
  |
  V
C D
  |
  V
CFG
```

```
A
  |
  V
B
  |
  V
C D
  |
  V
DT
```
Dominator Trees for ANF — 2

Cross-jumps:

```
A
  / \  /
B   E
  \ / \   \\
 C   D
```

CFG
DOMINATOR TREES FOR ANF — 2

Cross-jumps:

CFG

DT
Dominator Trees for ANF — 3

Sibling edges:

```
A
  /\       /
 B   E     D
  \ /     /\   
 C     CFG  D
```
Sibling edges:
Dominator Trees for ANF — 3

Sibling edges:

CFG

DT
IMPLEMENTATION

\[
\text{insert } dc \ dt \ (src, \ dst) = \\
\quad | \text{processed } dt \ dst = lift \ dc \ dt \ (src, \ dst) \\
\quad | \text{otherwise } = \text{process } dst \ (dc \cup dst) \ (dt \cup (src, \ dst))
\]

\[
lift \ dc \ dt \ (src, \ dst) = \\
\quad | (parent \ dt \ dst) \in dc = dt \\
\quad | \text{otherwise } = lift \ dc \ (lift' dt (src, \ dst)) \ (src, \ dst)
\]

\[
lift' \ dt \ src \ dst = \\
\quad \text{foldS lift" } (dt \cup (parent \ dt \ (parent \ dt \ dst)), \ dst) \ \text{cfg dst}
\]

\[
lift" \ dt \ sib = \\
\quad | \text{parent } dt \ sib == \text{parent } dt \ dst = lift' dt \ sib \\
\quad | \text{otherwise } = dt
\]
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

PARAMETER COMPUTATION
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: redundant.
➜ 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?