SSA, CPS, ANF: THREE APPROACHES TO COMPILING FUNCTIONAL LANGUAGES

PATRYK ZADARNOWSKI

patrykz@cse.unsw.edu.au
INTRODUCTION

→ Intermediate Representation:
   A *programming language used by the compiler to represent the source program internally through the translation process:*

   \[ source \rightarrow HIR \rightarrow LIR \rightarrow assembly \]

→ Features:
Intermediate Representation:
A *programming language used by the compiler to represent the source program internally through the translation process:*

```
source → HIR → LIR → assembly
```

Features:
- complete
Intermediate Representation:
A programming language used by the compiler to represent the source program internally through the translation process:

source $\rightarrow$ HIR $\rightarrow$ LIR $\rightarrow$ assembly

Features:
- complete
- flexible
Intermediate Representation:
A *programming language used by the compiler to represent the source program internally through the translation process:*

\[ \text{source} \rightarrow \text{HIR} \rightarrow \text{LIR} \rightarrow \text{assembly} \]

Features:
- complete
- flexible
- well-defined
**INTRODUCTION**

→ Intermediate Representation:
   A *programming language used by the compiler to represent the source program internally through the translation process:*

   \[ \text{source} \rightarrow \text{HIR} \rightarrow \text{LIR} \rightarrow \text{assembly} \]

→ Features:
   → complete
   → flexible
   → well-defined
   → robust
INTRODUCTION

➜ Intermediate Representation:
   A programming language used by the compiler to represent the source program internally throughout the translation process:

   source $\rightarrow$ HIR $\rightarrow$ LIR $\rightarrow$ assembly

➜ Features:
   ➜ complete
   ➜ flexible
   ➜ well-defined
   ➜ robust
   ➜ portable
Intermediate Representation:
A *programming language* used by the compiler to represent the source program internally through the translation process:

\[ \text{source} \rightarrow \text{HIR} \rightarrow \text{LIR} \rightarrow \text{assembly} \]

Features:
- complete
- flexible
- well-defined
- robust
- portable

Compilation by correctness-preserving transformations
set i, 0
L1: add i, i, 1
cmp t, i, 5
breq L1
DATA FLOW ANALYSIS

- Needed by many (most?) optimization algorithms
- Expensive to compute
- Reusable
- Should be stored in the IR
DATA FLOW ANALYSIS

- Needed by many (most?) optimization algorithms
- Expensive to compute
- Reusable
- Should be stored in the IR
  - D-U chains (Dragon Book)
  - SSA (most traditional compilers)
  - Lambda calculus
DATA FLOW ANALYSIS

- Needed by many (most?) optimization algorithms
- Expensive to compute
- Reusable
- Should be stored in the IR
  - D-U chains (Dragon Book)
  - SSA (most traditional compilers)
  - Lambda calculus
    - CPS (most Scheme & ML compilers)
    - ANF (GHC, TIL)
SSA — Static Single Assignment Form

- Only one definition of each variable
- Every definition “dominates” each use
SSA — Static Single Assignment Form

- Only one definition of each variable
- Every definition “dominates” each use
- Example:

  \[
  \begin{align*}
  i_0 &= 0 \\
  L1: \quad i_1 &= \phi i_0, i_2 \\
  i_2 &= \text{add} \ i_1, 1 \\
  t &= \text{sub} \ i_2, 5 \\
  \text{breq} \ L1
  \end{align*}
  \]
SSA — Static Single Assignment Form

- Only one definition of each variable
- Every definition “dominates” each use
- Example:

  \[
  \begin{align*}
  i0 &= 0 \\
  \text{L1: } i1 &= \phi i0, i2 \\
  i2 &= \text{add } i1, 1 \\
  t &= \text{sub } i2, 5 \\
  \text{breq L1}
  \end{align*}
  \]

- Quasi-functional three-address code
- Still uses control flow graph
- Explicit data-flow information
SSA — EVALUATION

➤ Uses:
   ➤ Simplifies data-flow based algorithms
   ➤ New optimization opportunities
SSA — Evaluation

➡ Uses:
  ➡ Simplifies data-flow based algorithms
  ➡ New optimization opportunities

➡ Pitfalls:
  ➡ Arrays difficult to represent
  ➡ Poor choice for formal reasoning
  ➡ Horrible for strong typing
  ➡ Can’t move code across basic block
CPS — Continuation-Passing Style

- A lambda-calculus variant
- All control flow information explicit
- Only tail-calls allowed
CPS — CONTINUATION-PASSING STYLE

- A lambda-calculus variant
- All control flow information explicit
- Only tail-calls allowed
- Purely-functional
- Data-flow information explicit
- Each function takes the rest of the program as an argument
CPS — EVALUATION

→ Uses:
  → Good for formal reasoning
  → Can express more optimization algorithms
  → Easily expresses even most complex control flow constructs (longjmp, exceptions)
**CPS — Evaluation**

- **Uses:**
  - Good for formal reasoning
  - Can express more optimization algorithms
  - Easily expresses even most complex control flow constructs (longjmp, exceptions)

- **Pitfalls:**
  - Verbose!!!
  - Too much information confuses function returns and jumps
  - Most control-flow information redundant
  - Encourages repeated analysis along each execution path
  - Difficult to translate into assembly language
ANF — A-NORMAL FORM

- Direct-style lambda-calculus
- Each subexpression named explicitly
- Normal and tail-calls distinguished
ANF — A-normaL Form

- Direct-style lambda-calculus
- Each subexpression named explicitly
- Normal and tail-calls distinguished
- Example:

```ml
let fun(i0) =
  let i1 = add i0, 1 in
  let t = sub i1, 5 in
  if0 t then 0 else fun i1
in fun 0
```
ANF — A-NORMAL FORM

- Direct-style lambda-calculus
- Each subexpression named explicitly
- Normal and tail-calls distinguished
- Example:
  
  ```
  let fun(i0) =
    let i1 = add i0, 1 in
    let t = sub i1, 5 in
    if0 t then 0 else fun i1
  in fun 0
  ```

- Purely-functional
- Explicit data-flow information
- Control-flow driven by data-flow
- Similar to three-address code
ANF — EVALUATION

➜ Uses:
  ➜ Perfect for data flow algorithms
  ➜ Easy to type-check
  ➜ No duplication of mechanisms for intra- and inter-procedural data flow
  ➜ Trivial to translate into assembly language
ANF — EVALUATION

➤ Uses:
   ➤ Perfect for data flow algorithms
   ➤ Easy to type-check
   ➤ No duplication of mechanisms for intra- and inter-procedural data flow
   ➤ Trivial to translate into assembly language

➤ Pitfalls:
   ➤ Complex control-flow constructs hard to express
COMPARISON

→ SSA-CPS by Kelsey
→ SSA-ANF by Chakravarty, Keller & Zadarnowski

→ Flexibility:
  ① CPS
  ② ANF
  ③ SSA

→ ANF ideal w.r.t. efficiency-to-flexibility tradeoff
BIBLIOGRAPHY

Appel *Modern Compiler implementation in ML*

Chakravarty, Keller, Zadarnowski *A Functional Perspective on SSA Optimization Algorithms*

Cytron, et.al. *Efficiently computing SSA form and the control dependance graph*

Flanagan, Sabry, Duba and Felleisen *The Essence of Compiling with Continuations*

Kelsey *A correspondence between continuation-passing style and SSA form*