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 throught the translation process:

  source $\longrightarrow$ *HIR* $\longrightarrow$ *LIR* $\longrightarrow$ *assembly*

- Features:

INTRODUCTION

➜ Intermediate Representation:

A programming language used by the compiler to represent the source program internally throught the translation process:

source $\longrightarrow$ *HIR* $\longrightarrow$ *LIR* $\longrightarrow$ *assembly*

➜ Features:

➜ complete

INTRODUCTION

- Intermediate Representation:

  A programming language used by the compiler to represent the source program internally throught the translation process:

  source $\longrightarrow$ *HIR* $\longrightarrow$ *LIR* $\longrightarrow$ *assembly*

- Features:
  - complete
  - flexible

INTRODUCTION

➜ Intermediate Representation:

A programming language used by the compiler to represent the source program internally throught the translation process:

source $\longrightarrow$ *HIR* $\longrightarrow$ *LIR* $\longrightarrow$ *assembly*

➜ Features:

- ➜ complete
- ➜ flexible
- ➜ well-defined

INTRODUCTION

➜ Intermediate Representation:

A programming language used by the compiler to represent the source program internally throught the translation process:

source $\longrightarrow$ *HIR* $\longrightarrow$ *LIR* $\longrightarrow$ *assembly*

➜ Features:

- ➜ complete
- ➜ flexible
- ➜ well-defined
- ➜ robust

INTRODUCTION

- Intermediate Representation:

  A programming language used by the compiler to represent the source program internally throught the translation process:

  source $\longrightarrow$ *HIR* $\longrightarrow$ *LIR* $\longrightarrow$ *assembly*

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

INTRODUCTION

- Intermediate Representation:

  A programming language used by the compiler to represent the source program internally throught the translation process:

  source $\longrightarrow$ *HIR* $\longrightarrow$ *LIR* $\longrightarrow$ *assembly*

- Features:
  - complete
  - flexible
  - well-defined
  - robust
  - portable
- Compilation by correctness-preserving transformations

Example — Three Address Code

```
      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:

```
     i0 = 0
L1:  i1 = φ i0, i2
     i2 = add i1, 1
     t = sub i2, 5
     breq L1
```

SSA — Static Single Assignment Form

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

```
     i0 = 0
L1:  i1 = φ i0, i2
     i2 = add i1, 1
     t = sub i2, 5
     breq L1
```

- 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 explicitely
- ➜ Normal and tail-calls distinguished

ANF — A-normal Form

- Direct-style lambda-calculus
- Each subexpression named explicitely
- 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
```

ANF — A-normal Form

- Direct-style lambda-calculus
- Each subexpression named explicitely
- 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*