

CSE 231: ANF Conversion

Ranjit Jhala May 18, 2026

Normal Forms

A key step in compilation is the conversion to **A-Normal Form (ANF)** where, informally speaking, each call or primitive operation's arguments are **immediate** values, i.e. constants or variable lookups whose values can be loaded with a single machine instruction.

```
((2 + 3) * (12 - 4)) * (7 + 8)
```

QUIZ: *What is the A-Normal Form of the above?*

Source Language

Lets start with a source language that we wish to work with:

```
type Id = String;

enum Op { Add, Sub, Mul }

enum Exp {
  Var(String),           // vars: x, y, ...
  Num(i32),              // numbers: 0, 1, ...
  Bin(Op, Box<Exp>, Box<Exp>), // binops
  Let(Id, Box<Exp>, Box<Exp>), // let-binders
}
```

The language, defined by `Exp`, has integers, variables, binary operators, and let-binders. For example, the source expression above corresponds to:

```
// ((2 + 3) * (12 - 4)) * (7 + 8)
fn exp0() -> Exp {
  bin(Mul, bin(
    Mul,
    bin(Add, num(2), num(3)),
    bin(Sub, num(12), num(4)),
  ),
```

```

        bin(Op::Add, num(7), num(8)),
    )
}

```

A-Normal Form

Before we can describe a *conversion* to A-Normal Form (ANF), we must pin down what ANF *is*. Our informal description was:

Immediate Values: “constants or variable lookups whose values can be loaded with a single machine instruction”

ANF Values: “each call or primitive operation’s arguments are *immediate* values, i.e. constants or variable lookups”

We can encode these as Rust predicates `is_imm` and `is_anf`

```

impl Exp {
    #[spec(fn(&Exp[@e]) -> bool[e.imm])]
    fn is_imm(&self) -> bool {
        match self {
            Var(_) => true,
            Num(_) => true,
            Bin(_, _, _) => false,
            Let(_, _, _) => false,
        }
    }

    #[spec(fn(&Exp[@e]) -> bool[e.anf])]
    fn is_anf(&self) -> bool {
        match self {
            Var(_) => true,
            Num(_) => true,
            Bin(_, e1, e2) => e1.is_imm() && e2.is_imm(),
            Let(_, e1, e2) => e1.is_anf() && e2.is_anf(),
        }
    }
}

```

We *could* define brand new types, say `ImmExp` and `AnfExp`, whose values correspond to *immediate* and *ANF* terms. Unfortunately, doing so leads to a bunch of code duplication, e.g. duplicate printers for `Exp` and `AnfExp`. Try it, as an exercise.

ANF Conversion: Intuition

Recall that our goal is to convert expressions like

```
((2 + 3) * (12 - 4)) * (7 + 8)
```

into a let-bound expression of the form

```
(let (?tmp0 (+ 2 3))
  (let ?tmp1 = (- 12 4) in
    (let ?tmp2 = (* ?tmp0 ?tmp1) in
```

```
(let ?tmp3 = (+ 7 8) in
  (* ?tmp2 ?tmp3))))
```

Generalizing a bit, we want to convert

```
(+ e1 e2)
```

into a sequence of let-bindings that introduce the names needed to make the arguments e1 and e2 immediate

```
(let (x1 a1) ... (let (xn an)
  (let (x1' a1') ... (let (xm' am')
    (+ v1 v2) ... )
  )
```

where v1 and v2 are immediate, and each ai is ANF.

Forcing Arguments to be Immediate

The key requirement is a way to **force** arbitrary *argument expressions* like e1 into a **pair**:

- a vector of bindings [(x1, a1), ..., (xn, an)] where each ai is Anf, and
- an immediate expression v1 of type Imm.

so e1 is *equivalent* to (let (x1 a1) ... (let (xn an) v1)).

Thus, we need a method to **make arguments immediate**, yielding a top-level conversion method:

```
fn to_imm(&self,
  count: &mut usize,
  binds: &mut Vec<Id, Anf>
) -> Imm;

fn to_anf(&self, count: &mut usize) -> Anf;
```

QUIZ: Making Arguments Immediate

```
fn to_imm(&self, count: &mut usize,
  binds: &mut RVec<Id, Anf>) -> Imm {
  match self {
    Exp::Var(_) | Exp::Num(_) =>

    Exp::Bin(op, e1, e2) => {

  }
  Exp::Let(..) => {
```

```

    }
  }
}

```

- **Numbers and variables** are already immediate, and are returned directly.
- **Binary operators** recursively make their operands immediate, generate a fresh binder for the operation, and return the fresh variable.
- **Let-binders** are converted to ANF and assigned to a fresh binder.

Fresh Variables

We use a `count: &mut usize` to produce fresh names:

```

fn fresh(count: &mut usize) -> Id {
  let name = format!("?tmp{}", count);
  *count += 1;
  name.to_string()
}

```

QUIZ: Top-Level Conversion: to_anf

```

fn to_anf(&self, count: &mut usize) -> Anf {
  match self {
    Exp::Var(_) | Exp::Num(_) =>

    Exp::Let(x, e1, e2) => {

    }

    Exp::Bin(op, e1, e2) => {

    }

  }
}

```

In `to_anf`, the real work happens inside `to_imm` which takes an arbitrary *argument* expression and makes it **immediate** by generating temporary (ANF) bindings.

The resulting bindings (and immediate values) are composed by popping from the binds vector and wrapping the result with `let_` to stitch them into a single Anf expression.