15-411: Tail Call Optimization

Jan Hoffmann

```
int powacc (int b, int e, int a)
{
  if (e == 0)
    return a;
  else
    return powacc(b,e-1,a*b);
}

int pow(int b, int e)
{
  return powacc(b,e,1)
}
```

```
int powloop(int b, int e)
{
  int acc = 1;
  while (e>0)
  {
    e = e - 1;
    acc = acc * b;
  }
  return acc;
}
```

Which implementation is more efficient?

Two Implementations of the Power Function

Compiling Tail Calls

Function overhead mainly comes from maintaining stack frames

Why do we have call frames?

- Store local variables and (temporarily) registers

➡ Resume computation after function call

Maintaining call a frame is not necessary if the function body ends with a function call

Tail call.

- Instead of creating a new frame, we can reuse the frame of the caller

Tail call optimization.

Tail Call Optimization

- Reusing the frame of the caller is easier for recursive calls since the frame “fits”

➡ Many compilers implement tail-call optimization only for recursive calls

Example: Java

- But: It’s not difficult to adjust the frame size to account for tail calls to other functions

- Implementation: Replace a function call in abstract assembly with a jump

Works best after function parameters are removed.

```
int powacc (int b, int e, int a)
{
  if (e == 0)
    return a;
  else
    return powacc(b,e-1,a*b);
}

int pow(int b, int e)
{
  return powacc(b,e,1)
}
```

Example: powacc

1. Compile to Abstract Assembly

```
int powacc (int b, int e, int a)
{
  if (e == 0)
    return a;
  else
    return powacc(b,e-1,a*b);
}

int pow(int b, int e)
{
  return powacc(b,e,1)
}
```

```
powacc$(b, e, a)$ :
    if $(e = 0)$ goto done
    $t_0 \leftarrow e - 1$
    $t_1 \leftarrow a * b$
    $t_2 \leftarrow$ powacc$(b, t_0, t_1)$
    return $t_2$

done :
    return $a$

pow$(b, e)$ :
    $t_0 \leftarrow$ powacc$(b, e, 1)$
    return $t_0$
```

C0

High-level Abstract Assembly

2. Replace Parameters with Abstract Registers

```
powacc$(b, e, a)$ :
    if $(e = 0)$ goto done
    $t_0 \leftarrow e - 1$
    $t_1 \leftarrow a * b$
    $t_2 \leftarrow$ powacc$(b, t_0, t_1)$
    return $t_2$

done :
    return $a$

pow$(b, e)$ :
    $t_0 \leftarrow$ powacc$(b, e, 1)$
    return $t_0$
```

High-level Abstract Assembly

```
powacc :
    $b \leftarrow$ arg$_1$
    $e \leftarrow$ arg$_2$
    $a \leftarrow$ arg$_3$
    if $(e = 0)$ goto done
    $t_0 \leftarrow e - 1$
    $t_1 \leftarrow a * b$
    arg$_1 \leftarrow b$
    arg$_2 \leftarrow t_0$
    arg$_3 \leftarrow t_1$
    call powacc
    $t_2 \leftarrow$ res
    res $\leftarrow t_2$
    ret

done :
    res $\leftarrow a$
    ret

pow :
    $b \leftarrow$ arg$_1$
    $e \leftarrow$ arg$_2$
    arg$_1 \leftarrow b$
    arg$_2 \leftarrow 2$
    arg$_3 \leftarrow 1$
    call powacc
    $t_0 \leftarrow$ res
    res $\leftarrow t_0$
    ret
```

Low-level Abstract Assembly

3. Apply Optimizations (Remove Null Sequences)

```
powacc :
    $b \leftarrow \text{arg}_1$
    $e \leftarrow \text{arg}_2$
    $a \leftarrow \text{arg}_3$
    if $(e = 0)$ goto done
    $t_0 \leftarrow e - 1$
    $t_1 \leftarrow a * b$
    $\text{arg}_1 \leftarrow b$
    $\text{arg}_2 \leftarrow t_0$
    $\text{arg}_3 \leftarrow t_1$
    call powacc
    ~~$t_2 \leftarrow \text{res}$~~
    ~~$\text{res} \leftarrow t_2$~~
    ret

done :
    $\text{res} \leftarrow a$
    ret

pow :
    $b \leftarrow \text{arg}_1$
    $e \leftarrow \text{arg}_2$
    $\text{arg}_1 \leftarrow b$
    $\text{arg}_2 \leftarrow 2$
    $\text{arg}_3 \leftarrow 1$
    call powacc
    ~~$t_0 \leftarrow \text{res}$~~
    ~~$\text{res} \leftarrow t_0$~~
    ret
```

Low-level Abstract Assembly

4. Introduce Tail Recursion

Replace recursive calls followed by returns with jumps:

```
call $f$        $\rightsquigarrow$        goto $f$
ret
```

```
powacc :
    $b \leftarrow$ arg$_1$
    $e \leftarrow$ arg$_2$
    $a \leftarrow$ arg$_3$
    if $(e = 0)$ goto done
    $t_0 \leftarrow e - 1$
    $t_1 \leftarrow a * b$
    arg$_1 \leftarrow b$
    arg$_2 \leftarrow t_0$
    arg$_3 \leftarrow t_1$
    goto powacc
```

```
done :
    res $\leftarrow a$
    ret

pow :
    $b \leftarrow$ arg$_1$
    $e \leftarrow$ arg$_2$
    arg$_1 \leftarrow b$
    arg$_2 \leftarrow 2$
    arg$_3 \leftarrow 1$
    call powacc
    ret
```

Non-Recursive Tail Calls

We also replace non-recursive tail calls with jumps

```
pow :
    $b \leftarrow \mathsf{arg}_1$
    $e \leftarrow \mathsf{arg}_2$
    $\mathsf{arg}_1 \leftarrow b$
    $\mathsf{arg}_2 \leftarrow 2$
    $\mathsf{arg}_3 \leftarrow 1$
    goto powacc
```

In general it’s not sound to jump to a block of a different function

However, it’s okay to merge pow and powacc into one function if powacc is only called from pow

Register Allocation

After tail call optimization, we can make the assignment:

b	$\mapsto$	arg_1
e	$\mapsto$	arg_2
a	$\mapsto$	arg_3
t_0	$\mapsto$	arg_2
t_1	$\mapsto$	arg_3

Eliminate self moves

```
powacc :
    if $(e = 0)$ goto done
    $\text{arg}_2 \leftarrow \text{arg}_2 - 1$
    $\text{arg}_3 \leftarrow \text{arg}_3 * \text{arg}_1$
    goto powacc
```

```
done :
    res $\leftarrow \text{arg}_3$
    ret

pow :
    $\text{arg}_1 \leftarrow 1$
    goto powacc
```