Intermediate representation (1)

In general, once the semantic analysis has been completed without errors, a compiler will generate an intermediate representation (IR) of the program, which will be used in the following compilation stages

The main benefit from using an intermediate representation of the program comes from separating the program analysis from the final code generation, which allows:

- Sharing the analysis code between different computer architectures
- Only having to implement architecture independent optimisations once
- Building a compiler for a different programming language changing solely the front-end of the compiler

Intermediate representation (2)

The intermediate representation, or intermediate code, may be regarded as the language for some abstract machine

The level (or the granularity) of the operations of the intermediate representation is, in general, closer to that of machine code than to that of a high-level language

An intermediate representation usually incorporates temporary locations (or temporary registers, or simply temporaries) for the storage of values

The number of temporary locations is unbounded

There may be different kinds of temporary locations, for dealing with different types of values

Three-address code (1)

The three-address code is a common form for the intermediate representation

The name comes from the fact that many of its instructions have 3 arguments, which are the addresses of their arguments

The arguments of an instruction include the source(s) of values and the destination(s) of the result

Three-address code (2)

A typical instruction in a three-address code is the instruction that adds two values together and stores the result in a temporary location

This instruction may come in many forms:

- add $t_1, t_2 \Rightarrow t_3$
- add t_3, t_1, t_2
- $t_3 \leftarrow$ add t_1, t_2 *syntax used in TAC*
- $t_3 \leftarrow t_1 + t_2$

The operation it performs is the sum and the 3 addresses are the temporary locations t_1, t_2 and t_3, of the instruction arguments

In the notation chosen, when an instruction produces a value, its destination appears to the left of the arrow: $\leftarrow$

Control flow

In the intermediate representation, control flow is usually expressed through low-level constructs, such as (conditional and unconditional) jumps to labelled instructions

Intermediate representation for TACL (1)

Starting version (v0)

Three-address code

Temporaries

Integer temporaries

t_i, $i = 0, 1, 2, \ldots$

Floating-point temporaries

fp_i, $i = 0, 1, 2, \ldots$

Labels

l_i, $i = 0, 1, 2, \ldots$

Names

@name

where *name* is a program identifier

Intermediate representation for TACL (2)

Starting version (v0)

Arithmetic operations

Integer

$t_i \leftarrow op\ t_j, t_k$

where *op* is one of

i_add i_sub i_mul i_div mod

$t_i \leftarrow$ i_inv t_j (additive inverse)

Floating point

$fp_i \leftarrow op\ fp_j, fp_k$

where *op* is one of

r_add r_sub r_mul r_div

$fp_i \leftarrow$ r_inv fp_j (additive inverse)

Example TACL code and IR

Homework

Complete the IR for the TACL function below

TACL function

```
fun real f(int n)
[
  var int r = 1;

  while (n > 0)
  [
    r = r * n;
    n = n - 1;
  ]

  ^ r
]
```

Partial IR

```
        t0 <- i_value 1
        @r <- i_lstore t0
        t1 <- i_aload @n
        t2 <- i_value 0
        t3 <- i_lt t2, t1
        cjump t3, l1, l2
l1:     t4 <- i_lload @r
        t5 <- i_aload @n
        t6 <- i_mul t4, t5
        @r <- i_lstore t6
        ...
```

Intermediate representation for TACL (1)

Definitive version (v1)

Three-address code

Temporaries

Integer temporaries

t_i, $i = 0, 1, 2, \ldots$

Floating-point temporaries

fp_i, $i = 0, 1, 2, \ldots$

Labels

l_i, $i = 0, 1, 2, \ldots$

Names

@name

where *name* is a program identifier

Intermediate representation for TACL (2)

Definitive version (v1)

Arithmetic operations

Integer

$t_i \leftarrow op\ t_j, t_k$

where *op* is one of

i_add   i_sub   i_mul   i_div   mod

$t_i \leftarrow \text{i\_inv}\ t_j$ (additive inverse)

Floating point

$fp_i \leftarrow op\ fp_j, fp_k$

where *op* is one of

r_add   r_sub   r_mul   r_div

$fp_i \leftarrow \text{r\_inv}\ fp_j$ (additive inverse)

Intermediate representation for TACL (3)

Definitive version (v1)

Comparison operations

Integer

$t_i \leftarrow op\ t_j, t_k$

where op is one of

i_eq   i_lt   i_ne   i_le

t_i gets 1 if $t_j\ op\ t_k$ is true, otherwise it gets 0

Floating point

$t_i \leftarrow op\ fp_j, fp_k$

where op is one of

r_eq   r_lt   r_ne   r_le

t_i gets 1 if $fp_j\ op\ fp_k$ is true, otherwise it gets 0

Intermediate representation for TACL (4)

Definitive version (v1)

Integer coercion

$fp_i \leftarrow \text{itor } t_j$

Logical operations

$t_i \leftarrow \text{not } t_j$

Value copy

$t_i \leftarrow \text{i\_copy } t_j$

$fp_i \leftarrow \text{r\_copy } fp_j$

Intermediate representation for TACL (5)

Definitive version (v1)

Data transfer

Integer

$t_i \leftarrow$ i_value n (load immediate value)

$t_i \leftarrow$ i_gload $@name$ (global variable)

$t_i \leftarrow$ i_lload $@name$ (local variable)

$t_i \leftarrow$ i_aload $@name$ (function argument)

$@name \leftarrow$ i_gstore t_i (global variable)

$@name \leftarrow$ i_lstore t_i (local variable)

$@name \leftarrow$ i_astore t_i (function argument)

Intermediate representation for TACL (6)

Definitive version (v1)

Data transfer (cont.)

Floating point

$fp_i \leftarrow \text{r\_value } x$ (load immediate value)

$fp_i \leftarrow \text{r\_gload } @name$ (global variable)

$fp_i \leftarrow \text{r\_lload } @name$ (local variable)

$fp_i \leftarrow \text{r\_aload } @name$ (function argument)

$@name \leftarrow \text{r\_gstore } fp_i$ (global variable)

$@name \leftarrow \text{r\_lstore } fp_i$ (local variable)

$@name \leftarrow \text{r\_astore } fp_i$ (function argument)

Intermediate representation for TACL (7)

Definitive version (v1)

Flow of control

Jump

jump l_i

Conditional jump

cjump t_i, l_j, l_k

Jump to l_j if t_i holds 1, jump to l_k if it holds 0

Intermediate representation for TACL (8)

Definitive version (v1)

Functions

Function call

$t_i \leftarrow \text{i\_call } @name, [a_1, \ldots, a_n]$

$fp_i \leftarrow \text{r\_call } @name, [a_1, \ldots, a_n]$

where $n \geq 0$ and each a_i is either t_j or fp_j, for some j

Call function $name$ with arguments $a_1, \ldots, a_n$

Function return value

$\text{i\_return } t_i$

$\text{r\_return } fp_i$

Set the value returned by the function and return

Intermediate representation for TACL (9)

Definitive version (v1)

Procedures

Procedure call

call @$name, [a_1, \ldots, a_n]$

where $n \geq 0$ and each a_i is either t_j or fp_j, for some j

Call procedure $name$ with arguments $a_1, \ldots, a_n$

Procedure return

return

Procedure end

Intermediate representation for TACL (10)

Definitive version (v1)

Print

i_print t_i

b_print t_i

r_print fp_i

Intermediate representation for TACL (11)

Definitive version (v1)

Remarks

1. The *t*_ne and *t*_le instructions are not useful for expressions appearing in control flow statements, such as the `if` and `while` statements

   They are only useful with expressions whose value will be assigned to a variable, used in a function or procedure call, or as the argument of a `print` statement, where they allow saving an additional not instruction

2. The not is equally not useful for the conditions of control flow statements

Intermediate representation for TACL (12)

Definitive version (v1)

Remarks (cont.)

3. The use of names in the instructions that deal with local variables and function arguments (t_lload and t_lstore, t_aload and t_astore) reflects the fact that we are in a stage where their location (whether they will reside in memory or in a register) has not yet been decided

   When the time to make that decision comes, the information contained in the symbol table must still available

4. Program identifiers appearing in the (written) intermediate representation are tagged with the `@` symbol so it is easy to distinguish them from IR instruction names and temporaries (e.g., temporary t_3 and variable `t3`)

Intermediate representation for TACL (13)

Definitive version (v1)

Example (1)

If `v` is an integer global variable, the statement

```
v = 1;
```

may have the following intermediate representation

$t_0 \leftarrow$ i_value 1
$@v \leftarrow$ i_gstore t_0

Intermediate representation for TACL (14)

Definitive version (v1)

Example (2)

If a, b an c are integer local variables, to the statement

```
a = b + c * 2;
```

may correspond the intermediate representation

$t_0 \leftarrow \text{i\_lload } @b$
$t_1 \leftarrow \text{i\_lload } @c$
$t_2 \leftarrow \text{i\_value } 2$
$t_3 \leftarrow \text{i\_mul } t_1, t_2$
$t_4 \leftarrow \text{i\_add } t_0, t_3$
$@a \leftarrow \text{i\_lstore } t_4$

Intermediate representation for TACL (15)

Definitive version (v1)

Example (3)

If x is an integer local variable and y is a real local variable, the intermediate representation of the statement below may be the one on the right

```
if (x < 0)
  y = -x;
else
  y = x;
```

$$
\begin{array}{ll}
 & t_1 \leftarrow \mathsf{i\_lload}\ @x \\
 & t_2 \leftarrow \mathsf{i\_value}\ 0 \\
 & t_3 \leftarrow \mathsf{i\_lt}\ t_1, t_2 \\
 & \mathsf{cjump}\ t_3,\ l_t,\ l_f \\
l_t: & t_4 \leftarrow \mathsf{i\_lload}\ @x \\
 & t_5 \leftarrow \mathsf{i\_inv}\ t_4 \\
 & fp_1 \leftarrow \mathsf{itor}\ t_5 \\
 & @y \leftarrow \mathsf{r\_lstore}\ fp_1 \\
 & \mathsf{jump}\ l_d \\
l_f: & t_6 \leftarrow \mathsf{i\_lload}\ @x \\
 & fp_2 \leftarrow \mathsf{itor}\ t_6 \\
 & @y \leftarrow \mathsf{r\_lstore}\ fp_6 \\
l_d: &
\end{array}
$$

Intermediate representation for TACL (16)

Definitive version (v1)

Example (4)

If `n` is an integer function argument and `r` an integer local variable, the code below could have the intermediate representation on the right

```
r = 1;
while (n > 0)
[
  r = r * n;
  n = n - 1;
]
^ r
```

$$
\begin{array}{lll}
 & t_0 \leftarrow \text{i\_value } 1 & \\
 & @r \leftarrow \text{i\_lstore } t_0 & \texttt{\# r = 1} \\
l_0: & t_1 \leftarrow \text{i\_aload } @n & \\
 & t_2 \leftarrow \text{i\_value } 0 & \\
 & t_3 \leftarrow \text{i\_lt } t_2, t_1 & \texttt{\# 0 < n?} \\
 & \text{cjump } t_3, l_1, l_2 & \\
l_1: & t_4 \leftarrow \text{i\_lload } @r & \\
 & t_5 \leftarrow \text{i\_aload } @n & \\
 & t_6 \leftarrow \text{i\_mul } t_4, t_5 & \texttt{\# r * n} \\
 & @r \leftarrow \text{i\_lstore } t_6 & \\
 & t_7 \leftarrow \text{i\_aload } @n & \\
 & t_8 \leftarrow \text{i\_value } 1 & \\
 & t_9 \leftarrow \text{i\_sub } t_7, t_8 & \texttt{\# n - 1} \\
 & @n \leftarrow \text{i\_astore } t_9 & \\
 & \text{jump } l_0 & \\
l_2: & t_{10} \leftarrow \text{i\_lload } @r & \\
 & \text{i\_return } t_{10} &
\end{array}
$$

Intermediate representation for TACL (17)

Definitive version (v1)

Further remarks

The intermediate representation is not streamlined

- The number of temporaries used is unnecessarily high

  Temporary location t_0 only appears on the first two lines of Example (4) and could be reused

- It contains redundant operations

  There are two instructions loading the same value to temporaries t_5 and t_7

Generating the intermediate representation (1)

The intermediate representation is generated by walking the decorated abstract syntax tree

In general, the intermediate representation only exists inside the compiler, and does not have a visible representation

The compiler may implement it internally as an instruction list or as a tree (like the abstract syntax tree)

Generating the intermediate representation (2)

IR generation for conditions

A condition (of an `if` or `while` statement) is just a boolean expression and its intermediate representation could be generated like that of any expression

However, its value is only needed for making a decision, and embedding the decision making into the evaluation of the condition means:

- Not having to save its value
- Having less jumps in the (intermediate) code

As one of the uses of conditions is controlling the execution of loops, saving a few steps in its evaluation may have a significant impact in the final code efficiency

Generating the intermediate representation (3)

IR generation for conditions (cont.)

The following is a simple and extreme example

```
if (true) s1 else s2
```

Treating the condition like any (boolean) expression, would lead to the intermediate representation on the left, while embedding the decision making in its evaluation would lead to the one on the right

$$
\begin{array}{rl}
 & t_0 \leftarrow \text{i\_value } 1 \\
 & \text{cjump } t_0, l_t, l_f \\
l_t: & \text{code for } \texttt{s1} \\
 & \text{jump } l_e \\
l_f: & \text{code for } \texttt{s2} \\
l_e: &
\end{array}
\qquad\qquad
\begin{array}{rl}
 & \text{jump } l_t \\
l_t: & \text{code for } \texttt{s1} \\
 & \text{jump } l_e \\
l_f: & \text{code for } \texttt{s2} \\
l_e: &
\end{array}
$$

The second version uses one less temporary, one less instruction, turns a conditional jump into an unconditional jump, and a simple analysis would allow eliminating everything but the code for `s1`

Function factorial (1)

TACL code

```
fun int factorial(int n)
[
  var int r = 1;

  if (n > 0)
    r = n * factorial(n - 1);

  ^ r
]
```

Function factorial (2)

Abstract syntax tree (linear representation)

```
(fun "factorial" [(arg "n" int)]
  (body [ (local "r" int (int_literal 1): int) ]
    (if
      (gt (id "n" arg int): int (int_literal 0): int): bool
      (assign (id "r" local int)
        (times
          (id "n" arg int): int
          (call "factorial" [
              (minus (id "n" arg int): int
                (int_literal 1): int): int]): int): int)
      nil)
    (id "r" local int): int))
```

Function factorial (3)

Intermediate representation

```
          t0 <- i_value 1
          @r <- i_lstore t0
          t1 <- i_aload @n
          t2 <- i_value 0
          t3 <- i_lt t2, t1
          cjump t3, l0, l1
l0:       t4 <- i_aload @n
          t5 <- i_aload @n
          t6 <- i_value 1
          t7 <- i_sub t5, t6
          t8 <- i_call @factorial, [t7]
          t9 <- i_mul t4, t8
          @r <- i_lstore t9
          jump l2
l1:
l2:       t10 <- i_lload @r
          i_return t10
```

Function factorial (4)

Intermediate representation (smarter version)

```
        t0 <- i_value 1
        @r <- i_lstore t0
        t1 <- i_aload @n
        t2 <- i_value 0
        t3 <- i_lt t2, t1
        cjump t3, l0, l1
l0:     t4 <- i_aload @n
        t5 <- i_aload @n
        t6 <- i_value 1
        t7 <- i_sub t5, t6
        t8 <- i_call @factorial, [t7]
        t9 <- i_mul t4, t8
        @r <- i_lstore t9
l1:     t10 <- i_lload @r
        i_return t10
```

Function factorial (5)

Remark

The difference between the two IR versions for the factorial function is that in the first, the IR for the `if` statement is generated as if it were

```
if (n > 0)
  r = n * factorial(n - 1);
else
  []
```

In this case, there is an additional jump over the (empty) `else` part, following the code for `r = n * factorial(n - 1);`

The smarter version results from handling the `if` without an `else` part as a special case, and the only jump needed is the conditional jump

Function g (1)

TACL code

```
fun real g(int n)
[
  var real s = 0;

  while (n >= 1)
    [
      s = s + n;
      n = n - 1;
    ]

  ^ s
]
```

Function g (2)

Intermediate representation

```
        t0 <- i_value 0
        fp0 <- itor t0
        @s <- r_lstore fp0
l0:     t1 <- i_aload @n
        t2 <- i_value 1
        t3 <- i_lt t1, t2
        cjump t3, l2, l1
l1:     fp1 <- r_lload @s
        t4 <- i_aload @n
        fp2 <- itor t4
        fp3 <- r_add fp1, fp2
        @s <- r_lstore fp3
        t5 <- i_aload @n
        t6 <- i_value 1
        t7 <- i_sub t5, t6
        @n <- i_astore t7
        jump l0
l2:     fp4 <- r_lload @s
        r_return fp4
```

Function fibonacci (1)

TACL code

```
fun int fibonacci(int n)
[
  var int r;

  if (n == 0 || n == 1)
    r = n;
  else
    r = fibonacci(n - 1) + fibonacci(sub(n, 2));

  ^ r
]
```

Function fibonacci (2)

Intermediate representation

```
        t0 <- i_aload @n
        t1 <- i_value 0
        t2 <- i_eq t0, t1
        cjump t2, l0, l3
l3:     t3 <- i_aload @n
        t4 <- i_value 1
        t5 <- i_eq t3, t4
        cjump t5, l0, l1
l0:     t6 <- i_aload @n
        @r <- i_lstore t6
        jump l2
```

...

Function fibonacci (3)

Intermediate representation (cont.)

```
l1:     t7 <- i_aload @n
        t8 <- i_value 1
        t9 <- i_sub t7, t8
        t10 <- i_call @fibonacci, [t9]
        t11 <- i_aload @n
        t12 <- i_value 2
        t13 <- i_call @sub, [t11,t12]
        t14 <- i_call @fibonacci, [t13]
        t15 <- i_add t10, t14
        @r <- i_lstore t15
l2:     t16 <- i_lload @r
        i_return t16
```

Function fibonacci (v2) (1)

```
fun int fibonacci(int n)
[
  var int r;
  var bool c;

  c = n == 0 || n == 1;

  if (c)
    r = n;
  else
    r = fibonacci(n - 1) + fibonacci(sub(n, 2));

  ^ r
]
```

Function fibonacci (v2) (2)

Intermediate representation

```
        t0 <- i_aload @n
        t1 <- i_value 0
        t2 <- i_eq t0, t1
        cjump t2, l0, l1
l1:     t3 <- i_aload @n
        t4 <- i_value 1
        t5 <- i_eq t3, t4
        t2 <- i_copy t5
l0:     @c <- i_lstore t2
        t6 <- i_lload @c
        cjump t6, l2, l3
l2:     t7 <- i_aload @n
        @r <- i_lstore t7
        jump l4
```

...

Function fibonacci (v2) (3)

Intermediate representation (cont.)

```
l3:     t8 <- i_aload @n
        t9 <- i_value 1
        t10 <- i_sub t8, t9
        t11 <- i_call @fibonacci, [t10]
        t12 <- i_aload @n
        t13 <- i_value 2
        t14 <- i_call @sub, [t12,t13]
        t15 <- i_call @fibonacci, [t14]
        t16 <- i_add t11, t15
        @r <- i_lstore t16
l4:     t17 <- i_lload @r
        i_return t17
```

Function fibonacci (v2) (4)

Remark

In this second version of the fibonacci function, the value of the expression

```
n == 0 || n == 1
```

is assigned to the variable `c`

The IR for the expression computes its value and saves it in the `t2` temporary, from where it is then stored into the variable

The highlighted IR parts in the previous example and in this one illustrate the difference between the intermediate representations for a boolean expression when it is used as a condition and for the same boolean expression when its value will be used later in the program

Function main (1)

```
var int n = 26;

fun int main()
[
  var int fib = fibonacci(n);

  # all three should print the same
  print fib;
  print fibonacci(14 + 3 * 4);
  print 121393;

  ^ 0 # return 0 to the system
]
```

Function main (2)

Intermediate representation

```
t0 <- i_gload @n
t1 <- i_call @fibonacci, [t0]
@fib <- i_lstore t1
t2 <- i_lload @fib
i_print t2
t3 <- i_value 14
t4 <- i_value 3
t5 <- i_value 4
t6 <- i_mul t4, t5
t7 <- i_add t3, t6
t8 <- i_call @fibonacci, [t7]
i_print t8
t9 <- i_value 121393
i_print t9
t10 <- i_value 0
i_return t10
```