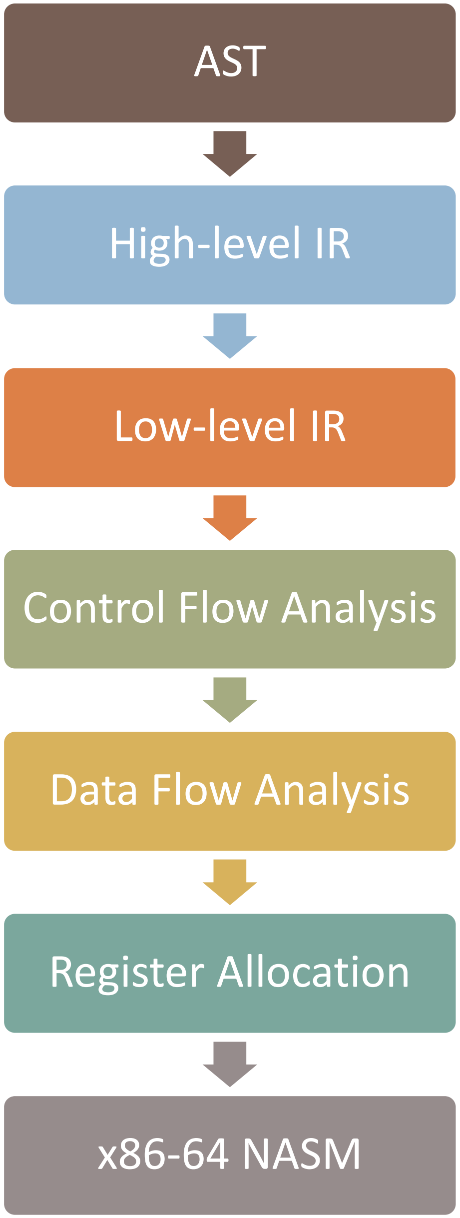
Malic Compiler

Design & Implementation

Lianmin Zheng
2017.6

Architecture

70+ days, 90+ commits, 12,000+ lines of code



- output-irrelevant elimination

- ✓ function inline
- ✓ instruction selection

- redundant jump elimination

- ✓ constant propagation and folding
- ✓ common sub-expression elimination
- ✓ dead code elimination

- global graphing coloring

- ✓ some peepholes

Abstract Syntax Tree

with the powerful tool ANTLR, maybe the AST is the easiest part of our compilers.

Output Irrelevant Elimination

- eliminate output-irrelevant and return-irrelevant code

```
int[] copy(int [] src, int n) {
    int [] dest    = new int[n];
    int [] useless = new int[n];

    for (int i = 0; i < n; i++) {
        dest[i]     = src[i];
        useless[i]  = src[i];
    }

    return dest;
}
```

- do this in the AST, because AST is easier and sufficient to analysis dependency

- if do this in IR, you will struggle with memory and alias

Output Irrelevant Maker

- Build dependency graph, nodes are variables and functions, edges are dependency relationship.
- Three kinds of dependency (edge):
  1. control dependency (thank god for no “goto” in the M* language)
  2. assign dependency
  3. function dependency (return, parameter)
- Source nodes in graph :
  - all variables assigned to parameter of output-relevant function (print, user func, ...)
  - output-relevant functions
- Iteration:

```
while (not stable) {
    visit AST, build dependency graph
    dfs from source, mark all reachable graph nodes output-relevant
    visit AST, mark AST nodes
}
in IRBuilder, do not generate IR for output-irrelevant AST nodes
```

- Alias: when meets copy between pointers (array, class), do not eliminate them.

Intermediate Representation

multi layer makes structure clearer and thing easier

High Level IR

- linear for statement
- tree for expression
- machine irrelevant

- Node design:

  Call, Label, Cjump, Jump, Return,

  Expr, Binary, Unary, Bin, Var, Const, Mem, Addr

Low Level IR

- linear for all
- instruction level

  add, sub, div, mul, xor, jmp, pop, push, sal, sar, …
- custom-made for x86 instruction set

  mov, lea, cmp, inc, dec, …

Optimization

an act, process, or methodology of making something (such as a design or system) as fully perfect, functional, or effective as possible

- Webster

Instruction Selection

- x86 support many forms of address
  - full form
    - mov rdx, [rax + rbx * 8 + 12]
  - reduced form
    - mov rdx, [rax + 12]
    - mov rdx, [rax + rbx * 8]
    - …
- lea : load effective address
  - lea reg, addr   -> reg = addr
  - mov reg, addr   -> reg = mem[addr]

Examples of Instruction Selection

- `b, c, d, i, h are registers`
- `a is an array of int32`

- `h = c * 4 + 3      ->  lea h, [c * 4 + 3]`
- `h = b + c * 4 + 3  ->  lea h, [b + c * 4 + 3]`
- `h = c * 5 + 2      ->  lea h, [c + c * 4 + 2]`
- `h = a[i]           ->  mov h, [a + i * 4]`
- `a[i] = 12          ->  mov [a + i * 4], 12`

- do this by sub-tree matching when converting high-level IR to low-level IR

Function Inline

- for non-recursive functions
  - if it is small, inline it
  - my criterion: #statements < 8
  - do inline recursively
- for recursive functions:
  - you can also inline it! it make some test cases much faster.
  - my criterion : #statements ^ depth < 40 && depth < 3

```
int f(int n) {
    return n <= 1 ? 1 : n * f(n-1)
}
```

```
int f(int n) {
    return n <= 1 ? 1 : n * ((n-1) <= 1 ? 1 : (n-1) * f((n-1) - 1);
}
```

Control Flow Analysis

- extract basic blocks and build control flow graph

- optimization
  - merge
  - path compression
  - organize trace greedily to maximize #fall-through jump

Data Flow Analysis

- (local) Constant Propagation and Folding
- (local) Common Sub-expression Elimination
- (global) Dead code Elimination

no matter how brute-forceful you IR is, these optimizations can eliminate almost all redundant code!

so if you implement these optimizations, you can feel free when generating IR.

Data Flow Analysis

- constant propagation and folding

  can be combined with inline!

```
int add(int x, int y) { return x + y; }
int sub(int x, int y) { return x - y; }
int mul(int x, int y) { return x * y; }

int main() {
    int a = 14;
    int b = 99;
    int c = 11;
    int d = add(a, sub(mul(a, b), c));
    int e = mul(add(d, c), sub(a, b));

    return a / b ^ c % d & e;
}
```

->

```
int main() {
    return 8;
}
```

Data Flow Analysis

- Common Sub-expression Elimination
  - common sub-expression is common in array indexing

    ```
    a[i][i] = a[i][j] + a[i][k]  ->  t = a[i]; t[i] = t[j] + t[k]
    ```

  - one-pass linear scan inside a basic block
  - do renaming and copy propagation simultaneously to find more common sub-expression

    copy propagation:

    ```
    a = b                  a = b                a = b
    x = a + c    ->        x = b + c    ->      x = b + c
    y = b + c              y = b + c            y = x
    ```

    renaming

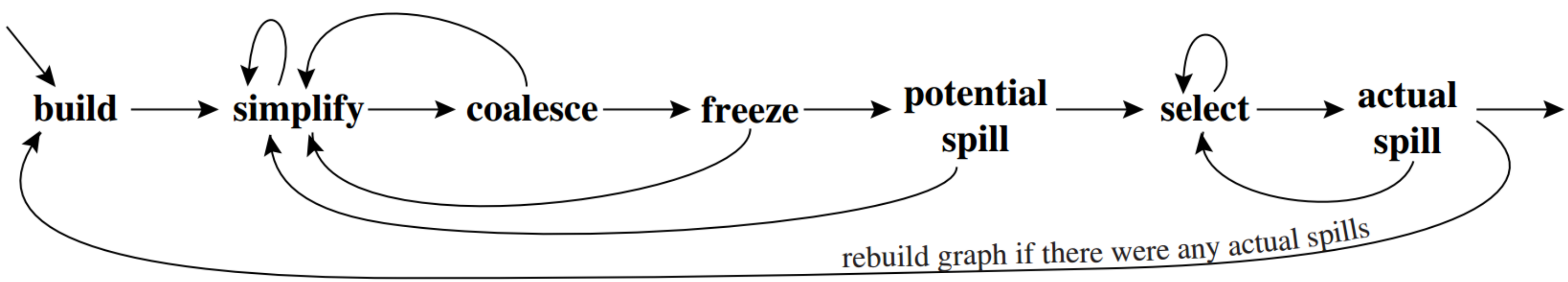
    ```
    a = c                  a = c
    a = a + b              t = a + b
    a = a * 2    ->        a = t * 2
    x = c + b              x = t
    y = c + b              y = t
    ```

Data Flow Analysis

- Dead code Elimination
  - after liveliness analysis
  - for every instruction x, if def(x) $\notin$ out(x), remove x

Register Allocation

- liveliness analysis
  - in single basic block: linear scan
  - among basic blocks: solve data flow equation by iteration
- build inference graph
  - linear scan in every basic block
- iterated allocation
  - see George, Lal; Appel, Andrew W. (May 1996). "Iterated Register Coalescing“
  - or chapter 11 of tiger book (the same as the above paper)



Register Allocation

- By iteration, every virtual register will be allocated to a physical register, feel free when write translator!
- How to meet the specific machine convention?
  - use pre-colored node!
  - inference graph should also contain the nodes that represents physical registers, but they are pre-colored
  - add ‘mov’ to make invalid instruction become valid

see next page

Meet the Machine Convention

- for division in x86, dividend must be rax. After division, quotient will be put into rax, remainder will be put into rdx.

- let x, y be virtual registers

  for an instruction x = x / y, it may be invalid if x is not allocated to rax, so we should add move to make it valid.

  let r_rax be a pre-colored virtual register, since it is pre-colored, it must be allocated to rax. let r_rdx be a pre-colored virtual register, rdx be the physical register.

```
raw IR:

x = x / y
```

->

```
modified IR:

r_rax = x
r_rax = r_rax / y  ; now it is always valid
r_rdx = rdx        ; r_rdx is changed, refresh it
x = r_rax
```

- redundant ‘mov’ will be eliminated in coalesce phase
- do the same modify for calling convention, ret, sal, sar, …

Other small optimizations

- leaf function
  - allocate register for global variable in leaf function
  - don’t need sub rsp in leaf function
- expand print
  - print(“aha” + toString(5)) -> print(“aha”); printInt(5);
  - do it recursively
- use 32-bit division
  - div rax -> div eax
  - 32-bit division is 2 times faster than 64-bit division
- use gcc –O3 to generate built-in functions
  - I write toString, substr, str_concate in C and use gcc to generate asm
  - faster than my hand-code asm which calls strcpy, sprintf, …

Other small optimizations

- short-cut evaluation
    - push down label info, instead of calculating value every time

```
Cjump(cond, trueLabel, falseLabel) is condition jump

Cjump(!a, L1, L2)      -> Cjump(a, L2, L1)
Cjump(a && b, L1, L2) -> Cjump(a, goon, L2)
                         goon:
                         Cjump(b, L1, L2)
Cjump(a || b, L1, L2) -> Cjump(a, L1, goon)
                         goon:
                         Cjump(b, L1, L2)
```

    - do it recursively

Performance Analysis

the malic compiler is ranked first in the final board

Final Board

Test case	272	274	277	279	282	284	337	338	344	348
Rank	2	2	2	1	1	1	3	1	1	3

Test case	349	350	352	353	357	360	361	362	363	364
Rank	1	1	7	1	1	2	1	2	1	1

Experiment

- register allocation is the most general optimization
  typically make T_old / T_new = 1.5 ~ 3.0

Testcase	Optimization	T_before / T_after
sha1	register allocation	2.89

- some optimizations perform well in specific test cases

Testcase	Optimization	T_before / T_after
superloop	Cjump	3.26
expr	common expression elimination	7.28
hanoi	inline recursive function	2.76
segtree	32-bit division	1.95
cnf-lp	gcc –O3 for builtin function	2.01
useless	output-irrelevant elimination	2.02

combining various optimizations can get better result, 1 + 1 > 2 (inline, cse, allocation ...)

In the End

enjoy writing compiler!

Book & Reference

- *ふつうのコンパイラをつくろう:*
  very practical, good explanation of x86 and asm
- *Tiger:*
  my backend almost follows this book
- *Engineering A Compiler:*
  cover many things, some people think it's useful,
  but I felt it is not very practical

Helping

- Give advise on framework and optimization to some classmates
- Help TA to modify invalid test cases

Acknowledge

- Thank Lequn Chen for his wonderful Online Judge
- Thank Lequn Chen, Zhijian Liu for their reports and open code
- Thank Tiancheng Xie for his advices
- Thank Rong Ma for his teaching
- Thank Zhekai Zhang and other classmates for our discussing