Calculator Compiler Final Project

5/17/2019

CPEG 621 – Compiler Design

Benjamin Steenkamer

Specifications

- Fully implements the calculator language
- Frontend TAC generation with up to 3 nested if/else statements
- Data dependence works simultaneously with TAC generation
  - Can work for n-nested if/else statements
  - Correctly tracks all 3 dependence types through every if/else branch
- CSE and Copy-statement elimination after the TAC generation
  - Can work for n-nested if/else statements
  - Cooperative heuristic
    - They should not undo each other's work
  - Goal is to reduce the number of variables needed in registers
- Backend creates C from unoptimized and optimized TAC
  - Adapts to changes in TAC made by optimizations
  - Makes timing versions of each that loops the code

Calculator Language

- Flex Rules

```
[A-Za-z][A-Za-z0-9]* {  // Variable names are case insensitive
    #ifdef DEBUG
    printf("token %s at line %d\n", yytext, flex_line_num);
    #endif

    yylval.str = strdup(yytext); // Must be freed in bison grammar rules
    return VARIABLE;
    }

[0-9]+ {
    #ifdef DEBUG
    printf("token %s at line %d\n", yytext, flex_line_num);
    #endif

    yylval.str = strdup(yytext); // Must be freed in bison grammar rules
    return INTEGER;
    }

"**" {
    #ifdef DEBUG
    printf("token %s at line %d\n", yytext, flex_line_num);
    #endif

    return POWER;
    }

[-+()=*/!?] {
        #ifdef DEBUG
        printf("token %s at line %d\n", yytext, flex_line_num);
        #endif

        return *yytext;     // Return character literal as the token value
    }

"\n" {
    #ifdef DEBUG
    printf("token \\n at line %d\n", flex_line_num);
    #endif

    flex_line_num++;
    return *yytext;
    }
```

- Bison Rules
  - Generates the TAC expressions
  - Matches if/else statements
    - **Difficult for 3 deep if/else!**

```
calc :
    calc expr '\n'      { my_free($2); gen_tac_else(NULL); }
    |
    ;

expr :
    INTEGER             { $$ = $1; }
    | VARIABLE          { $$ = lc($1); track_user_var(lc($1), 0); }
    | VARIABLE '=' expr { $$ = lc($1); gen_tac_assign(lc($1), $3); my_free($3); }
    | expr '+' expr     { $$ = gen_tac_expr($1, "+", $3); my_free($1); my_free($3); }
    | expr '-' expr     { $$ = gen_tac_expr($1, "-", $3); my_free($1); my_free($3); }
    | expr '*' expr     { $$ = gen_tac_expr($1, "*", $3); my_free($1); my_free($3); }
    | expr '/' expr     { $$ = gen_tac_expr($1, "/", $3); my_free($1); my_free($3); }
    | '!' expr          { $$ = gen_tac_expr(NULL, "!", $2); my_free($2); }      // Bitwise-not in calc lang
    | expr POWER expr   { $$ = gen_tac_expr($1, "**", $3); my_free($1); my_free($3);}
    | '(' expr ')'      { $$ = $2; }             // Will give syntax error for unmatched parens
    | '(' expr ')' '?' { gen_tac_if($2); } '(' expr ')'
                        {
                            $$ = $7;
                            track_if_else();    // Keep track of how many closing elses are need for
                            // printf("do_gen_else incremented by \"%s\": %d\n", $7, do_gen_else);
                            my_free($2);    // nested if/else cases
                        }
    ;
```

Compiler Frontend and Backend

- Converts calculator language into TAC
  - Inserts temporary variables as needed: _t#

```
if(a) {
    _t7 = a + 1;
    if(_t7) {
        _t8 = x + 1;
        if(_t8) {
            _t9 = !9;
            _t10 = y ** _t9;
            q = _t10;
        } else {
            q = 0;
        }
    } else {
        q = 0;
    }
    t = q;
} else {
    t = 0;
}
_t11 = q + t;
```

- Adds initialization of variables, printfs(), inputs, labels, and timing version

```
#include <time.h>
#include <stdio.h>
#include <math.h>

int main() {
    int a = 0, y = 0, x = 0, z = 0, u = 0, q = 0, p = 0
    int _t0 = 0, _t1 = 0, _t2 = 0, _t3 = 0, _t4 = 0, _t

    printf("a=");
    scanf("%d", &a);

    struct timespec _begin_time, _end_time;
    double _elapsed_time;
    int _iter;
    clock_gettime(CLOCK_MONOTONIC, &_begin_time);

    for(_iter = 0; _iter < 500000; _iter++){
    S0:     _t12 = a - 2;
    S1:     _t0 = _t12;
    S2:     y = _t0;
    S3:     if(y) {
    S4:     x = y;
            } else {
    S5:     x = 0;
            }
    S6:     _t1 = _t12;
    S7:     _t2 = x * _t1;
    S8:     z = _t2;
```

Data Dependency Analysis

- Flow (read after write), Anti (write after read), and Write (write after write)
  - Printed for every TAC line
  - Only “1 hop” of dependence
  - Looks backward from current line to find dependence
    - Stops when it finds a write to the variable (flow or write dependence) in a guaranteed path

- Steps taken for one TAC line:

1. Record variables in TAC line, the if/else depth, and whether in if or else

2. Check dependencies of each variable in TAC by looking backward
   1. If written to in current TAC, check for write and anti
   2. If read from in current TAC, check for flow

3. Record the dependences as they are encountered
   1. If a flow or write dep. in a guaranteed path is found, stop looking back
      1. Flow or write dep. outside if/else
      2. Flow or write in 1-deep if and else statements
   2. For potential dep. found in if/else, it will look forward from the potential dep. to see if future statement in the if/else will block it
      1. If checker in same if structure, can also tell if that dependence will block it from further up ones

Optimization - CSE

- Only recognized the form: x = <u>a op b</u>
  - Ignores … = !a (assume this is fast enough to do every time)
  - Recognizes communitive operations: a + b and a * b
- Inserts in the form
  - **_c1** = a op b
  - c = **_c1**
  - …
  - d = **_c1**
- Only create temp variable if subexpression used more than <u>once</u> afterward
  - And if it is not assigned to _c#
    - In this case, reuse this temp (prevents redundant, but not infinite stacking)
  - Does this by checking for future invalidation before inserting
  - Prevents waste from inserting and using only once => not a <u>common</u> subexpression
  - Prevents undoing work by copy statement
- The _c# differentiates it from the _t# temporaries from the frontend
- Invalidate common subexpression if:
  - A variable used in the subexpression was assigned a value on any path
  - the if/else context the common subexpression was created in is left (goes <u>out</u> of)
    - Works for n-deep nested if/else

Optimization – Copy-statement Elim.

- The types recognized:

```
a = b            a = b            a = b        a = b
x = a op c       x = c op a       x = !a       c = a

a = !v           a = b op c
x = a            x = a
```

- Propagate copy statement(s) for current line, then record current assignment
  - Records version before inserting (may need a second pass)
  - **Will <u>not</u> record with _c = …** (prevents <u>infinite stacking</u> issue)
  - Will not record x = x
- Invalidate when:
  - any of the variables in the recorded copy statement are written to
  - if/else context the assignment occurred in is exited
    - Works for n-nested if/else
- Will delete dead temporary variables
  - _t# assigned but never used again
  - Less values need to be in registers, reduce code size

Optimization – Heuristic

- **Goal** : Reduce # of redundant math operations and variables that need to be in registers
- CSE and copy-statement elimination shouldn't undo each other's work
  - CSE shouldn't introduce redundant work
    - e.g.: reuses _c#
    - Only insert temp. if subexpression is "common"
- Copy statement elimination creates/reveals common subexpressions that can be eliminated by CSE
- Each optimization records how many changes they made
- When both return zero changes, the fix point is reached
  - Otherwise, continue to loop
  - Since they don't undo each other's work, the fix point is guaranteed to be reached

Optimization – Heuristic Testing

```
_t0 = 1 + a;
x = _t0;
_t1 = 1 + a;
y = _t1;
z = a;
_t2 = 1 + a;
f = _t2;
```

Frontend TAC
a, _t0, _t1, _t2

CSE

```
_c0 = 1 + a;
_t0 = _c0;
x = _t0;
_t1 = _c0;
y = _t1;
z = a;
_t2 = 1 + z;
f = _t2;
```

Cpy st

Copy statement
ignores _c0

```
_c0 = 1 + a;
_t0 = _c0;
x = _c0;
_t1 = _c0;
y = _c0;
z = a;
_t2 = 1 + a;
f = 1 + z;
```

Dead var

```
_c0 = 1 + a;
x = _c0;
y = _c0;
z = a;
f = 1 + z;
```

CSE

```
_c0 = 1 + a;
x = _c0;
y = _c0;
z = a;
f = 1 + z;
```

Cpy st

```
_c0 = 1 + a;
x = _c0;
y = _c0;
z = a;
f = 1 + a;
```

CSE

CSE reuses _c0

```
_c0 = 1 + a;
x = _c0;
y = _c0;
z = a;
f = _c0;
```

Optimized TAC: a, _c0

Optimization – Heuristic Testing

```
_t0 = 1 + a;
x = _t0;
_t1 = 1 + a;
y = _t1;
z = a;
_t2 = 1 + a;
f = _t2;
```

0.009495 seconds for 5M loops

```
_c0 = 1 + a;
x = _c0;
y = _c0;
z = a;
f = _c0;
```

0.009452 seconds for 5M loops

1.0045 speedup, 0.45% faster

1% ~ 3% faster for larger tests
(more if large numbers and difficult math)