Feasible C

By Yan Yao
Advisor: Dr. Chris Pollett

Introduction

- A translator parses our feasible C code, and outputs usual C code.
- It only parses polynomial tagged function.
- Functions parsed by this translator are polynomial in its inputs.
  - Assumption: it takes a OS effectively constant time to perform certain operations such as creation of local variables, variable assigment.

Tools: Lex and Yacc

- Lexer recognizes tokens from the input strings.
  - The patterns lex uses to match the token are expressions.
  - For every expression there is an associated action which usually returns a token to the parser.

Tools:Lex and Yacc

- Yacc recognizes grammars
  - It reads token, and recognizes rules.
  - Example (a grammar might be used to build a calculator program):

    statement: NAME = *expression*;

    expression: NUMBER PLUS NUMBER | NUMBER MINUS NUMBER ;

Implementation: Restrictions

- Variable Declaration
  - Checks the symbol table to find if it is a variable redefinition.
  - If it’s a valid declaration, puts the variable into symbol table.

Implementation: Restrictions

- Assignment
  - Checks if the expression on the left of the assignment operator is one of the function input parameter.
  - Checks if the expression is a valid variable (if it's declared).

Implementation: Restrictions

- for loop
  - If the expression_3 is decrement, the comparison operator in expression_2 has to be ‘>=‘ or ‘>’. Otherwise it might be a infinite loop. We don’t allow this situation.

```
for (i = 0; i<=10; i--)
{   statement }
```

  - Needs all three expressions, otherwise it might be a infinite loop.

```
for(expression_1;expression_2; expression_3)
{
    statement_1;
    statement_2;
}
```

Implementation: Restrictions

- while loop
  - Requires programmer to add an expression maxcount(expression). A feasible C while loop has the form:

```
while((expression)&&maxcount(argi))
{
statement;
}
```

Implementation: Restrictions

- while loop in input:

```
while((expression)&&maxcount(arg))
        {
        statement;
        }
```

arg has to be a function input or a constant.

- The preprocessor ouputs:

```
int loop_control_0 = arg;
while((expression)&&
    (loop_control_0--)>0)
{
 statement;
}
```

Implementation: Restrictions

- Recursive function
  - recursion control variable slot.
    - If a polynomial function is a recursive function, then there must be a recursion control variable slot in its input parameter list.

```
polynomial void myfunction(int i,  int $j)
{
statement
myfunction( i, $j);
}
```

Implementation: Restrictions

- Recursive function

```
polynomial void myfunction(int i, int $j)
                    {
                    statement
                    myfunction( i, $j);
                    }
```

- The preprocessor outputs:

```
void myfunction(int i, int $j)
{
if((j--)==0) return ;
statement
myfunction(i , $j);
}
```

Implementation: Restrictions

- Data Types
  - Allows primary data types: int, char, double, float, signed, unsigned.
  - No pointers. (it might cause non-polynomial execution)
  - Struct/union are allowed, but only has primary types.
  - Arrays: needs boundary checking. So we have struct:

```
typedef struct ArrayInt
 {
int *arr;
 int len;
 };
```

If sees int[] a =new int[*expression*];

it outputs:

```
ArrayInt a;
a.arr = (int *)malloc(expression *sizeof(int));
 a.len = expression;
```

Maps a[expression] to: a[((i<a.len&& expression >= 0)? expression : 0)]

Also adds: free(a.arr) at the end of the block.

Implementation: Restrictions

- Function calls
  - Arguments type checking
  - Only function has recursion control variable slot can make function call.
  - Only functions in the polynomial function list can be called.

Implementation: Restrictions

- Function prototype
  - A polynomial tagged function prototype will be put into the function list.
  - The information of function input parameters types are also stored in the list.

First Main Result – Complexity

Expression compleixty

- The primary expressions are variables and constants of type *int*, *long*, or *double*. The operation complexity of primary expressions is defined as constant 1.
- Expressions with unary operators such as ++*expression*, --*expression,* and -*expression* have the complexity of complexity of *expression* plus 1.
- The expressions with multiplicative operators of the form of
  - *expression _1* expression_2*
  - *expression_1 / expression_2*
  - *expression_1 % expression_2*

  have the complexity of complexity of (*expression_1*+1) * (complexity of *expression_2*+1).

First Main Result – Define Complexity

- Operation complexity
  - The complexity of expressions with additive operators of the form of
    - *expression_1* + *expression_2*
    - *expresiion_1* – *expression_2*
  - is defined as complexity of *expression_1* + complexity of *expression_2*.
  - The expressions with relational or equality operators of the form of
    - *expression_1* < *expression_2*
    - *expression_1* > *expression_2*
    - *expression_1* <= *expression_2*
    - *expression_1* >= *expression_2*
    - *expression_1* == *expression_2*
    - *expression_1* != *expression_2*
  - have the complexity of complexity of *expression_1* + complexity of *expression_2*.

First Main Result – Complexity

- Expression complexity
  - The complexity of expressions with bitwise/logical AND and OR operators of the form of
    - *expression_1* ^ *expression_2*
    - *expression_1* | *expression_2*
    - *expression_1* && *expression_2*
    - *expression_1* || *expression_2*
  - is defined as complexity of *expression_1* + complexity of *expression_2.*
  - For the expressions with operators such as conditional operator and assignment operators, the complexity is also defined as the sum of the complexity of every expression.
  - If the expression is simply a function call, we handle it the same as the function calls. As we defined above the complexity of function call in the complexity of the function.
  - If the expression is used to assign a function to a variable, we say the complexity of this expression is the complexity of the function plus 1.

First Main Result – Complexity

- Complexity of Statements
  - a variable declaration statement in the form:

    char a, b, c;

  complexity = number of items in the list;
- For an assignment such as

  a = *expression*;

  complexity = 1 + complexity of *expression*;
- for a *return* statement
  - *return expression*
  - complexity = complexity of the expression;

First Main Result – Complexity

- *for* loop

  for (*i=0; i* < *j; i*++)

  {statement}

  - complexity = j * complexity(statement(i)) ;
- *for* (*i=10; i* > *j; i*--) statement
- if….else….

  if(*expression*) statement_1 else statement_2

  - The complexity of this statement is the larger one of the complexity of statement_1 and statement_2, plus the complexity of expression of *expression*.

First Main Result – Complexity

- in a *while* loop

  while (*expression1&maxcount(expression2)*) statement;

  complexity = *expression2* * complexity(statement) ;

- for a *switch* statement,

  switch (*expression*){

  case *constant1*: statement;

  break;

  case *constant2*: statement;

  break;

  ….

  case *constantn*: statement;

  break;

  }

  – complexity = sum of complexity of every *case* statement + complexity of *expression* ;

First Main Result – Complexity

- statement block:

  {
  statement_1;
  statement_2
  …
  statement_n;
  }

  - complexity of statement block = sum of the complexity of all the statements in the block.
- function call
  - The complexity of function call = the complexity of the polynomial function

First Main Result – Theorem 1

- Theorem 1.
  - The runtime of any C expression of the formexpr(arg_1, … arg_n) where arg_n are the variables appearing in the expression or the runtime of any C function of the form

    type_of_return_value function_name (arg_1, arg_2… arg_n)

    parsed by this preprocessor is bounded by

    $O((abs(argument_1)+abs(argument_2)+\ldots+abs(argument_n)\ )f(m')+\ f(m'))$

    $f(m') = 2m'$

    If argumenti is an array, then in the above we replace this argument by the sum of the absolute values of its elements. That is:

    abs(argumenti [0])+abs(argumenti [1])+…+abs(argumenti [n-1])

    where n is the length of the array.

First Main Result – Theorem 1

- Proof by induction
  - An expression containing two primary expressions and one operator might look like:

    $expr_1(arg_1, arg_2 \ldots arg_m)$ op $expr_2(arg_1, arg_2 \ldots arg_n)$.

    op ‘ *’ is representative of all the other cases. By the induction hypothesis, the runtime of $expr_1$ is .

$$O(\sum_{i=0}^{m} abs(\arg_i)^{f(m_1')} + f(m_1'))$$

First Main Result – Theorem 1

- The runtime of $expr_2$ is $O(\sum_{j=0}^{n} abs(\arg_j)^{f(m_2')} + f(m_2'))$

  So by the induction hypothesis we have that the runtime of

  $expr_1(arg_1, \ldots arg_m) * expr_2(arg_1, \ldots arg_n)$

  can be bounded by .

$$O(\sum_{i=0}^{m}(abs(\arg_i)^{f(m_1')} + f(m_1') + 1) * (\sum_{i=0}^{n}(abs(\arg_i)^{f(m_2')} + f(m_2') + 1))$$

First Main Result – Theorem 1

- Let = z. Using this, the run-time of expr1(arg1…) * expr2(arg1…) is A:

$$O((z^{2^{m_1}} + 2^{m_1'} + 1) * (z^{2^{m_2}} + 2^{m_2'} + 1)) \leq O((z^{2^{m_1}'} + 2^{m_1'} + 2)(z^{2^{m_1}} + 2^{m_2}{}' + 2))$$

First Main Result – Theorem 1

- Let z≥2(we assume the inputs to the expression is greater or equal two)

$$\mathrm{A} \le O(z^{2^{m_1'}+2^{m_2'}} + z^{2^{m_1'}+2^{m_2'+1}} + z^{2^{m_1'2}+2^{m_1'+1}} + 2^{m_1'+m_2'+2}) \le O(3z^{2^{m_1'}+2^{m_2'+1}} + 2^{m_1'+m_2'+2})$$

$$A \le O(3z^{2^{m_1'+m_2'+1}} + 2^{m_1'+m_2'+2})$$

$$\because 3z^{2^{m_1'+m_2'+1}} \le z^{2^{m_1'+m_2'+1}} z^2$$

$$\therefore A \le O(z^{2^{m_1'+m_2'+2}} + 2^{m_1+m_2+2'}) \le O(z^{2^{(m_1'+1)*(m_2'+1)}} + 2^{(m_1+1)*(m_2+1)'})$$

First Main Result – Theorem 1

– Let $m'' = (m_1' + 1) * (m_2' + 1), f(m'') = 2^{(m_1' + 1) * (m_2' + 1)}$

and the runtime of the expression is bounded by

$$O(z^{2^{m_1' * m_2'}} + 2^{m_1' * m_2'})$$

So induction hypothesis holds. Therefore the theorem is true.

First Main Result – Theorem 1

- In the base case which is a function contains zero statement in the function body:

```
type_of_returnvalue myfunction(argument1, …argumentn)
{
}
```

We can see that the runtime of an empty function is bounded by a constant. Therefore the theorem is true whenthe function has zero statement.

First Main Result – Theorem 1

- In functions which contains more than three statements
  - myfunction (argument1, …argumentn)

```
{
    statement_1;
    statement_2;
    statement_3;
    …..
    statement_n;
}
```

  - myfunction is equivalent to

```
myfunction2(argument1, …argumentn)
{
    variable declarations;
function_1(argument1, …argumentn,&additional_arguments);
function_2(argument1, …argumentn, &additional_arguments);
}
```

Here the first statement contains the local variable declarations statements of myfunction. They might contain statements such as `char a;', but as we have specified in our restrictions they cannot contain declarations and assignments such as `char a=10;'.

First Main Result – Theorem 1

The additional_arguments passed to function1 and function2 come from these local variables of myfunction. Roughly, function1 and function2 will look like:

```
function_1(argument1, …argumentn, &additional_arguments)
{
    statement_1;
    statement_2;
    ……..
    statement_n-3;
}
function_2(argument1, …argumentn, &additional_arguments)
{
statement_n-2;
statement_n-1;
statement_n;
}
```

First Main Result – Theorem 1

- By the induction hypothesis, the runtime of function_1 is $O(\sum_{i=0}^{n}(abs(\arg_i)^{f(m_1')} + f(m_1')))$
- The runtime of function_2 is

$$O(\sum_{i=0}^{n}(abs(\arg_i) + \sum_{i=0}^{n} abs(\arg_i)^{f(m_1')} + f(m_1')^{f(m_1')})^{f(m_2')} + f(m_2')))$$

First Main Result – Theorem 1

- Let g = $\sum_{i=0}^{n} abs(\arg_i)$
- the runtime of myfunction A

$$O(g^{2^{m1'}} + 2^{m_1'} + (g + g^{2^{m1'}} + 2^{m_1'})^{2^{m_2'}} + 2^{m2'}$$

First Main Result – Theorem 1

- The value of g is greater or equal 2 on all but finitely many inputs (namely, the input 1). So we have $A \le O(g^{2^{m_1'+m_2'}+2^{m_2'}+1} + 2^{m_1'+m_2'}) \le O(g^{2^{m_1'+m_2'+1}} + 2^{m_1'+m_2'+1})$

  Let $m'' = (m_1' + m_2' + 1), f(m'') = 2^{(m_1'+m_2'+1)}$

  and the runtime of the expression is bounded by

  $O(g^{2^{m_1'*m_2'+1}} + 2^{m_1'*m_2'+1})$

  So induction hypothesis holds. Therefore the theorem is true.

Simulating polynomial in the argument time Turing Machines

- Assumptions:
  - The Turing Machine never uses more tape squares than can be indexed by one int (roughly, 232 squares) and the total number of steps of the computation can be stored in one int.
  - The output of the computation is the contents of the tape when the halt state is entered.
  - The Turing Machine only has one tape.
  - The alphabet of the Turing Machine only has two symbols 0,1 (and of course blank).

Simulating polynomial in the argument time Turing Machines

- Theorem
  - Let P be a Turing Machine which on all inputs x runs in time bounded by x^k for some fixed k and otherwise restricted as above. Then there is a function f_P that our preprocessor will validate as polynomial such that on input and outputs satisfying the above restrictions f_P will output the same value as P.

Simulating polynomial in the argument time Turing Machines

- Proof
  - Let P be a Turing Machine as above. Its input given the restrictions above will be a string of 0's and 1's. We assume these are passed to our polynomial function as an array of int's called input. The output will be returned by reference in an array called output which we assume can hold all of the tape squares needed for the computation. Below is a skeleton of what our function will look like:

Simulating polynomial in the argument time Turing Machines

```
polynomial  void  f_P(int[] input, int[] output)
{
int len, i, maxtime, state;
int head_pos;
len = input.len;
for ( i = len-1; i>=0; i--)
{
output[i] = input[i];
maxtime = 2*maxtime + input[i];
}
/*new now use output as the tape we do our simulation on */
maxtime = maxtime * … *maxtime; //k th power
head_pos =0;
for( i = maxtime; i>0; i--)
{
        switch (state)
{
                case 0:
if (output[head_pos] == 0)
{
        state = new state;
        output[head_pos] = new value;
        head_pos = new_head_pos;
}
else if (output[head_pos] == 1)
{
        state = new state;
        output[head_pos] = new value;
        head_pos = new_head_pos;
```