Project 1: detecting and analyzing loop constructs

October 12, 2017

1 Project description

The goal of this project is to program some part of a compiler that can handle loop constructs in C, performing some dataflow analysis. You will build a simple tool that takes as input a C program, and outputs information about the looping constructs in the program.
The first step of your project should be to build an AST (abstract syntax tree) of the input program and understand it. Some libraries already provide functions to iterate over statement of a function body for example, and you are definitely encouraged to use them. The syntax of the programs you can expect as input is described in Section 3. It is a subset of C, so any tool that can analyze C code can help you, but you can use this as a reference if you have a doubt.
In our subset of C, right-hand side of assignments do not have side-effects, and the only pointers we handle are array pointers. We also remove `typedef` constructs and use only simple scalar types in C and the arrays we can build with them.
Once you have the AST, you can start to analyze the program.

The goal of the preliminary analysis should be detect the loops used in the function bodies. Then, for each loop, you should be able to output the following information (without using external libraries):

- Control-flow and data-flow information:
  - live variables at the entry of the loop and reaching expressions.
  - the set of read/write variables in the loop body.
  - the set of variables that are used after the last iteration of loop in the function.
  - the loop body's AST.
- when possible, extract the index, the guard condition and the update operation of the loop. For example, in the following loop:

  ```
  for(int i = 0; i++; i<n){
    a[i] = a[i-1] + a[i];
  }
  ```

  the index variable `i` is updated by the assignment `i++` and the guard condition is `i<n`. This can be harder in a `while` loop!
- Nested loops information. Each statement in the loop nest (except iteration statements) should have an identifier. You should then be able to output the loop carried dependence graph and iteration space traversal graph. For this task you should not expect too complicated loop bodies, and array subscripts will use only linear combination of the loop indexes.

Your program should output this information in a synthetic manner in a text file, but you are free to choose the format yourselves.

Notation The project has three main parts: getting the control-flow and data-flow information (40pts), extracting index, guard condition and index update (30pts) and the information about nested loops (30pts).

Bonus Once you have the dependence graphs for the nested loops, you can try to output information about what loop transformations are applicable!

Language and provided code

Since you can use **language of your choice**, there is **no code provided** for the project. You will have to build everything yourselves, however you should use available libraries or packages for the tasks that are not essential to the project. In particular, you should use an available lexer and parser to transform the C code into a suitable format that you can manipulate.
Choosing a language you are comfortable with and that has helpful libraries for this project will save you a lot of time!

Examples

- OCaml with CIL.
- Python with PyCParser.
- Racket and c-utils.
- and many others ...

In any case, parsing shouldn't be the problem, you can always start with the assumption that your program takes as ainput the AST of a program. If you have any doubt about your choice, e-mail victorn@cs.toronto.edu or meet during office hours.

2 Deliverables

- the source code of your tool.
- a manual, explaining how to use it. It should be compilable and executable on a CDF/Teaching Labs machine.
- a short report explaining your approach and the difficulties you encountered.

3 Program syntax

This is a subset of the grammar defined in ISO/IEC 9899:TC3.

We will use the list construction with any syntax object, it refers to an empty list or a comma-separated list of the syntax objects:

$\langle \textit{List X} \rangle$ ::= $\langle \textit{empty} \rangle$
  | $\langle \textit{NonEmptyList X} \rangle$

$\langle \textit{NonEmptyList X} \rangle$ ::= $\langle X \rangle$
  | $\langle X \rangle$ ',' $\langle \textit{List X} \rangle$

The syntactic element specified in brackets (ex: {'+', '-'} means one of the contained elements can be used to produce the syntax object.

3.1 Expressions

This is inspired from the C99 standard with some restrictions:

- there are no *assignment-expressions*. For example, the following expression is illegal in our subset:

  ```
  char c;
  int i;
  long l;
  l=(c=i);
  ```

- The only pointers are used in array subscripts, therefore we removed address and indirection operators **&** and .

To summarize the features of the expressions of the language: we have function calls, array subscripts, unary expressions and binary expressions.

⟨*primary-expression*⟩ ::= ⟨*identifier*⟩
 | ⟨*constant*⟩
 | ⟨*string-literal*⟩
 | '(' ⟨*expression*⟩ ')'

⟨*postfix-expression*⟩ ::= ⟨*primary-expression*⟩
 | ⟨*postfix-expression*⟩
 | ⟨*postfix-expression*⟩ '[' ⟨*expression*⟩ ']'
 | ⟨*postfix-expression*⟩ '('⟨*List expression*⟩] ')'

⟨*unary-expression*⟩ ::= ⟨*postfix-expression*⟩
 | '+' | '-' | ' ' | '!' ⟨*unary-expression*⟩

⟨*multiplicative-expression*⟩ ::= ⟨*unary-expression*⟩
 | ⟨*multiplicative-expression*⟩ { '*' '/' '%' } ⟨*unary-expression*⟩

⟨*additive-expression*⟩ ::= ⟨*unary-expression*⟩
 | ⟨*additive-expression*⟩ {'+' '-' } ⟨*multiplicative-expression*⟩

⟨*shift-expression*⟩ ::= ⟨*additive-expression*⟩
 | ⟨*shift-expression*⟩ { '<<' '>>' } ⟨*additive-expression*⟩

⟨*relational-expression*⟩ ::= ⟨*shift-expression*⟩
 | ⟨*relational-expression*⟩ {'<' '>' '<=' '>=' } ⟨*shift-expression*⟩

⟨*equality-operators*⟩ ::= ⟨*relational-expression*⟩
 | ⟨*equality-expression*⟩ { '==' '!=' } ⟨*relational-expression*⟩

⟨*AND-expression*⟩ ::= ⟨*equality-expression*⟩
 | ⟨*AND-expression*⟩ '&' ⟨*equality-expression*⟩

⟨*exclusiveOR-expression*⟩ ::= ⟨*AND-expression*⟩
 | ⟨*exclusiveOR-expression*⟩ '^' ⟨*AND-expression*⟩

⟨*inclusiveOR-expression*⟩ ::= ⟨*exclusiveOR-expression*⟩
 | ⟨*inclusiveOR-expression*⟩ '|' ⟨*exclusiveOR-expression*⟩

⟨*logicalAND-expression*⟩ ::= ⟨*inclusiveOR-expression*⟩
 | ⟨*logicalAND-expression*⟩ '&&' ⟨*inclusiveOR-expression*⟩

⟨*logicalOR-expression*⟩ ::= ⟨*logicalAND-expression*⟩
 | ⟨*logicalOR-expression*⟩ '||' ⟨*logicalAND-expression*⟩

⟨*conditional-expression*⟩ ::= ⟨*logicalOR-expression*⟩
 | ⟨*logicalOR-expression*⟩ '?' ⟨*expression*⟩ ':' ⟨*conditional-expression*⟩

⟨*expression*⟩ ::= ⟨*conditional-expression*⟩

3.2 Declarations

⟨*declaration*⟩ ::= ⟨*type-specifier*⟩ ⟨*List init-declarator*⟩

⟨*init-declarator*⟩ ::= ⟨*declarator*⟩
| ⟨*declarator*⟩ ’=’ ⟨*initializer*⟩

⟨*declarator*⟩ ::= ⟨*identifier*⟩
| ’*’ ⟨*declarator*⟩
| ⟨*identifier*⟩ ’[’ ⟨*constant*⟩ ’]’

We use the following type specifiers - or *basic types* :

⟨*type-specifier*⟩ ::= ’void’
| ’char’
| ’int’
| ’float’
| ’_Bool’

3.3 Statements and blocks

⟨*statement*⟩ ::= ⟨*compound-statement*⟩
| ⟨*expression-statement*⟩
| ⟨*selection-statement*⟩
| ⟨*iteration-statement*⟩
| ⟨*return-statement*⟩

⟨*compound-statement*⟩ ::= ” ⟨*block-item-list*⟩ ”

⟨*block-item-list*⟩ ::= ⟨*block-item*⟩
| ⟨*block-item*⟩ ⟨*block-item-list*⟩

⟨*block-item*⟩ ::= ⟨*declaration*⟩ ’;’
| ⟨*statement*⟩

⟨*expression-statement*⟩ ::= ⟨*expression-assign*⟩ ’;’

⟨*expression-assign*⟩ ::= ⟨*expression*⟩
| ⟨*expression*⟩ { ’=’ ’+=’ ’-=’ ’&=’ ’||=’ } ⟨*expression*⟩
| ⟨*expression*⟩ { ’++’ ’–’ }
| { ’++’ ’–’ } ⟨*expression*⟩

⟨*selection-statement*⟩ ::= ’if’ ’(’ ⟨*expression*⟩ ’)’ ⟨*statement*⟩
| ’if’ ’(’ ⟨*expression*⟩ ’)’ ⟨*statement*⟩ ’else’ ⟨*statement*⟩

⟨*iteration-statement*⟩ ::= ’while’ ’(’ expression ’)’ ⟨*statement*⟩
| ’do’ ⟨*statement*⟩ ’while’ ’(’ ⟨*expression*⟩ ’)’ ’;’
| ’for’ ’(’ ⟨*expression-statement*⟩ ’;’ ⟨*expression*⟩ ’;’ ⟨*expression-statement*⟩ ’)’ ⟨*statement*⟩
| ’for’ ’(’ ⟨*declaration*⟩ ’;’ ⟨*expression*⟩ ’;’ ⟨*expression-statement*⟩ ’)’ ⟨*statement*⟩

⟨*return-statement*⟩ ::= ’return’ ⟨*expression*⟩* ’;’

3.4 Program

A program is simply a list of *external-declarations*:

⟨*program*⟩ ::= ⟨*List external-declaration*⟩

⟨*external-declaration*⟩ ::= ⟨*function-definition*⟩
| ⟨*declaration*⟩

$\langle$*function-definition*$\rangle$ ::= $\langle$*declaration*$\rangle$ '(' $\langle$*List declaration*$\rangle$ ')' $\langle$*compound-statement*$\rangle$