CS323 Project Phrase 3 Report

12112421 王德涵 12112620 徐霄阳 12112822 孔繁初

Assumptions

As described in the requirement document, our implementation is based on the following assumptions:

Assumption 1 All tests are free of lexical/syntax/semantic errors (suppose there are also no logical errors).
Assumption 2 There are only integer primitive type variables.
Assumption 3 There are no global variables, and all identifiers are unique.
Assumption 4 The only return data type for all functions is **`int`**.
Assumption 5 All functions are defined directly without declaration.
Assumption 6 There are no structure variables or arrays.
Assumption 7 Function's parameters are never structures or arrays.

Additionally, we also **modified (or revoked) assumption 6 and 7**, that is:

Modified Assumption 6 and 7 Structure variables can appear in the program, and they can be declared as function parameters. Still, assignment operations will not be directly performed on a structure variable.

We also added other assumptions:

Assumption 8 Non-conditional expressions can also be used in control flow statements. If the value of the expression is **`0`**, the expression is regarded as FALSE. Otherwise, the expression is regarded as TRUE.

Translation Schemes

Most of our translation schemes are identical to the schemes described in the requirement document. However, to **lessen the number of instructions of control flow statements**, we largely modified **`translate_cond_Exp`** and **`translate_Stmt`**. In **`translate_cond_Exp`**, we added two more arguments, **`jump_only_if_true`** and **`jump_only_if_false`**. If **`jump_only_if_true`** is set as **TRUE**, the generated code **won't jump** if the conditional expression is **FALSE**. The **`jump_only_if_false`** does similar things. Furthermore, to support **Assumption 8**, in **`translate_cond_Exp`**, we also added the case where the input **Exp** is an arithmetic expression.
Our modified **`translate_cond_Exp`** and **`translate_Stmt`** is shown below:

`translate_cond_Exp(Exp, label_true, label_false, jump_only_if_true, jump_only_if_false) = case Exp of`	
`Exp1 relop Exp2`	**`tmp1 = new_place()`** **`tmp2 = new_place()`**

	code1 = translate_Exp(Exp1, tmp1)
	code2 = translate_Exp(Exp2, tmp2)
	if jump_only_if_true:
	  code3 = [IF tmp1 relop tmp2 GOTO label_true]
	  return code1 + code2 + code3
	else if jump_only_if_false:
	  code3 = [IF tmp1 reversed_relop tmp2 GOTO label_true]
	  return code1 + code2 + code3
	else:
	  code3 = [IF tmp1 relop tmp2 GOTO label_true]
	  code4 = [GOTO label_false]
	  return code1 + code2 + code3 + code4

Exp1 AND Exp2	if jump_only_if_true:
	  label1 = new_label();
	  code1 = translate_cond_Exp(Exp1, NULL, label1, 0, 1)
	  code2 = translate_cond_Exp(Exp2, label_true, NULL, 1, 0)
	  code3 = [LABEL label1]
	  return code1 + code2 + code3
	else if jump_only_if_false:
	  code1 = translate_cond_Exp(Exp1, NULL, label_false, 0, 1)
	  code2 = translate_cond_Exp(Exp2, NULL, label_false, 0, 1)
	  return code1 + code2
	else:
	  code1 = translate_cond_Exp(Exp1, NULL, label_false, 0, 1)
	  code2 = translate_cond_Exp(Exp2, label_true, label_false, 0, 0)
	  return code1 + code2

Exp1 OR Exp2	if jump_only_if_true:
	  code1 = translate_cond_Exp(Exp1, label_true, NULL, 1, 0)
	  code2 = translate_cond_Exp(Exp2, label_true, NULL, 1, 0)
	  return code1 + code2
	else if jump_only_if_false:
	  label1 = new_label();
	  code1 = translate_cond_Exp(Exp1, label1, NULL, 1, 0)
	  code2 = translate_cond_Exp(Exp2, NULL, label_false, 0, 1)
	  code3 = [LABEL label1]
	  return code1 + code2 + code3
	else:
	  code1 = translate_cond_Exp(Exp1, label_true, NULL, 1, 0)
	  code2 = translate_cond_Exp(Exp2, label_true, label_false, 0, 0)
	  return code1 + code2

NOT Exp	return translate_cond_Exp(Exp, label_false, label_true, jump_only_if_false, jump_only_if_true)

LP Exp RP	return translate_cond_Exp(Exp, label_true, label_false, jump_only_if_true, jump_only_if_false)

else (arithmetic expressions)	if jump_only_if_true:
	  tmp = new_place()
	  code1 = translate_Exp(Exp, tmp)
	  code2 = [IF tmp == #0 label_true]
	  return code1 + code2
	else if jump_only_if_false:
	  tmp = new_place()
	  code1 = translate_Exp(Exp, tmp)
	  code2 = [IF tmp != #0 label_false]
	  return code1 + code2
	else:
	  tmp = new_place()
	  code1 = translate_Exp(Exp, tmp)
	  code2 = [IF tmp == #0 label_true]
	  code3 = [GOTO label_false]
	  return code1 + code2 + code3

translate_Stmt(Stmt) = case Stmt of	
(Other cases are identical to what described in official document)	
IF LP Exp RP Stmt	label_false = new_label() code1 = translate_cond_Exp(Exp, NULL, label_false, 0, 1) code2 = translate_Stmt(Stmt) + [LABEL label_false] return code1 + code2
IF LP Exp RP Stmt1 ELSE Stmt2	label_else = new_label() label_endif = new_label() code1 = translate_cond_Exp(Exp, NULL, label_else, 0, 1) code2 = translate_Stmt(Stmt1) + [GOTO label_endif] + [LABEL else] code3 = translate_Stmt(Stmt2) + [LABEL endif] return code1 + code2 + code3
WHILE LP Exp RP Stmt	label1 = new_label() label2 = new_label() code1 = [LABEL label1] + translate_cond_Exp(Exp, NULL, label2, 0, 1) code2 = translate_Stmt(Stmt) + [GOTO label1] + [LABEL label2] return code1 + code2

Optimization

We use DAG to optimize our generated codes. The optimization contains following steps:

1. Generate blocks:
2. Generate DAGs according to the blocks
3. Optimize useless registers and codes
4. Rebuild our code

The details are following:

0. Data structure:

(1). We use structure Code to store our codes, one pointer Code* represent one single code. Var “type” represent the command of the code:

type 0: LABEL tk1 :
type 1: FUNCTION tk1 :
type 2: tk1 := tk2
type 3: tk1 := tk2 + tk3
type 4: tk1 := tk2 - tk3
type 5: tk1 := tk2 * tk3
type 6: tk1 := tk2 / tk3
type 7: tk1 := &tk2
type 8: tk1 := *tk2
type 9: *tk1 := tk2
type 10: GOTO tk1
type 11: IF tk1 relop tk2 GOTO tk3
type 12: RETURN tk1
type 13: DEC tk1 size
type 14: PARAM tk1
type 15: ARG tk1
type 16: tk1 := CALL tk2
type 17: READ tk1
type 18: WRITE tk1

(2). We use structure Block to store the message in one block

(3). We use structure Dnode to store the nodes in DAGs, specially, the var “operator” represent the operation type of the node:

0. init
1. +
2. -
3. *
4. /
5. fun
6. read
7. if
8. assign

(4). We use structure Export to store extra operator of a node, the type of structure represent the operation type, and relop represent to the signal type of if operation

type 1. var
type 2. write
type 3. arg
type 4. return
type 5. if
relop 0: <
relop 1: <=
relop 2: >
relop 3: >=
relop 4: !=
relop 5: ==

1. **Generate blocks**

   We scan all code from top to bottom. When we meet command type = 0,1,13, We immediately generate a new block. When we meet command type = 10,11,12, we immediately end this block and generate a new block for next line's code.

2. **Generate DAGs**

   We scan the codes in each block:

   For Type 2, we will check whether the token on the right side of equal mark is a const or has used in other nodes. If the in degree of the right side is not 0 or the right side is a const, we generate a new node for this command, and set the operator = 8. Else, we attach the token on the left side to the right side

   For Type3-6, we will check whether there is a node satisfied that it's in degree is zero and it's all other parameters is similar to the code. If there is, attach left token to the node, or we need to generate a now node for it.

   For Type 11, we generate a special "if" node, set operator to 7
   For Type 13, we assign the token is a structure
   For Type 15, we generate a new structure Export, attach it to the corresponding node(according to token).
   For Type 16, we generate a new node, set operator to 5
   For Type 17, we generate a new node, set operator to 6
   For Type 18, we attach an export to corresponding node.

3. **Optimize useless registers and codes**

   a) For each node with multiple registers, we delete useless registers.
   b) For each zero in degree node without an export, delete it

4. **Rebuild our code**

   We sort all nodes by topology. Specially, when two nodes have the same in degree, we will compare their location in the origin code. The deeper it is, the bigger it is.
   For the biggest node, if it has export type, deal with it, delete the export type and put it

back to the node list. If it isn't, generate code during to it's operator and delete node. Repeating this until the queue is empty.

Extra Test Cases

In out extra test cases, both **test case 1 and 2** corresponds to **modified assumption 6 and 7**. In **test case 1**, nested struct is tested, showing that our translation can identify struct members in a struct variable. In **test case 2**, structure variable is declared as function parameters. **Test case 3** corresponds to **assumption 8**, where arithmetic expressions can be regarded as conditional expressions. **Test case 4** is not special, we use it to **test our optimization**.