Intermediate Code Generation implementaion report

2013-11394 김형모
2013-11412 오평석

1 Overview

In this phase of the project, we converted the abstract syntax tree into intermediate representation in three-address code form. The classes of AST nodes are : Scope, Statement, Expression. In general, scope has code symbols and temp symbols for statements. Each statement is interpreted into one(or more) three-address code. Each expression is translated to evaluated values. We may need temp symbols to use the value.

Array designators are also handled with expressions, but it has some issues including calcuating and calling pre-defined functions. Short-circuit codes are also difficult part of this phase. It requires generating many labels and gotos.

Lastly, we decided to merge 'opNeg' and integer constant even though our negation policy is simple.

2 Scope's ToTac()

Scope's ToTac() is trivial. It just attaches label to each statement (include last empty statement), and add 'goto' instruction to next label. At the end of ToTac(), it calls CleanupControlFlow() method to remove redundant labels and goto instructions.

```
0:
  statement 0
  goto 1
1:
  statement 1
  goto 2
2:
  ...
n-1:
  statement n-1
  goto n
n:
```

The last 'goto' instruction and the label 'n' will be removed by executing CleanupControlFlow().

3 Statement's ToTac()

Statement's ToTac() is pretty easy. When we face some statement, just gets the TAC of the expression and apply the exact operation. More detail descriptions are follwing:

3.1 CAstStatAssign::ToTac(CCodeBlock *cb)

It gets the TAC of RHS as src, and the TAC of LHS as dst. Next, assign the src to the dst. Finally the form of the TAC will be following:

```
assign dst <- src
```

3.2 CAstStatCall::ToTac(CCodeBlock *cb)

When we form the TAC of subroutineCall, we build the form as follows:

```
param n−1 <− arg(n−1)
...
param 1 <− arg(1)
param 0 <− arg(0)
call f
```

So we make the TAC of each arguments and build the instruction ‘param’ using this. At the end, we add the instruction ‘call’.

3.3 CAstStatReturn::ToTac(CCodeBlock *cb)

We build the TAC by using the instruction ‘return’. If we have return expression, we push the TAC of this expression to the instruction. Otherwise, just leaves the expression as NULL.

3.4 CAstStatIf::ToTac(CCodeBlock *cb)

The statement ‘if’ has the TAC form as follows:

```
if (the condition is true) goto if_true
goto if_false
if_true:
  (ifBody statement sequence)
  goto next:
if_false:
  (elseBody statement sequence)
next:
```

First, we create two labels with name “if_true” and “if_false”. By using these labels, we build two ‘goto’ instruction dependent on the condition’s result. Second, add ifBody’s TAC to the label “if_true” and ‘goto’ isntruction at the end of the “if_true” label. Finally, add elseBody’s Tac to the label “if_false”.

3.5 CAstStatWhile::ToTac(CCodeBlock *cb)

The statement ‘while’ has the TAC form as follows:

```
while_cond:
  if (the condition is true) goto while_body
  goto next
while_body:
  (whileBody statement sequence)
  goto while_cond
next:
```

The way to make TAC of ‘while’ statement is similar to ‘if’ statement.

4 Expression's ToTac()

Expression's ToTac() is a little different since it has value, especially boolean type. As the skeleton code gives us two overloading ToTac() function, we adapted this functions to make three-address-code. Usually, when the expression has boolean type, it creates more labels and 'goto' instructions to apply the appropriate control flow.

4.1 CAstStatBinaryOp::ToTac(CCodeBlock *cb)

When the binary operator is opAdd, opSub, opMul, or opDiv, the expression is integer type. In this case, we create temporary variable(will be returned at the end) and saves the result of the expression to the temporary variable.

Otherwise, the expression is boolean type. It creates three labes with name 'ltrue', 'lfalse', and 'lend', and creates the control flow by calling ToTac(cb, ltrue, lfalse). After this call, we create the temporary variable (will be returned at the end) and assigns true or false depending the result of the expression.

For example,

1. a := i + 3

```
add temp <- i , 3
assign a <- temp
```

2. b := i > 3

```
  if i > 3 goto ltrue
  goto lfalse
ltrue:
  assign temp <- 1
  goto lend
lfalse:
  assign temp <- 0
lend:
  assign b <- temp
  goto next
next:
```

4.2 CAstStatBinaryOp::ToTac(CCodeBlock *cb, CTacLabel *ltrue, CTacLabel *lfalse)

When the binary operator is RelOP, &&, or ||, the expression is boolean type. Since SnuPL/1 evaluates the expression's value by using lazy evaluation, we apply the short circuit if the operator is && or ||. At the start of the ToTac, we create new label with name 'nextCond'. We will use this label when we build short circuit.

RelOp

It gets the TAC of left operand and right operand. If (leftTac oper rightTac) is true, goto ltrue label. Else, goto lfalse label.

opAnd

First, it gets the TAC of left operand by calling left->ToTac(cb, nextCond, ltrue). Next, it attaches the 'nextCond' label. Finally, it gets the TAC of right operand by calling right->ToTac(cb, ltrue, lfalse). It says that "If the current condtion is true, then look the next condition. Otherwise the total condtion is false regardless of the remaining condition's result." It behaves as follows:

```
  if (cond1 is true) goto nextCond_1
  goto lfalse
nextCond_1:
  if (cond2 is true) goto nextCond_2
  goto lfalse
nextCond_2:
  ...
nextCond_n:
  if (condn is true) goto ltrue
  goto lfalse
ltrue:
  (some expression when the condition is true)
  goto next
lfalse:
  (some expression when the conditoin is false)
next:
```

opOr

It behaves similar when the operator is ‘opAnd’. It says that “If the current condition is true, the total condition is true regardless of the remaining condition’s result. Otherwise, then look the next condtion.”

It behaves as follows:

```
  if (cond1 is true) goto ltrue
  if (cond2 is true) goto ltrue
  ...
  if (condn is true) goto ltrue
  goto lfalse
ltrue:
  (some expression when the condition is true)
  goto next
lfalse:
  (some expression when the condition is false)
next:
```

4.2.1 CAstUnaryOp::ToTac(CCodeBlock *cb)

When the unary operator is opPos or opNeg, it should be an integer type. Usually it creates temporary variable and assigns the result of the TAC of operand to the temporary variable. Finally, it returns the temporary variable. But if the operand’s node is CAstConstant, it just returns constant with the value of CAstConstant considering the unary operator.

For example, consider the node : UnaryOp(“-”, 2147483648). It will returns “constant(-2147483648)”, instead of “neg t <- 2147483648”.

If the unary operator is opNot, now it shoule be a boolean type. We create three labels with name “ltrue”, “lfalse”, and “lend”. And we create the temporary variable (will be returned at the end) and construct the TAC of the control flow by calling ToTac(cb, ltrue, lfalse). Finally, we attaches the labels and add the ‘goto’ instruction to complete the control flow.

For example, when we face “a := !b” expression, our ToTac() function will works as follows:

```
  if (b is true) goto 1
  assign tmp <- true
  goto 2
1:
  assign tmp <- false
2:
  assign a <- tmp
```

4.2.2 CAstUnaryOp::ToTac(CCodeBlock *cb, CTacLabel *ltrue, CTacLabel *lfalse)

This fucntion will be called only when the unary operator is opNot. It just calles the operand’s ToTac function by swapping the argument ltrue and lfalse.

4.2.3 CAstStatSpecialOp::ToTac(CCodeBlock *cb)

Although the special operator is opAddress, opReference, and opCast, we only use the operator ‘opAddress’. So it just creates the temporary variable (will be returned at the end) and add ‘opAddress’ instruction to the variable. Remark that this function is not overloadded.

4.2.4 CAstFunctionCall:ToTac(CCodeBlock *cb)

It is similar to CAstStatCall::ToTac(CCodeBlock *cb), except for return value.

4.2.5 CAstFunctionCall:ToTac(CCodeBlock *cb, CTacLabel *ltrue, CTacLabel *lfalse)

It calls ToTac(cb) and saves it to retval. If retval is true, then goto ltrue label. Else, goto lfalse label.

```
if (retval is true) goto ltrue
goto lfalse
```

4.2.6 CAstDesignator::ToTac(CCodeBlock *cb)

It just returns the symbol's name.

4.2.7 CAstDesignator::ToTac(CCodeBlock *cb, CTacLabel *ltrue, CTacLabel *lfalse)

it calls ToTac(cb) and saves it. If it is true, then goto ltrue label. Else, goto lfalse label.

```
if (the designator is true) goto ltrue
goto lfalse
```

4.2.8 CAstArrayDesignator::ToTac(CCodeBlock *cb)

It calculates pointer of exact data and reference the pointer acourding to following steps.

First, we address pure array into pointer if needed. we should use pointer until return. Second, we calculate pointer to data location

The formular to evaluate location inside array as follows:

$$a[x1]...[xn] = (array\_offset) + (...((x1*C1 + x2)*C2 + x3)*C3 + \ldots + xn)*Cn$$

- &a = pointer to array 'a'
- array_offset = &a+ DOFS(&a)
- Ck = DIM(&a, k+1) when k = 1 to n-1
- Ck = data size of basetype

Last, we reference array location using CTacReference.

4.2.9 CAstArrayDesignator::ToTac(CCodeBlock *cb, CTacLabel *ltrue, CTacLabel *lfalse)

it calls ToTac(cb) and saves it. If it is true, then goto ltrue label. Else, goto lfalse label.

```
if (the array designator is true) goto ltrue
goto lfalse
```

4.2.10 CAstConstant::ToTac(CCodeBlock *cb)

It just returns the constant.

4.2.11 CAstConstant::ToTac(CCodeBlock *cb, CTacLabel *ltrue, CTacLabel *lfalse)

It saves the constant's value to cond. If cond is true, then goto ltrue label. Else, goto lfalse label.

```
if (cond is true) goto ltrue
goto lfalse
```

4.2.12 CAstStringConstant::ToTac(CCodeBlock *cb)

4.2.13 CAstStringConstant::ToTac(CCodeBlock *cb, CTacLabel *ltrue, CTacLabel *lfalse)

It just returns its symbol.