Compiler C to PDL

Philippe Geraldeli Araujo e Allan Patrick

09-08-2018

1 C to PDL

Fork do miniC feito por eubnara

Este compilador tem o objetivo de converter C para PDL(Propositional Dynamic Logic).

Program := (DeclList)? (FuncList)? // DeclList FuncList ou DeclList ou FuncList
DeclList := (Declaration)+ // Declaration ou DeclList Declaration
FuncList := (Function)+
Declaration := Type IdentList
IdentList := identifier (, identifier)* // identifier ou IdentList , identifier
Identifier := id ou id [intnum] // (Note) [,] are not symbols used in regular expression
Function := Type id ((ParamList)?) CompoundStmt
ParamList := Type identifier (, Type identifier)*
Type := int ou float
CompoundStmt := (DeclList)? StmtList
StmtList := (Stmt)*
Stmt := AssignStmt ou CallStmt ou RetStmt ou WhileStmt ou ForStmt ou IfStmt ou CompoundStmt ou ;
AssignStmt :=Assign
Assign := id = Expr ou id [Expr] = Expr
CallStmt := Call ;
Call := id ((ArgList)?)
RetStmt := return (Expr)? ;
Expr := MINUS Expr | MathRel Eqltop Expr | MathRel | Call | Ids
MathRel := MathEql Relaop MathRel | MathEql
MathEql := TERM Addiop MathEql | TERM
TERM := FACTOR Multop TERM | FACTOR
FACTOR := '(' Expr ')' | FLOATNUM | INTNUM
Id := ID | ID [Expr]

3 Prova de corretude do algoritmo:

So, Our miniC program doesn't follow the rule below.
1. ++;
2. According to this rule := CompoundStmt := (DeclList)? StmtList

Em PDL temos duas estruturas importantes, sendo elas programas e formulas, programas atômicos são os menores programas possíveis e formulas atômicas são as menores formulas possíveis. Nesse algoritmo um programa atômico tem no máximo 1 operando, foi escolhido essa forma por convenção de 3 registradores. Sendo assim, toda declaração de variável e função de atribuição com apenas 1 operador, um programa atômico, logo uma lista de instruções é um conjunto de programas atômicos. Por exemplo:

2 Algorithm Converter

Input = Arquivo em C
Output = Arvore/Arquivo

int i → [int i]

Int i = 10 → declaração: [int i] | Função: [i = 10]

Um programa pode ser descrito como uma lista de declarações ou uma lista de funções, sendo que pela recursão toda concatenação dos mesmos é tratada e as declarações são tratadas de forma vista anteriormente.

Algorithm 1: BuildTree(Program* head)

```
1 if headDeclaration != NULL then
2     visitDeclaration(headDeclaration);
3 if headFunction != NULL then
4     visitFunction(headFunction);
```

Algorithm 2: visitDeclaration(DECLARATION* decl)

```
1 if DeclList then
2     if Declaration then
3         insert(declarationType);
4         insert(declarationId);
5     if DeclList then
6         visitDeclaration(previousDeclaration);
```

Algorithm 3: visitFunction(FUNCTION* func)

```
1 if FunctionList then
2     if FunctionList then
3         visitFunction(previousFunction);
4     if Function then
5         insert('(');
6         if funcParameter != NULL then
7             insert(funcParameter);
8         visitCompoundStmt(FunctionCstmt);
```

Algorithm 4: visitCompoundStmt(COMPOUNDSTMT* cstmt)

```
1 if cstmtDeclaration != NULL then
2     visitDeclarationcstmtDeclaration;
3 if cstmtStatement != NULL then
4     visitStmt(cstmtStatement);
```

Algorithm 5: visitStmt(Stmt* stmt)

```
1  switch stmtS do
2      case Assign do
3          InsertSemicolon();
4          insert(stmtS_AssignID);
5          insert("=");
6          insert(stmtS_AssignExpression);
7      case Call do
8          insertSemicolon();
9          insert(stmtSCallIdentifier);
10         insert(CallArg);
11     case Return do
12         if Stmt_Return == NULL then
13             insert("return");
14         else
15             insert("return");
16             visitStmt(stmtS_Return);
17     case While do
18         if StmtSdo_while == true then
19             visitStmt(stmtS_while);
20             insert(WhileCondition);
21             visitStmt(stmtS_while);
22             insert(")*;¬(");
23         else
24             insert(WhileCondition);
25             visitStmt(stmtS_while);
26             insert(")*¬");
27             insert(WhileCondition);
28     case For do
29         visitStmtS(StmtSAssign);
30         insert(ForCondition);
31         visitStmt(stmtS_For);
32         visitStmt(stmtS_ForInc);
33         insert(ForCondition);
34     case If do
35         insert(If_condition);
36         VisitStmt(stmtS_if);
37         if stmtSelse != NULL then
38             insert(IfCondition);
39             visitStmt(stmtSelse);
40     case CompoundStmt do
41         visitCompoundStmtstmtS;
42     case Semicolon do
43         insert(";");
```

No caso de termos uma função vai recair em casos mais específicos que vão ser tratados isoladamente. Como por construção a recursão cuida de tratar o caso mais interno para o mais externo recairemos sempre sobre o caso mais básico. Temos então os casos das estruturas de repetição, condição e escopo. A condição de escopo é a mais simples, pois como em PDL é ignorado, um escopo é estruturado toda vez que a recursão entra em uma função, logo esse caso é ignorado pela conversão. Logo mostraremos que para um caso genérico Sigma de um subconjunto de C todas as transformações feitas são válidas já que temos prova de que as transformações de uma linguagem de programação para PDL está correto comprovadamente, logo iremos mostrar que a nossa conversão de algoritmo equivale a conversão que já está comprovada.

Dado pela definição de PDL no livro [1] o Assign é dado pela seguinte forma:

int i = 1 + 2 + 3 → declaração: [int i] | função: [i = 1 + 2 + 3]

Como visto anteriormente toda função Assign utiliza apenas 1 operando para o nosso algoritmo, logo no caso de mais operandos é utilizado Identifiers auxiliares para esse quesito, por exemplo:

int j = 1 → declaração: [int j] | função: [j = 1]

Algorithm 6: void Parts(struct EXPR* expr)

```
1  char var[] = "_tX";
2  int buffersize = 100;
3  char* variable = malloc(buffersize);
4  switch exprE do
5      case eId do
6          var[2] = ++x + '0';
7          strncpy(variable,var,buffersize);
8          Parts3[tam][0] = variable;
9          tam++;
10     case eIntnum do
11         var[2] = ++x + '0';
12         strncpy(variable,var,buffersize);
13         Parts3[tam][0] = variable;
14         tam++;
15     case eFloatnum do
16         var[2] = ++x + '0';
17         strncpy(variable,var,buffersize);
18         Parts3[tam][0] = variable;
19         tam++;
```

Na função Parts temos que:

Na linha 1 é criado o nome da variável. Na linha 2,3 é criado um buffer de tamanho 100, para alocar todas as variáveis necessárias. Na linha 8,13,18 é adicionado na tabela o nome da variável.

Logo temos que para o caso simples de 1 operando é trivial verificar que as transformações são idênticas, pois é fácil ver que é uma adição direta na arvore. Para n ¿ 1 operandos temos que é fácil ver ficar também visto que para uma função genérica teremos:

int g = n + (n+1) + (n+2) + (n+3) + ... + (n+m)

logo a transformação da declaração ficará:

aux n = (n + (m-1)) + (n+m) aux n-1 = (n + (m-2)) + aux n aux n-2 = (n + (m-3)) + aux n-1 ... aux = (n+1) + aux 2

g = (n) + aux

O compilador faz essa conversão na função visitExpr2, tendo então que esta transformação é trivialmente transformada em:

int g = n + (n+1) + (n+2) + (n+3) + ... + (n+m)

Logo como a conversão é equivalente a definição está correto.

3.1 Caso do IF

A conversão para PDL de um if simples ocorre da seguinte forma:

Theorem 1 *If a then b else c* $\rightarrow$ *a?;bU¬a;c*

Temos que o compilador faz essa conversão da seguinte forma:

Na linha 4 adiciona dois escopos de parêntese na arvore. Na linha 6 procura a condição do if para adicionar em seguida no visitExpr2. Na linha 7 fecha o parêntese e adiciona o "?". Na linha 10 vai para o Stmt do if. Na linha 13 é checado se existe um else, caso exista é adicionado o "U(¬(" na linha 12. Na linha 17 é adicionado a condição novamente. Na linha 18 termina o escopo do if.

Logo podemos inferir que os 2 casos de if são trivialmente equivalentes aos casos que ocorrem na definição de PDL.O algoritmo faz essa conversão e insere na árvore os casos possíveis gerando ramificações. Assim levando ao caso base de cada ramificação e sendo tratada pela recursão.

3.2 Caso Call

Nesse caso o que estamos fazendo na prática é nada mais nada menos que entrando em uma função normalmente, logo quando temos a chamada de uma função externa é fácil ver que é apenas uma função que será inserida na árvore normalmente.

3.3 Caso While e doWhile

O * no PDL é representado da forma que pode ser repetido zero ou mais vezes, logo não temos controle por quantas iterações que serão executadas no programa, no while temos essa abstração.

Temos que a conversão de um while para PDL ocorre da seguinte forma de acordo com a definição:

Algorithm 7: (void visitIf_s2(struct IF_S* if_s))

```
1  scopeTail = newScope(sIF, scopeTail);
2  _isTitlePrinted2 = false;
3  scopeTail→parent→if_n++;
4  insert(aux, "child", "( (");
5  aux2 = aux;
6  visitExpr2(if_s→cond);
7  insert(aux,"child",")? ");
8  if if_s→if_→s then
9      insert(aux,"child",";");
10 visitStmt2(if_s→if_);
11 insert(aux,"child"," )");
12 if if_s→else_ != NULL then
13     insert(aux, "child", "U(¬(");
14     aux2 = aux;
15     visitExpr2(if_s→cond);
16     insert(aux,"child",")? ");
17     if if_s→else_→s then
18         insert(aux,"child",";");
19     visitStmt2(if_s→else_);
20     insert(aux, "child", ")");
```

Theorem 2 *While a do b → (a?;b)*;¬a?*

Temos então o algoritmo do while:

```
1  if while_s→do_while == true then
2  |  insert(aux, "child", "(");
3  |  visitStmt2(while_s→stmt);
4  |  insert(aux, "child", "(");
5  |  aux2 = aux;
6  |  visitExpr2(while_s→cond);
7  |  insert(aux,"child",")?");
8  |  insert(aux, "child", ")*;¬(");
9  |  aux2 = aux;
10 |  visitExpr2(while_s→cond);
11 |  insert(aux,"child",")? ");
12 else
13 |  insert(aux, "child", "( (");
14 |  aux2 = aux;
15 |  visitExpr2(while_s→cond);
16 |  insert(aux,"child",")?");
17 |  visitStmt2(while_s→stmt);
18 |  insert(aux, "child", ")*;¬(");
19 |  aux2 = aux;
20 |  visitExpr2(while_s→cond);
21 |  insert(aux,"child",")?");
```

Nas linhas 1,12 é feito um teste caso seja um dowhile ou um while, no caso de ser um do while ele irá continuar na linha 2, caso contrário na linha 12. Na linha 2 é inserida um parêntese. Na linha 3 é adicionado uma vez tudo que possui dentro do while. Na linha 4 é feito mais uma adição de parêntese. Na linha 6 é adicionado a condição do dowhile. Na linha 8 é tratado da forma de um while normal. Na linha 13 é adicionado 2 parênteses. Na linha 15 é adicionado a condição do while. Na linha 16 se fecha o parênteses e adiciona o "?". Na linha 17 é adicionado tudo que o while possui internamente. Na linha 18,20,21 é adicionado a negação da condição e adicionado um ponto de interrogação.

4 Caso For

O for também pode ser tratada da mesma forma que o while mas com algumas ressalvas.Logo temos que a conversão de for para PDL ocorre da seguinte forma, no for temos 3 campos, o de declaração, condição e incremento. Sendo d a declaração, c a condição e i o incremento, temos o seguinte exemplo:

Assim a conversão é feita da seguinte forma:

(d;c?;E;i)*¬c?

No algoritmo temos que:

```
1 for c do
2 |_ E;
```

Algorithm 8: visitFor_s2 (struct FOR_S* for_s)

```
1  insert(aux, "child","(");
2  visitAssignStmt2(for_s→init);
3  insert(aux,"child",";(");
4  aux2 = aux;
5  visitExpr2(for_s→cond);
6  insert(aux,"child",")?");
7  insert(aux, "child", "; ");
8  visitStmt2(for_s→stmt);
9  insert(aux,"child", "; ");
10 visitAssignStmt2(for_s→inc);
11 insert(aux,"child",")*;¬(");
12 aux2 = aux;
13 visitExpr2(for_s→cond);
14 insert(aux,"child",")?¬for_s→cond");
```

Na linha 1 é adicionado um parêntese. Na linha 2 é adicionado a declaração dentro do for, dado como "for_s→init" Na linha 3 é adicionado mais um parêntese. Na linha 5 é adicionada toda a condição do for. Na linha 6 se fecha o parêntese da condição do for. Na linha 8 é adicionado tudo que estava dentro do for. Na linha 10 é adicionado o campo de incremento.

Logo podemos verificar trivialmente que a definição do for no PDL dado em [1] e a conversão do compilador são equivalentes e portanto temos que a corretude do for do compilador está correta visto que a corretude de conversão para o pdl está correta.Por fim como estes elementos englobam todo o escopo do subconjunto de c utilizado pelo algoritmo temos que a corretude do mesmo está correta visto que a corretude de cada elemento está correta

References