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).

(Verifiquem o que os caracteres especiais dessa gramática significa. (Não achei nada referente a essa extensão do BNF, estou entrando em contato com o eubnara para saber mais sobre isso, se for o caso refatoro)) Program := (DeclList)? (FuncList)?
DeclList := (Declaration)+
FuncList := (Function)+
Declaration := Type IdentList
IdentList := identifier (, identifier)*
Identifier := id ou id [intnum]
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]

O nosso programa não segue a regra abaixo:
1. ++, –
2. De acordo com essa regra := CompoundStmt := (DeclList)? StmtList

2 Algorithm Converter

(Tá uma mistureba de inglês e português. Definam um único idioma. (o codigo tá em ingles, o algoritmo em portugues, se quiser refatoro pra tudo em ingles))
Entrada = Arquivo em C
Saída = Arvore/Arquivo

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)

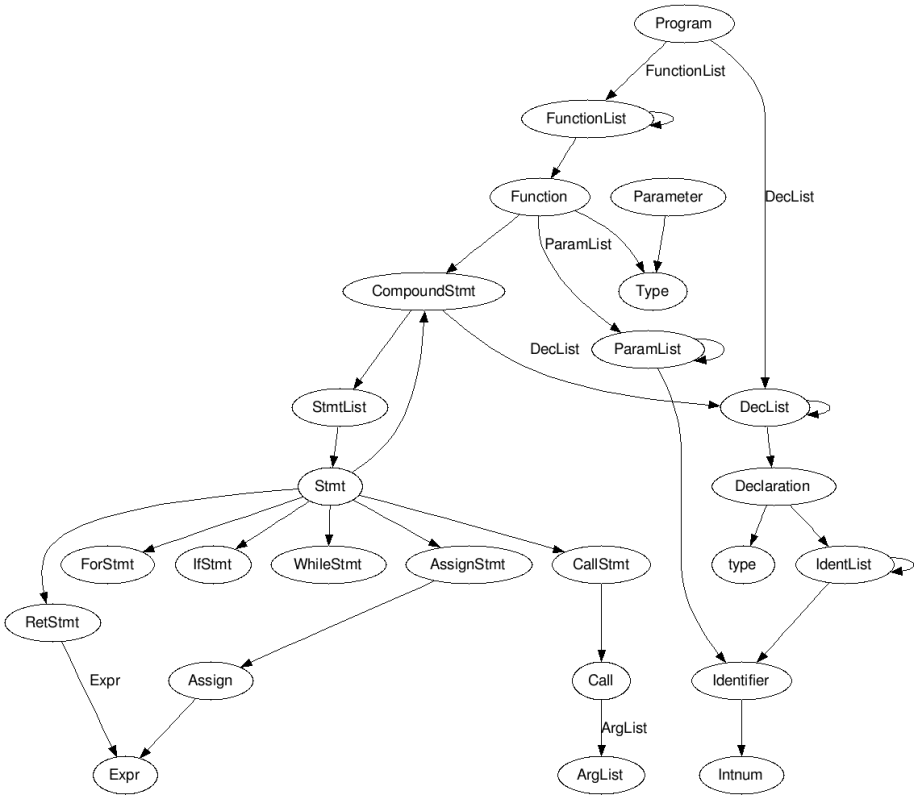
```
1  if cstmtDeclaration != NULL then
2      visitDeclarationcstmtDeclaration;
3  if cstmtStatement != NULL then
4      visitStmt(cstmtStatement);
```

Algorithm 5: visitStmt(Stmt* stmt)

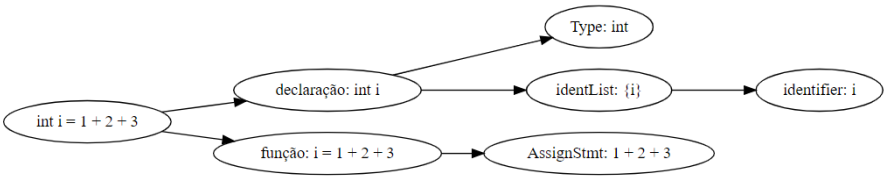
```
1  switch stmtS do
2      case Assign do
3          InsertSemicolon();
4          insert(stmtS_AssignID);
5          insert("=");
6          insert(stmtS_AssignExpression);
7      case Call do
8          insertSemicolon();
9          insert(stmtSCallIdentifier);
10         insert(CallArg);
11     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(";");
```

3 Prova de corretude do algoritmo:

O Propositional Dynamic Logic(PDL) é uma lógica completa, isto é, se existir um caminho correto, iremos chegar até ele. Porém, precisamos partir da hipótese que o programa sempre vai parar. Nesse algoritmo foi utilizado o Flex e o Bison como ferramentas, nele é gerado uma árvore de execução que segue o Formalismo de Backus-Naur(BNF) apresentado no capitulo 1. O algoritmo então opera nessa árvore, que por sua vez é construída com a estratégia "bottom-top". Sendo assim, todo caso é tratado isoladamente, isto é, se todo caso base é comprovadamente correto, a composição deles também é. Temos então os casos das estruturas de repetição, condição e escopo. A condição de escopo é a mais simples, pois como em PDL é ignorada, um escopo é estruturado toda vez que a recursão entra em uma função, logo esse caso é ignorado pela conversão. Assim mostraremos que para um caso genérico Sigma de um subconjunto da linguagem C todas as transformações feitas são válidas, já que é comprovado que todas as conversões de uma linguagem de programação para PDL estão corretas. Agora iremos mostrar que o algoritmo de conversão gera um resultado equivalente a conversão que já está comprovada.



Uma estrutura de Assign por exemplo se comporta do seguinte modo:



3.1 Estruturas de condição

Todas as definições a seguir são explicitadas em [1].

A conversão para PDL de um `if` simples ocorre assim:

Definição 1 *If φ then α else $\beta \rightarrow \varphi?\ ;\ \alpha \cup \neg\alpha?\ ;\ \beta$*

O compilador faz essa conversão da seguinte forma:

Algorithm 6: (void visitIf_s2(struct IF_S* if_s))

Data: Codigo em c
Result: Programa em PDL correspondente ao if

```
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", "∪(¬(");
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", ")");
```

Na linha 4 adiciona dois escopos de parêntese na árvore.
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
Nessa etapa do programa temos um `If` simples, sendo φ = cond e α = Stmt temos então: ($\varphi?\ ;\ \alpha$)
Na linha 13 é checado se existe um else, caso exista é adicionado o "∪(¬(" na linha 12.

Na linha 17 é adicionado a condição novamente.
Na linha 18 termina o escopo do if.
Percebe-se que da mesma forma que a etapa anterior, o else possui o mesmo comportamento, porém com uma negação, sendo o stmt do else igual a β, temos: $((\varphi? \ ; \alpha) \cup (\neg\alpha? \ ; \beta))$

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

3.2 Caso Call

Ao passar pela função o compilador cria diferentes árvores. Cada árvore tem seu nome armazenado como uma tag, sendo assim, quando uma função é chamada é adicionada essa tag na conversão referenciando a árvore em questão.

3.3 Estruturas de repetição

O * no PDL significa que o escopo é repetido zero ou mais vezes, logo não temos controle de quantas iterações serão executadas no programa, mas sabemos que é um numero finito de vezes. Assim, essa abstração as vezes pode significar uma perda parcial de expressividade.

3.3.1 Caso While e doWhile

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

Definição 2 *While* φ *do* $\beta \rightarrow (\varphi? \ ; \beta)^\star \ ; \neg\varphi?$

Definição 3 *do* β *While* $\varphi \rightarrow \beta \ ;((\varphi? \ ; \beta)^\star \ ; \neg\varphi?)$

Temos então o algoritmo do while:

```
   Data: Codigo em c
   Result: Programa em PDL correspondente ao while ou doWhile
 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 |   visitStmt2(while_s→stmt);
 9 |   insert(aux, "child", ")*;¬(");
10 |   aux2 = aux;
11 |   visitExpr2(while_s→cond);
12 |   insert(aux,"child",")? ");
13 else
14 |   insert(aux, "child", "( (");
15 |   aux2 = aux;
16 |   visitExpr2(while_s→cond);
17 |   insert(aux,"child",")?");
18 |   visitStmt2(while_s→stmt);
19 |   insert(aux, "child", ")*;¬(");
20 |   aux2 = aux;
21 |   visitExpr2(while_s→cond);
22 |   insert(aux,"child",")?");
```

Nas linhas 1,12 é feito um teste para saber se é 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 inserido um parêntese.
Na linha 3 é adicionado tudo que possui dentro do while.
Nessa etapa como é adicionado a execução do programa temos: β
Na linha 4 é feita mais uma adição de parêntese.
Na linha 6 é adicionada a condição do dowhile.
Na linha 8 é tratada da forma de um while normal.

A partir desse processo temos que o programa PDL formado sendo φ a condição e β o stmt equivale a: $(\beta\ ;((\varphi?\ ;\ \beta)^\star\ ;\ \neg\varphi?))$

Na linha 14 é adicionado 2 parênteses.
Na linha 15 é adicionada a condição do while.
Na linha 17 se fecha o parênteses e adiciona o "?".
Na linha 18 é adicionado tudo que o while possui internamente.
Na linha 19,21,22 é adicionado a negação da condição e adicionado um ponto de interrogação.
Tendo como condição e stmt o mesmos simbolos temos: $(\varphi?\ ;\ \beta)^\star\ ;\ \neg\varphi?)$

Logo, podemos afirmar que a partir da execução desse algoritmo temos a conversão em PDL do While/doWhile, pois há equivalência no programa gerado pelo algoritmo e a definição citada.

3.3.2 Caso For

O for também pode ser tratado da mesma forma que o while, mas com algumas ressalvas. Logo a conversão de for para PDL ocorre da seguinte forma: no "for"temos 3 campos, o de declaração, condição e incremento. Sendo "decl"a declaração, "cond"a condição e "inc"o incremento, temos como exemplo:

```
for (decl ; cond ; inc){
        stmt
}
```

Assim, a conversão é equivalente a um while e é feita da seguinte forma:

decl ; ((cond? ; stmt ; inc)* ; ¬cond?)

Algorithm 7: visitFor_s2 (struct FOR_S* for_s)

Data: Codigo em c
Result: Programa em PDL correspondente ao for

```
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.
Seja a declaração um programa π, a condição φ, o incremento ϖ, e o stmt ϑ, temos:
$(\pi \ ; \ (\varphi? \ ; \ \varpi \ ; \ \vartheta)^\star)$

Logo, podemos verificar trivialmente que a equivalência a um while com uma declaração e um incremento e a conversão do compilador são equivalentes. Portanto, pode-se concluir que a corretude do for do compilador está correta visto que a corretude de conversão para o pdl também 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.

Referências

[1] M.J. Fischer, R.E. Ladner, Propositional dynamic logic of regular programs, J. Comput. System Sci.18 (2):194-211, 1979