Project 6: Program Normalizer and Control-Flow Graph Visualizer

Overview

The goal of this project is to automatically transform a concurrent program with a fixed number of threads into an equivalent program who's control-flow graph has at most one back-edge (i.e. a program with at most one loop), and visualize all the intermediate steps.

Implementation

This project is split into a number of sub-components, each with its own defined input and output. The implementation can be in any language available on the CDF computers. Each needed data-structure will be designed by yourselves.

Component 0: WHILE program parser (Optional, 10%)

Input: A WHILE program represented as text.

You may design the grammar yourselves as long as it supports all the necessary constructs, as well as the ability to declare variables and separate code into a fixed number of threads (you do not need to support constructs for dynamic thread creation).

Output: A WHILE program represented as an abstract syntax tree with the following structure:

```
PROGRAM ::= STMT
STMT ::= LOOP(STMT)        (while * do s)
       | AMB(STMT, STMT)   (if * then s1 else s2)
       | SEQ(STMT, STMT)   (s1 s2)
       | ASSUME(EXPR)      ([e])
       | ASSIGN(VAR, EXPR) (v := e)
EXPR ::= ...
```

The set of expressions is left unspecified. You may implement any expressions you like as long as they do not produce side-effects. You will almost certainly need to implement TRUE, FALSE, and NOT(EXPR) in order to complete Component 5.

The parenthesized code to the right designates shorthands that will be used in the description of Component 5. Your grammar does not need to accept programs in this format.

Component 1: WHILE program to Control-Flow Graph (10%)

Input: A WHILE program represented as an abstract syntax tree (as in Component 0).

Output: A control-flow graph representing the program.

Component 2: Control-Flow Graph Composition (Optional, 10%)
**Input:** A list of control-flow graphs.

**Output:** The concurrent composition of the list of control-flow graphs. Given a list of directed graphs \( \langle V_1, E_1 \rangle \ldots \langle V_n, E_n \rangle \), the concurrent composition of these graphs is defined as \( G = \langle V, E \rangle \) where

\[
\begin{align*}
V &= V_1 \times \ldots \times V_n \\
E &= \bigcup_{i=1}^n \{ ((v_1, \ldots, v_n), (v'_1, \ldots, v'_n)) \in V \times V \mid (v_i, v'_i) \in E_i \land \forall j. i \neq j \Rightarrow v_j = v'_j \}
\end{align*}
\]

**Note:** This step essentially constructs a sequential program that represents all possible executions of a concurrent program.

**Component 3: Control-Flow Graph to GOTO program (10%)**

**Input:** A control-flow graph.

**Output:** An equivalent program represented as an abstract syntax tree with the following structure:

```plaintext
PROGRAM ::= STMT
STMT ::= LABEL
| GOTO(LABEL)
| AMB(STMT, STMT)
| SEQ(STMT, STMT)
| ASSUME(EXPR)
| ASSIGN(VAR, EXPR)
EXPR ::= ...
```

**Component 4: GOTO program to WHILE program (30%)**

**Input:** A GOTO program (as in Component 3).

**Output:** An equivalent WHILE program (as in Component 1).

**Algorithm:** You must come up with the translation algorithm yourself. You can utilize any references you like, but you must cite them. If the input program is a WHILE program converted to a GOTO program, then the resulting WHILE program should bear as much resemblance to the original WHILE program as possible.

**Component 5: WHILE program normalization (30%)**

**Input:** A WHILE program (as in Component 1).

**Output:** An equivalent WHILE program that has at most one loop construct.

**Algorithm:** The following transformation based on the paper "Kleene Algebra with Tests" and works by recursively transforming each statement into the form \( (\text{pre}) (\text{while} * \text{do} (\text{body})) (\text{post}) \). In each case, the variable \( \text{flag} \) represents a freshly generated boolean variable that is not used anywhere else.

```
(AMB)
if * then ... else ...
≡ if * then (pre1)
   while * do (body1)
   (post1)
else (pre2)
```

while * do (body2)  
(post2)
≡ if * then flag := true  
(pre1)  
else flag := false  
(pre2)  
while * do  
if flag then (body1) else (body2)  
if flag then (post1) else (post2)
(LOOP)  
while * do ...
≡ while * do  
(pre)  
while * do (body)  
(post)
≡ if * then  
(pre)  
while * do  
if * then (post)  
(pre)  
else (body)  
(post)
≡ if * then flag := true  
(pre)  
else flag := false  
while * do  
if flag ∧ * then (post)  
(pre)  
else (body)  
if flag then (post)
(SEQ)  
...
≡ (pre1)  
while * do (body1)  
(post1)  
(pre2)  
while * do (body2)  
(post2)
≡ (pre1)  
flag := true  
while * do  
if flag ∧ * then flag := false  
(post1)  
(pre2)  
if flag then (body1)  
else (body2)  
[¬flag]  
(post2)

Some of the code above uses constructs that are not available in the abstract syntax for WHILE programs. You must figure out how to translate these expressions into the core forms.
As an example, a regular if expression

```
if flag then
    ⟨body1⟩
else
    ⟨body2⟩
```

is equivalent to

```
if * then
    [flag]
    ⟨body1⟩
else
    [-flag]
    ⟨body2⟩
```

**Component 6: Control-Flow Graph to DOT (20%)**

**Input:** A control-flow graph.

**Output:** The control-flow graph in the [GraphViz DOT format](https://graphviz.org/).

**Evaluation**

You will be primarily evaluated based on correctness. Performance is not the focus of this project, but you will suffer penalties if any part of your implementation is impractically slow.

The components labelled as Optional may be completed for bonus marks.

If at any point you discover that two or more components can be more easily written by combining them (e.g. implementing a component that directly converts a control-flow graph to a WHILE program) then feel free to do so. The component will be worth as much as the components it replaces.