

Translating Between PEGs and CFGs

Ross Tate Michael Stepp Zachary Tatlock Sorin Lerner
Department of Computer Science and Engineering
University of California, San Diego
{rtate, mstepp, ztatlock, lerner} @cs.ucsd.edu

1 Introduction

In recent research, we have developed a new approach to structuring optimizers, and to make this approach effective, we have designed a new IR called Program Expression Graphs (PEG). The details of the PEG representation can be found elsewhere. In this technical report, we only present the algorithms for translating from CFGs to a PEGs (Section 2) and from PEGs to CFGs (Section 3).

2 Transforming a CFG into a PEG

We transform a CFG into a PEG in two steps. First, we transform the CFG into an Abstract PEG (A-PEG). Conceptually, an A-PEG is a PEG that operates over program stores rather than individual variables, and whose nodes represent the basic blocks of the CFG. Figure 1(b) shows a sample A-PEG, derived from the CFG in Figure 1(a). The A-PEG captures the structure of the original CFG using θ , $pass$ and $eval$ nodes, but does not capture the flow of individual variables, nor the details of how each basic block operates.

For each basic block n in the CFG, there is a node SE_n in the corresponding A-PEG that represents the execution of the basic block (SE stands for Symbolic Evaluator): given a store at the input of the basic block, SE_n returns the store at the output. For basic blocks that have multiple CFG successors, meaning that the last instruction in the block is a branch, we assume that the store returned by SE contains a specially named boolean variable whose value indicates which way the branch will go. The function $cond$ takes a program store, and selects this specially named variable from it. As a result, for a basic block n that ends in a branch, $cond(SE_n)$ is a boolean stating which way the branch should go.

Once we have an A-PEG, the translation from A-PEG to PEG is simple – all that is left to do is expand the A-PEG to the level of individual variables by replacing each SE_n node with a dataflow representation of the instructions in block n . For example, in Figure 1, if there were two variables being assigned in all the basic blocks, then the PEG would essentially contain two structural copies of the A-PEG, one copy for each variable.

Our algorithm for converting a CFG into an A-PEG starts with a reducible CFG (all CFGs produced from valid Java code are reducible, and furthermore, if a CFG is not reducible, it can be transformed

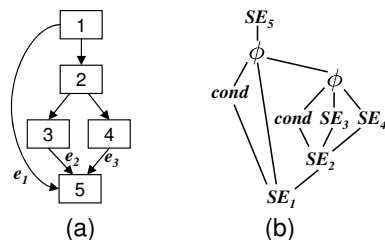


Figure 1: Sample CFG and corresponding A-PEG

to an equivalent reducible one at the cost of some node duplication [1]). Using standard techniques, we identify loops, and for each loop we identify (1) the loop header node, which is the first node that executes when the loop begins (this node is guaranteed to be unique because the graph is reducible), (2) back edges, which are edges that connect a node inside the loop to the loop header node and (3) break edges, which are edges that connect a node inside the loop to a node that is not in the loop.

From the CFG we build what is called a *Forward Flow Graph* (FFG), which is an acyclic version of the CFG. In particular, the FFG contains all the nodes from the CFG, plus a node n' for each loop header node n ; it also contains all the edges from the CFG, except that any back edge connected to a loop header n is instead connected to n' .

We use N to denote the set of nodes in the FFG, and E the set of edges. For any $n \in N$, $in(n)$ and $out(n)$ are the set of incoming and outgoing edges of n . If n is a basic block that ends in a branch statement, we use $out_{\text{true}}(n)$ and $out_{\text{false}}(n)$ for the true and false outgoing edge of n . We use $a \xrightarrow{*} b$ to represent that there is a path in the FFG from the node (or edge) a to the node (or edge) b . We identify a loop by its loop header node l , and for any $n \in N$, we use $loops(n) \subseteq N$ to denote the set of loops that n belongs to (precisely, $l \in loops(n) \Leftrightarrow (l \xrightarrow{*} n \wedge n \xrightarrow{*} l')$). Finally, we use overbars to denote constructors of A-PEG nodes – in particular, for any mathematical operator g , the function \bar{g} constructs an A-PEG node labeled with “ g ”.

Our conversion algorithm from CFG to A-PEG is shown in Figure 2. We describe each function in turn.

ComputeAPEG. Once the FFG is constructed, our conversion algorithm calls the ComputeAPEG function. Throughout the rest of the description, we assume that the FFG and CFG are globally accessible. ComputeAPEG starts by creating, for each node n in the CFG (line 1), a globally accessible A-PEG node SE_n (line 2), and a globally accessible A-PEG node c_n (line 3). The conversion algorithm then sets the input of each SE_n node to the A-PEG expression computed by ComputeInputs(n) (lines 4-5).

ComputeInputs. The ComputeInputs function starts out by calling Decide on the incoming edges of n (lines 7-9). Intuitively, Decide computes an A-PEG expression that, when evaluated, will decide between different edges (we describe Decide and its arguments in more detail shortly). After calling Decide, ComputeInputs checks if n is a loop header node (line 10). If it is *not*, then one can simply return the *result* from Decide (line 13). On the other hand, if n is a loop header node, then its FFG predecessors are nodes from *outside* the loop (since back edges originating from within the loop now go to n'). In this case, the *result* computed on line 9 only accounts for values coming from *outside* of the loop, and so we need lines 11-12 to adjust the result so that it also accounts for values coming from *inside* the loop. In particular, we use ComputeInputs(n') to compute the A-PEG expression for values coming from inside the loop, and then we create the θ_i expression that combines the two (with i being the loop nest depth of n).

Decide. The Decide function is used to create an expression that decides between a set of edges. In particular, suppose we are given a set of FFG edges, and we already know that the program will definitely reach one of these edges starting from the root node – then Decide returns an A-PEG expression that, when evaluated, computes which of these edges will be reached from the beginning of the FFG. The first parameter to Decide is the set of edges; the second parameter is a function mapping edges to A-PEG nodes – this function is used to create an A-PEG node from each edge; and the third parameter is a looping context, which is the set of loops that Decide is currently analyzing.

Decide starts by calling $least_dominator(\mathcal{E})$ to compute d , the least dominator (in the FFG) of the given set of edges, where least means furthest away from the root (line 14). If d is in the current looping context (line 15), then, after optimizing the case where *value* maps all edges to the same A-PEG node (lines 16-17), Decide calls itself to decide between the true and false cases (lines 18-19). Decide then creates the appropriate ϕ node, using the c_d node created in ComputeAPEG (line 20).

For example, suppose ComputeInputs is called on node 5 from Figure 1. Since there are no loops in Figure 1, ComputeInputs simply returns Decide($\{e_1, e_2, e_3\}, value_fn, \emptyset$), and Decide only executes lines 14-20. As a result, this leads to the following steps (where we’ve omitted the last two parameters to

Function ComputeAPEG()

```

1: for each CFG node  $n$  do
2:   let global  $SE_n = \overline{\text{create A-PEG node labeled "SE}_n\text{"}}$ 
3:   let global  $c_n = \overline{\text{cond}(SE_n)}$ 
4:   for each CFG node  $n$  do
5:     set child of  $SE_n$  to ComputeInputs( $n$ )
6:   return resulting A-PEG

```

Function ComputeInputs($n : N$)

```

7: let  $in\_edges = in(n)$ 
8: let  $value\_fn = \lambda e : in\_edges . SE_{src(e)}$ 
9: let  $result = \text{Decide}(in\_edges, value\_fn, loops(n))$ 
10: if  $n$  is a loop header node then
11:   let  $i = |loops(n)|$ 
12:   let  $result = \overline{\theta}_i'(result, \text{ComputeInputs}(n'))$ 
13: return  $result$ 

```

Function Decide($\mathcal{E} : 2^E$, $value : \mathcal{E} \rightarrow N_{\text{A-PEG}}$, $\mathcal{L} : 2^N$)

```

14: Let  $d = \text{least\_dominator}(\mathcal{E})$ 
15: if  $loops(d) \subseteq \mathcal{L}$  then
16:   if  $\exists v . \forall e \in \mathcal{E} . value(e) = v$  then
17:     return  $v$ 
18:   let  $t = \text{Decide}(\{e \in \mathcal{E} \mid out_{\text{true}}(d) \xrightarrow{*} e\}, value, \mathcal{L})$ 
19:   let  $f = \text{Decide}(\{e \in \mathcal{E} \mid out_{\text{false}}(d) \xrightarrow{*} e\}, value, \mathcal{L})$ 
20:   return  $\overline{\phi}(c_d, t, f)$ 
21: else
22:   let  $l$  be the outermost loop in  $loops(d)$  that is not in  $\mathcal{L}$ 
23:   let  $i$  be the nesting depth of  $l$ 
24:   let  $break\_edges = \text{ComputeBreakEdges}(l)$ 
25:   let  $break = \text{BreakCondition}(l, break\_edges, \mathcal{L} \cup \{l\})$ 
26:   let  $val = \overline{\text{Decide}}(\mathcal{E}, value, \mathcal{L} \cup \{l\})$ 
27:   return  $\overline{eval}_i(val, \overline{pass}_i(break))$ 

```

Function BreakCondition($l : N$, $break_edges : 2^E$, $\mathcal{L} : 2^N$)

```

28: let  $all\_edges = break\_edges \cup in(l')$ 
29: let  $value\_fn = \lambda e : all\_edges . (\text{if } e \in break\_edges \text{ then } \overline{\text{true}} \text{ else } \overline{\text{false}})$ 
30: return Simplify(Decide( $all\_edges$ ,  $value\_fn$ ,  $\mathcal{L}$ ))

```

Figure 2: CFG to A-PEG conversion algorithm

Decide because they are always the same):

$$\begin{aligned}
& \text{Decide}(\{e_1, e_2, e_3\}) = \\
& \phi(c_1, \text{Decide}(\{e_1\}), \text{Decide}(\{e_2, e_3\})) = \\
& \phi(c_1, \text{Decide}(\{e_1\}), \phi(c_2, \text{Decide}(\{e_2\}), \text{Decide}(\{e_3\}))) = \\
& \phi(c_1, value_fn(e_1), \phi(c_2, value_fn(e_2), value_fn(e_3))) = \\
& \phi(c_1, SE_1, \phi(c_2, SE_3, SE_4))
\end{aligned}$$

This is exactly the A-PEG expression used for the input to node 5 in Figure 1.

Going back to the code for Decide, if the dominator d is *not* in the same looping context, then the edges that we are deciding between originate from a more deeply nested loop. We therefore need to compute the appropriate “break” condition – the right combination of *eval/pass* that will convert

values from the more deeply nested loop into the current looping context. To do this, `Decide` picks the outermost loop l of the more deeply nested loops that are not in the context (line 22); it computes the set of edges that break out of l using `ComputeBreakEdges`, a straightforward function not shown here (line 24); it computes the break condition for l using `BreakCondition` (line 25); it then computes an expression that decides between the edges \mathcal{E} , but this time adding l to the loop context (line 26); finally `Decide` puts it all together in an *eval/pass* expression (line 27).

BreakCondition. The `BreakCondition` function creates a boolean A-PEG node that evaluates to true when the given loop l breaks. Deciding whether or not a loop breaks amounts to deciding if the loop, when started at its header node, reaches the break edges (*break_edges*) or the back edges (*in(l')*). We can reuse our `Decide` function described earlier for this purpose (lines 29-30). Finally, we use the `Simplify` function, not shown here, to perform basic boolean simplifications on the result of `Decide` (line 30).

3 Transforming a PEG into a CFG

The translation from PEG to CFG assumes that the PEG has certain properties, the most important of which are: (1) there is a partial order \leq on the loops \mathcal{L} such that for every θ'_ℓ node n , each ℓ' in *variance*(n) satisfies $\ell' \leq \ell$; (2) the second child of an *eval* node is a *pass* node – if a PEG is not in this form, it can be transformed to an equivalent PEG by pushing *eval*'s up; (3) by removing the second outgoing edge of all θ' nodes, the PEG becomes acyclic – this property is guaranteed to hold because it is encoded in the constraints the Pseudo-Boolean solver uses to select a PEG.

Our translation uses the notion of a *PEG block*, which is a container that originally just stores a PEG. As the algorithm progresses, each PEG block will eventually get translated to the sub-CFG that represents the PEG it contains. There are two kinds of PEG blocks: fall-through blocks, which eventually reduce to a single-entry single-exit sub-CFG; and branch blocks, which eventually reduce to a single-entry two-exit CFG, with the two exits being the true and false side of a branch that occurs at the end of the block.

To start, there is a single fall-through PEG block containing the PEG for the entire function we are translating. `TranslateBlock`, shown in Figure 3, is called on this block.

Ignoring line 2 for now, `TranslateBlock` first processes *eval* and ϕ nodes using `DoEvals` and `DoPhis` (lines 1 and 3). Intuitively, these procedures move entire PEG sub-graphs into PEG blocks that are then inserted as PEG nodes in lieu of the original sub-graphs. In particular, `DoEvals` replaces *eval/pass* nodes with a new kind of *loop* node, and `DoPhis` replaces ϕ nodes with a new kind of *branch* node. The *loop* and *branch* PEG nodes contain in them newly created PEG blocks (which in turn contain PEGs) to be processed recursively during the call to `Serialize`. The next step is to perform loop fusion (line 4), which essentially merges independent *loop* nodes together that have the same break condition. Finally, the last step is to serialize (line 5). When serialization begins, θ' nodes have already been removed. The PEG is therefore acyclic and so nodes can be processed topologically. Serialization creates a CFG, and each node (in topological order) is converted into CFG instructions. When a *loop* or *branch* node is encountered, a recursive call is made to `TranslateBlock` in order to construct a sub-CFG just for that *loop* or *branch* node, and this sub-CFG is spliced into the currently constructed CFG.

Returning to line 2, if the block b is a branch block that has at least one ϕ node in it, then it should be split. The `Split` function takes a PEG branch block and returns a CFG. It splits on its branch condition, moving the true and false computations into two new PEG blocks. `TranslateBlock` is then called on these two new blocks, and the original branch-block, with whatever PEG nodes have remained there. `Split` then returns a CFG that connects the 3 CFGs returned by the calls to `TranslateBlock` in an if-then-else pattern.

We now describe `DoEvals` and `DoPhis`, shown in Figure 3, in more detail. We denote by *Block* the set of PEG blocks, N_{PEG} the set of PEG nodes, *Var* the set of CFG variables, and *Select* the set of functions from N_{PEG} to booleans (ie : $Select = N_{PEG} \rightarrow \mathbb{B}$). For $n \in N_{PEG}$, *variance*(n) is the set of loops that n varies on. In particular, $variance(n) = \{\ell \mid \neg invariant_\ell(n)\}$, where $invariant_\ell(n)$ is true iff the value of n does not vary on loop ℓ . Invariance is computed using the following four rules, where \bullet denotes “don’t care”:

Function TranslateBlock($b : Block$)

```
1: DoEvals( $b$ )
2: if ShouldSplit( $b$ ) then return Split( $b$ )
3: DoPhis( $b$ )
4: FuseLoops( $b$ )
5: return Serialize( $b$ )
```

Procedure DoEvals($b : Block$)

```
6: while there are eval nodes in  $b.peg$  do
7:   let  $e = eval_\ell(a, pass_\ell(b)) \in b.peg$  be an eval node such that there is no  $eval_{\ell'} \in b.peg$  such that  $\ell' < \ell$ 
8:   let  $init = NewFallThroughBlock()$ 
9:   let  $body = NewBranchBlock()$ 
10:  let  $varies = \lambda n . variance(n) \neq \emptyset$ 
11:  let  $varies\_not\_theta = \lambda n . varies(n) \wedge n$  is not a  $\theta_\ell$  node
12:   $body.SetBranch(b, varies\_not\_theta)$ 
13:  for each  $t = \theta_\ell(c, d) \in Reach(e, varies)$  do
14:     $init.Modify(var(t), c, varies\_not\_theta)$ 
15:     $body.Modify(var(t), \bar{\phi}(b, var(t), d), varies\_not\_theta)$ 
16:   $body.Modify(var(e), \bar{\phi}(b, a, var(e)), varies\_not\_theta)$ 
17:  let  $inputs = init.inputs \cup body.inputs$ 
18:  let  $loop\_node = \overline{loop}_{(init, body, var(b))}(inputs)$ 
19:   $b.peg[e \mapsto \bar{p}_{var(e)}(loop\_node)]$ 
```

Procedure DoPhis($b : Block$)

```
20: let  $\mathcal{G} = GroupPhis(b.peg)$ 
21: let  $nodes =$  new map of type  $\mathcal{G} \rightarrow N_{PEG}$ 
22: let  $blocks =$  new map of type  $\mathcal{G} \rightarrow Block$ 
23: for each  $G \in \mathcal{G}$  do
24:   let  $block = NewBranchBlock()$ 
25:    $block.SetBranch(ChooseBranch(G), \lambda n . n \in G)$ 
26:   for each  $n \in G$  do
27:      $block.Copy(n, \lambda n . n \in G)$ 
28:    $nodes(G) = \overline{branch}_{block}(block.inputs)$ 
29:    $blocks(G) = block$ 
30: let  $all = \bigcup_{G \in \mathcal{G}} G$ 
31: let  $S = b.significant$ 
32: while  $\exists s \in S . \exists n \in Border(s, \lambda n . n \notin all)$  do
33:   let  $G \in \mathcal{G}$  be such that  $n \in G$ 
34:    $blocks(G).Modify(var(n), n, \lambda n . n \in G)$ 
35:    $b.peg[n \mapsto \bar{p}_{var(n)}(nodes(G))]$ 
```

Figure 3: PEG to CFG conversion algorithm

1. all constant and method params nodes are *invariant_ℓ*, for all ℓ
2. if all children of an operator node op are *invariant_ℓ* and $op \neq \theta'_\ell$, then op is *invariant_ℓ*
3. *invariant_ℓ*($pass_\ell(\bullet)$) = **true**
4. *invariant_ℓ*(A) implies *invariant_ℓ*($eval_\ell(\bullet, A)$) = **true**.

We associate with each $n \in N_{PEG}$ a variable denoted $var(n)$. For $b \in Block$, $b.peg$ refers to the PEG in that block; $b.inputs$ is a set for storing PEG nodes that the block b reads from outside of this block; and $b.significant$ is the set of PEG nodes that are outputs of the block, as well as the branch condition if the block is a branch block. Given a PEG peg and two PEG nodes a and b , we use the notation $peg[a \mapsto b]$ to denote destructively updating peg to replace a with b (in particular, the parents of a are redirected to point to b).

DoEvals and DoPhis use the following helper functions:

- **Reach** : $N_{PEG} \times Select \rightarrow 2^{N_{PEG}}$;
Reach(n, f) returns the set of nodes reachable from n by following children links while f returns true.
- **Border** : $N_{PEG} \times Select \rightarrow 2^{N_{PEG}}$;
Border(n, f) returns the set of nodes reachable from n where f first becomes false. This is the border region of Reach(n, f), that is to say, all the nodes “one beyond” Reach(n, f).
- **Copy** : $Block \times N_{PEG} \times Select \rightarrow void$;
 $b.Copy(n, f)$ copies the PEG rooted at n into $b.peg$, replacing each node $n' \in Border(n, f)$ with $var(n')$. Each time a node n' is converted to $var(n')$, n' is added to the set $b.inputs$.
- **Modify** : $Block \times Var \times N_{PEG} \times Select \rightarrow void$;
 $b.Modify(v, n, f)$ calls $b.Copy(n, f)$ and then records that block b on exit needs to set v to the copy of n created by Copy.
- **SetBranch** : $Block \times N_{PEG} \times Select \rightarrow void$;
 $b.SetBranch(n, f)$ calls $b.Copy(n, f)$ and then sets the branch condition for b to the copy of n created by Copy.

DoEvals. The DoEvals procedure starts off by selecting an *eval* node e to convert into a loop (line 7). It then creates a fall-through block called *init* for the instructions before the loop, and a branch block *body* for the body of the loop (lines 8-9). DoEvals then records the computations that must be performed in *init* and *body* (lines 10-16). In copying computations to *init* and *body*, *varies_not_theta* causes invariant nodes and θ'_ℓ nodes to be replaced with their variables (line 11). DoEvals starts by stating that *body* must compute the branch b (line 12). Then, for each $\theta'_\ell(c, d)$ node t in Reach($e, varies$), DoEvals records that *init* should set $var(t)$ to the initial value c (line 14), and that *body* should set $var(t)$ to the recursive computation d only when the break condition b is false (line 15). DoEvals then states that *body* should set $var(e)$ to the a computation, but only when its break condition b is true (line 16). This sets the value of e to be used after the loop. Finally, DoEvals creates the *loop* node (lines 17-18), and replaces e with it (line 19). Because *loop* represents a block that can return many values, we need to select $var(e)$ from its result, which we do using a selector function $\rho_{var(e)}$.

DoPhis. The DoPhis procedure starts by grouping ϕ nodes (line 20). The ϕ nodes in the same group will be branched upon in a nested way. It is always safe to group all ϕ nodes together, but to avoid an exponential blowup, we want to create as many groups as possible. In particular, GroupPhis returns a partition of a subset of the nodes in $b.peg$. This function is the key to performing branch fusion. GroupPhis starts with all nodes in separate sets, and then groups nodes by iteratively applying the following rules: (1) ϕ nodes whose conditions are related by simple boolean relations are grouped together; (2) any nodes reachable by following children links from a ϕ node n , excluding nodes that are always evaluated, are included in n 's group; (3) any node that is reachable from a group G , and also reaches G is merged into G ; (4) if any two groups overlap, they are merged. Once GroupPhis has computed groups, DoPhis creates, for each group of nodes G , a branch block for the group (line 24), and selects some branch condition for it using ChooseBranch (line 25). ChooseBranch(G) picks a $\phi(b, x, y)$ node in G that has no parents in G , and returns its condition b . DoPhis then copies all nodes from G into the block (lines 26-27), and creates a *branch* PEG node for the block (line 28). Finally, DoPhis adjusts all nodes in the current block b to use the newly created *branch* PEG nodes (lines 30-35).

References

- [1] S. Muchnick. *Advanced Compiler Design And Implementation*. Morgan Kaufmann Publishers, 1997.