TRANSLATING BETWEEN PEGS AND CFGS

ROSS TATE, MICHAEL STEPP, ZACHARY TATLOCK, AND SORIN LERNER

Department of Computer Science and Engineering, University of California, San Diego
e-mail address: {rtate,mstepp,ztatlock,lerner}@cs.ucsd.edu

Abstract. In this technical report, we present the algorithms for translating control flow graphs (CFGs) to program expression graphs (PEGs) and translating PEGs to CFGs. In order to explain these algorithms, we first define a simple imperative language, SIMPLE, and show the algorithms for translating between SIMPLE programs and PEGs, followed by the techniques for extending these algorithms to CFGs. For an introduction to PEGs, please refer to the original conference paper [4]. For more detail on PEGs, SIMPLE, and guarantees of the algorithms presented here, please refer to the more recent journal paper [5]. The journal paper already contains the algorithms for SIMPLE programs, but not the algorithms for CFGs.

1. Converting CFGs to PEGs

In this section we describe how programs written in an imperative style can be transformed to work within our PEG-based optimization system. We first define a minimal imperative language (Section 1.1) and then present ML-style functional pseudocode for converting any program written in this language into a PEG (Section 1.2). Next, we present a formal account of the conversion process using type-directed translation rules in the style of [2] (Section 1.3). Finally, we present an algorithm for presenting CFGs to PEGs (Section 1.4).

1.1. **The SIMPLE programming language.** We present our algorithm for converting to the PEG representation using a simplified source language. In particular, we use the SIMPLE programming language, the grammar of which is shown in Figure 1. A SIMPLE program contains a single function `main`, which declares parameters with their respective types, a body which uses these variables, and a return type. There is a special variable `retvar`, the value of which is returned by `main` at the end of execution. SIMPLE programs may have an arbitrary set of primitive operations on an arbitrary set of types; we only require that there is a Boolean type for conditionals (which makes the translation simpler). Statements in SIMPLE programs have four forms: statement sequencing (using semicolon), variable assignment (the variable implicitly inherits the type of the expression), if-then-else

1998 ACM Subject Classification: D.3.4.

Key words and phrases: Compiler Optimization, Equality Reasoning, Intermediate Representation.

An earlier version of this work appeared at the 36th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL 2009).

Supported in part by NSF CAREER grant CCF-0644306.

LOGICAL METHODS
IN COMPUTER SCIENCE

DOI:10.2168/LMCS-???

$$
\begin{array}{l}
p ::= \texttt{main}(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau\ \{s\} \\
s ::= s_1; s_2 \mid x := e \mid \texttt{if}\ (e)\ \{s_1\}\ \texttt{else}\ \{s_2\} \mid \texttt{while}\ (e)\ \{s\} \\
e ::= x \mid op(e_1, \ldots, e_n)
\end{array}
$$

Figure 1: Grammar for SIMPLE programs

$\boxed{\vdash p \quad \text{(programs)}}$

$$\text{Type-Prog}\frac{x_1 : \tau_1, \ldots, x_n : \tau_n \vdash s : \Gamma \qquad \Gamma(\texttt{retvar}) = \tau}{\vdash \texttt{main}(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau\ \{s\}}$$

$\boxed{\Gamma \vdash s : \Gamma' \quad \text{(statements)}}$

$$\text{Type-Seq}\frac{\Gamma \vdash s_1 : \Gamma' \qquad \Gamma' \vdash s_2 : \Gamma''}{\Gamma \vdash s_1; s_2 : \Gamma''} \qquad \text{Type-Asgn}\frac{\Gamma \vdash e : \tau}{\Gamma \vdash x := e : (\Gamma, x : \tau)}$$

$$\text{Type-If}\frac{\Gamma \vdash e : \texttt{bool} \qquad \Gamma \vdash s_1 : \Gamma' \qquad \Gamma \vdash s_2 : \Gamma'}{\Gamma \vdash \texttt{if}\ (e)\ \{s_1\}\ \texttt{else}\ \{s_2\} : \Gamma'} \qquad \text{Type-While}\frac{\Gamma \vdash e : \texttt{bool} \qquad \Gamma \vdash s : \Gamma}{\Gamma \vdash \texttt{while}\ (e)\ \{s\} : \Gamma}$$

$$\text{Type-Sub}\frac{\Gamma \vdash s : \Gamma' \qquad \Gamma'' \subseteq \Gamma'}{\Gamma \vdash s : \Gamma''}$$

$\boxed{\Gamma \vdash e : \tau \quad \text{(expressions)}}$

$$\text{Type-Var}\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \text{Type-Op}\frac{op : (\tau_1, \ldots, \tau_n) \to \tau \qquad \Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash op(e_1, \ldots, e_n) : \tau}$$

Figure 2: Type-checking rules for SIMPLE programs

branches and while loops. Expressions in SIMPLE programs are either variables or primitive operations (such as addition). Constants in SIMPLE are nullary primitive operations.

The type-checking rules for SIMPLE programs are shown in Figure 2. There are three kinds of judgments: (1) judgment $\vdash p$ (where p is a program) states that p is well-typed; (2) judgment $\Gamma \vdash s : \Gamma'$ (where s is a statement) states that starting with context Γ, after s the context will be Γ'; (3) judgment $\Gamma \vdash e : \tau$ (where e is an expression) states that in type context Γ, expression e has type τ.

For program judgments, there is only one rule, Type-Prog, which ensures that the statement inside of `main` is well-typed; $\Gamma(\texttt{retvar}) = \tau$ ensures that the return value of `main` has type τ. For statement judgments, Type-Seq simply sequences the typing context through two statements. Type-Asgn replaces the binding for x with the type of the expression assigned into x (if Γ contains a binding for x, then $(\Gamma, x : \tau)$ is Γ with the binding for x replaced with τ; if Γ does not contain a binding for x, then $(\Gamma, x : \tau)$ is Γ extended with a binding for x). The rule Type-If requires the context after each branch to be the same. The rule Type-While requires the context before and after the body to be the same, specifying

```
1: function TranslateProg(p : Prog) : N =
2:   let m = InitMap(p.params, λx. param(x))
3:   in TS(p.body, m, 0)(retvar)

4: function TS(s : Stmt, Ψ : map[V, N], ℓ : ℕ) : map[V, N] =
5:   match s with
6:     "s1; s2" ⇒ TS(s2, TS(s1, Ψ, ℓ), ℓ)
7:     "x := e" ⇒ Ψ[x ↦ TE(e, Ψ)]
8:     "if (e) {s1} else {s2}" ⇒ PHI(TE(e, Ψ), TS(s1, Ψ, ℓ), TS(s2, Ψ, ℓ))
9:     "while (e) {s}" ⇒
10:      let vars = Keys(Ψ)
11:      let Ψt = InitMap(vars, λv. TemporaryNode(v))
12:      let Ψ' = TS(s, Ψt, ℓ + 1)
13:      let Ψθ = THETA_{ℓ+1}(Ψ, Ψ')
14:      let Ψ'θ = InitMap(vars, λv. FixpointTemps(Ψθ, Ψθ(v)))
15:      in EVAL_{ℓ+1}(Ψ'θ, pass_{ℓ+1}(¬(TE(e, Ψ'θ))))

16: function TE(e : Expr, Ψ : map[V, N]) : N =
17:   match e with
18:     "x" ⇒ Ψ(x)
19:     "op(e1, ..., ek)" ⇒ op(TE(e1, Ψ), ..., TE(ek, Ψ))

20: function PHI(n : N, Ψ1 : map[V, N], Ψ2 : map[V, N]) : map[V, N] =
21:   Combine(Ψ1, Ψ2, λ t f . φ(n, t, f))

22: function THETA_{ℓ:ℕ}(Ψ1 : map[V, N], Ψ2 : map[V, N]) : map[V, N] =
23:   Combine(Ψ1, Ψ2, λ b n . θ_ℓ(b, n))

24: function EVAL_{ℓ:ℕ}(Ψ : map[V, N], n : N) : map[V, N] =
25:   InitMap(Keys(Ψ), λv . eval_ℓ(Ψ(v), n)

26: function Combine(m1 : map[a, b], m2 : map[a, c], f : b * c → d) : map[a, d] =
27:   InitMap(Keys(m1) ∩ Keys(m2), λk.f(m1[k], m2[k]))
```

Figure 3: ML-style pseudo-code for converting SIMPLE programs to PEGs

the loop-induction variables. The rule Type-Sub allows the definition of variables to be "forgotten", enabling the use of temporary variables in branches and loops.

1.2. **Translating SIMPLE Programs to PEGs.** Here we use ML-style functional pseudo-code to describe the translation from SIMPLE programs to PEGs. Figure 3 shows the entirety of the algorithm, which uses a variety of simple types and data structures, which we explain first.

In the pseudo-code (and in all of Sections 1 and 2), we use the notation $\overline{a}(n_1, \ldots, n_k)$ to represent a PEG node with label a and children n_1 through n_k. Whereas previously the distinction between creating PEG nodes and applying functions was clear from context, in

a computational setting like pseudo-code we want to avoid confusion between for example applying negation in the pseudo-code $\neg(\ldots)$, vs. creating a PEG node labeled with negation $\overline{\neg}(\ldots)$. For parameter nodes, there can't be any confusion because when we write $param(x)$, $param$ is actually not a function – instead $param(x)$ as a whole is label. Still, to be consistent in the notation, we use $\overline{param}(x)$ for constructing parameter nodes.

We introduce the concept of a *node context* Ψ, which is a set of bindings of the form $x : n$, where x is a SIMPLE variable, and n is a PEG node. A node context states, for each variable x, the PEG node n that represents the current value of x. We use $\Psi(x) = n$ as shorthand for $(x : n) \in \Psi$. Aside from using node contexts here, we will also use them later in our type-directed translation rules (Section 1.3). For our pseudo-code, we implement node contexts as an immutable map data structure that has the following operations defined on it

- *Map initialization.* The InitMap function is used to create maps. Given a set K of keys and a function f from K to D, InitMap creates a map of type $map[K, D]$ containing, for every element $k \in K$, an entry mapping k to $f(k)$.
- *Keys of a map.* Given a map m, Keys(m) returns the set of keys of m.
- *Map read.* Given a map m and a key $k \in$ Keys(m), $m(k)$ returns the value associated with key k in m.
- *Map update.* Given a map m, $m[k \mapsto d]$ returns a new map in which key k has been updated to map to d.

The pseudo-code also uses the types *Prog*, *Stmt*, *Expr* and V to represent SIMPLE programs, statements, expressions and variables. Given a program p, *p.params* is a list of its parameter variables, and *p.body* is its body statement. We use syntax-based pattern matching to extract information from *Stmt* and *Expr* types (as shown on lines 6,7 for statements, and 18, 19 for expressions).

Expressions. We explain the pieces of this algorithm one-by-one, starting with the TE function on line 16. This function takes a SIMPLE expression e and a node context Ψ and returns the PEG node corresponding to e. There are two cases, based on what type of expression e is. Line 18 states that if e is a reference to variable x, then we simply ask Ψ for its current binding for x. Line 19 states that if e is the evaluation of operator op on arguments $e_1, \ldots, e_k$, then we recursively call TE on each e_i to get n_i, and then create a new PEG node labeled op that has child nodes $n_1, \ldots, n_k$.

Statements. Next we explore the TS function on line 4, which takes a SIMPLE statement s, a node context Ψ, and a loop depth ℓ and returns a new node context that represents the changes s made to Ψ. There are four cases, based on the four statement types.

Line 6 states that a sequence $s_1; s_2$ is simply the result of translating s_2 using the node context that results from translating s_1. Line 7 states that for an assignment $x := e$, we simply update the current binding for x with the PEG node that corresponds to e, which is computed with a call to TE.

Line 8 handles if-then-else statements by introducing ϕ nodes. We recursively produce updated node contexts Ψ_1 and Ψ_2 for statements s_1 and s_2 respectively, and compute the PEG node that represents the guard condition, call it n_c. We then create PEG ϕ nodes by calling the PHI function defined on line 20. This function takes the guard node n_c and the two node contexts Ψ_1 and Ψ_2 and creates a new ϕ node in the PEG for each variable that

is defined in both node contexts. The true child for each ϕ node is taken from Ψ_1 and the false child is taken from Ψ_2, while all of them share the same guard node n_c. Note that this is slightly inefficient in that it will create ϕ nodes for all variables defined before the if-then-else statement, whether they are modified by it or not. These can be easily removed, however, by applying the rewrite $\phi(C, A, A) = A$.

Finally we come to the most complicated case on line 9, which handles while loops. In line 10 we extract the set of all variables defined up to this point, in the set *vars*. We allocate a temporary PEG node for each item in *vars* on line 11, and bind them together in the node context Ψ_t. We use TemporaryNode(v) to refer to a temporary PEG node named v, which is a new kind of node that we use only for the conversion process. We then recursively translate the body of the while loop using the context full of temporary nodes on line 12. In the resulting context Ψ', the temporary nodes act as placeholders for loop-varying values. Note that here is the first real use of the loop depth parameter ℓ, which is incremented by 1 since the body of this loop will be at a higher loop depth than the code before the loop. For every variable in *vars*, we create $\theta_{\ell+1}$ nodes using the THETA function defined on line 22. This function takes node contexts Ψ and Ψ', which have bindings for the values of each variable before and during the loop, respectively. The binding for each variable in Ψ becomes the first child of the θ (the base case) and the binding in Ψ' becomes the second child (the inductive case). Unfortunately, the θ expressions we just created are not yet accurate, because the second child of each θ node is defined in terms of temporary nodes. The correct expression should replace each temporary node with the new θ node that corresponds to that temporary node's variable, to "close the loop" of each θ node. That is the purpose of the FixpointTemps function called on line 14. For each variable $v \in$ *vars*, FixpointTemps will rewrite $\Psi_\theta(v)$ by replacing any edges to TemporaryNode(x) with edges to $\Psi_\theta(x)$, yielding new node context Ψ'_θ. Now that we have created the correct θ nodes, we merely need to create the *eval* and *pass* nodes to go with them. Line 15 does this, first by creating the $pass_{\ell+1}$ node which takes the break condition expression as its child. The break condition is computed with a call to TE on e, using node context Ψ'_θ since it may reference some of the newly-created θ nodes. The last step is to create *eval* nodes to represent the values of each variable after the loop has terminated. This is done by the EVAL function defined on line 24. This function takes the node context Ψ'_θ and the $pass_{\ell+1}$ node and creates a new $eval_{\ell+1}$ node for each variable in *vars*. This final node context that maps each variable to an *eval* is the return value of TS. Note that, as in the case of if-then-else statements, we introduce an inefficiency here by replacing all variables with $eval's$, not just the ones that are modified in the loop. For any variable v that was bound to node n in Ψ and not modified by the loop, its binding in the final node context would be $eval_{\ell+1}(T, pass_{\ell+1}(C))$, where C is the guard condition node and $T = \theta_{\ell+1}(n, T)$ (i.e. the θ node has a direct self-loop). We can easily remove the spurious nodes by applying a rewrite to replace the *eval* node with n.

Programs. The TranslateProg function on line 1 is the top-level call to convert an entire SIMPLE program to a PEG. It takes a SIMPLE program p and returns the root node of the translated PEG. It begins on line 2 by creating the initial node context which contains bindings for each parameter variable. The nodes that correspond to the parameters are opaque parameter nodes, that simply name the parameter variable they represent. Using this node context, we translate the body of the program starting at loop depth 0 on line 3. This will yield a node context that has PEG expressions for the final values of all the

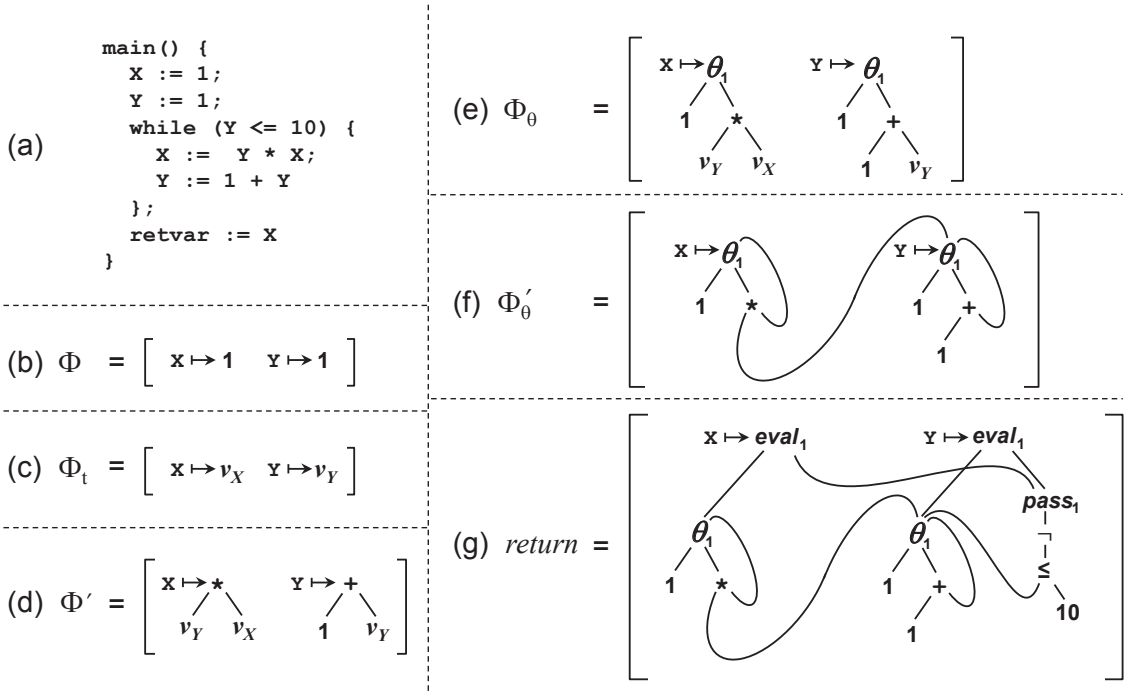

Figure 4: Steps of the translation process. (a) shows a SIMPLE program computing the factorial of 10; (b) through (f) show the contents of the variables in TS when processing the `while` loop; and (g) shows the return value of TS when processing the `while` loop.

variables in the program. Hence, the root of our translated PEG will be the node that is bound to the special return variable `retvar` in this final node context.

Example. We illustrate how the translation process works on the SIMPLE program from Figure 4(a), which computes the factorial of 10. After processing the first two statements, both `X` and `Y` are bound to the PEG node 1. Then TS is called on the `while` loop, at which point Ψ maps both `X` and `Y` to 1. Figures 4(b) through 4(g) show the details of processing the `while` loop. In particular, (b) through (f) show the contents of the variables in TS, and (g) shows the return value of TS. Note that in (g) the node labeled i corresponds to the *pass* node created on line 15 in TS. After the loop is processed, the assignment to `retvar` simply binds `retvar` to whatever `X` is bound to in Figure 4(g).

1.3. **Type-directed translation.** In this section we formalize the translation process described by the pseudo-code implementation with a type-directed translation from SIMPLE programs to PEGs, in the style of [2]. The type-directed translation in Figure 5 is more complicated than the implementation in Figure 3, but it makes it easier to prove the correctness of the translation. For example, the implementation uses maps from variables to PEG nodes, and at various points queries these maps (for example, line 18 in Figure 3 queries the Ψ map for variable x). The fact that these map operations never fail relies on implicit properties which are tedious to establish in Figure 3, as they rely on the fact

$\boxed{\vdash p \quad \triangleright \quad n \quad \text{(programs)}}$

$$\text{Trans-Prog}\ \frac{\{x_1 : \tau_1, \ldots, x_k : \tau_k\} \vdash s : \Gamma \quad \triangleright \quad \{x_1 : \overline{param}(x_1), \ldots, x_k : \overline{param}(x_k)\} \rightsquigarrow_0 \Psi \qquad n = \Psi(\texttt{retvar}) \text{ (well defined because } \Gamma(\texttt{retvar}) = \tau)}{\vdash \texttt{main}(x_1 : \tau_1, \ldots, x_k : \tau_k) : \tau\ \{s\} \quad \triangleright \quad n}$$

$\boxed{\Gamma \vdash s : \Gamma' \quad \triangleright \quad \Psi \rightsquigarrow_\ell \Psi' \quad \text{(statements)}}$

$$\text{Trans-Seq}\ \frac{\Gamma \vdash s_1 : \Gamma' \quad \triangleright \quad \Psi \rightsquigarrow_\ell \Psi' \qquad \Gamma' \vdash s_2 : \Gamma'' \quad \triangleright \quad \Psi' \rightsquigarrow_\ell \Psi''}{\Gamma \vdash s_1; s_2 : \Gamma'' \quad \triangleright \quad \Psi \rightsquigarrow_\ell \Psi''}$$

$$\text{Trans-Asgn}\ \frac{\Gamma \vdash e : \tau \quad \triangleright \quad \Psi \rightsquigarrow n}{\Gamma \vdash x := e : (\Gamma, x : \tau) \quad \triangleright \quad \Psi \rightsquigarrow_\ell (\Psi, x : n)}$$

$$\text{Trans-If}\ \frac{\begin{array}{c}\Gamma \vdash e : \texttt{bool} \quad \triangleright \quad \Psi \rightsquigarrow n \\ \Gamma \vdash s_1 : \Gamma' \quad \triangleright \quad \Psi \rightsquigarrow_\ell \Psi_1 \qquad \Gamma \vdash s_2 : \Gamma' \quad \triangleright \quad \Psi \rightsquigarrow_\ell \Psi_2 \\ \{x : n_{(x,1)}\}_{x \in \Gamma'} = \Psi_1 \qquad \{x : n_{(x,2)}\}_{x \in \Gamma'} = \Psi_2 \\ \Psi' = \{x : \overline{\phi}(n, n_{(x,1)}, n_{(x,2)})\}_{x \in \Gamma'}\end{array}}{\Gamma \vdash \texttt{if}\ (e)\ \{s_1\}\ \texttt{else}\ \{s_2\} : \Gamma' \quad \triangleright \quad \Psi \rightsquigarrow_\ell \Psi'}$$

$$\text{Trans-While}\ \frac{\begin{array}{c}\Gamma \vdash e : \texttt{bool} \quad \triangleright \quad \Psi \rightsquigarrow n \\ \Psi = \{x : v_x\}_{x \in \Gamma} \text{ each } v_x \text{ fresh} \qquad \ell' = \ell + 1 \qquad \Gamma \vdash s : \Gamma \quad \triangleright \quad \Psi \rightsquigarrow_{\ell'} \Psi' \\ \{x : n_x\}_{x \in \Gamma} = \Psi_0 \qquad \{x : n'_x\}_{x \in \Gamma} = \Psi' \\ \Psi_\infty = \{x : \overline{eval_{\ell'}}(v_x, \overline{pass_{\ell'}}(\overline{\neg}(n)))\}_{x \in \Gamma} \text{ with each } v_x \text{ unified with } \overline{\theta_{\ell'}}(n_x, n'_x)\end{array}}{\Gamma \vdash \texttt{while}\ (e)\ \{s\} : \Gamma \quad \triangleright \quad \Psi_0 \rightsquigarrow_\ell \Psi_\infty}$$

$$\text{Trans-Sub}\ \frac{\begin{array}{c}\Gamma \vdash s : \Gamma' \quad \triangleright \quad \Psi \rightsquigarrow_\ell \Psi' \\ \{x : n_x\}_{x \in \Gamma'} = \Psi' \qquad \Psi'' = \{x : n_x\}_{x \in \Gamma''} \text{ (well defined because } \Gamma'' \subseteq \Gamma')\end{array}}{\Gamma \vdash s : \Gamma'' \quad \triangleright \quad \Psi \rightsquigarrow_\ell \Psi''}$$

$\boxed{\Gamma \vdash e : \tau \quad \triangleright \quad \Psi \rightsquigarrow n \quad \text{(expressions)}}$

$$\text{Trans-Var}\ \frac{n = \Psi(x) \text{ (well defined because } \Gamma(x) = \tau)}{\Gamma \vdash x : \tau \quad \triangleright \quad \Psi \rightsquigarrow n}$$

$$\text{Trans-Op}\ \frac{\Gamma \vdash e_1 : \tau_1 \quad \triangleright \quad \Psi \rightsquigarrow n_1 \qquad \ldots \qquad \Gamma \vdash e_k : \tau_k \quad \triangleright \quad \Psi \rightsquigarrow n_k}{\Gamma \vdash op(e_1, \ldots, e_k) : \tau \quad \triangleright \quad \Psi \rightsquigarrow \overline{op}(n_1, \ldots, n_k)}$$

Figure 5: Type-directed translation from SIMPLE programs to PEGs

that the program being translated is well-typed. In the type-directed translation, these properties are almost immediate since the translation operates on an actual proof that the program is well-typed.

In fact, the rules in Figure 5 are really representations of each case in a constructive total deterministic function defined inductively on the proof of well-typedness. Thus when we use the judgment $\Gamma \vdash e : \tau \quad \triangleright \quad \Psi \rightsquigarrow n$ as an assumption, we are simply binding n to the result of this constructive function applied to the proof of $\Gamma \vdash e : \tau$ and the PEG context Ψ. Likewise, we use the judgment $\Gamma \vdash s : \Gamma' \quad \triangleright \quad \Psi \rightsquigarrow_\ell \Psi'$ to bind Ψ' to the result of the constructive function applied to the proof of $\Gamma \vdash s : \Gamma'$ and the PEG context Ψ. Here we explain how this type-directed translation works.

Expressions. The translation process for an expression e takes two inputs: (1) a derivation showing the type correctness of e, and (2) a node context Ψ. The translation process produces one output, which is the node n that e translates to. We formalize this with a judgment $\Gamma \vdash e : \tau \quad \triangleright \quad \Psi \rightsquigarrow n$, which states that from a derivation of $\Gamma \vdash e : \tau$, and a node context Ψ (stating what node to use for each variable), the translation produces node n for expression e. For example, consider the Trans-Op rule, which is used for a primitive operation expression. The output of the translation process is a new PEG node with label op, where the argument nodes $n_1 \ldots n_k$ are determined by translating the argument expressions $e_1 \ldots e_k$.

The Trans-Var rule returns the PEG node associated with the variable x in Ψ. The definition $n = \Psi(x)$ is well defined because we maintain the invariant that $\Gamma \vdash e : \tau \quad \triangleright \quad \Psi \rightsquigarrow n$ is only used with contexts Γ and Ψ that are defined on precisely the same set of variables. Thus, because the Type-Var rule requires $\Gamma(x) = \tau$, Γ must be defined on x and so we know Ψ is also defined on x.

Note that a concrete implementation of the translation, like the one in Figure 3, would explore a derivation of $\Gamma \vdash e : \tau \quad \triangleright \quad \Psi \rightsquigarrow n$ bottom-up: the translation starts at the bottom of the derivation tree and makes recursive calls to itself, each recursive call corresponding to a step up in the derivation tree. Also note that there is a close relation between the rules in Figure 5 and those in Figure 2. In particular, the formulas on the left of the $\triangleright$ correspond directly to the typing rules from Figure 2.

Statements. The translation for a statement s takes as input a derivation of the type-correctness of s, a node context capturing the translation that has been performed up to s, and returns the node context to be used in the translation of statements following s. We formalize this with a judgment $\Gamma \vdash s : \Gamma' \quad \triangleright \quad \Psi \rightsquigarrow_\ell \Psi'$, which states that from a derivation of $\Gamma \vdash s : \Gamma'$, and a node context Ψ (stating what node to use for each variable in s), the translation produces an updated node context Ψ' after statement s (ignore ℓ for now). For example, the rule Trans-Asgn updates the node context to map variable x to the node n resulting from translating e (which relies on the fact that e is well typed in type context Γ).

Again, we maintain the invariant that in all the derivations we explore, the judgment $\Gamma \vdash s : \Gamma' \quad \triangleright \quad \Psi \rightsquigarrow \Psi'$ is only used with contexts Γ and Ψ that are defined on precisely the same set of variables, and furthermore the resulting contexts Γ' and Ψ' will always be defined on the same set of variables (although potentially different from Γ and Ψ). It is fairly obvious that the rules preserve this invariant, although Trans-Sub relies on the fact that Γ'' must be a subcontext of Γ'. The Trans-Seq and Trans-Asgn rules are self explanatatory, so below we discuss the more complicated rules for control flow.

The rule Trans-If describes how to translate if-then-else statements in SIMPLE programs to ϕ nodes in PEGs. First, it translates the Boolean guard expression e to a node n which will later be used as the condition argument for each ϕ node. Then it translates

the statement s_1 for the "true" branch, producing a node context Ψ_1 assigning each live variable after s_1 to a PEG node representing its value after s_1. Similarly, it translates s_2 for the "false" branch, producing a node context Ψ_2 representing the "false" values of each variable. Due to the invariant we maintain, both Ψ_1 and Ψ_2 will be defined on the same set of variables as Γ'. For each x defined in Γ', we use the name $n_{(x,1)}$ to represent the "true" value of x after the branch (taken from Ψ_1) and $n_{(x,2)}$ to represent the "false" value (taken from Ψ_2). Finally, the rule constructs a node context Ψ' which assigns each variable x defined in Γ' to the node $\overline{\phi}(n, n_{(x,1)}, n_{(x,2)})$, indicating that after the if-then-else statement the variable x has "value" $n_{(x,1)}$ if n evaluates to true and $n_{(x,2)}$ otherwise. Furthermore, this process maintains the invariant that the type context Γ' and node context Ψ' are defined on exactly the same set of variables.

The last rule, Trans-While, describes how to translate while loops in SIMPLE programs to combinations of θ, *eval*, and *pass* nodes in PEGs. The rule starts by creating a node context Ψ which assigns to each variable x defined in Γ a fresh temporary variable node v_x. The clause $\ell' = \ell + 1$ is used to indicate that the body of the loop is being translated at a higher loop depth. In general, the ℓ subscript in the notation $\Psi \rightsquigarrow_\ell \Psi'$ indicates the loop depth of the translation. Thus, the judgment $\Gamma \vdash s : \Gamma' \ \triangleright \ \Psi \rightsquigarrow_{\ell'} \Psi'$ translates the body of the loop s at a higher loop depth to produce the node context Ψ'. The nodes in Ψ' are in terms of the temporary nodes v_x in Γ and essentially represent how variables change in each iteration of the loop. Each variable x defined in Γ has a corresponding node n_x in the node context Ψ_0 from before the loop, again due to the invariants we maintain that Γ and Ψ are always defined on the same set of variables. This invariant also guarantees that each variable x defined in Γ also has a corresponding node n'_x in the node context Ψ'. Thus, for each such variable, n_x provides the base value and n'_x provides the iterative value, which can now combined using a θ node. To this end, we unify the temporary variable node v_x with the node $\overline{\theta_{\ell'}}(n_x, n'_x)$ to produce a recursive expression which represents the value of x at each iteration of the loop. Lastly, the rule constructs the final node context Ψ_∞ by assigning each variable x defined in Γ to the node $\overline{\mathit{eval}_{\ell'}}(v_x, \overline{\mathit{pass}_{\ell'}}(\overline{\neg}(n)))$ (where v_x has been unified to produce the recursive expression for x). The node $\overline{\neg}(n)$ represents the break condition of the loop; thus $\overline{\mathit{pass}_{\ell'}}(\overline{\neg}(n))$ represents the number of times the loop iterates. Note that the same *pass* node is used for each variable, whereas each variable gets its own θ node. In this manner, the rule Trans-While translates while loops to PEGs, and furthermore preserves the invariant that the type context Γ and node context Ψ_∞ are defined on exactly the same set of variables.

Programs. Finally, the rule Trans-Prog shows how to use the above translation technique in order to translate an entire SIMPLE program. It creates a node context with a PEG parameter node for each parameter to `main`. It then translates the body of the `main` at loop depth 0 to produce a node context Ψ. Since the return `retvar` is guaranteed to be in the final context Γ, the invariant that Γ and Ψ are always defined on the same variables ensure that there is a node n corresponding to `retvar` in the final node context Ψ. This PEG node n represents the entire SIMPLE program.

Translation vs. pseudo-code. The pseudo-code in Figure 3 follows the rules from Figure 5 very closely. Indeed, the code can be seen as using the rules from the type-directed translation to find a derivation of $\vdash p \ \triangleright \ n$. The search starts at the end of the derivation

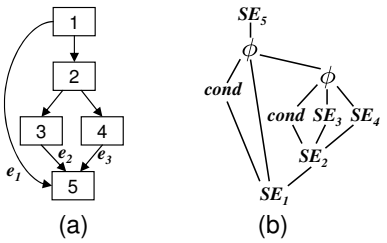

Figure 6: Sample CFG and corresponding A-PEG

tree, and moves up from there. The entry-point of the pseudo-code is TranslateProg, which corresponds to rule Trans-Prog, the last rule to be used in a derivation of $\vdash p \quad \triangleright \quad n$. TranslateProg calls TS, which corresponds to finding a derivation for $\Gamma \vdash s : \Gamma' \quad \triangleright \quad \Psi \rightsquigarrow_\ell \Psi'$. Finally, TS calls TE, which corresponds to finding a derivation for $\Gamma \vdash e : \tau \quad \triangleright \quad \Psi \rightsquigarrow n$. Each pattern-matching case in the pseudo-code corresponds to a rule from the type-directed translation.

The one difference between the pseudo-code and the type-directed translation is that in the judgments of the type-directed translation, one of the inputs to the translation is a derivation of the type correctness of the expression/statement/program being translated, whereas the pseudo-code does not manipulate any derivations. This can be explained by a simple erasure optimization in the pseudo-code: because of the structure of the type-checking rules for SIMPLE (in particular there is only one rule per statement kind), the implementation does not need to inspect the entire derivation – it only needs to look at the final expression/statement/program in the type derivation (which is precisely the expression/statement/program being translated). It is still useful to have the derivation expressed formally in the type-directed translation, as it makes the proof of correctness more direct. Furthermore, there are small changes that can be made to the SIMPLE language that prevent the erasure optimization from being performed. For example, if we add subtyping and implicit coercions, and we want the PEG translation process to make coercions explicit, then the translation process would need to look at the type derivation to see where the subtyping rules are applied.

Because the type-directed translation in Figure 5 is essentially structural induction on the proof that the SIMPLE program is well typed, we can guarantee that its implementation in Figure 3 terminates. Additionally, because of the invariants we maintain in the type-directed translation, we can guarantee that the implementation always successfully produces a translation. We discuss the correctness guarantees provided by the translation below.

1.4. **Converting a CFG to a PEG.** We convert a CFG into a PEG in two steps. First, we convert the CFG into an Abstract PEG (A-PEG). Conceptually, an A-PEG is a PEG that operates over programs stores rather than individual variables, and whose nodes represent the basic blocks of the CFG. Figure 6(b) shows a sample A-PEG, derived from the CFG in Figure 6(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 6, 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 to an equivalent reducible one at the cost of some node duplication [3]). 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 $out_{\mathbf{true}}(n)$ and $out_{\mathbf{false}}(n)$ 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 $\overline{g}$ constructs an A-PEG node labeled with "g".

Our conversion algorithm from CFG to A-PEG is shown in Figure 7. 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

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

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

16: **function** Decide($\mathcal{E} : 2^E,\ value : \mathcal{E} \to N_{\text{A-PEG}},\ \mathcal{L} : 2^N$)
17: Let $d = least\_dominator(\mathcal{E})$
18: **if** $loops(d) \subseteq \mathcal{L}$ **then**
19: **if** $\exists v\ .\ \forall e \in \mathcal{E}\ .\ value(e) = v$ **then**
20: **return** v
21: **let** $t = \text{Decide}(\{e \in \mathcal{E} \mid out_{\mathbf{true}}(d) \xrightarrow{*} e\}, value, \mathcal{L})$
22: **let** $f = \text{Decide}(\{e \in \mathcal{E} \mid out_{\mathbf{false}}(d) \xrightarrow{*} e\}, value, \mathcal{L})$
23: **return** $\overline{\phi}(c_d, t, f)$
24: **else**
25: **let** l be the outermost loop in $loops(d)$ that is not in $\mathcal{L}$
26: **let** i be the nesting depth of l
27: **let** $break\_edges = \text{ComputeBreakEdges}(l)$
28: **let** $break = \text{BreakCondition}(l, break\_edges, \mathcal{L} \cup \{l\})$
29: **let** $val = \text{Decide}(\mathcal{E}, value, \mathcal{L} \cup \{l\})$
30: **return** $\overline{eval_i}(val, \overline{pass_i}(break))$

31: **function** BreakCondition($l : N, break\_edges : 2^E, \mathcal{L} : 2^N$)
32: **let** $all\_edges = break\_edges \cup in(l')$
33: **let** $value\_fn = \lambda e : all\_edges\ .\ \left(\mathbf{if}\ e \in break\_edges\ \mathbf{then}\ \overline{\mathbf{true}}\ \mathbf{else}\ \overline{\mathbf{false}}\right)$
34: **return** Simplify(Decide($all\_edges, value\_fn, \mathcal{L}$))

Figure 7: CFG to A-PEG conversion algorithm

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 created 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 6. Since there are no loops in Figure 6, ComputeInputs simply returns $\text{Decide}(\{e_1, e_2, e_3\}, value\_fn, \emptyset)$, 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 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 6.

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).

2. Reverting PEGs to CFGs

In this section we present the complement to the previous section: a procedure for converting a PEG back to a SIMPLE program. Whereas the translation from SIMPLE programs to PEGs was fairly simple, the translation back (which we call reversion) is much

more complicated. Since the order of SIMPLE execution is specified explicitly, SIMPLE programs have more structure than PEGs, and so it is not surprising that the translation from PEGs back to SIMPLE is more complex than the translation in the forward direction. Because of the complexity involved in reversion, we start by presenting a simplified version of the process in Sections 2.1 through 2.5. This simple version is correct but produces SIMPLE programs that are inefficient because they contain a lot of duplicated code. We then present in Sections 2.6 through 2.9 several optimizations on top of the simple process to improve the quality of the generated code by removing the code duplication. These optimizations include branch fusion, loop fusion, hoisting code that is common to both sides of a branch, and hoisting code out of loops. In the setting of SIMPLE, these optimizations are optional – they improve the performance of the generated code, but are not required for correctness. However, if we add side-effecting operations like heap reads/writes (discussed in detail in the journal paper [5]), these optimizations are not optional anymore: they are needed to make sure that we don't incorrectly duplicate side-effecting operations. Finally, we present more advanced techniques for reverting PEGs to CFGs rather than SIMPLE programs, taking advantage of the flexibility of CFGs in order to produce even more efficient translations of PEGs. This is particularly important when reverting PEGs translated from programs in a language with more advanced control flow such as `break`s and `continue`s.

2.1. **CFG-like PEGs.** Before we can proceed, we need to define the precondition of our reversion process. In particular, our reversion process assumes that the PEGs we are processing are *CFG-like*, as formalized by the following definition.

Definition 1 (CFG-like PEG context)**.** *We say that a PEG context* Ψ *is CFG-like if* $\Gamma \vdash \Psi : \Gamma'$ *using the rules in Figure 8.*

The rules in Figure 8 impose various restrictions on the structure of the PEG which makes our reversion process simpler. For example, these rules guarantee that the second child of an *eval* node is a *pass* node, and that by removing the second outgoing edge of each θ node, the PEG becomes acyclic. All PEGs produced by our SIMPLE-to-PEG translation process from Section 1 or CFG-to-PEG conversion process from Section 1.4 are CFG-like.

In Figure 8, Γ is a type context assigning parameter variables to types. ℓ is the largest loop-depth with respect to which the PEG node n is allowed to be loop-variant; initializing ℓ to 0 requires n to be loop-invariant with respect to all loops. Θ is an assumption context used to type-check recursive expressions. Each assumption in Θ has the form $\ell \vdash n : \tau$, where n is a θ_ℓ node; $\ell \vdash n : \tau$ states that n has type τ at loop depth ℓ. Assumptions in Θ are introduced in the Type-Theta rule, and used in the Type-Assume rule. The assumptions in Θ prevent "unsolvable" recursive PEGs such as $x = 1 + x$ or "ambiguous" recursive PEGs such as $x = 0 * x$. The requirement in Type-Eval-Pass regarding Θ prevents situations such as $x = \overline{\theta_2}(eval_1(eval_2(x, \dots), \dots), \dots)$, in which essentially the initializer for the nested loop is the final result of the outer loop.

Although $\Gamma \vdash \Psi : \Gamma'$ is the syntactic form which we will use in the body of the text, our diagrams will use the visual representation of $\Gamma \vdash \Psi : \Gamma'$ shown in Figure 9. Part (a) of the figure shows the syntactic form of the judgment (used in the text); (b) shows an example of the syntactic form; and finally (c) shows the same example in the visual form used in our diagrams.

$\boxed{\Gamma \vdash \Psi : \Gamma'}$

$$\text{Type-PEG-Context}\frac{\forall (x:\tau) \in \Gamma'.\ \ \Psi(x) = n \Rightarrow \Gamma \vdash n : \tau}{\Gamma \vdash \Psi : \Gamma'}$$

$\boxed{\Gamma \vdash n : \tau}$

$$\text{Type-PEG}\frac{\Gamma, 0, \varnothing \vdash n : \tau}{\Gamma \vdash n : \tau}$$

$\boxed{\Gamma, \ell, \Theta \vdash n : \tau}$

$$\text{Type-Param}\frac{\Gamma(x) = \tau}{\Gamma, \ell, \Theta \vdash \overline{param}(x) : \tau}$$

$$\text{Type-Op}\frac{op : (\tau_1, \ldots, \tau_n) \to \tau \quad \Gamma, \ell, \Theta \vdash n_1 : \tau_1 \quad \ldots \quad \Gamma, \ell, \Theta \vdash n_n : \tau_n}{\Gamma, \ell, \Theta \vdash \overline{op}(n_1, \ldots, n_n) : \tau}$$

$$\text{Type-Phi}\frac{\Gamma, \ell, \Theta \vdash c : \texttt{bool} \quad \Gamma, \ell, \Theta \vdash t : \tau \quad \Gamma, \ell, \Theta \vdash f : \tau}{\Gamma, \ell, \Theta \vdash \overline{\phi}(c, t, f) : \tau}$$

$$\text{Type-Theta}\frac{\begin{array}{c}\ell' = \ell - 1 \\ \Gamma, \ell', \Theta \vdash b : \tau \quad \Gamma, \ell, (\Theta, (\ell \vdash \overline{\theta_\ell}(b, n) : \tau)) \vdash n : \tau\end{array}}{\Gamma, \ell, \Theta \vdash \overline{\theta_\ell}(b, n) : \tau}$$

$$\text{Type-Eval-Pass}\frac{\begin{array}{c}\ell = \ell' + 1 \quad \forall \ell_1, n, \tau'.[(\ell_1 \vdash n : \tau') \in \Theta \Rightarrow \ell_1 < \ell'] \\ \Gamma, \ell', \Theta \vdash v : \tau \qquad\qquad \Gamma, \ell', \Theta \vdash c : \texttt{bool}\end{array}}{\Gamma, \ell, \Theta \vdash \overline{eval_{\ell'}}(v, \overline{pass_{\ell'}}(c)) : \tau}$$

$$\text{Type-Reduce}\frac{\ell \geq \ell' \quad \Theta \supseteq \Theta' \quad \Gamma, \ell', \Theta' \vdash n : \tau}{\Gamma, \ell, \Theta \vdash n : \tau}$$

$$\text{Type-Assume}\frac{(\ell \vdash n : \tau) \in \Theta}{\Gamma, \ell, \Theta \vdash n : \tau}$$

Figure 8: Rules for defining CFG-like PEGs

2.2. **Overview.** We can draw a parallel between CFG-like PEG contexts and well-typed statements: both can be seen as taking inputs Γ and producing outputs Γ'. With this parallel in mind, our basic strategy for reverting PEGs to SIMPLE programs is to recursively translate CFG-like PEG contexts $\Gamma \vdash \Psi : \Gamma'$ to well-typed SIMPLE statements $\Gamma \vdash s : \Gamma'$. Therefore, the precondition for our reversion algorithm is that the PEG context we revert must be CFG-like according to the rules in Figure 8.

For the reversion process, we make two small changes to the type checking rules for SIMPLE programs. First, we want to allow the reversion process to introduce temporary

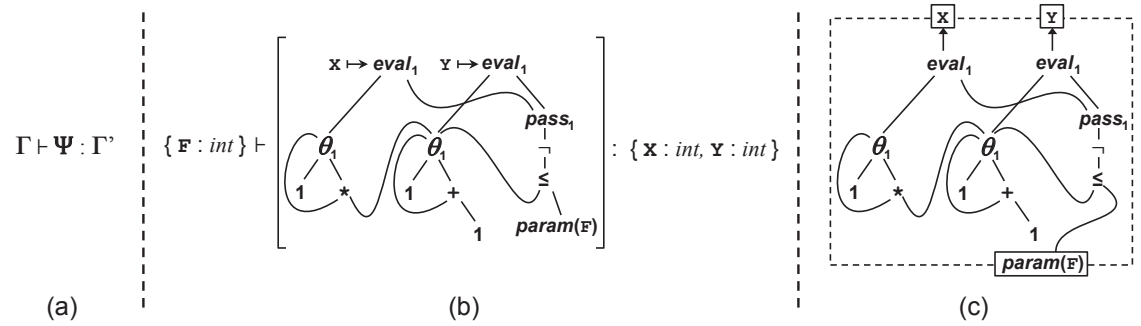

Figure 9: Visual representation of the judgment for CFG-like PEG contexts. (a) shows the syntactic form of the judgment; (b) shows an example of the syntactic form; and (c) shows the same example in visual form.

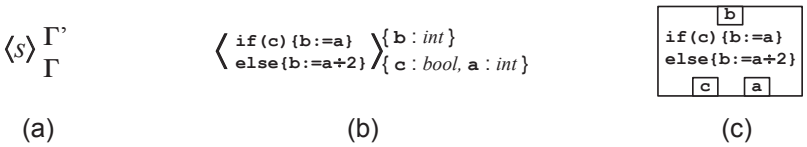

Figure 10: Diagrammatic representation of a statement node

variables without having to add them to Γ' (in $\Gamma \vdash s : \Gamma'$). To this end, we allow intermediate variables to be dropped from Γ', so that $\Gamma \vdash s : \Gamma'$ and $\Gamma'' \subseteq \Gamma'$ implies $\Gamma \vdash s : \Gamma''$.

Second, to more clearly highlight which parts of the generated SIMPLE code modify which variables, we introduce a notion $\Gamma_0; \Gamma \vdash s : \Gamma'$ of SIMPLE statements s which use but do not modify variables in Γ_0 (where Γ and Γ_0 are disjoint). We call Γ_0 the *immutable context*. The rules in Figure 2 can be updated appropriately to disallow variables from Γ_0 to be modified. Similarly, we add Γ_0 to the notion of CFG-like PEG contexts: $\Gamma_0; \Gamma \vdash \Psi : \Gamma'$. Since PEG contexts cannot modify variables anyway, the semantics of bindings in Γ_0 is exactly the same as bindings in Γ (and so we do not need to update the rules from Figure 8). Still, we keep an immutable context Γ_0 around for PEG contexts because Γ_0 from the PEG context gets reflected into the generated SIMPLE statements where it has a meaning. Thus, our reversion algorithm will recursively translate CFG-like PEG contexts $\Gamma_0; \Gamma \vdash \Psi : \Gamma'$ to well-typed SIMPLE statements $\Gamma_0; \Gamma \vdash s : \Gamma'$.

Throughout the rest of section 2, we assume that there is a PEG context $\overline{\Gamma_0}; \overline{\Gamma} \vdash \overline{\Psi} : \overline{\Gamma'}$ that we are currently reverting. Furthermore, we define Γ_0 to be:

$$\Gamma_0 = \overline{\Gamma_0} \cup \overline{\Gamma} \tag{2.1}$$

As will become clear in Section 2.3, the Γ_0 from Equation (2.1) will primarily be used as the immutable context in recursive calls to the reversion algorithm. The above definition of Γ_0 states that when making a recursive invocation to the reversion process, the immutable context for the recursive invocation is the entire context from the current invocation.

Statement nodes. The major challenge in reverting a CFG-like PEG context lies in handling the primitive operators for encoding control flow: ϕ for branches and *eval*, *pass* and θ

for loops. To handle such primitive nodes, our general approach is to repeatedly replace a subset of the PEG nodes with a new kind of PEG node called a *statement node*. A statement node is a PEG node $\langle s \rangle_{\Gamma}^{\Gamma'}$ where s is a SIMPLE statement satisfying $\Gamma_0; \Gamma \vdash s : \Gamma'$ (recall that Γ_0 comes from Equation (2.1)). The node has many inputs, one for each variable in the domain Γ, and unlike any other node we've seen so far, it also has many outputs, one for each variable of Γ'. A statement node can be perceived as a primitive operator which, given an appropriately typed list of input values, executes the statement s with those values in order to produce an appropriately typed list of output values. Although $\langle s \rangle_{\Gamma}^{\Gamma'}$ is the syntactic form which we will use in the body of the text, our diagrams will use the visual representation of $\langle s \rangle_{\Gamma}^{\Gamma'}$ shown in Figure 10. Part (a) of the figure shows the syntactic form of a statement node (used in the text); (b) shows an example of the syntactic form; and finally (c) shows the same example in the visual form used in our diagrams. Note that the visual representation in part (c) uses the same visual convention that we have used throughout for all PEG nodes: the inputs flow into the bottom side of the node, and the outputs flow out from the top side of the node.

The general approach we take is that *eval* nodes will be replaced with while-loop statement nodes (which are statement nodes $\langle s \rangle_{\Gamma}^{\Gamma'}$ where s is a while statement) and ϕ nodes will be replaced with if-then-else statement nodes (which are statement nodes $\langle s \rangle_{\Gamma}^{\Gamma'}$ where s is an if-then-else statement). To this end, our most simplistic reversion algorithm, which we present first, converts PEG contexts to statements in three steps:

(1) We replace all *eval*, *pass*, and θ nodes with while-loop statement nodes. This results in an acyclic PEG.
(2) We replace all ϕ nodes with if-then-else statement nodes. This results in a PEG with only statement nodes and domain operators such as $+$ and $*$ (that is to say, there are no more *eval*, *pass*, θ or ϕ nodes).
(3) We sequence the statement nodes and domain operators into successive assignment statements. For a statement node $\langle s \rangle_{\Gamma}^{\Gamma'}$, we simply inline the statement s into the generated code.

We present the above three steps in Sections 2.3, 2.4 and 2.5, respectively. Finally, since the process described above is simplistic and results in large amounts of code duplication, we present in sections 2.6 through 2.9 several optimizations that improve the quality of the generated SIMPLE code.

2.3. **Translating Loops.** In our first pass, we repeatedly convert each loop-invariant *eval* node, along with the appropriate *pass* and θ nodes, into a while-loop statement node. The nested loop-variant *eval* nodes will be taken care of when we recursively revert the "body" of the loop-invariant *eval* nodes to statements. For each loop-invariant $eval_\ell$ node, we apply the process described below.

First, we identify the set of θ_ℓ nodes reachable from the current $eval_\ell$ node or its $pass_\ell$ node without passing through other loop-invariant nodes (in particular, without passing through other $eval_\ell$ nodes). Let us call this set of nodes S. As an illustrative example, consider the left-most PEG context in Figure 11, which computes the factorial of 10. When processing the single $eval_1$ node in this PEG, the set S will contain both θ_1 nodes. The intuition is that each θ node in S will be a loop variable in the SIMPLE code we generate. Thus, our next step is to assign a fresh variable x for each θ_ℓ node in S; let b_x refer to the first child of the θ_ℓ node (i.e. the base case), and i_x refer to the second child (i.e. the

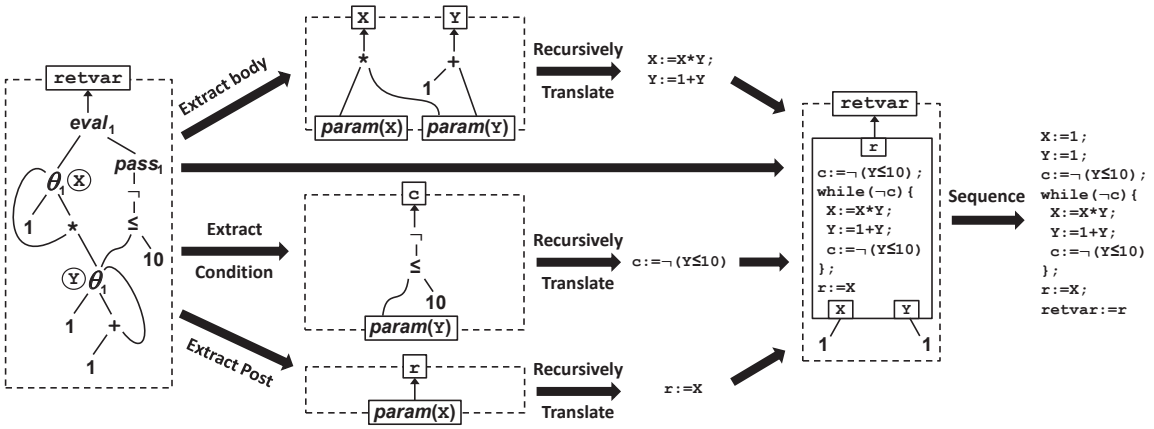

Figure 11: Example of converting *eval* nodes to while-loop statement nodes

iterative case); also, given a node $n \in S$, we use $var(n)$ for the fresh variable we just created for n. In the example from Figure 11, we created two fresh variables `x` and `y` for the two θ_1 nodes in S. After assigning fresh variables to all nodes in S, we then create a type context Γ as follows: for each $n \in S$, Γ maps $var(n)$ to the type of node n in the PEG, as given by the type-checking rules in Figure 8. For example, in Figure 11 the resulting type context Γ is $\{\texttt{x} : \texttt{int}, \texttt{y} : \texttt{int}\}$.

Second, we construct a new PEG context Ψ_i that represents the body of the loop. This PEG context will state in PEG terms how the loop variables are changed in one iteration of the loop. For each variable x in the domain of Γ, we add an entry to Ψ_i mapping x to a copy of i_x (recall that i_x is the second child of the θ node which was assigned variable x). The copy of i_x is a fully recursive copy, in which descendants have also been copied, but with one important modification: while performing the copy, when we reach a node $n \in S$, we don't copy n; instead we use a parameter node referring to $var(n)$. This has the effect of creating a copy of i_x with any occurrence of $n \in S$ replaced by a parameter node referring to $var(n)$, which in turn has the effect of expressing the next value of variable x in terms of the current values of all loop variables. From the way it is constructed, Ψ_i will satisfy $\Gamma_0; \Gamma \vdash \Psi_i : \Gamma$, essentially specifying, in terms of PEGs, how the loop variables in Γ are changed as the loop iterates (recall that Γ_0 comes from Equation (2.1)). Next, we recursively revert Ψ_i satisfying $\Gamma_0; \Gamma \vdash \Psi_i : \Gamma$ to a SIMPLE statement s_i satisfying $\Gamma_0; \Gamma \vdash s_i : \Gamma$. The top-center PEG context in Figure 11 shows Ψ_i for our running example. In this case Ψ_i states that the body of the loop modifies the loop variables as follows: the new value of `x` is `x*y`, and the new value of `y` is `1+y`. Figure 11 also shows the SIMPLE statement resulting from the recursive invocation of the reversion process.

Third, we take the second child of the *eval* node that we are processing. From the way the PEG type rules are setup in Figure 8, this second child must be the *pass* node of the *eval*. Next, we take the first child of this *pass* node, and make a copy of this first child with any occurrence of $n \in S$ replaced by a parameter node referring to $var(n)$. Let c be the PEG node produced by this operation, and let Ψ_c be the singleton PEG context $\{x_c : c\}$, where x_c is fresh. Ψ_c represents the computation of the break condition of the loop in terms of the loop variables. From the way it is constructed, Ψ_c will satisfy $\Gamma_0; \Gamma \vdash \Psi_c : \{x_c : \texttt{bool}\}$. We then recursively revert Ψ_c satisfying $\Gamma_0; \Gamma \vdash \Psi_c : \{x_c : \texttt{bool}\}$ to a SIMPLE statement s_c

satisfying $\Gamma_0; \Gamma \vdash s_c : \{x_c : \texttt{bool}\}$. s_c simply assigns the break condition of the loop to the variable x_c. The middle row of Figure 11 shows the PEG context for the break condition and the corresponding SIMPLE statement evaluating the break condition.

Fourth, we take the first child of the *eval* node and make a copy of this first child with any occurrence of $n \in S$ replaced with a parameter node referring to $var(n)$. Let r be the PEG node produced by this operation, and let Ψ_r be the singleton PEG context $\{x_r : r\}$, where x_r is fresh. Ψ_r represents the value desired after the loop in terms of the loop variables. From the way it is constructed, Ψ_r will satisfy $\Gamma_0; \Gamma \vdash \Psi_r : \{x_r : \tau\}$, where τ is the type of the first child of the *eval* node in the original PEG. We then recursively revert Ψ_r satisfying $\Gamma_0; \Gamma \vdash \Psi_r : \{x_r : \tau\}$ to a SIMPLE statement s_r satisfying $\Gamma_0; \Gamma \vdash s_r : \{x_r : \tau\}$. s_r simply assigns the value desired after the loop into variable x_r. The bottom row of Figure 11 shows the PEG context for the value desired after the loop and the corresponding SIMPLE statement evaluating the desired value. Often, but not always, the first child of the $eval_\ell$ node will be a θ_ℓ node, in which case the statement will simply copy the variable as in this example.

Finally, we replace the $eval_\ell$ node being processed with the while-loop statement node $\langle s_c; \texttt{while}\ (\neg x_c)\ \{s_i; s_c\}; s_r \rangle_\Gamma^{\{x_r:\tau\}}$. Figure 11 shows the while-loop statement node resulting from translating the *eval* node in the original PEG context. Note that the while-loop statement node has one input for each loop variable, namely one input for each variable in the domain of Γ. For each such variable x, we connect b_x to the corresponding x input of the while-loop statement node (recall that b_x is the first child of the θ node which was assigned variable x). Figure 11 shows how in our running example, this amounts to connecting 1 to both inputs of the while-loop statement node. In general, our way of connecting the inputs of the while-loop statement node makes sure that each loop variable x is initialized with the its corresponding base value b_x. After this input initialization, s_c assigns the status of the break condition to x_c. While the break condition fails, the statement s_i updates the values of the loop variables, then s_c assigns the new status of the break condition to x_c. Once the break condition passes, s_r computes the desired value in terms of the final values of the loop variables and assigns it to x_r. Note that it would be more "faithful" to place s_r inside the loop, doing the final calculations in each iteration, but we place it after the loop as an optimization since s_r does not affect the loop variables. The step labeled "Sequence" in Figure 11 shows the result of sequencing the PEG context that contains the while-loop statement node. This sequencing process will be covered in detail in Section 2.5.

Note that in the computation of the break condition in Figure 11, there is a double negation, in that we have `c := ¬... ;` and `while(¬c)`. The more advanced reversion techniques described in Section 2.10 prevents such double negations.

2.4. **Translating Branches.** After our first pass, we have replaced *eval*, *pass* and θ nodes with while-loop statement nodes. Thus, we are left with an acyclic *PEG* context that contains ϕ nodes, while-loop statement nodes, and domain operators like $+$ and $*$. In our second pass, we repeatedly translate each ϕ node into an if-then-else statement node. In order to convert a ϕ node to an if-then-else statement node, we must first determine the set of nodes which will always need to be evaluated regardless of whether the guard condition is true or false. This set can be hard to determine when there is another ϕ node nested inside the ϕ node. To see why this would be the case, consider the example in Figure 12, which we will use as a running example to demonstrate branch translation. Looking at part (a), one might think at first glance that the $\div$ node in the first diagram is always evaluated by

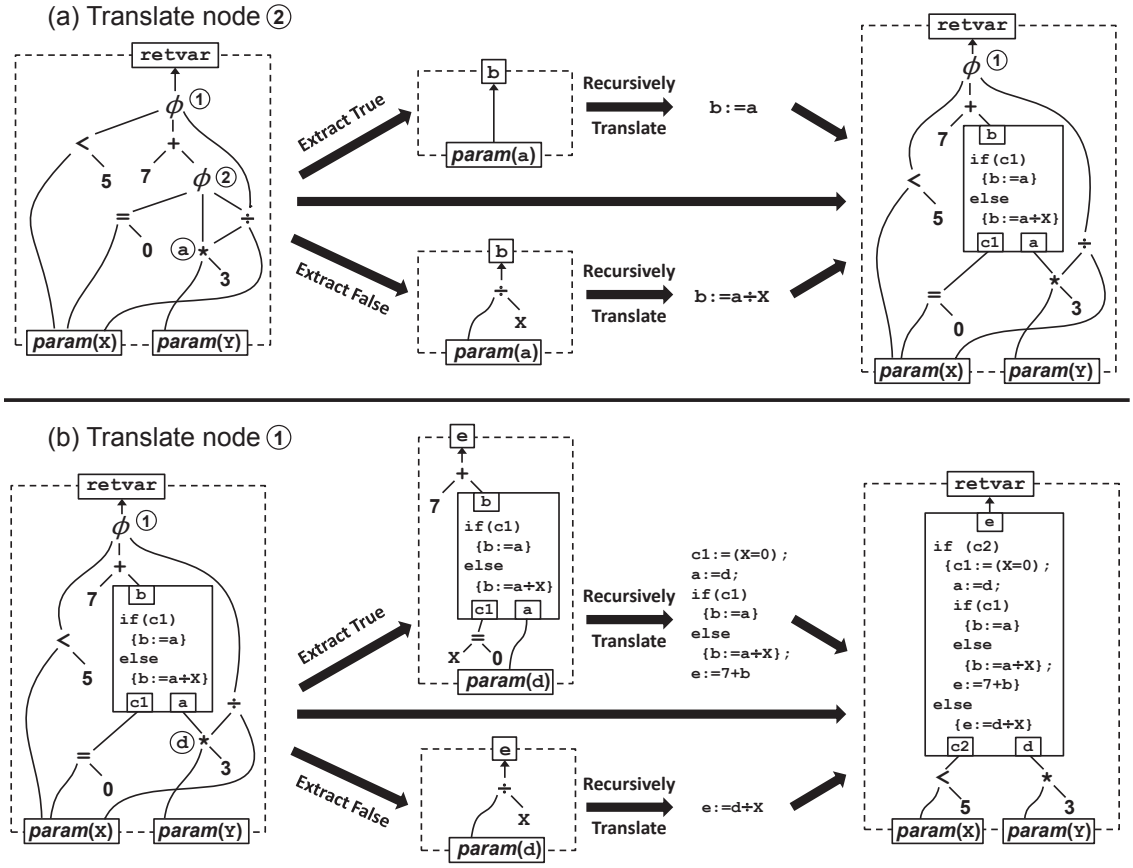

Figure 12: Example of converting ϕ nodes to if-then-else statement nodes

the ϕ node labeled ① since it is used in the PEGs for both the second and third children. However, upon further examination one realizes that actually the ÷ node is evaluated in the second child only when `x`$\neq$`0` due to the ϕ node labeled ②. To avoid these complications, it is simplest if we first convert ϕ nodes that do not have any other ϕ nodes as descendants. After this replacement, there will be more ϕ nodes that do not have ϕ descendants, so we can repeat this until no ϕ nodes are remaining. In the example from Figure 12, we would convert node ② first, resulting in (b). After this conversion, node ① no longer has any ϕ descendants and so it can be converted next. Thus, we replace ϕ nodes in a bottom-up order. For each ϕ node, we use the following process.

First, we determine the set S of nodes that are descendants of both the second and third child of the current ϕ node (i.e. the true and false branches). These are the nodes that will get evaluated regardless of which way the ϕ goes. We assign a fresh variable to each node in this set, and as in the case of loops we use $var(n)$ to denote the variable we've assigned to n. In Figure 12(a), the `*` node is a descendant of both the second and third child of node ②, so we assign it the fresh variable `a`. Note that the `3` node should also be assigned a variable, but we do not show this in the figure since the variable is never used. Next, we take the second child of the ϕ node and make a copy of this second child in which any occurrence of $n \in S$ has been replaced with a parameter node referring to $var(n)$. Let

t be the PEG node produced by this operation. Then we do the same for the third child of the ϕ node to produce another PEG node f. The t and f nodes represent the true and false computations in terms of the PEG nodes that get evaluated regardless of the direction the ϕ goes. In the example from Figure 12(a), t is $\overline{param}(\texttt{a})$ and f is $\overline{\div}(\overline{param}(\texttt{a}), \overline{param}(\texttt{x}))$. Examining t and f, we produce a context Γ of the newly created fresh variables used by either t or f. In the example, Γ would be simply $\{\texttt{a} : \texttt{int}\}$. The domain of Γ does not contain `x` since `x` is not a new variable (i.e. `x` is in the domain of Γ_0, where Γ_0 comes from Equation (2.1)). Thus, t and f are PEGs representing the true and false cases in terms of variables Γ representing values that would be calculated regardless.

Second, we invoke the reversion process recursively to translate t and f to statements. In particular, we create two singleton contexts $\Psi_t = \{x_\phi : t\}$ and $\Psi_f = \{x_\phi : f\}$ where x_ϕ is a fresh variable, making sure to use the same fresh variable in the two contexts. From the way it is constructed, Ψ_t satisfies $\Gamma_0; \Gamma \vdash \Psi_t : \{x_\phi : \tau\}$, where τ is the type of the ϕ node. Thus, we recursively revert Ψ_t satisfying $\Gamma_0; \Gamma \vdash \Psi_t : \{x_\phi : \tau\}$ to a SIMPLE statement s_t satisfying $\Gamma_0; \Gamma \vdash s_t : \{x_\phi : \tau\}$. Similarly, we revert Ψ_f satisfying $\Gamma_0; \Gamma \vdash \Psi_f : \{x_\phi : \tau\}$ to a statement s_f satisfying $\Gamma_0; \Gamma \vdash s_f : \{x_\phi : \tau\}$. The steps labeled "Extract True" and "Extract False" in Figure 12 show the process of producing Ψ_t and Ψ_f in our running example, where the fresh variable x_ϕ is `b` in part (a) and `e` in part (b). Note that Ψ_t and Ψ_f may themselves contain statement nodes, as in Figure 12(b), but this poses no problems for our recursive algorithm. Finally, there is an important notational convention to note in Figure 12(a). Recall that Ψ_f satisfies $\Gamma_0; \Gamma \vdash \Psi_f : \{x_\phi : \tau\}$, and that in Figure 12(a) $\Gamma_0 = \{\texttt{x} : \texttt{int}\}$ and $\Gamma = \{\texttt{a} : \texttt{int}\}$. In the graphical representation of $\Gamma_0; \Gamma \vdash \Psi_f : \{x_\phi : \tau\}$ in Figure 12(a), we display variable `a` as a real boxed input (since it is part of Γ), whereas because `x` is in Γ_0, we display `x` without a box, and using the shorthand of omitting the *param* (even though in reality it is there).

Finally, we replace the ϕ node we are processing with the if-then-else statement node $\langle \texttt{if}\ (x_c)\ \{s_t\}\ \texttt{else}\ \{s_f\} \rangle^{\{x_\phi : \tau\}}_{(\Gamma, x_c : \texttt{bool})}$ (where x_c is fresh). This statement node has one input for each entry in Γ, and it has one additional input x_c for the guard value. We connect the x_c input to the first child of the ϕ node we are currently processing. For each variable x in the domain of Γ, we connect the x input of the statement node to the "always-evaluated" node n for which $var(n) = x$. Figure 12 shows the newly created if-then-else statement nodes and how they are connected when processing ϕ node ① and ②. In general, our way of connecting the inputs of the if-then-else statement node makes sure that each always-evaluated node is assigned to the appropriate variable, and the guard condition is assigned to x_c. After this initialization, the statement checks x_c, the guard condition, to determine whether to take the true or false branch. In either case, the chosen statement computes the desired value of the branch and assigns it to x_ϕ.

2.5. **Sequencing Operations and Statements.** When we reach our final pass, we have already eliminated all primitive operators (*eval*, *pass*, θ, and ϕ) and replaced them with statement nodes, resulting in an acyclic PEG context containing only statement nodes and domain operators like + and $*$. At this point, we need to sequence these statement nodes and operator nodes. We start off by initializing a statement variable S as the empty statement. We will process each PEG node one by one, postpending lines to S and then replacing the PEG node with a parameter node. It is simplest to process the PEG nodes from the bottom up, processing a node once all of its inputs are parameter nodes. Figure 13

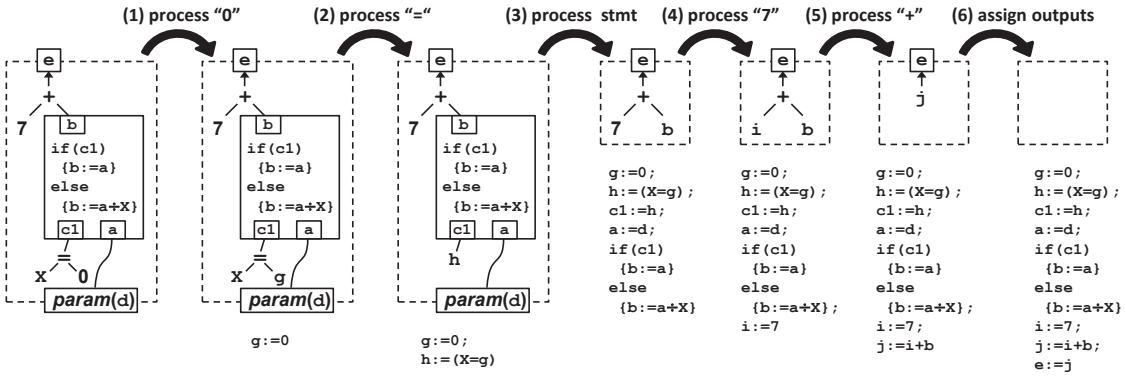

Figure 13: Example of sequencing a PEG context (with statement nodes) into a statement

shows every stage of converting a PEG context to a statement. At each step, the current statement S is shown below the PEG context.

If the node being processed is a domain operator, we do the following. Because we only process nodes where all of its inputs are parameter nodes, the domain operator node we are processing must be of the following form: $\overline{op}(\overline{param}(x_1), \ldots, \overline{param}(x_k))$. We first designate a fresh variable x. Then, we postpend the line $x := op(x_1, \ldots, x_k)$ to S. Finally, we replace the current node with the node $\overline{param}(x)$. This process is applied in the first, second, fourth, and fifth steps of Figure 13. Note that in the first and fourth steps, the constants 0 and 7 are a null-ary domain operators.

If on the other hand the node being processed is a statement node, we do the following. This node must be of the following form: $\langle s \rangle_{\Gamma}^{\Gamma'}$. For each input variable x in the domain of Γ, we find the $\overline{param}(x_0)$ that is connected to input x, and postpend the line $x := x_0$ to S. In this way, we are initializing all the variables in the domain of Γ. Next, we postpend s to S. Finally, for each output variable x' in the domain of Γ', we replace any links in the PEG to the x' output of the statement node with a link to $\overline{param}(x')$. This process is applied in the third step of Figure 13.

Finally, after processing all domain operators and statement nodes, we will have each variable x in the domain of the PEG context being mapped to a parameter node $\overline{param}(x')$. So, for each such variable, we postpend the line $x' := x$ to S. All these assignments should intuitively run in parallel. This causes problems if there is a naming conflict, for example x gets y and y gets x. In such cases, we simply introduce intermediate fresh copies of all the variables being read, and then we perform all the assignments by reading from the fresh copies. In the case of x and y, we would create copies x' and y' of x and y, and then assign x' to y, and y' to x. This process is applied in the sixth and last step of Figure 13 (without any naming conflicts). The value of S is the final result of the reversion, although in practice we apply copy propagation to this statement since the sequencing process produces a lot of intermediate variables.

2.6. **Loop Fusion.** Although the process described above will successfully revert a PEG to a SIMPLE program, it does so by duplicating a lot of code. Consider the reversion process in Figure 14. The original SIMPLE program for this PEG was as follows:

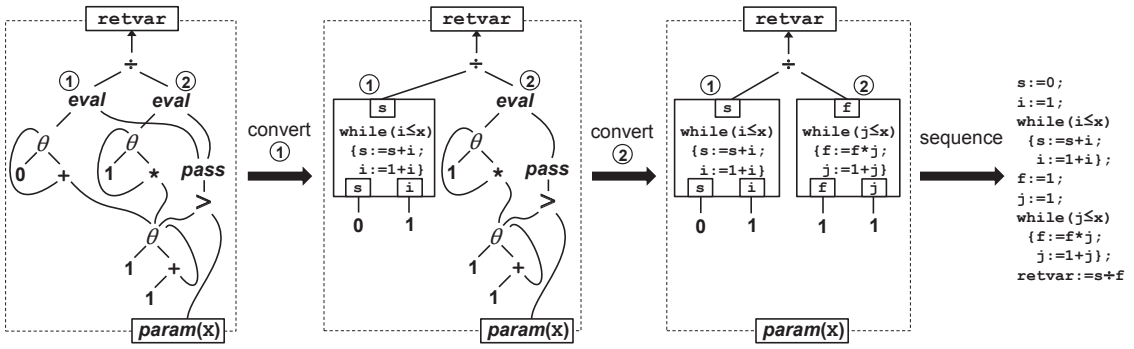

Figure 14: Reversion of a PEG without applying loop fusion

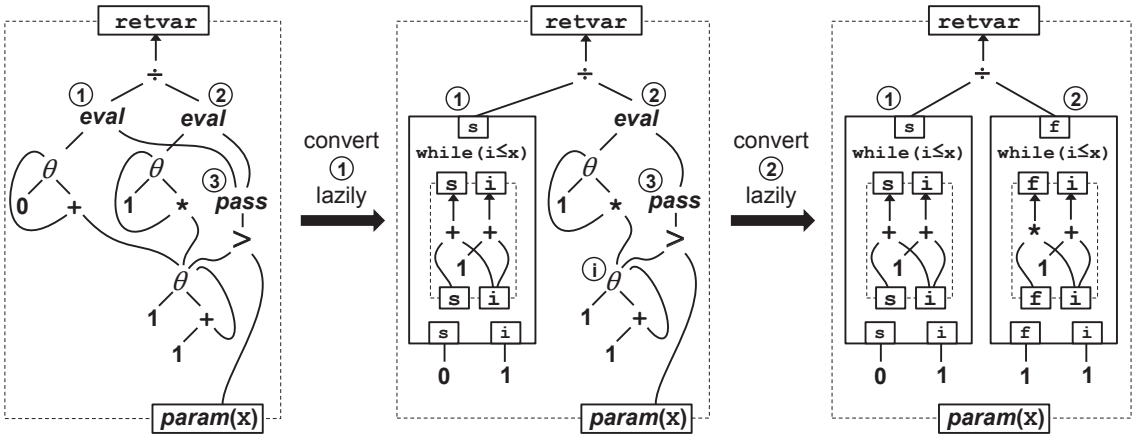

Figure 15: Conversion of *eval* nodes to revised loop nodes

```
s:=0; f:=1; i:=1;
while(i<=x) { s:=s+i; f:=f*i; i:=1+i };
retvar:=s÷f
```

The conversion of the above code to PEG results in two *eval* nodes, one for each variable that is used after the loop. The reversion process described so far converts each *eval* node separately, resulting in two separate loops in the final SIMPLE program. Here we present a simple optimization that prevents this code duplication by fusing loops during reversion. In fact, this added loop-fusion step can even fuse loops that were distinct in the original program. Thus, loop fusion can be performed simply by converting to a PEG and immediately reverting back to a SIMPLE program, without even having to do any intermediate transformations on the PEG.

We update our reversion process to perform loop-fusion by making several changes. First, we modify the process for converting *eval* nodes to while-loop statement nodes in three ways; the revised conversion process is shown in Figure 15 using the same PEG as before. The first modification is that we tag each converted θ node with the fresh variable we designate for it. For example, the conversion process for the first *eval* node in Figure 15 generates a fresh variable `i` for one of the θ nodes, and so we tag this θ node with `i`. If we

ever convert a θ node that has already been tagged from an earlier *eval* conversion, we reuse that variable. For example, when converting the second *eval* node in Figure 15, we reuse the variable `i` unlike in Figure 14 where we introduced a fresh variable `j`. This way all the *eval* nodes are using the same naming convention. The second modification is that when processing an *eval* node, we do not immediately revert the PEG context for the loop body into a statement, but rather we remember it for later. This is why the bodies of the while-loop statement nodes in Figure 15 are still PEG contexts rather than statements. Thus, we have to introduce a new kind of node, which we call a *loop node*, which is like a while-loop statement node, except that it stores the body (and only the body) of the loop as a PEG context rather than a statement – the remaining parts of the loop are still converted to statements (in particular, the condition and the post-loop computation are still converted to statements, as was previously shown in Figure 11). As an example, nodes ① and ② in the right most part of Figure 15 are loop nodes. Furthermore, because we are leaving PEGs inside the loop nodes to be converted for later, we use the term "convert lazily" in Figure 15. The third modification is that the newly introduced loop nodes store an additional piece of information when compared to while-loop statement nodes. In particular, when we replace an *eval* node with a loop node, the new loop node will store a link back to the *pass* node of the *eval* node being replaced. We store these additional links so that we can later identify fusable loops: we will consider two loop nodes fusable only if they share the same *pass* node. We do not show these additional links explicitly in Figure 15, but all the loop nodes in that Figure implicitly store a link back to the same *pass* node, namely node ③.

Second, after converting the ϕ nodes but before sequencing, we search for loop nodes which can be fused. Two loop nodes can be fused if they share the same *pass* node and neither one is a descendant of the other. For example, the two loop nodes in Figure 15 can be fused. If one loop node is a descendant of the other, then the result of finishing the descendant loop is required as input to the other loop, and so they cannot be executed simultaneously. To fuse the loops, we simply union their body PEG contexts, as well as their inputs and their outputs. The step labeled "fuse ① & ②" in Figure 16 demonstrates this process on the result of Figure 15. This technique produces correct results because we used the same naming convention across *eval* nodes and we used fresh variables for all θ nodes, so no two distinct θ nodes are assigned the same variable. We repeat this process until there are no fusable loop nodes.

Finally, the sequencing process is changed to first convert all loop nodes to while-loop statement nodes, which involves recursively translating the body PEG context inside the loop node to a statement. This additional step is labeled "convert loops to statements" in Figure 16. The final SIMPLE program has only one while loop which simultaneously calculates both of the desired results of the loop, as one would expect.

To summarize, the process described so far is to (1) translate *eval* nodes into loop nodes, (2) translate ϕ nodes into if-then-else statements, (3) perform fusion of loop nodes and (4) sequencing step. It is important to perform the fusion of loop nodes *after* converting ϕ nodes, rather than after converting *eval* nodes. Consider for example two loops with the same break condition, neither of which depend on the other, but where one is always executed and the other is only executed when some branch guard is true (that is to say, its result is used only on one side of a ϕ node). If we perform fusion of loop nodes before converting ϕ nodes, then these two loop nodes appear to be fusable, but fusing them would cause both loops to always be evaluated, which is not semantics preserving. We avoid this problem by processing all ϕ nodes first, after which point we know that all the remaining

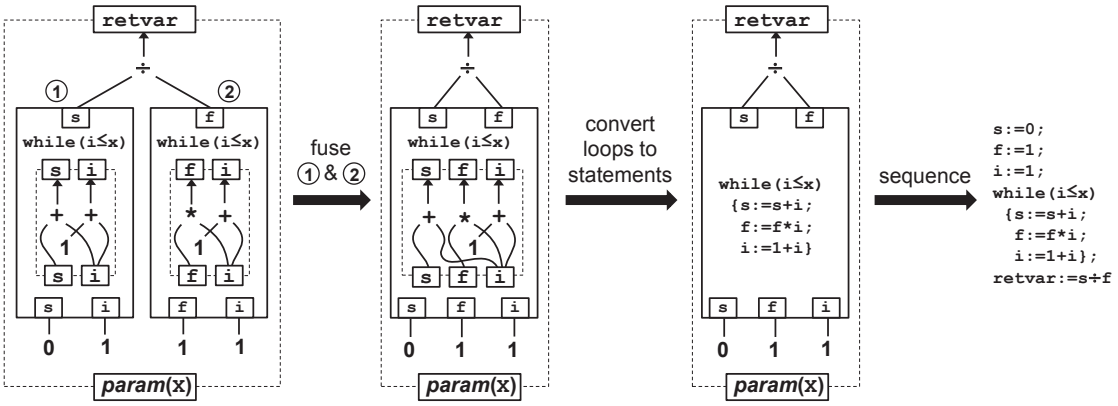

Figure 16: Fusion of loop nodes

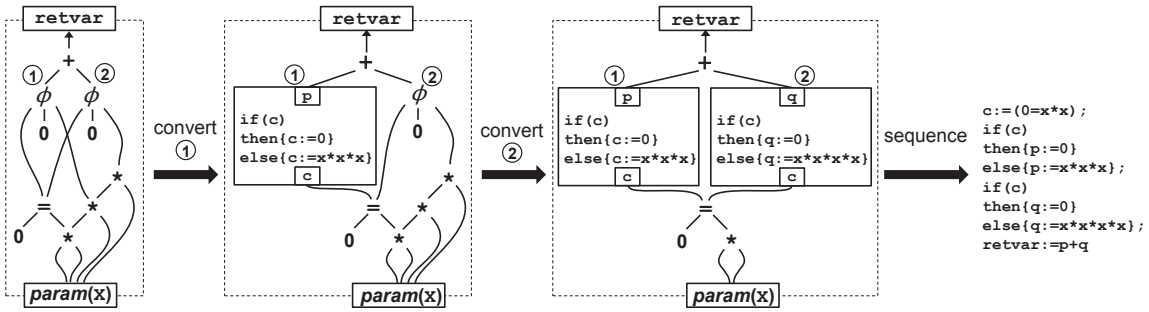

Figure 17: Reversion of a PEG without applying branch fusion

nodes in the PEG context must be executed (although some of these nodes may be branch nodes). In the example just mentioned with two loops, the loop which is under the guard (and thus under a ϕ node) will be extracted and recursively processed when the ϕ node is transformed into a branch node. In this recursive reversion, only one loop will be present, the one under the guard, and so no loop fusion is performed. After the ϕ node is processed, there will be only one remaining loop node, the one which is executed unconditionally. Again, since there is only one loop node, no loop fusion is performed, and so the semantics is preserved.

2.7. **Branch Fusion.** In the same way that our previously described reversion process duplicated loops, so does it duplicate branches, as demonstrated in Figure 17. Similarly to loop fusion, our reversion process can be updated to perform branch fusion.

First, we modify the processing of ϕ nodes to make the reversion of recursive PEG contexts lazy: rather than immediately processing the extracted true and false PEG contexts, as was done in Figure 12, we instead create a new kind of node called a *branch node* and store the true and false PEG contexts in that node. A branch node is like an if-then-else statement node, except that instead of having SIMPLE code for the true and false sides of the statement, the branch node contains PEG contexts to be processed later. As with if-then-else statement nodes, a branch node has a guard input which is the first child of

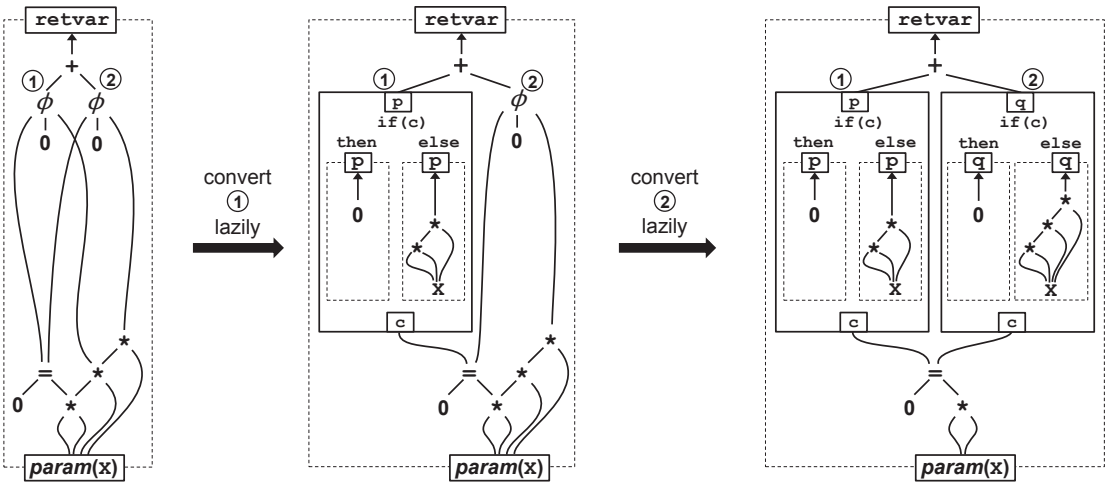

Figure 18: Conversion of ϕ nodes to revised branch nodes

the ϕ node being replaced (that is to say, the value of the branch condition). For example, Figure 18 shows this lazy conversion of ϕ nodes on the same example as Figure 17. The nodes labeled ① and ② in the right most part of Figure 18 are branch nodes.

Second, after all ϕ nodes have been converted to branch nodes, we search for branch nodes that can be fused. If two branch nodes share the same guard condition input, and neither one is a descendant of the other, then they can be fused. Their true PEG contexts, false PEG contexts, inputs, and outputs are all unioned together respectively. This process is much like the one for loop fusion, and is demonstrated in Figure 19 in the step labeled "fuse ① & ②". Notice that when we union two PEGs, if there is a node in each of the two PEGs representing the exact same expression, the resulting union will only contain one copy of this node. This leads to an occurrence of common sub-expression elimination in Figure 19: when the false and true PEGs are combined during fusion, the resulting PEG only has one `x*x*x`, which allows the computation for `q` in the final generated code to be `c*x`, rather than `x*x*x*x`.

Finally, the sequencing process is changed to first convert all branch nodes into statement nodes, which involves recursively translating the true and false PEG contexts inside the branch node to convert them to statements. This additional step is labeled "convert branches to statements" in Figure 19. The final SIMPLE program has only one if-then-else which simultaneously calculates both of the desired results of the branches, as one would expect.

To summarize, the process described so far is to (1) translate *eval* nodes into loop nodes, (2) translate ϕ nodes into branch nodes, (3) perform fusion of loop nodes (4) perform fusion of branch nodes (5 sequencing step. As with loop fusion, it is important to perform fusion of branch nodes after each and every ϕ node has been converted to branch nodes. Otherwise, one may end up fusing two branch nodes where one branch node is used under some ϕ and the other is used unconditionally.

If two branch nodes share the same guard condition input, but one is a descendant of the other, we can even elect to fuse them vertically, as shown in Figure 20. In particular, we sequence the true PEG context of one branch node with the true PEG context of the

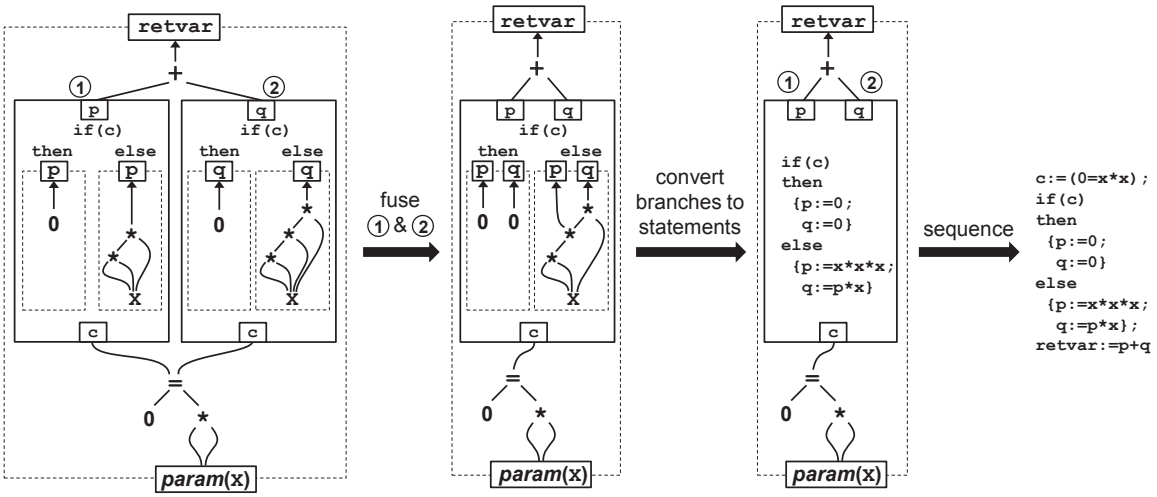

Figure 19: Fusion of branch nodes

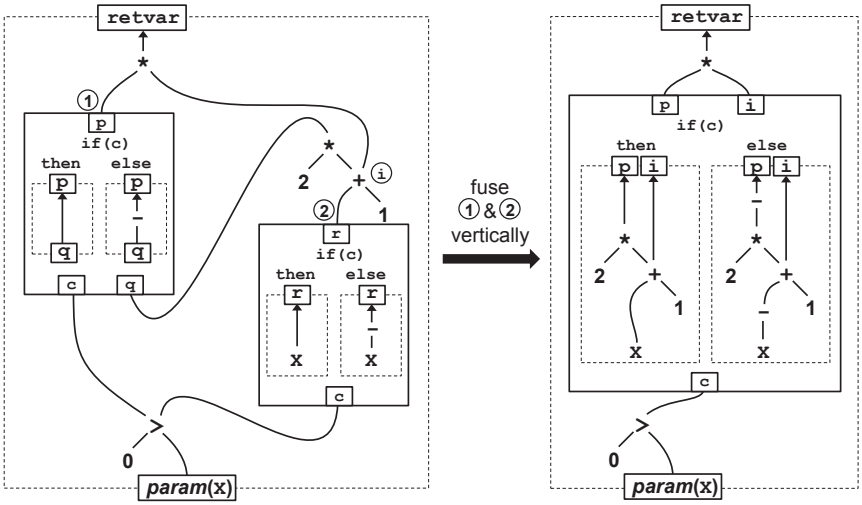

Figure 20: Vertical fusion of branch nodes

other, and do the same with the false PEG contexts. Note how, because the node labeled ⓘ is used elsewhere than just as an input to branch node ①, we added it as an output of the fused branch node.

2.8. **Hoisting Redundancies from Branches.** Looking back at the branch fusion example from Figures 18 and 19, there is still one inefficiency in the generated code. In particular, `x*x` is computed in the false side of the branch, even though `x*x` has already been computed before the branch.

In our original description for converting ϕ nodes in Section 2.4, we tried to avoid this kind of redundant computation by looking at the set of nodes that are reachable from both the true and false children (second and third children) of a ϕ node. This set was meant to

capture the nodes that, for a given ϕ, are need to be computed regardless of which side the ϕ node goes – we say that such nodes execute unconditionally with respect to the given ϕ node. These nodes were kept outside of the branch node (or the if-then-else statement node if using such nodes). As an example, the node labeled ⓐ in Figure 12 was identified as belonging to this set when translating ϕ node ②, and this is why the generated if-then-else statement node does not contain node ⓐ, instead taking it as an input (in addition to the `c1` input which is the branch condition).

It is important to determine as completely as possible the set of nodes that execute unconditionally with respect to a ϕ. Otherwise, code that intuitively one would think of executing unconditionally outside of the branch (either before the branch or after it) would get duplicated in one or both sides of branch. This is precisely what happened in Figure 18: our approach of computing the nodes that execute unconditionally (by looking at nodes reachable from the true and false children) returned the empty set, even though `x*x` actually executes unconditionally. This is what lead to `x*x` being duplicated, rather than being kept outside of the branch (in the way that `a` was in Figure 12). A more precise analysis would be to say that a node executes unconditionally with respect to a ϕ node if it is reachable from the true and false children of the ϕ (second and third children), *or* from the branch condition (first child). This would identify `x*x` as being executed unconditionally in Figure 12. However, even this more precise analysis has limitations. Suppose for example that some node is used only on the true side of the ϕ node, and never used by the condition, so that the more precise analysis would not identify this node as always executing. However, this node could be used unconditionally higher up in the PEG, or alternatively it could be the case that the condition of the ϕ node is actually equivalent to true. In fact, this last possibility points to the fact that computing exactly what nodes execute unconditionally with respect to a ϕ node is undecidable (since it reduces to deciding if a branch is taken in a Turing-complete computation). However, even though the problem is undecidable, more precision leads to less code duplication in branches.

MustEval Analysis. To modularize the part of the system that deals with identifying nodes that must be evaluated unconditionally, we define a MustEval analysis. This analysis returns a set of nodes that are known to evaluate unconditionally in the current PEG context. An implementation has a lot of flexibility in how to define the MustEval analysis. More precision in this analysis leads to less code duplication in branches.

Figure 21 shows the example from Figure 18 again, but this time using a refined process that uses the MustEval analysis. The nodes which our MustEval analysis identifies as always evaluated have been marked, including `x*x`. After running the MustEval analysis, we convert all ϕ nodes which are marked as always evaluated. All remaining ϕ nodes will be pulled into the resulting branch nodes and so handled by recursive calls. Note that this is a change from Section 2.4, where ϕ nodes were processed starting from the lower ones (such as node ② in Figure 12) to the higher ones (such as node ① in Figure 12). Our updated process, when running on the example from Figure 12, would process node ① first, and in doing so would place node ② in the true PEG context of a branch node. Thus, node ② would get processed later in a recursive reversion.

Going back to Figure 21, both ϕ nodes are marked as always evaluated, and so we process both of them. To have names for any values that are always computed, we assign a fresh variable to each node that is marked by the MustEval analysis, reusing the same fresh variables for each ϕ node we convert. For example, in Figure 21 we use the fresh variable ⓢ

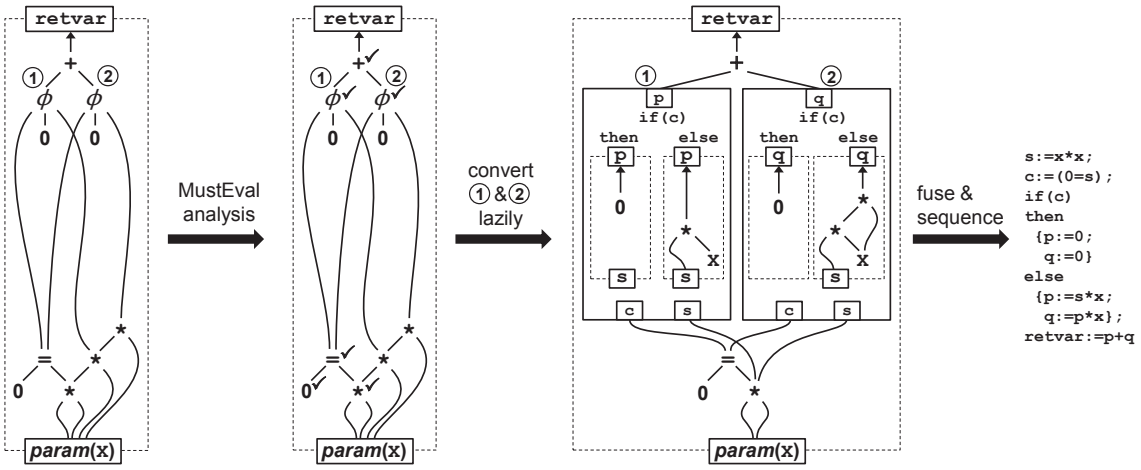

Figure 21: Same example as in Figure 18, but this time hoisting redundancies from branches

for the node `x*x`. We then produce a t and f node for each ϕ node as before, replacing nodes which will always be evaluated with parameter nodes of appropriate variables. Figure 18 shows that t for the first ϕ node is simply `0` whereas f is `s*x`, using `s` in place of `x*x`. After all the ϕ nodes have been converted, we perform branch fusion and sequencing as before. Figure 18 shows this updated process. The resulting code now performs the `x*x` computation before the branch.

One subtlety is that the MustEval analysis must satisfy some minimal precision requirements. To see why this is needed, recall that after the MustEval analysis is run, we now only process the ϕ nodes that are marked as always evaluated, leaving the remaining ϕ nodes to recursive invocations. Thus, if MustEval doesn't mark any nodes as being always evaluated, then we would not process any ϕ nodes, which is a problem before after the ϕ-processing stage, we require there to be no more ϕ nodes in the PEG. As a result, we require the MustEval analysis to be *minimally precise*, as formalized in the following definition.

Definition 2. *We say that a MustEval analysis is minimally precise if for any PEG context* Ψ, $S = \mathit{MustEval}(\Psi)$ *implies the following properties:*

$$\begin{aligned}
(x, n) \in \Psi &\Rightarrow n \in S \\
\overline{op}(n_1 \ldots n_k) \in S &\Rightarrow n_1 \in S \wedge \ldots \wedge n_k \in S \\
\overline{\phi}(c, a, b) \in S &\Rightarrow c \in S \\
\overline{\langle s \rangle}(n_1 \ldots n_k) \in S &\Rightarrow n_1 \in S \wedge \ldots \wedge n_k \in S
\end{aligned}$$

In essence the above simply states that MustEval must at least return those nodes which can trivially be identified as always evaluated. To see why this is sufficient to guarantee that we make progress on ϕ nodes, consider the worst case, which is when MustEval returns nothing more than the above trivially identified nodes. Suppose we have a ϕ node that is not identified as always evaluated. This node will be left to recursive invocations of reversion, and at some point in the recursive invocation chain, as we translate more and more ϕ nodes into branch nodes, our original ϕ node will become a top-level node that is always evaluated

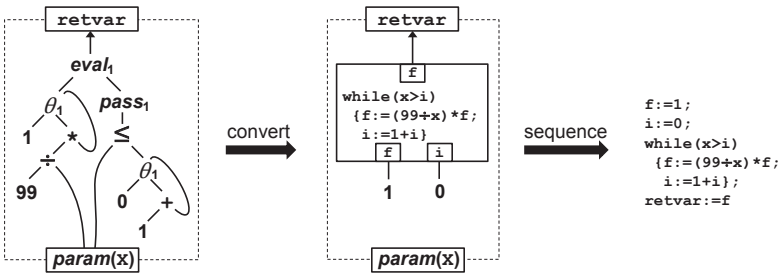

Figure 22: Reversion of a PEG without applying loop-invariant code motion

The $invariant_\ell(n)$ predicate is the largest predicate that satisfies the following three rules:

(1) if $L(n)$ is θ_ℓ, then n is not $invariant_\ell$
(2) if $L(n)$ is $eval_\ell$, then if the second child of n is not $invariant_\ell$ then n is also not $invariant_\ell$
(3) otherwise if $L(n)$ is not $pass_\ell$, then if any child of n is not $invariant_\ell$ then n is also not $invariant_\ell$

Figure 23: Definition of the Syntactic Loop-Invariance Predicate

(in the PEG context being processed in the recursive invocation). At that point we will process it into a branch node.

2.9. **Loop-Invariant Code Motion.** The process described so far for converting *eval* nodes in Section 2.3 duplicates code unnecessarily. In particular, every node used by the loop body is copied into the loop body, including loop-invariant nodes. Figure 22 shows how placing the loop-invariant operation `99÷x` into the loop body results in the operation being evaluated in every iteration of the loop. In the same way that in Section 2.7 we updated our processing of ϕ nodes to hoist computations that are common to both the true and false sides, we can also update our processing of *eval* nodes to hoist computations that are common across all loop iterations, namely loop-invariant computations.

In Figure 23 we define a predicate $invariant_\ell(n)$ which is true if the value of n does not vary in loop ℓ, meaning that n is invariant with respect to loop ℓ. The general approach will therefore be as follows: when converting $eval_\ell$, we simply keep any node n that is invariant with respect to ℓ outside of the loop body. Unfortunately, there are some subtleties with making this approach work correctly. For example, consider the PEG from Figure 22. In this PEG the ÷ node is invariant with respect to loop 1 (`99÷x` produces the same value no matter what iteration the execution is at). However, if we were to evaluate the loop-invariant operation `99÷x` before the loop, we would change the semantics of the program. In particular, if `x` were 0, the ÷ operation would fail, whereas the original program would simply terminate and return 1 (because the original program only evaluates `99÷x` if `x` is strictly greater than 0). Thus, by pulling the loop-invariant operation out of the loop, we have changed the semantics of the program.

Even traditional formulations of loop-invariant code motion must deal with this problem. The standard solution is to make sure that pulling loop-invariant code outside of a loop does not cause it to execute in cases where it would not have originally. In our PEG

Rewrite $eval_\ell(a, pass_\ell(c))$ to $\phi(eval_\ell(c, \mathrm{Z}), eval_\ell(a, \mathrm{Z}), eval_\ell(peel_\ell(a), pass_\ell(peel_\ell(c))))$ to start loop peeling. Then apply the following rewrite rules until completion:

$$peel_\ell(n) \text{ rewrites to } \begin{cases} n = \theta_\ell(a, b) & b \\ invariant_\ell(n) & n \\ \textbf{otherwise } n = op(a_1, \ldots, a_k) & op(peel_\ell(a_1), \ldots, peel_\ell(a_k)) \end{cases}$$

$$eval_\ell(n, \mathrm{Z}) \text{ rewrites to } \begin{cases} n = \theta_\ell(a, b) & a \\ invariant_\ell(n) & n \\ \textbf{otherwise } n = op(a_1, \ldots, a_k) & op(eval_\ell(a_1, \mathrm{Z}), \ldots, eval_\ell(a_k, \mathrm{Z})) \end{cases}$$

Figure 24: Rewrite Process for Loop Peeling

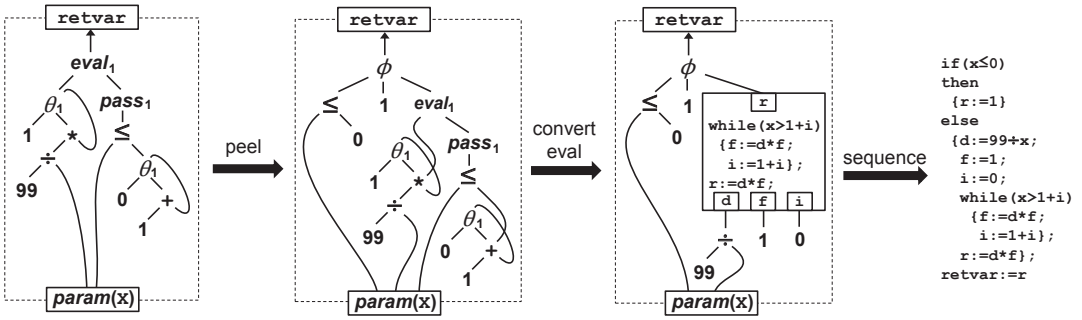

Figure 25: Reversion of a PEG after peeling the loop once

setting, there is a simple but very conservative way to guarantee this: when processing an $eval_\ell$ node, if we find a node n that is invariant with respect to ℓ, we pull n outside of the loop only if there are no θ or ϕ nodes between the $eval_\ell$ node and n. The intuition behind disallowing ϕ and θ is that both of these nodes can bypass evaluation of some of their children: the ϕ chooses between its second and third child, bypassing the other, and the θ node can bypass its second child if the loop performs no iterations. This requirement is more restrictive than it needs to be, since a ϕ node always evaluates its first child, and so we could even allow ϕ nodes, as long as the loop invariant node was used in the first child, not the second or third. In general, it is possible to modularize the decision as to whether some code executes more frequently than another in an *evaluation-condition analysis*, or EvalCond for short. An EvalCond analysis would compute for every node in the PEG context an abstract evaluation condition capturing under which cases the PEG node is evaluated. EvalCond is a generalization of the MustEval analysis from Section 2.8, and as with MustEval, an implementation has a lot of flexibility in defining EvalCond. In the more general setting of using an EvalCond analysis, we would only pull a loop-invariant node if its evaluation condition is implied by the evaluation condition of the *eval* node being processed.

Since we now prevent loop-invariant code from being hoisted if it would execute more often after being hoisted, we correctly avoid pulling `99÷x` out of the loop. However, as is well known in the compiler literature [1], even in such cases it is still possible to pull the loop-invariant code out of the loop by performing loop peeling first. For this reason, we perform loop peeling in the reversion process in cases where we find a loop invariant-node

that (1) cannot directly be pulled out because doing so would make the node evaluate more often after hosting and (2) is always evaluated provided the loop iterates a few times. Figure 24 shows the rewrite process for loop peeling on a CFG-like PEG. Loop peeling, the operator $peel_\ell$, and constant Z are explained in more detail in the journal paper [5]. In short, $peel_\ell$ essentially gets the value of a loop variable in the "next" iteration of the loop while $eval(-, \mathrm{Z})$ gets the value of a loop variable upon entering the loop. Using the same starting example as before, Figure 25 shows the result of this peeling process (step labeled "peel"). After peeling, the ϕ node checks the entry condition of the original loop and evaluates the peeled loop if this condition fails. Notice that the *eval* and $\leq$ nodes in the new PEG loop refer to the second child of the θ nodes rather than θ nodes themselves, effectively using the value of the loop variables after one iteration. An easy way to read such nodes is to simply follow the edges of the PEG; for example, the "+" node can be read as "$1 + \theta_1(\ldots)$".

In general, we repeat the peeling process until the desired loop-invariant nodes used by the body of the loop are also used before the body of the loop. In our example from Figure 25, only one run of peeling is needed. Notice that, after peeling, the $\div$ node is still loop-invariant, but now there are no ϕ or θ nodes between the *eval* node and the $\div$ node. Thus, although `99÷x` is not always evaluated (such as when `x≤0` is true), it is always evaluated whenever the *eval* node is evaluated, so it is safe to keep it out of the loop body. As a result, when we convert the *eval* nodes to loop nodes, we no longer need to keep the $\div$ node in the body of the loop, as shown in Figure 25. Figure 25 also shows the final generated SIMPLE program for the peeled loop. Note that the final code still has some code duplication: `1+i` is evaluated multiple times in the same iteration, and `d*f` is evaluated both when the while-loop guard succeeds and the guard fails. These redundancies are difficult to remove without using more advanced control structures that are not present in SIMPLE. Our implementation can take advantage of more advanced control structures to remove these remaining redundancies. We explain these techniques in Section 2.10.

We should also note that, since the EvalCond analysis can handle loop operators and subsumes the MustEval analysis, it is possible to convert ϕ nodes before converting *eval* nodes, although both still need to happen after the loop peeling stage. This rearrangement enables more advanced redundancy elimination optimizations.

2.10. **Advanced Control Flow.** So far we have been reverting PEGs to SIMPLE programs. However, SIMPLE has simplistic control flow, whereas more realistic languages such as Java have more advanced and very useful control flow: specifically `break`s and `continue`s. Although programs using these constructs can be translated to use only basic branch and loop structures, this translation often results in less efficient and more complicated CFGs. In order to revert PEGs resulting from realistic languages to efficient CFGs, the reversion process needs to be able to handle advanced control structures. Here we explain how the reversion process can be revised to accomplish this.

Figure 26 shows how a SIMPLE program with a `break` is represented as a PEG. The PEG is significantly more complicated since `break` introduces a new exit point for the loop. In particular, the loop has two break conditions: the negation of the loop guard, and the guard of the `break` statement. The ϕ node used by the *pass* node indicates that if the loop guard fails then the loop ends, otherwise if the `break` guard passes then the loop ends. The ϕ node used by the *eval* node indicates that if the loop guard holds in the last iteration of the loop then the loop must have broken via the `break` statement so it must return the

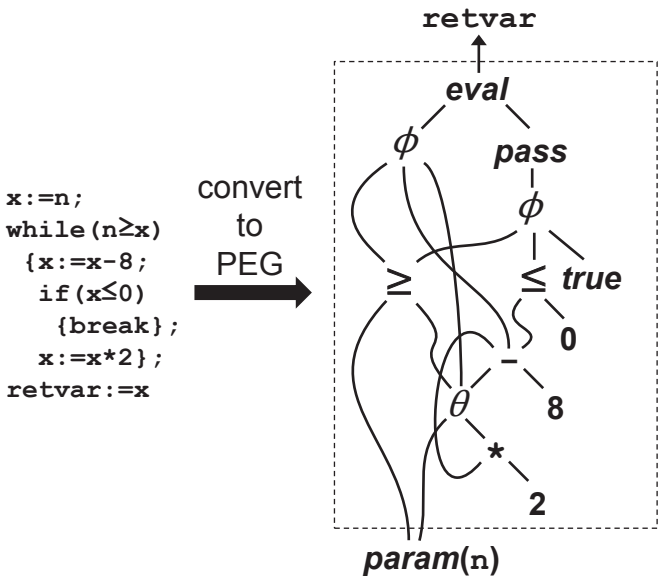

Figure 26: Conversion of a SIMPLE program with a `break` to a PEG

state of `x` at that point (which is the state of `x` at the beginning of the loop iteration minus 8), otherwise the loop must have ended due to the loop guard so it can simply return the state of `x` at the beginning of the loop iteration. In order to untangle this PEG and revert it to an efficient CFG, we must take advantage of the fact that the result of the loop and the break condition of the loop are both ϕ nodes with the same guard. This requires a more complex reversion process, but this process rewards us by producing more efficient CFGs even for simple loops. For example, this revised process would prevent the code duplication of `1+i` in the result in Figure 25. The revised process is accomplished by making two major changes: introducing *guarded* PEG contexts which are eventually converted to branching CFGs, and changing loop nodes to use these guarded PEG contexts.

Before when we replaced an *eval* node with a loop node, we would create three new PEG contexts: one for the condition, one for the body, and one for the final result. After loop fusion, we would convert each of these to statements and then use the results to construct a while-loop statement node. Now, instead of creating three PEG contexts, we will construct a single PEG context to describe the loop. This single PEG context updates the values of the loop variables if the break condition fails, and it updates the value of the result variable if the break condition passes. This PEG context also has a distinguished node labeled as the guard, which in this case is the break condition of the loop. We replace the *eval* node with a revised loop node which instead stores this guarded PEG context (although it still also remembers the associated pass node). The top-left diagram in Figure 27 is the PEG from Figure 26. Figure 27 shows how the loop is extracted into a guarded PEG context (below) and the *eval* node is replaced with the revised loop node containing that guarded PEG context (to the right). The process for constructing the guarded PEG context is described later.

We can still apply loop fusion to these revised loop nodes as we did before by taking the union of the guarded PEG contexts (the distinguished guard node will be the same in the

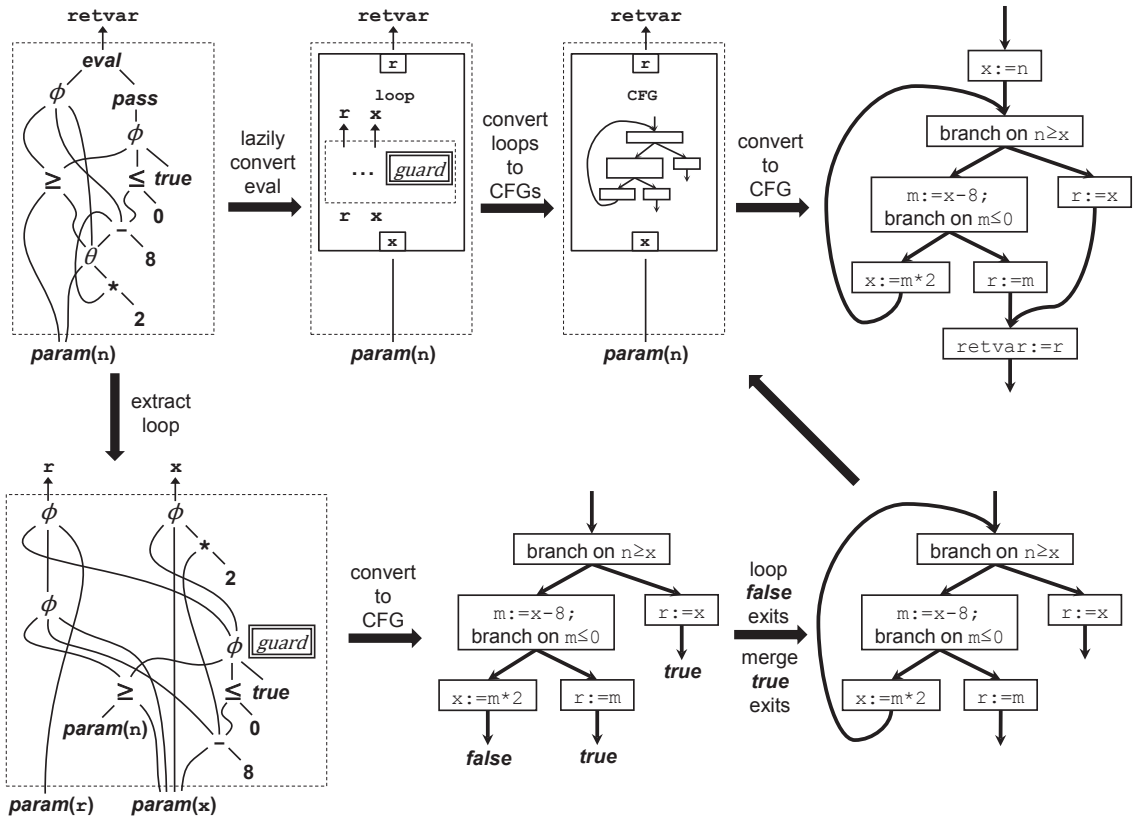

Figure 27: Reversion of a PEG to a CFG

PEG contexts of both loop nodes since they pertain to the same *pass* node). Afterwards, when it comes time to sequentialize the PEG, we need to first turn this loop node into a CFG node (instead of a statement node). A CFG node contains a CFG with a single entry and multiple exits. When sequentializing the PEG, all of these exits are routed to whatever CFG block sequentialization determines is next after the CFG node; thus a CFG node is simply a generalization of a statement node.

In order to convert a loop node to a CFG node, we first convert the contained guarded PEG context into a special single-entry CFG. This process is described later, but Figure 27 shows the CFG resulting from our example guarded PEG context. Each exit in this CFG is marked as either true or false, indicating whether that edge is taken if the distinguished guard node is true or false. Since the distinguished guard node of our guarded PEG context was the break condition of the loop, the exits marked as true are taken when the loop should end, and the exits marked as false are taken when the loop should continue. So, we can turn this CFG into the desired loop by routing all the exits marked as false back to the head of the loop, as exemplified in Figure 27. We then replace the loop node with the CFG node containing the resulting CFG, as shown in Figure 27.

In the steps described above we left two processes unexplained. The first such process is how to construct the guarded PEG context. We assign a fresh variable x to each relevant θ node as before. Also like before, we construct nodes c, n_x, and r representing the loop's break condition, loop-variable update, and desired result respectively in terms

of these fresh variables. Then, we construct a PEG context mapping each θ variable x to $\overline{\phi}(c, \overline{param}(x), n_x)$ and mapping x_r (fresh) to $\overline{\phi}(c, r, \overline{param}(x_r))$. The assignment for the θ variable x means that, if the loop's break condition holds then the loop is ending immediately so there is no need to update the loop variable, otherwise the loop is iterating one more time so we should update the loop variable to its next value. The guarded PEG context in Figure 27 says that loop variable `x` should not change if the loop's break condition holds, and should be updated to $(\texttt{x} - 8) * 2$ otherwise. The assignment for x_r means that, if the loop's break condition holds then the loop is ending so we should finally calculate the desired value, and otherwise we should do nothing. The use of $\overline{param}(x_r)$ is a slight trick, relying on the fact that no one actually uses this variable until after the loop terminates, by which point it would have been assigned the value of r. The guarded PEG context in Figure 27 says that the "return" variable `r` should not be changed if the loop's break condition is false, should be $\texttt{x} - 8$ if otherwise the loop's guard holds (indicating that the loop exited through the `break` statement), and should be just `x` otherwise (because the loop exited from the beginning). c is the distinguished guard node so that edges marked true or false after translating this guarded PEG context to a CFG (described next) correspond to edges exiting the loop or continuing the loop respectively. Figure 27 shows the loop's break condition labeled as the guard.

Lastly, we describe how a guarded PEG context can be reverted to a single-entry CFG with multiple exits marked as either true or false. We step through the reversion of the guarded PEG context constructed in Figure 27. This reversion is shown in Figure 28. To make the explanation simpler, let us assume that during the entire reversion process the following simplifying replacements are applied automatically should the opportunity arise: a ϕ node whose first child is $\overline{true}$ is replaced with its second child; a ϕ node whose first child is $\overline{false}$ is replaced with its third child; a node of the form $\overline{\phi}(\overline{\neg}(c), t, f)$ is replaced with $\overline{\phi}(c, f, t)$. How we revert a guarded PEG context depends on the distinguished guard node, so we describe the various cases in turn as we step through the reversion process.

If the distinguished guard node is neither true, false, nor a negation, then we must determine what condition we want to branch on first. This condition may be the distinguished guard node, but if the guard node is a ϕ then the guard node itself depends on another condition, so it would be more efficient to branch on that condition first. To determine the appropriate branch node C, first initialize C as the guard node. While the current value of C is itself a ϕ node, update the value of C to the first child of that ϕ node. Afterwards, C is the node for the first branch condition that must be evaluated. In the top diagram of Figure 28, C will be $\texttt{n} \geq \texttt{x}$. After determining the node C to branch on, we construct a guarded PEG context Ψ_t like the current one except with all uses of C replaced with $\overline{true}$ (and simplifying ϕs automatically). Similarly we construct Ψ_f by replacing C with $\overline{false}$ (and simplifying). We recursively convert each of these guarded PEG contexts to CFGs, then connect them with a CFG block which branches to the first if C is true and to the second if C is false. In our example, the branch for the first stage is $\texttt{n} \geq \texttt{x}$ and the resulting guarded PEG contexts Ψ_t and Ψ_f are shown underneath in Figure 28.

In the second stage (reading left to right) we can apply the same process as above, branching on $\texttt{x-8} \leq \texttt{0}$ this time. However, we have an opportunity here to prevent some code duplication. In particular, the node `x-8` will be evaluated regardless of the branch condition, so we can improve by assigning its value to a variable before branching and then having each branch use that variable instead of the original node. Thus after determining the branch condition but before creating Ψ_t and Ψ_f, we determine the set of nodes which

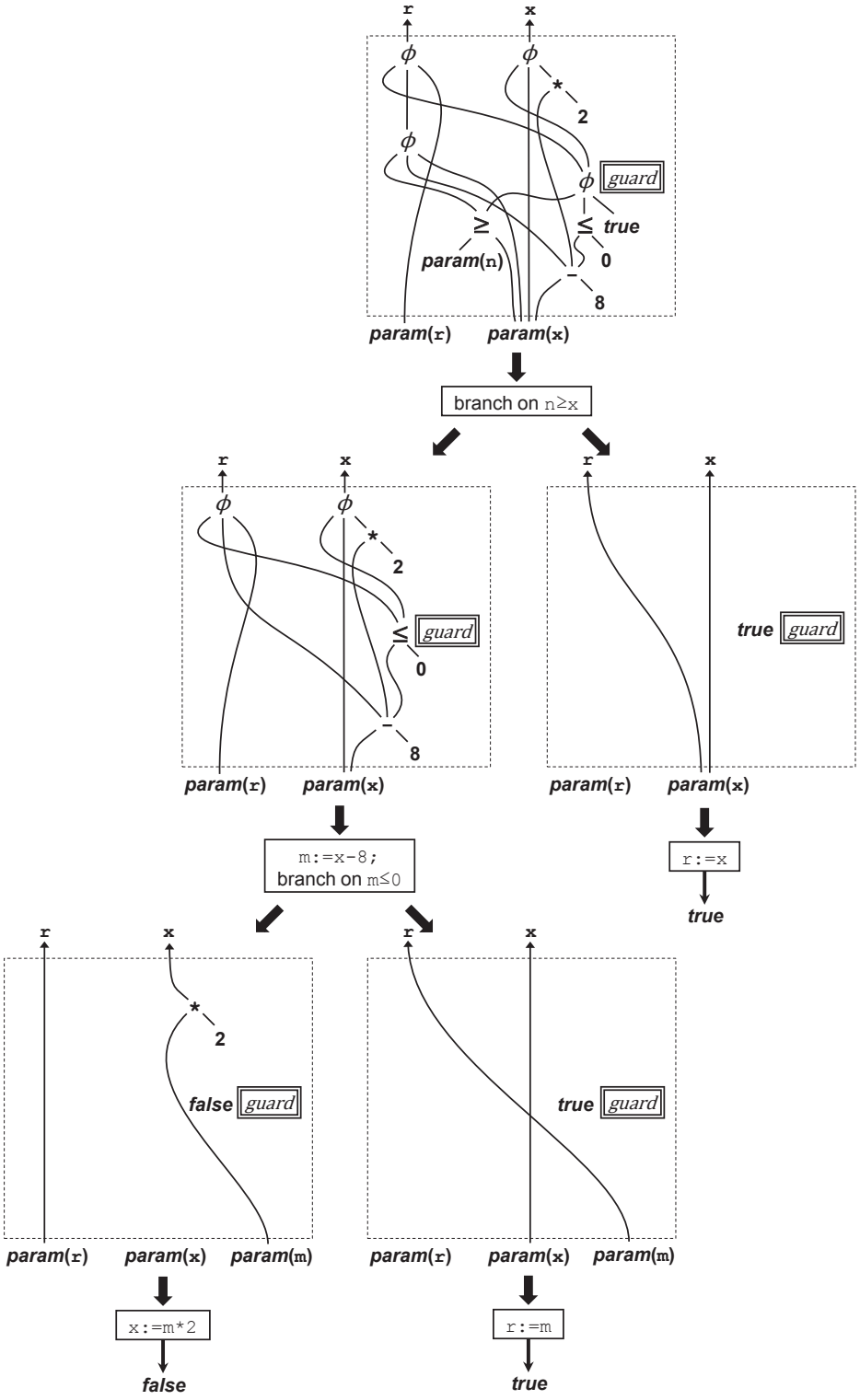

Figure 28: Reversion of a guarded PEG context to a CFG with exits marked as true or false

will always be evaluated regardless of the branch condition. Then, when creating Ψ_t and Ψ_f, each use of one of these nodes is replaced with a parameter node of a fresh variable, but making sure to reuse the same variables across Ψ_t and Ψ_f. We construct a context Ψ which maps each of these fresh variables to its appropriate node and also calculates the branch condition, and then convert Ψ to a CFG. Finally, we route all the exits of this CFG to the branch block, which branches to the CFGs for the revised Ψ_t and Ψ_f as before. In the example in Figure 28, Ψ simply assigns `x-8` to the fresh variable `m` and computes the branch condition as $\texttt{m} \leq \texttt{0}$, so we combined the resulting simple CFG together with the branch block.

The remaining stages in the example all have either a true or false as their distinguished guard node. In these cases, we simply convert the PEG context (without the distinguished guard node) to a CFG. Then we label all the exits of this CFG with either true or false depending on the guard node, as shown in Figure 28. The final result of the reversion of this example is the bottom-center diagram in Figure 27.

There is one remaining case we need to cover not illustrated in the example: when the distinguished guard node is of the form $\neg(c)$. In this case, we make a recursive call with c as the distinguished guard node, and then simply swap the true and false markings on the exits in the resulting CFG.

We have shown how the improved reversion process can be applied to loops with advanced control structure. The process for converting ϕ nodes can be modified in a similar way. When the conditions of different ϕ nodes are related (say due to short-circuiting), this can produce more efficient CFGs with more advanced control flow. There are more optimizations besides the ones we have presented above, but we feel that these are the most critical to producing efficient CFGs and the most challenging.

Acknowledgements

We would like to thank Jeanne Ferrante, Todd Millstein, Christopher Gautier, members of the UCSD Programming Systems group, and the anonymous reviewers for giving us invaluable feedback on earlier drafts of this paper.

References

[1] A. Appel and J. Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, 2002.
[2] Daan Leijen. A type directed translation of MLF to System F. In *The International Conference on Functional Programming (ICFP'07)*. ACM Press, October 2007.
[3] S. Muchnick. *Advanced Compiler Design And Implementation*. Morgan Kaufmann Publishers, 1997.
[4] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. In *POPL*, January 2009.
[5] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. In *LMCS*, December 2010.