



# Beyond Relooper: Recursive Translation of Unstructured Control Flow to Structured Control Flow (Functional Pearl)

NORMAN RAMSEY, Tweag, France and Tufts University, USA

In many compilers, control flow is represented using an arbitrary directed graph. But in some interesting target languages, including JavaScript and WebAssembly, intraprocedural control flow can be expressed only in structured ways, using loops, conditionals, and multilevel breaks or exits. As was shown by Peterson, Kasami, and Tokura in 1973, such structured control flow can be obtained by translating arbitrary control flow. The translation uses two standard analyses, but as published, it takes three passes—which may explain why it was overlooked by Emscripten, a popular compiler from C to JavaScript. By tweaking the analyses and by applying fundamental ideas from functional programming (recursive functions and immutable abstract-syntax trees), the translation, along with a couple of code improvements, can be implemented in a single pass. This new implementation is slated to be added to the Glasgow Haskell Compiler. Its single-pass translation, its immutable representation, and its use of dominator trees make it much easier to reason about than the original translation.

CCS Concepts: • **Software and its engineering** → **Compilers; Functional languages.**

Additional Key Words and Phrases: WebAssembly, control-flow analysis, dominator tree, reverse postorder numbering, Haskell

## ACM Reference Format:

Norman Ramsey. 2022. Beyond Relooper: Recursive Translation of Unstructured Control Flow to Structured Control Flow (Functional Pearl). *Proc. ACM Program. Lang.* 6, ICFP, Article 90 (August 2022), 22 pages. <https://doi.org/10.1145/3547621>

## 1 INTRODUCTION

Compilers don't just generate machine code any more. Many compilers generate JavaScript, and some generate WebAssembly, a relatively new low-level intermediate code that was originally motivated by a desire to make in-browser code faster than JavaScript [Haas et al. 2017]. JavaScript and WebAssembly don't look like traditional target machines; JavaScript is a full-blown, general-purpose programming language with structured control flow, high-level data structures, and operations that check types at run time. WebAssembly also has structured control flow, but it looks more machine-like: its data is machine-level data, and its computations operate only on integers and floating-point numbers of 32 and 64 bits, with no dynamic type checks.

Because of their structured control flow, neither JavaScript nor WebAssembly can express a goto instruction. But goto is a valuable tool. In many compilers, goto and its variants provide the only representation of control flow in low-level intermediate code. Such a representation is conveniently close to machine code, and it also supports a panoply of optimizations, as shown by Davidson and Fraser [1984] and implemented in gcc, for example. For a compiler writer who wants to leverage these advantages, a lack of goto presents a problem.

---

Author's address: Norman Ramsey, Tweag, Paris, France and Tufts University, Medford, Massachusetts, USA.

---



This work is licensed under a Creative Commons Attribution-NoDerivatives 4.0 International License.

© 2022 Copyright held by the owner/author(s).

2475-1421/2022/8-ART90

<https://doi.org/10.1145/3547621>

Fortunately for compiler writers, `goto` statements in low-level intermediate code can be translated into structured control flow using a translation developed by [Peterson, Kasami, and Tokura \[1973\]](#). Their translation preserves semantics, and unlike the heuristic Relooper algorithm used in Emscripten [[Zakai 2011](#)], it introduces additional code or additional computation only when necessary. Given importance of JavaScript and WebAssembly as compiler targets, this translation deserves to be more widely known. But the translation is more complicated than it needs to be, and it works with representations that are too much of their time. The translation is set up by two analyses of path properties: dominance and all-pairs, loop-free reachability. It represents target code as a sequence of statements in a hybrid language that includes both structured control flow *and* `goto`. And it translates a control-flow graph in three passes: mutate the source code, translate it to the target language, and mutate the translated code. This three-pass translation works, but it's not easy to understand why—and implementing it can be a challenge.

These are problems that functional programming can solve. The contribution of this paper is to reimplement [Peterson, Kasami, and Tokura](#)'s translation in a single pass (after the initial two analyses), using classic ideas of functional programming: recursive functions operating on immutable representations of programs. The reimplementation uses no fancy language features: only recursion and abstract-syntax trees are necessary. (First-class functions are also used, but only as a convenience.) The recursive translation follows the inductive structure of a dominator tree—a concept not mentioned in the original work. This structure, and the fact that the translation is implemented in one pass, make the recursive version much easier to reason about than the original (section 6). The key parts of the code are shown in section 5, which should make the details clear and the technique easy to adapt to many compilers.

## 2 A CASE STUDY: THE TRANSLATION PROBLEM IN GHC

The work described in this paper should apply to any compiler that uses `goto` in its intermediate code. But the work has been motivated by a desire to generate WebAssembly in the Glasgow Haskell Compiler (GHC).

WebAssembly is an intriguing platform. Like the Java Virtual Machine, WebAssembly offers a write-once, run-anywhere, portable, secure bytecode. Compared with the JVM, however, WebAssembly promises faster startup times, and it does not need a heavyweight managed run-time system, a sophisticated compiler, or a beefy CPU. In addition to browsers and servers, WebAssembly code can run on edge-computing platforms and even low-energy embedded devices. At the time of writing, WebAssembly is targeted by maybe a couple dozen compilers for high-level languages. But its ecosystem is evolving rapidly, and it looks poised to take off.

GHC is the premier compiler for Haskell; it currently translates Haskell into LLVM code, native code for several machine platforms, and its own bytecode. GHC transforms Haskell source first into Haskell Core [[Sulzmann et al. 2007](#)], then into spineless, tagless G-code [[Peyton Jones 1992](#)], then into Cmm, and finally into machine code. Translated machine code is supported by a run-time system that provides memory management, exception handling, and other services.

Both Haskell Core and spineless, tagless G-code are functional languages, so why not generate WebAssembly from one of these forms? Because compiled Haskell code needs a run-time system. To support copying garbage collection, exceptions, and other features, GHC's run-time system must be able to scan a stack to find roots, move a stack safely, and unwind a stack when an exception occurs—as well as perform other operations. So the compiler must emit code that represents and manipulates stacks in the way that the run-time system expects. In particular, compiled code must correctly allocate frames, store live pointers in the frames where the collector can find them, keep the stack pointer where the run-time system expects, and so on. Meeting all these obligations is a ton of work, and there are ample opportunities to get things wrong. Fortunately, GHC's translation

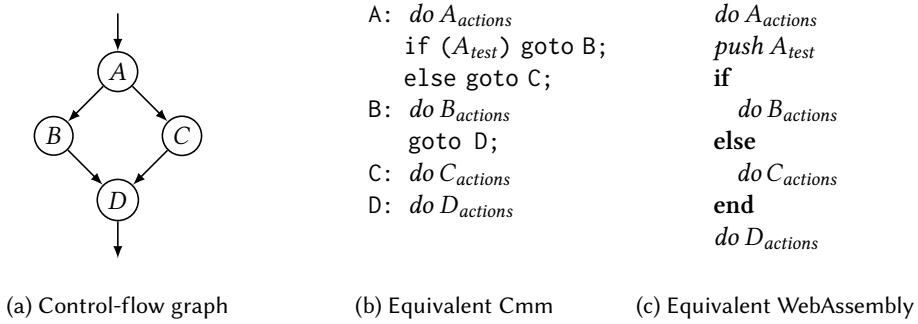


Fig. 1. Simple conditional

into Cmm takes care of everything—which makes Cmm the form from which we should generate WebAssembly.

Cmm is an imperative language based on C-- [Peyton Jones, Ramsey, and Reig 1999]. In Cmm, each function is represented as an immutable control-flow graph [Ramsey and Dias 2005] in which control flow is represented as `goto`, conditional `goto`, or computed `goto`. Each Cmm control-flow graph must be translated into a WebAssembly function, and the translation must reconcile the control structures of the two languages. Cmm’s primary control structures are `goto` and conditional `goto`; WebAssembly’s control structures include `if`, `loop`, and a multilevel exit-or-continue primitive called `br` (branch). Cmm and WebAssembly are otherwise similar: both languages are imperative, and within a function, imperative actions may have side effects on shared global state, local variables, and in the case of WebAssembly, an evaluation stack.

Example Cmm control-flow graphs and WebAssembly equivalents are shown in Figures 1 and 2. In each figure, a source control-flow graph is shown on the left. Nodes are labeled A, B, C, and so on. Every node contains imperative *actions*; for example, A’s actions are written  $A_{actions}$ . And if a node has two successors, it ends in a conditional test, which is written  $A_{test}$ . The control-flow graph’s internal representation is what GHC must translate, but for comparison purposes, equivalent Cmm code is shown in the middle of each figure. WebAssembly is shown on the right.

The control-flow graph in Figure 1 has the classic diamond shape of a conditional. It is translated using WebAssembly’s `if` form; actions from node A are placed before the `if`, actions from nodes B and C are placed inside the `if`, and actions from the “merge node” D are placed after the `if`. The control-flow graph in Figure 2 is a loop with two nodes and one exit. It doesn’t correspond to a traditional `while` loop, but `while` loops aren’t part of WebAssembly anyway; WebAssembly has only `loop` . . . `end`, which loops forever. WebAssembly’s `loop` may be exited or continued using a `br` instruction, and it may be exited using `return`. In Figure 2c, the 1 in `br 1` causes WebAssembly to ignore 1 level of nesting (the `if`) and to transfer control to the next enclosing level, which is `loop`. Transfer to a `loop` always means “continue,” as shown in the comment next to `br 1`.

With these examples for context, the control constructs in Cmm and WebAssembly can be characterized more precisely. Cmm’s control flow may be arbitrary. Cmm code is essentially a simplified form of C-- code [Ramsey, Peyton Jones, and Lindig 2005], represented as an applicative control-flow graph [Ramsey and Dias 2005; Ramsey, Dias, and Peyton Jones 2010]. A control-flow graph is a set of basic blocks, each of which ends in a control transfer. A basic block is not the same as WebAssembly’s `block` form, so to avoid confusion, in this paper Cmm’s basic blocks are called

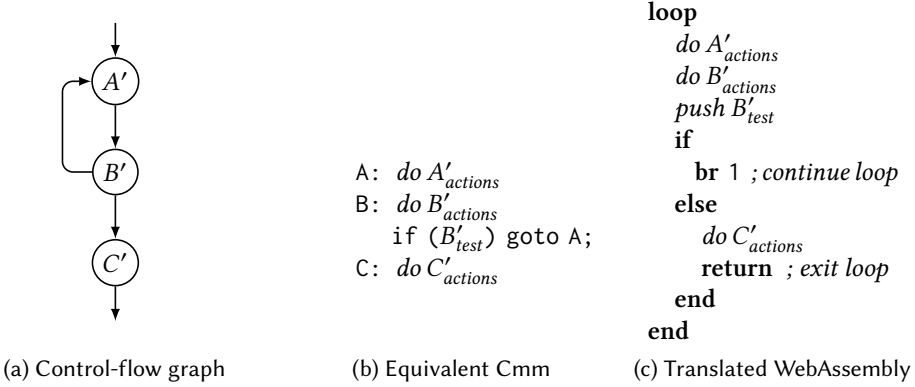


Fig. 2. Simple loop

*nodes*. Every node begins with a label, continues with a sequence of actions, and ends in one of these control transfers:

- Unconditionally transfer control to the node labeled  $L$ .
- Depending on the value of a Boolean test expression, conditionally transfer control either to the node labeled  $L_t$  or to the node labeled  $L_f$ .
- Return from the current function.
- End the current function by making a tail call.
- Switch based on the value of an expression. That is, compute an integer  $k$ , and transfer control to the corresponding label in a static collection of labels indexed by  $k$ . If  $k$  does not index any label, transfer control to a default label.

The form of the terminating control transfer influences the translation of a node from Cmm to WebAssembly.

WebAssembly’s control flow is structured control flow with one looping construct, some simple conditionals, **return**, and a multilevel break/continue statement (**br**). A function’s body is a sequence of *instructions*  $e^*$ , where the grammar of instructions  $e$  includes the following syntactic forms [Haas et al. 2017]:<sup>1</sup>

- A loop **loop**  $e^*$  **end**. Instructions within the loop are repeated indefinitely until the loop is terminated by an instruction that exits it.
- A conditional **if**  $e^*$  **else**  $e^*$  **end**, where the condition is always “the value on top of the evaluation stack is nonzero.”
- A block **block**  $e^*$  **end**. Instructions within the block are executed once, unless the block is terminated early by an instruction that exits it.
- An exit instruction **br**  $i$ . The index  $i$  is a natural number that identifies an enclosing **loop**, **if**, or **block** form, and the **br** instruction either *continues* the identified form (if it is **loop**) or *exits* the identified form (if it is **if** or **block**).

These forms, plus some special cases of **br**, like **br\_if**, **br\_table**, and **return**, are the tools with which WebAssembly code determines control flow within a procedure.

Setting aside Cmm’s switch form, a translation from a Cmm control-flow graph to a WebAssembly function body has to handle two kinds of intraprocedural control transfers. A conditional transfer is translated into an **if** form. An unconditional control transfer is translated in one of two ways: either

<sup>1</sup>For simplicity, type annotations are omitted.

control “falls through” to the next instruction in a sequence of WebAssembly instructions, or the control transfer is translated as a **br**.

An ideal translation would have three properties:

- The translation preserves the semantics of the program.
- Dynamically, the translation introduces no additional computation. That is, when the original Cmm and the translated WebAssembly start execution in related states, they execute identical sequences of actions and tests.
- Statically, the translation introduces no additional code. That is, every action and test in the original Cmm has at most one counterpart in the translated code.

As shown by [Peterson, Kasami, and Tokura \[1973\]](#), an ideal translation exists when the original control-flow graph is *reducible*.

### 3 MOMENTARY DIGRESSION: REDUCIBILITY

The central algorithm of [Peterson, Kasami, and Tokura](#), which this paper reimplements using ideas from functional programming, works only on *reducible* flow graphs. Reducibility is equivalent to a whole bunch of lovely properties; for this paper, the most relevant one is that a control-flow graph is reducible if and only if every loop has a unique entry node.

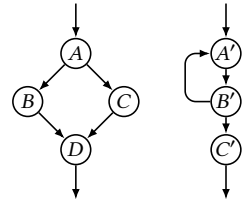
GHC typically produces reducible control-flow graphs. But if a programmer pursues efficiency through an aggressive combination of mutual recursion and unlifted types, GHC can produce an irreducible control-flow graph. In that case, before it performs the translation described in this paper, GHC transforms the irreducible flow graph into an equivalent reducible flow graph using *node splitting* [[Aho et al. 2007](#), section 9.7.6]. The equivalent graph does no more dynamic computation than the original graph, but statically it contains more code. The node-splitting transformation is described in detail in appendix A.

### 4 KEYS TO A SOLUTION: DOMINANCE AND REVERSE POSTORDER NUMBERING

A reducible control-flow graph can be translated to structured control flow by a recursive function, which works top-down over the graph’s *dominator tree*. In a dominator tree, the children of each node are normally unordered; my function sorts the children using a *reverse postorder numbering*. Both dominator trees and reverse postorder numberings, which are used in many static analyses, are explained in this section, along with their influence on the recursive translator. The recursive translator itself is presented in the following section, and a third section explains why the translator works.

A dominator tree expresses a *dominance* relation between nodes of a control-flow graph. Dominance is defined as follows: node  $X$  *dominates* node  $Y$  if and only if  $X$  appears on every path from the entry point to  $Y$ . The set of nodes that dominate  $Y$  is  $Y$ ’s *dominator set*.

To show examples, the control-flow graphs from Figures 1 and 2 are duplicated on the right. In the first control-flow graph, node  $D$  is reachable by two paths:  $ABD$  and  $ACD$ . Nodes  $A$  and  $D$  appear on both paths, so node  $D$  is dominated by itself and by node  $A$ . Although node  $D$  is a successor of node  $B$  and of node  $C$ , neither  $B$  nor  $C$  dominates  $D$ . Node  $B$  is reachable only by path  $AB$ , so it is dominated by  $A$  and by itself. Node  $C$  is also dominated by  $A$  and by itself. In the second graph, node  $B'$  is reachable by the infinite set of paths  $(A'B')^*A'B'$ , and it is dominated by  $A'$  and by itself. And node  $C'$  is reachable by paths  $(A'B')^*A'B'C'$ ; it is dominated by  $A'$  and  $B'$  and by itself.



The dominator relation of any control-flow graph has these properties:

- Every node  $Y$  dominates itself.<sup>2</sup>
- A node may dominate more than one other node, and it may be dominated by more than one other node.
- If both  $W$  and  $X$  dominate  $Y$ , then either  $W$  dominates  $X$  or  $X$  dominates  $W$ .
- If  $Y$  is the entry point, its dominator set is  $\{Y\}$ .
- If  $Y$  is not the entry point, its dominator set contains at least one other node besides itself. Moreover,  $Y$ 's dominator set contains a unique node different from  $Y$  that is dominated by all the other nodes in the set, excluding  $Y$ . That node is  $Y$ 's *immediate dominator*.

In the control-flow graph from Figure 1, node  $D$  is dominated only by  $A$  (and itself), so  $D$ 's immediate dominator is  $A$ . Node  $B$ 's immediate dominator is also  $A$ , and likewise for node  $C$ . In the graph from Figure 2, node  $C'$  is dominated by both  $A'$  and  $B'$ . Node  $B'$  is itself dominated by  $A'$ , which means that  $C'$ 's immediate dominator is  $B'$ . In any control-flow graph, immediate dominators define a structure in which each node of a control-flow graph points to its immediate dominator. That structure is always a tree rooted at the entry point: the *dominator tree*. Dominator trees for the graphs from Figures 1 and 2 are shown in Figure 3 (on the right).

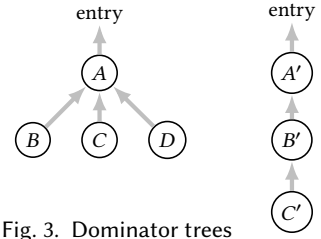
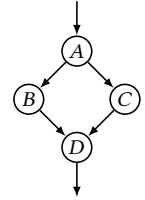


Fig. 3. Dominator trees

The dominator tree drives my recursive translator, which works from the root downward. The key function, `doTree`, is given a subtree rooted at a node  $X$ ; I write the call as “`doTree X`.” Calling `doTree X` returns the translation of  $X$  and of all of the nodes that  $X$  dominates (the nodes in the subtree). When translating  $X$ , `doTree` recursively translates the subtrees rooted at  $X$ 's children—the nodes that  $X$  immediately dominates. Then `doTree` places the subtrees' translations in locations that depend on the form of  $X$ .

A control-flow graph is translated by passing its complete dominator tree to `doTree`. For example, when the control-flow graph from Figure 1 is translated (the graph is repeated on the right), the translator calls `doTree A`. Function `doTree` then calls itself to compute the translations of  $A$ 's children: nodes  $B$ ,  $C$ , and  $D$ . Node  $A$  is a conditional and so is translated into an `if` form. Nodes  $B$  and  $C$  are reachable *only* from  $A$ , and their translations are placed into the true and false branches of the `if`, respectively. But node  $D$ , which is also immediately dominated by node  $A$ , is reachable from more than one predecessor—it is a *merge node*. So its translation is placed *after* the `if`. The result is the WebAssembly code shown in Figure 1c.



When a node immediately dominates more than one merge node, `doTree` has to work harder. A contrived example appears in Figure 4. On the left, the control flow graph has three conditionals ( $A$ ,  $B$ , and  $D$ ) and two merge nodes ( $E$  and  $F$ ). The contrivance resides in node  $E$ : node  $E$  looks like it should go in the false branch of node  $B$ , but it also looks like it should go in the true branch of node  $D$ . Since  $E$ 's code should not be duplicated, `doTree` needs to put  $E$ 's translation somewhere else. To see where, `doTree` revisits the dominator tree (Figure 4b).

When `doTree A` is called, `doTree` calls itself recursively to get the translations of the subtrees rooted at nodes  $B$ ,  $D$ ,  $E$ , and  $F$ . Node  $A$  ends in a conditional and so is translated into an `if` form,

<sup>2</sup>This information is rarely useful, and in an implementation,  $Y$  may not even be represented in its own dominator set. But defining  $Y$  to dominate itself leads to beautiful recursion equations: the dominator set of node  $Y$  is  $\{Y\}$  plus the intersection of the dominator sets of  $Y$ 's predecessors.



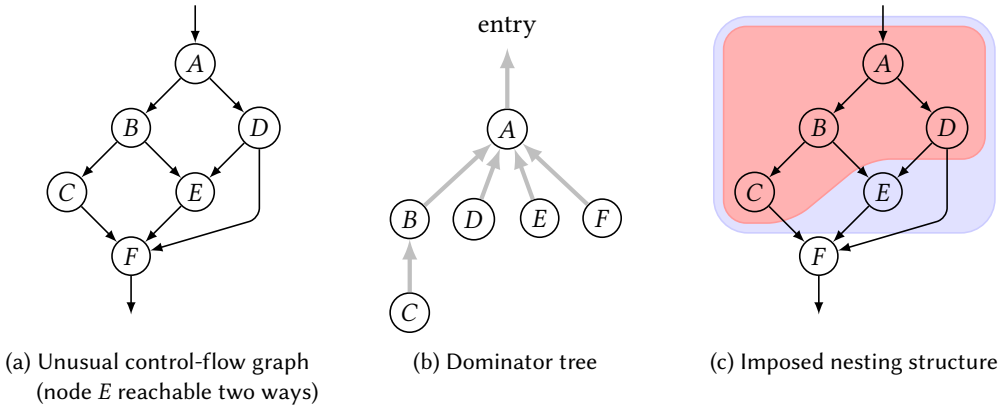


Fig. 4. Imposing nesting structure for merge nodes

and the translations of nodes *B* and *D* are placed in the true and false branches, respectively. Nodes *E* and *F* are merge nodes, so their translations are placed after the translation of node *A*. These translations must be placed in such a way as to be reachable by **br** instructions from the translations of nodes *B*, *C*, and *D*. To get an idea how that works, let's look at the recursive translation of the subtree rooted at *B*, which must be able to transfer control to *E*.

When `doTree A` calls `doTree B` recursively, node *B* is also a conditional and is translated into an **if**—and the translation of *B*'s child *C* can be placed into the true branch of *B*'s **if** form. What about *E*? Looking at the control-flow edges leaving *B*, it appears as if the translation of *E* should go in the false branch of *B*'s **if** form. But `doTree` doesn't look at control-flow edges; it looks at the dominator tree. Since node *E* isn't a child of node *B*, `doTree B` isn't responsible for *E*'s translation. Instead, `doTree B` fills the false branch with a **br** instruction. The **br** instruction exits a **block**  $\cdots$  **end** form, so when `doTree A` places *E*'s translation, it must ensure that *E*'s translation follows a **block**  $\cdots$  **end** form.

Calling `doTree A` places the translations of *E* and *F* as shown in Figure 5 (on the right). The translation of *E* is preceded by the translations of nodes *A*, *B*, *C*, and *D*, which are wrapped in **block**  $\cdots$  **end**. The translations of *C* and *D* include the **br** instructions that are needed to reach node *F*. The translation of *F* is preceded by the translations of nodes *A* through *E*, which are wrapped in another **block**  $\cdots$  **end**. The blocks nest as visualized in Figure 4c, in which each block is represented by a colored region. Given this nesting structure, any preceding translation can reach *E* or *F* using a **br** instruction.

How does `doTree A` determine that the translation of *E* goes on the inside? Nodes *E* and *F* are both immediately dominated by *A*, and both are merge nodes. But these properties don't suffice to determine which translation goes inside the other. And order matters! In the control-flow graph (Figure 4a), *E* branches to *F*, so *E*'s translation has to go on the inside. But what if neither node branches directly to the other? How does `doTree` place their translations?

Peterson, Kasami, and Tokura [1973] place translations by using a complicated predicate involving the existence of paths in a mutated control-flow graph. But there is a simpler way to do it. Translations can be placed using a purely local test on the original, unmutated control-flow graph: compare the *reverse postorder numbers* of the nodes being translated.

**block** ; blue block  
**block** ; red block  
*Translations of*  
*A, B, C, and D*  
*(with brs)*  
**end**  
*Translation of E*  
**end**  
*Translation of F*

Fig. 5. Schematic translation of Figure 4a

A reverse postorder numbering embodies a total order on nodes that respects our notions of what nodes “come first.” (For example, if a graph contains no loops, a reverse postorder numbering orders its nodes topologically.) Reverse postorder, like dominance, shows up in many static analyses; for example, iterative dataflow analyses converge most quickly when nodes are visited in reverse postorder (for a forward analysis) or postorder (for a backward analysis). A reverse postorder numbering is computed by a postorder traversal (depth-first search from the entry point) of the control-flow graph: after each node is visited, it is assigned a number in sequence from  $n$  down to 1, assuming the graph has  $n$  nodes. As examples, a reverse postorder numbering has been used to label the control-flow graphs in Figures 1, 2, and 4, assuming that  $A = 1$ ,  $B = 2$ , and so on.

Unlike a dominator tree, a reverse postorder numbering is not unique: whenever a node has multiple successors, the reverse postorder numbering depends on the order in which those successors are visited. But in every reverse postorder numbering, if there are two nodes  $X$  and  $Y$  and there is a path from  $X$  to  $Y$  that does *not* pass through the beginning of a loop, then  $Y$  has a larger reverse postorder number than  $X$ . Also, if  $X$  dominates  $Y$ , then every path to  $Y$  goes through  $X$ , so no matter what reverse postorder numbering is chosen,  $X$  has a smaller reverse postorder number.

A reverse postorder numbering tells doTree how to nest merge nodes: nodes with smaller reverse postorder numbers go on the inside. For example, in Figure 4,  $E$  has a smaller reverse postorder number than  $F$ , so  $E$ ’s block goes inside of  $F$ ’s block.

A reverse postorder numbering can be used not only to nest blocks but also to identify loops:

- Because each node has a distinct number, every loop (cycle) in the control-flow graph must contain an edge that goes from a higher-numbered node to a lower-numbered node. Such an edge is called a *back edge*. (An edge from a node to itself is also a back edge.) A single node may be the target of multiple back edges and thereby be involved in multiple loops. In Figures 1, 2, and 4, the only back edge is the edge from  $B$  to  $A$  in Figure 2.
- If and only if the original control-flow graph is reducible, every loop has a unique entry point, called the *loop header*. For example, in Figure 2 node  $A$  is a loop header. A loop can be entered only via its header, and so the header dominates every node in the loop. And any depth-first search must reach the header before it can reach any other node in the loop, so the header has the lowest reverse postorder number of any node in the loop—and it is the target of a back edge.

A control-flow graph is reducible if and only if the target of every back edge dominates the edge’s source, that is, if every loop has a header.

Together, a reverse postorder numbering and dominator tree provide all the information that doTree needs in order to place its translations correctly.

Loop headers and back edges can also help explain why nesting works. When node  $X$  immediately dominates both  $Y_1$  and  $Y_2$ ,  $Y_1$  is placed inside the block that  $Y_2$  follows. So if  $Y_1$  needs to branch to  $Y_2$ , or more generally if a node dominated by  $Y_1$  needs to branch to  $Y_2$ , it can use **br** to exit the block that  $Y_2$  follows. But what if  $Y_2$  (or a node it dominates) needs a path to  $Y_1$ ? Because  $Y_2$  appears later in the reverse postorder numbering, any path from  $Y_2$  to  $Y_1$  must pass through a back edge to a loop header. Because the control-flow graph is reducible, the header dominates every node in its body, including  $Y_2$ . Therefore  $Y_2$ ’s translation appears inside the **loop** form, and it can get to the header via a **br** instruction.

The nesting trick works only with reducible control-flow graphs. When a control-flow graph is irreducible, it resembles the graph in Figure 6 (on the right). Nodes  $Y$  and  $Z$  form a loop with no header; the loop can be entered either at  $Y$  or at  $Z$ . There is no way to nest translations of  $Y$  and  $Z$  such that each is reachable from the other. And the graph in Figure 6 isn’t just any example; it’s the canonical example: *every* irreducible flow graph contains

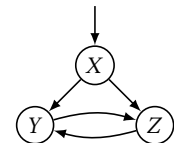


Fig. 6. Irreducible



a subgraph of that form [Hecht and Ullman 1972]. As noted above, GHC copes with an irreducible control-flow graph by first converting it to an equivalent reducible control-flow graph (appendix A). Only then does it invoke the translator described below.

A reverse postorder numbering not only tells doTree how to nest the blocks that precede merge nodes; it also helps doTree translate each individual node. The translation of a subtree rooted at node  $X$  depends on properties of the control-flow edges into and out of  $X$ , the forward or backward directions of which are determined by comparing reverse postorder numbers.

- A node  $X$  that has two or more forward inedges is a *merge node*. Being a merge node doesn't affect how  $X$  is translated, but it does affect the *placement* of  $X$ 's translation: the translation will follow a **block** form.
- A node  $X$  that ends in a conditional branch has two outedges and is translated into an **if** form. The translations of  $X$ 's children in the dominator tree are placed in one or the other branch of the **if** form, *except* for the translations of children that are rooted at merge nodes. Those translations are placed after **block** forms that enclose the **if**.
- A node  $X$  that has a *back* inedge is a loop header, and its translation is wrapped in a **loop** form. The translation of the subtree rooted at  $X$  is placed into the body of the **loop**.

These properties are independent; a given node may have any combination or none at all.

## 5 PURELY FUNCTIONAL, RECURSIVE IMPLEMENTATION

My translator's external interface is provided by function `structuredControl`, the key parts of which are shown in this section. This function translates *only* control flow; to translate an expression of the form  $X_{test}$  or actions of the form  $X_{actions}$ , it relies on other functions that are passed in. Those functions return WebAssembly code of type `WasmExpr` or `WasmActions`:

```
structuredControl
  :: (Label -> CmmExpr    -> WasmExpr)    -- translator for tests
  -> (Label -> CmmActions -> WasmActions) -- translator for actions
  -> CmmGraph                      -- CFG to be translated
  -> Wasm                          -- translation of the CFG
```

Most of these types relate to the source language of Cmm control-flow graphs. In Cmm, each node is named by a unique `Label`, like  $X$ , for example. A node labeled  $X$  is represented by a `CfgNode`, a conditional test  $X_{test}$  is represented by a `CmmExpr`, and a sequence of actions  $X_{actions}$  is represented by `CmmActions`.

Type `Wasm` includes all the WebAssembly instructions that affect intraprocedural control flow. Type `WasmExpr` represents WebAssembly code that pushes a value onto the evaluation stack but does not affect intraprocedural control flow. And type `WasmActions` represents WebAssembly code that may mutate the contents of memory or the values of variables but does not affect intraprocedural control flow or the evaluation stack.

A node's parts are extracted by these functions:

```
entryLabel  :: CfgNode -> Label
nodeBody    :: CfgNode -> CmmActions
flowLeaving :: CfgNode -> CmmControlFlow -- successors, possible test
```

A value of type `CmmControlFlow` describes how a node ends and what its successors are. Values of this type can represent GHC's unconditional and conditional forms, which have one and two

successors, respectively. They can also represent returns and tail calls, which have no successors. The description of the conditional form includes the condition, which has type `CmmExpr`.

```
data CmmControlFlow
  = Unconditional Label          -- go to another node
  | Conditional CmmExpr Label Label -- choose next node based on condition
  | TerminalFlow                -- leave function (return or tail call)
```

The `CmmControlFlow` type can also describe flow leaving a `Switch` form—the form that GHC uses to implement case expressions. But because `Switch` is messy, I’ve kept the details out of this paper.

The target language is described by the `Wasm` type. This type’s value constructors include WebAssembly’s control-flow constructs, plus constructor `WasmActions`, which embeds the translation of actions (a basic block).

```
data Wasm where
  WasmBlock    :: Wasm -> Wasm
  WasmLoop     :: Wasm -> Wasm
  WasmIf       :: WasmExpr -> Wasm -> Wasm -> Wasm

  WasmBr       :: Int -> Wasm
  WasmReturn   :: Wasm

  WasmActions  :: WasmActions -> Wasm
  WasmSeq      :: Wasm -> Wasm -> Wasm
```

One other type, `Context`, is used only in the translator. It describes the syntactic context into which WebAssembly code is placed. That context determines the behavior of `br` instructions. For example, in Figure 2 (page 4), at the end of node *B*, the translation of `goto A` is placed in the context `loop ... if ... • ... end ... end`, where `•` represents a “hole” into which the translation is placed. In this context, `goto A` must be translated into `br 1`, where the 1 tells the machine to ignore the `if` and continue with the `loop`. A syntactic context is represented by a value of type `Context`, which records the `br`-relevant syntactic forms that enclose the hole:

```
type Context = [ContainingSyntax] -- innermost form first
data ContainingSyntax
  = IfThenElse
  | LoopHeadedBy Label          -- label marks loop header
  | BlockFollowedBy Label       -- label marks code right after the block
```

During translation, every label that a node might need to branch to via `br` appears in the context, either as the header of a loop currently being translated or as an instruction that follows a block. For example, in Figure 4c (page 7), when nodes *A* through *D* are translated, labels *E* and *F* appear in the context as list elements `BlockFollowedBy E` (the red block) and `BlockFollowedBy F` (the blue block).

The context is initially empty and is extended by the `inside` function. For expository purposes, `inside` is just cons:

```
inside :: ContainingSyntax -> Context -> Context
inside = (:) 
```

The context is used by each of a group of three mutually recursive functions, which cooperate to implement `structuredControl`:

```
doTree      :: Tree.Tree CfgNode          -> Context -> Wasm
doBranch    :: Label -> Label             -> Context -> Wasm
nodeWithin  :: CfgNode -> [Tree.Tree CfgNode] -> Context -> Wasm
```

These functions operate as follows:

- Function `doTree` is called on a subtree of the dominator tree, rooted at node  $X$ ; `doTree` returns the translation of the subtree, which includes  $X$  and everything that  $X$  dominates. Function `doTree` first creates a syntactic template based on the properties of  $X$  from section 4, then fills the template with the translations of  $X$ 's children. These children are the nodes that  $X$  *immediately* dominates.
- Function `doBranch` is called on the labels of two nodes  $X$  and  $Y$ ; it returns code that, when placed after the translation of  $X$ , transfers control to the translation of  $Y$ . If  $X$  is  $Y$ 's only forward-edge predecessor, `doBranch` simply returns the translation of  $Y$  (and everything that  $Y$  dominates). Otherwise  $Y$ 's translation already appears in the context, and `doBranch` returns a `br` instruction.
- Function `nodeWithin` is an auxiliary function; it places the translation of a single node into a nest of blocks. Function `nodeWithin` is called by `doTree`  $X$ , which looks at  $X$ 's children in the dominator tree and computes  $Ys$ : the children of  $X$  that are merge nodes. Then `doTree` passes  $X$  and  $Ys$  to `nodeWithin`, which returns the translation of  $X$  (and its other children) nested inside one block for each element of  $Ys$ . The elements of  $Ys$  are ordered with higher reverse postorder numbers first, so in Figure 4, `doTree A` calls `nodeWithin A [F, E]`.

These three functions are worth presenting in detail, with numbered lines for easy reference.

To translate a subtree rooted at  $X$ , `doTree` sets  $X$  up for `nodeWithin` by selecting those children of  $X$  that are merge nodes. Each merge-node child is translated in an induction step of `nodeWithin`;  $X$  and its remaining children are translated in the base case. If  $X$  is a loop header, `doTree` wraps  $X$ 's translation in `loop`, and it adds the loop to the context passed to `nodeWithin`.

```
1 doTree (Tree.Node x children) context =
2   let codeForX = nodeWithin x (filter hasMergeRoot children)
3   in if isLoopHeader x then
4       WasmLoop (codeForX (LoopHeadedBy (entryLabel x) `inside` context))
5   else
6       codeForX context
7   where hasMergeRoot = isMergeNode . Tree.rootLabel
```

Auxiliary functions `isLoopHeader` and `isMergeNode` are predicates on nodes; they are implemented using the dominator tree and the reverse postorder numbering.

Function `nodeWithin` places the translations of  $X$  and of its merge-node children. Its induction step places  $X$ 's merge-node children when they take the form of a nonempty list  $(Y_n : Ys)$ . In this case, node  $Y_n$  must have a higher reverse postorder number than every node in  $Ys$ . Function `nodeWithin` creates a new block that is followed by  $Y_n$ , and because of  $Y_n$ 's high reverse postorder number, this block can safely enclose all the others.

```
8 nodeWithin x (y_n:ys) context =
9   WasmBlock (nodeWithin x ys (BlockFollowedBy ylabel `inside` context)) <>
10  doTree y_n context
11  where ylabel = entryLabel (Tree.rootLabel y_n)
```

The `<>` operator concatenates two fragments of WebAssembly.

The base case of `nodeWithin` translates node  $X$ . The translation ends in WebAssembly code returned by `doBranch`, or possibly in a `return`. The base case accounts for those children of  $X$  that are *not* merge nodes. Such children must be successors of  $X$ , and they are accounted for by calling `doBranch` on  $X$ 's successors. The successors are found by computing the control flow leaving  $X$ .

```

12 nodeWithin x [] context =
13   WasmActions (txBlock xlabel (nodeBody x)) <>    -- the actions of X
14   case flowLeaving x of
15     Unconditional l -> doBranch xlabel l context
16     Conditional e t f ->
17       WasmIf (txExpr xlabel e)                    -- X's final conditional test
18         (doBranch xlabel t (IfThenElse : context))
19         (doBranch xlabel f (IfThenElse : context))
20     TerminalFlow -> WasmReturn
21   where xlabel = entryLabel x

```

As suggested above, function `flowLeaving` examines the six different forms in which a `CfgNode` can end, determining what kind of control flow the form designates and what its successors are. The other functions used here have these types:

```

txBlock      :: Label -> CmmActions -> WasmActions
txExpr       :: Label -> CmmExpr   -> WasmExpr

```

Function `txBlock` is the second argument passed to `structuredControl`; it translates  $X_{actions}$ . And when  $X$  ends in a conditional, function `txExpr`, which is the first argument to `structuredControl`, translates condition  $X_{test}$ .

Control flow from  $X$  to its successor or successors is translated by `doBranch`, which is the simplest but most subtle of the three translation functions.

```

22 doBranch source target context
23   | isBackward source target = WasmBr i -- continue
24   | isMergeLabel      target = WasmBr i -- exit
25   | otherwise = doTree (subtreeAt target) context
26   where i = index target context    -- computed only on demand!

```

A backward branch goes to an enclosing loop header, and a forward branch to a merge node exits a **block**. Any other branch goes to a target node that immediately succeeds  $X$  and is reachable only from  $X$ , so the translation of the target is simply placed inline. The auxiliary functions have these types:

```

isBackward   :: Label -> Label -> Bool
isMergeLabel :: Label -> Bool
index        :: Label -> Context -> Int

```

A backward branch is identified by comparing reverse postorder numbers; because a self-edge counts as a back edge, a branch is backward if the number of the target is not greater than the number of the source. A merge node is any node that is entered by two or more forward control-flow edges; it is identified by testing its label for membership in a precomputed set.

Function `index` computes the index `i` that is passed to `WasmBr`, which is the number of enclosing syntactic forms that have to be skipped past before reaching the form associated with the target label.

```

27 index label (frame : context)
28   | matchesFrame label frame = 0
29   | otherwise = 1 + index label context
30 where matchesFrame label (BlockFollowedBy l) = label == l
31       matchesFrame label (LoopHeadedBy l)   = label == l
32       matchesFrame _ _ = False

```

The mutually recursive functions `doTree`, `nodeWithin`, and `doBranch` cooperate to translate every node in the dominator tree—which means every node in the control-flow graph. An entire control-flow graph is translated by translating its dominator tree in the empty context:

```

33 structuredControl txExpr txBlock g = doTree sortedDominatorTree []

```

## 6 WHY THE TRANSLATOR WORKS

The translator in section 5 has all three of the ideal properties described in section 2: it doesn't introduce additional computation, it doesn't duplicate any code, and it preserves semantics. The translator avoids duplicating code because it works by induction on the structure of the dominator tree of the original control-flow graph, in which every node of the graph appears exactly once. And at each node, it translates the actions (and test, if any) of that node exactly once.

To see how the translator preserves semantics, we need to consider the actions in the node bodies, the decisions that appear at the ends of conditional nodes, and the branches. The actions and the decisions are dealt with by inspecting the calls to `txBlock` and `txExpr`; an argument is presented by [Peterson, Kasami, and Tokura \[1973, Theorem 5\]](#). (The essence of their argument is that the translated actions and decisions appear in the right places according to the labels of the nodes.) The branches present more of a challenge.

The branches work because the `Context` parameter passed to each translation function accurately describes the syntactic context into which the translation is placed, including the position of every label mentioned in the context. (For example, in Figure 2, the branch from *B* to *A* is translated in the context `[IfThenElse, LoopHeadedBy A]`.) The accuracy of each context's description can be confirmed by an induction on the number of calls to the three translation functions; every inductive step reasons about one of the call sites where a value of type `Context` is passed. That is, every `Context` can be confirmed to be accurate by reasoning about numbered code lines 4, 6, 9, 10, 18, and 19. And in any context, a value returned from `index` codes for a `br` instruction that transfers control to the label named in the context. (For example, in Figure 2, `br 1` codes for a transfer to *A*, which is the loop header.) When function `index` returns a value, that value can be shown to code for the correct index by remembering the dynamic semantics of `br`, consulting the definition of `index`, and remembering the accuracy of the `Context` parameter. What's not obvious is why `index` should always return a value. Why, when a branch is translated, does the target label always appear in the context? That question can be answered by case analysis, starting with the direction of branches.

Given a reverse postorder numbering, every branch is either forward or backward. The backward case is simpler. Because the input control-flow graph is reducible, every loop has a unique entry point, and for every backward branch  $Y \rightarrow X$ , *Y* is dominated by *X*. (These three properties are equivalent.) And when `doTree` is called on the subtree rooted at *X*, the subtree includes *Y*. But because *X* is the target of a backward branch, it is a loop header, and code line 4 ensures that when `nodeWithin` is called, the loop headed by label *X* is in the context. So when branch  $Y \rightarrow X$  is translated, *X* is in the context. (An irreducible input graph could have two nodes *Y* and *Z*, each of

which branched to the other, and neither of which would be a loop header—so neither one would dominate the other. In that case no nesting could work.)

Now suppose the control-flow graph includes a forward branch  $Y_1 \rightarrow Y_2$ , where  $Y_2$  is a merge node. (Function `index` is called on a forward branch only when  $Y_2$  is a merge node.) Node  $Y_2$  has an immediate dominator, which we'll call  $X$ . Because  $Y_2$  is a merge node immediately dominated by  $X$ , when `doTree`  $X$  is evaluated  $Y_2$  is one of the nodes passed to `nodeWithin` on numbered code line 2. And so on code line 9,  $Y_2$  is added to a context.

Similarly, the placement of  $Y_1$ 's translation is determined by  $Y_1$ 's relation to  $X$  in the dominator tree. First, if  $Y_1$  is  $X$ , then line 9 ensures that when  $X$  is translated,  $Y_2$  appears in the context.

Next, if  $Y_1$  is different from  $X$ , then because  $X$  dominates  $Y_2$ , every path from the entry point to  $Y_2$  goes through  $X$ . In particular, every path that includes the branch  $Y_1 \rightarrow Y_2$  goes through  $X$ . Therefore  $X$  dominates  $Y_1$  as well. That means that  $Y_1$  descends from  $X$  in the dominator tree. Therefore there is a unique child of  $X$  that is an ancestor of  $Y_1$  in the dominator tree. (That ancestor might be  $Y_1$  itself.) The placement of  $Y_1$  depends on whether  $Y_1$ 's ancestor is a merge node.

- If  $Y_1$ 's ancestor is *not* a merge node, then the ancestor's translation is part of the translation returned by the base case of `nodeWithin` (code line 12), and the context parameter includes the labels of all the merge nodes that are immediately dominated by  $X$ , including  $Y_2$ .
- If  $Y_1$ 's ancestor is a merge node, then the ancestor, along with  $Y_2$ , is on the list of nodes passed as a second argument to `nodeWithin`  $X$  on code line 2. Because  $Y_1 \rightarrow Y_2$  is a *forward* edge,  $Y_2$  has a larger reverse postorder number than  $Y_1$ , and because the ancestor dominates  $Y_1$ , the ancestor's reverse postorder number is no greater than  $Y_1$ 's. (They are equal if and only if  $Y_1$  itself is the ancestor.) Therefore  $Y_2$  precedes  $Y_1$ 's ancestor in the list. Therefore, when  $Y_1$  is translated by `nodeWithin` on numbered code line 10, the context parameter already contains  $Y_2$ , which will have been put in the context by a previous call to `nodeWithin` on line 9.

Therefore, in every case in which `index` is called by `doBranch`, the target label appears in the context.

## 7 IMPLEMENTATION IN GHC

The translator described in this paper is part of GHC's WebAssembly back end, which is work in progress (snapshot at <https://zenodo.org/record/6727752>). Compared with the code in section 5, the translator used in GHC has been enhanced in three ways: it eliminates redundant `br` instructions, it does not wrap a `block` form directly around `if ... end`, and it handles Cmm's `Switch` form.

### 7.1 Redundant `br` Instructions

A `br` instruction is redundant if omitting it would not change the behavior of the program—that is, if simply “falling through” the point where the `br` appears would arrive at the target label. Redundant `br`'s are identified by enhancing the `Context` type to remember what label, if any, immediately follows the hole in the context. This “fallthrough label” is set in only two places: When the hole in the context is a loop body (numbered code line 4), it is set to the label of the loop header. And when the hole in the context is wrapped in a `block` that is followed by the translation of node  $Y_n$  (numbered code lines 9 and 10), it is set to  $Y_n$ . The fallthrough label is used by `doBranch` to optimize the code; when `doBranch` is given a target label that immediately follows the hole in its context, it omits the `br` instruction.



## 7.2 Unnecessary block Forms

GHC's second enhancement eliminates unnecessary **block** forms. An unnecessary block form may be introduced by `nodeWithin X Ys` when `Ys` is a singleton list. In section 5, `nodeWithin X [Y1]` always wraps `X`'s translation in `block ... end`, so that `Y1` can be reached by a **br** instruction. But when `X` is translated into an `if ... end` form, `Y1` can be reached by a **br** instruction that exits the `if`; no **block** is needed. As an example, the *Translations of A, B, C, and D* in Figure 5 stands for an `if ... end` form, so the `block ... end` form that wraps it (labeled "red block") is unnecessary. As another example, the code shown in Figure 1c is shown *after* unnecessary-block elimination; the code shown in section 5 would wrap that example's `if ... end` in a `block ... end`.

To know when a block is necessary, GHC's version of `nodeWithin` takes an additional parameter of type `Maybe Label`. When this parameter is `Just Z`, a block may be needed; `nodeWithin` must make `Z` reachable by returning a syntactic form that can be exited by a **br** instruction (a **block** or an `if`). In the inductive case, `nodeWithin` does not return an `if` form, so it makes `Z` reachable by wrapping a translation in `block ... end`.

```
nodeWithin x (y_n:ys) (Just zlabel) context =
  WasmBlock (nodeWithin x (y_n:ys) Nothing context')
  where context' = BlockFollowedBy zlabel `inside` context
```

When its additional parameter is `Nothing`, `nodeWithin` need not provide a means of reaching anything in its context. But when `nodeWithin` calls itself recursively on `Ys`, it must tell the recursive call to make `Yn` reachable by a **br** instruction, so it passes `Just Yn`:

```
nodeWithin x (y_n:ys) Nothing context =
  nodeWithin x ys (Just ylabel) (context `withFallthrough` ylabel) <>
  doTree y_n context
  where ylabel = entryLabel y_n
```

The `withFallthrough` function sets the `fallthrough` label to `Yn`.

These two cases of `nodeWithin` cooperate to push the code that wraps `X` all the way into `nodeWithin`'s base case. When the base case must fill a hole with a **br**-exitable form, `nodeWithin` uses a **block** form—*unless* `X` will translate into `if`:

```
nodeWithin x [] (Just zlabel) context
  | not (generatesIf x) = WasmBlock (nodeWithin x [] Nothing context')
  where context' = BlockFollowedBy zlabel `inside` context
```

When no **block** is needed, the remaining case of `nodeWithin` is implemented as in section 5, except that the `IfThenElse` form in the context now carries the value passed in the parameter of type `Maybe Label`, and this value is recognized by the index function.

## 7.3 Switch Statements

GHC's third enhancement handles a syntactic form not shown in section 5: Cmm's `Switch` form. The `Switch` form implements Haskell's case expression, and in the base case of `nodeWithin` it is translated into a WebAssembly `br_table` instruction (computed branch). Unlike `if ... else ... end`, `br_table` is an instruction, not a syntactic form, and it cannot contain other syntactic forms; it takes label indices as operands. So when translating a Cmm `Switch` form, `doTree` must place *every* child of `X` after a **block**, even if the child is not a merge node. Function `doTree` meets this obligation by passing *all* of `X`'s children to `nodeWithin`; when `X` ends in a `Switch`, the filter `hasMergeRoot` on numbered code line 2 is replaced by the identity function.

## 7.4 Preliminary Evaluation

Including all three enhancements, GHC’s implementation of the translator takes about 350 lines of code. Because GHC’s WebAssembly back end is not yet complete, we cannot translate GHC’s standard benchmarks. But the translator does have a simple test suite: about 30 Haskell and Cmm programs, which have been written to produce typical control-flow graphs (as in Figures 1 and 2), irreducible control-flow graphs, and unusual but reducible control-flow graphs (as in Figure 4).

Results obtained with the test suite suggest that the enhancements described above are worth implementing. For example, when translating all programs in the test suite, the translator in section 5 generates 171 `br` instructions. The enhanced version eliminates 149 of them, leaving only 22. The block-elimination enhancement also reduces code size. Block elimination does not apply to every `if` form; in section 5, an `if` form is wrapped in `block ... end` only if it has at least one merge-node child in the dominator tree. When translating all programs in the test suite, the translator generates 163 `if` forms, of which 59 have at least one merge-node child. The enhanced version eliminates the `block` forms around all 59.

Although these enhancements improve code size, they might not make a difference in run time. A compiler like Cranelift [Bytecode Alliance 2022], for example, which is used in the Wasm-time virtual machine, converts WebAssembly to static single-assignment form (SSA). It should translate GHC’s enhanced and unenhanced forms of WebAssembly into identical SSA forms. But WebAssembly can also be interpreted directly, without translation into an internal form [Titzer 2022]. In Titzer’s interpreter, GHC’s enhanced form will be interpreted more quickly and will require fewer entries in each function’s side table.

## 7.5 Supporting Analyses and Transformations

GHC’s translator is supported by three components not shown in section 5: reverse postorder numbering, dominator analysis, and node splitting. Reverse postorder numbering and dominator analysis build on existing code. Reverse postorder traversal is part of GHC’s dataflow-optimization framework [Ramsey, Dias, and Peyton Jones 2010]. The traversal returns a list of nodes, from which a numbering is obtained by zipping the list with  $[\emptyset . . ]$ . Dominator analysis is implemented using the algorithm of Lengauer and Tarjan [1979]. An existing implementation, which is used to lay out code in GHC’s x86 back end, almost suffices; it just needs to be wrapped so it can analyze a Cmm control-flow graph. The wrapper takes about 35 lines of code. Node splitting represents graphs using the library described by Erwig [2001], but the algorithm requires new code; the implementation described in appendix A takes about 260 lines of Haskell.

## 7.6 Polymorphism

The code deployed in GHC is more polymorphic than what you see above. Polymorphism arises because our design stratifies GHC’s representation of WebAssembly into three layers.

- The *compilation-unit* layer represents all the bureaucracy surrounding a collection of functions: information like each function’s WebAssembly type, its local variables, and so on.
- The compilation-unit layer is parameterized by the *control-flow layer*. That layer closely resembles the Wasm type shown in section 5, except that `WasmExpr` and `WasmActions` are type parameters.
- The type parameters are meant to be instantiated by types from the *basic-block layer*, which represents WebAssembly instructions that do not affect control flow within a procedure. That layer uses types to distinguish an “expression,” which pushes a value onto WebAssembly’s evaluation stack, from “actions,” which leave WebAssembly’s evaluation stack unchanged.

In GHC, the translation function is polymorphic in the same way: `WasmExpr` and `WasmActions` are replaced by type parameters. During testing, these type parameters are instantiated by string renderings of Cmm code, and a pair of interpreters is used to confirm that the original Cmm and its translation execute the same sequence of actions when run in the same states. This testing exposed a couple of faults in early versions of the code.

## 8 RELATED WORK

The work most related to this paper is the original translation algorithm of [Peterson, Kasami, and Tokura \[1973\]](#). Given a reducible control-flow graph, its dominance relation, and information about which nodes have paths to which other nodes without traversing a back edge to a loop header, the original translation uses three passes:

- (1) Duplicate loop headers and transform the control-flow graph to eliminate loops.
- (2) Emit code that uses `if-then-else`, `goto`, and labels.
- (3) Transform the emitted code: replace labels with **loops** and each `goto` with a multilevel exit.

My implementation combines these three passes into one. The work of pass 1 is done by using a reverse postorder numbering to identify back edges, which effectively eliminates loops. And once the target language is represented using an algebraic data type and the translation is written recursively, the work of passes 2 and 3 is easily done in just one pass.

The original translation targets a language that is simpler than WebAssembly: the original target language has just `if`, `loop`, and a multilevel `exit`. The original target language has no **block** form, and it lacks WebAssembly's split-personality `br`, which acts as an exit (break) from most forms but as `continue` when targeting a **loop** form. WebAssembly's `br` simplifies the problem, as the translation falls out exceptionally nicely, but my implementation could easily be adapted to the original target language.

[Peterson, Kasami, and Tokura](#) don't just present an ideal translation; they go deep into what can and cannot be accomplished. Their paper is full of useful examples, e.g., control-flow graphs whose translations must add either static code or dynamic computation, or a control-flow graph whose translation into an `if/loop` language requires multilevel exit. If you are interested in the capabilities and limitations of structured control flow, the paper is well worth reading.

[Peterson, Kasami, and Tokura](#) do not use the same data structures as my implementation—reverse postorder numbering and the dominator tree—but they use almost the same ideas:

- [Peterson, Kasami, and Tokura](#) define a relation “*A* is above *B*” to mean that (in my terms) there is a path from *A* to *B* that uses only forward edges. I simplify my implementation by using the stronger relation “*A* has a lower reverse postorder number than *B*,” which implies “above” but is easier to implement.
- [Peterson, Kasami, and Tokura](#) use the standard dominance relation, saying “cover” for “dominate” and “lowest cover” for “immediate dominator.”

I use two insights that seem not to be in the original paper: the well-known insight that the immediate-dominator relation defines a tree in which each node appears exactly once, and the new insight that this particular translation works by induction over the structure of that tree. This insight, in my opinion, makes it easier to argue for the correctness of the translation.

[Baker \[1977\]](#) describes another algorithm for translating a control-flow graph into structured code. Her algorithm targets Ratfor, whose control-flow forms resemble those of [Peterson, Kasami, and Tokura](#), plus a `next` form that continues execution of a loop, plus `goto`. Baker's translation uses `goto` when the source control-flow graph is irreducible, and sometimes even when it isn't. That choice makes Baker's algorithm unsuitable for generating JavaScript or WebAssembly, but it suits her goal: to produce code that is compact, correct, and readable. Her paper begins with

the ideas and terminology that we would use today: forward and back edges, reverse postorder numbering, loop headers, and immediate dominators. It then goes on to introduce new concepts that help place translated code in the positions that Baker finds most readable.

Ramshaw [1988] shows how to transform a source program that uses `goto` into a target program that uses multilevel `exit`, while retaining the structure of the original program. (He wished to translate the source of  $\text{\TeX}$  from Pascal into Mesa.) As Ramshaw notes, he picks up where Peterson, Kasami, and Tokura leave off. In particular, he builds on pass 3 of Peterson, Kasami, and Tokura, which works by rewriting the `gotos` out of an otherwise structured program. Ramshaw presents a different set of rewrite rules, based on an analysis of a directed graph of `gotos`.

Zakai [2011] describes Emscripten, a compiler from LLVM to JavaScript. Like WebAssembly, JavaScript offers only structured control flow and lacks `goto`. Zakai describes an algorithm called Relooper, which recursively takes a *set* of nodes from a control-flow graph and returns a structured representation of the control flow as Emscripten “blocks.” As observed by Iozzelli [2019], Relooper is a greedy, heuristic algorithm that works by identifying common patterns in a control-flow graph. Relooper appears not to rely on the dominance relation or on any ordering of nodes in the control-flow graph. When an Emscripten block can exit to more than one node, Relooper gathers the exit labels under a `switch` on a special variable `__label__`, which identifies the node to which control is intended to be transferred. This approach handles arbitrary control flow, including irreducible control flow, but it may add assignments to `__label__` and tests of `__label__` even when they are not needed. Although `__label__` can be optimized away in “many or even most” cases, Relooper provides no guarantees.

Iozzelli [2019] describes Stackifier, a translation used in Cheerp, which is a C++ compiler that generates both WebAssembly and JavaScript. Iozzelli begins with examples that expose problems with Relooper, noting that Relooper “is very easily tricked by more complex control flow into emitting unnecessary label assignments.” (Similar examples had caused problems for Cheerp in practice.) By contrast, Stackifier relies on the same compiler concepts that are used in this paper and in other related work: forward edges and back edges, dominance, and loop headers. Stackifier orders nodes by topologically sorting the subgraph containing forward edges; such a sort produces a reverse postorder ordering. A simple version of Stackifier produces code that uses too many `break` and `continue` forms (`br` instructions), making the resulting code hard to read. Readability is improved by moving non-merge nodes inside `if`, just as Peterson, Kasami, and Tokura do. (Interestingly, Iozzelli cites Ramshaw [1988] but not Peterson, Kasami, and Tokura [1973].)

Stackifier, like the algorithm of Peterson, Kasami, and Tokura, works only on reducible control-flow graphs. Reducibility is nicely introduced by Allen [1970]. Allen defines reducibility by first identifying *intervals* in a control-flow graph; an interval describes a region of the control-flow graph that has one entry and contains a loop. Intervals partition the graph, and the partition induces a “second-order control-flow graph” in which each interval is a node. The partitioning process can be iterated until it reaches a fixed point; if the fixed-point graph has but a single node, the original graph is reducible.

Hecht and Ullman [1972] show that a control-flow graph is reducible if and only if it is *collapsible* by a sequence of much simpler transformations, each of which either removes a self edge or absorbs a node into its unique predecessor. A graph is collapsible if and only if this process terminates in a single-node graph. Collapsibility offers the advantage of helping to identify nodes that can be split to convert an irreducible control-flow graph into one that is equivalent but reducible.

Node splitting is described by Aho et al. [2007, section 9.7.6]. Naïve node splitting may duplicate much more code than is necessary; Janssen and Corporaal [1997] report that on a set of benchmarks, a naïve approach more than triples the number of nodes in a program, on average. Janssen and Corporaal present heuristics that reduce the number of duplicated nodes to about 3% of the total.

## 9 DISCUSSION

The translator described in this paper produces code that is correct and well structured, but not perfectly readable. The translator does move actions into **if** forms in a way that produces readable code, without requiring the additional steps described by Iozzelli [2019]. But it moves actions into **loop** forms too aggressively: code that we’d like to see immediately *after* a loop is instead placed *inside* the loop and followed by a **br** or **return** instruction. For example, in Figure 2c, *do C<sub>actions</sub>* is placed inside the loop. Supposing anybody wanted to read the generated WebAssembly, as opposed to just run it, this placement would be suboptimal—as was well known to Baker [1977, Section 3]. We would prefer that *do C<sub>actions</sub>* be placed after the end of the loop.

The readability of the translator’s output is less important than the process by which the translator was developed. A friend of mine likes to tell young engineers, “Sometimes if you spend two weeks hacking, you can save an afternoon that you otherwise would have had to spend in the library.” This project used the opposite process: weeks went into finding good papers and understanding what was written in them, after which the first version of the new translator was written in an afternoon. And even in that first version, the three passes of the original translator were combined into one—although that hadn’t been a goal. The single-pass translator emerged as a natural, happy consequence of having the reverse postorder numbers handy and of tackling the coding using recursive functions and an algebraic data type. This new implementation serves a practical need, and the experience has reinforced my belief in the power of functional programming. I hope you have enjoyed reading about it as much as I enjoyed writing about it.

## ACKNOWLEDGMENTS

I was able to persuade GHC to generate an irreducible control-flow graph only with the help of Richard Eisenberg.

Helpful feedback on the manuscript was provided by Julien Debon, Ron Garcia, Manuel Serrano, and the anonymous referees.

## DATA-AVAILABILITY STATEMENT

The software described in this paper is openly available in the Zenodo repository [Ramsey 2022].

## A TRANSFORMING IRREDUCIBLE CONTROL-FLOW GRAPHS

GHC transforms an irreducible control-flow graph into a reducible control-flow graph by using an algorithm called *node splitting*. GHC’s algorithm is a simpler version of the algorithm described in the venerable “dragon book” [Aho et al. 2007, section 9.7.6]. GHC transforms an irreducible control-flow graph into a reducible control-flow graph in three steps: embed the original control-flow graph into a “supergraph,” collapse the supergraph down to a single node (modifying the control-flow graph in the process), and recover an equivalent control-flow graph from the supergraph. Since there are two graphs in play, I call the nodes of the supergraph “supernodes,” and I call nodes of the control-flow graph “control-flow nodes.” (A supernode is effectively the same as the “region” described in the dragon book, except that a supernode need not keep track of edges.)

Both GHC's node splitter and the dragon book's node splitter rely on the following invariants:

- Each supernode contains one or more control-flow nodes; the nodes of the supergraph partition the nodes of the control-flow graph.
- Each supernode contains a unique *head*, which is a control-flow node. The head dominates all the control-flow nodes contained in the supernode. All incoming control-flow edges from control-flow nodes contained in other supernodes point to the head.
- At each supernode  $Z$ , the set of control-flow nodes contained within  $Z$ , together with their incoming and outgoing edges, define a control-flow subgraph with these properties:
  - It is reducible.
  - It has a single entry point (the head).
  - It may have arbitrarily many exits.

In GHC, these invariants are first established by creating a supergraph in which each supernode contains exactly one control-flow node. In other words, each node of the original control-flow graph is initially the head of its own supernode.

It is convenient but not necessary to think of the supergraph as having edges: a superedge exists between two supernodes  $U$  and  $V$  if and only if there exists a control-flow edge from a control-flow node contained in  $U$  to a control-flow node contained in  $V$ . (The invariants imply that the control-flow node in  $V$  must be  $V$ 's head.)

A supergraph is collapsed by repeatedly applying one of the following two transformations until the supergraph has a single supernode:

#### MERGE

If the supergraph contains two supernodes  $U$  and  $V$  such that every control-flow edge that comes into  $V$ 's head originates with a control-flow node that is contained in  $U$ , then  $U$  and  $V$  can be merged.

The precondition for MERGE is usually expressed by saying "in the supergraph, supernode  $V$  has a unique predecessor  $U$ ." The MERGE transformation combines the two transformations of the graph-collapsing algorithm described by [Hecht and Ullman \[1972\]](#):  $T_2$ , which merges two (super) nodes, and  $T_1$ , which removes a self-edge from the (super) graph.

#### SPLIT

Choose a supernode  $X$ , and find the maximal set of supernodes  $\{W_i\}$  such that each  $W_i$  contains a control-flow node with an outgoing edge that points to the head of  $X$ . If the set  $\{W_i\}$  contains more than one supernode, *split*  $X$  as described below.

The precondition for SPLIT is usually expressed by saying "in the supergraph, supernode  $X$  has multiple predecessors  $\{W_i\}$ ."

A supernode  $X$  with predecessors  $\{W_i\}$  is split as follows:

- (1) By the invariant,  $X$  contains a nonempty set of control-flow nodes; call them  $\{X_j\}$ . For each  $W_i$ , create a new supernode  $X_i$  that is a "fresh" copy of  $X$  in the following sense:
  - For each control-flow node  $X_j$  in supernode  $X$ , create a copy  $X_{j,i}$  that has a fresh label.
  - For each  $j$  and  $j'$ , within the copy  $X_{j,i}$ , replace every reference to the label  $X_{j'}$  with a reference to the label  $X_{j',i}$ .
 The resulting set of control-flow nodes  $X_{j,i}$ , with  $i$  held fixed and  $j$  varying, forms the new supernode  $X_i$ .
- (2) For each  $W_i$ , and for each control-flow node  $W$  in supernode  $W_i$ , replace every reference to the head of  $X$  with a reference to the head of  $X_i$ .

The freshening of  $X$  is easier to implement than to describe: list the labels of the control-flow nodes  $\{X_j\}$ ; apply "gensym" to the list to get a list of fresh labels; zip them together to make



an association list, which defines a substitution; and map the substitution over the set  $\{X_j\}$ . (The substitution changes all mentions: uses as well as definitions.) Then to implement step 2, apply the same substitution to the control-flow nodes contained in  $W_i$ .

As an example, let's look again at the canonical irreducible control-flow graph from Figure 6, shown on the right as a supergraph (Figure 7). This is the supergraph that would be produced by a sequence of MERGE transformations from any control-flow graph that is irreducible because it has one loop with two entry points:  $X$  and  $W_2$ . The MERGE transformation can't be applied to this supergraph, but the SPLIT transformation can be applied to supernode  $X$  or to supernode  $W_2$ , each of which has two predecessors.

Suppose entry point  $X$ , which is reachable from both inside and outside the loop, is split into two new supernodes  $X_1$  and  $X_2$ , in such a way that  $W_1$  points to  $X_1$  and  $W_2$  points to  $X_2$ . The resulting supergraph is shown in Figure 8. Supernode  $X_1$  is now reachable only from outside the loop, and it is no longer an entry point; instead, it duplicates the code also found in  $X_2$ , then transfers control to  $W_2$ . And supernode  $X_2$  is now reachable only by paths that pass through  $W_2$ ; the loop cannot be entered at  $X_2$ , so  $X_2$  is no longer an entry point. In Figure 8, supernode  $W_2$  is the undisputed header, and the only entry point, of the loop between supernodes  $W_2$  and  $X_2$ . And the graph in Figure 8 is reducible: first  $X_1$ , then  $W_2$ , and finally  $X_2$  can be repeatedly MERGED into  $W_1$ .

The node-splitting algorithm described above is nondeterministic; there are many supergraphs to which either MERGE or SPLIT could be applied. GHC uses a naïve, greedy strategy: whenever possible, it applies MERGE. This strategy can duplicate control-flow nodes unnecessarily—and on benchmarks of irreducible control-flow graphs taken from real-world programs, it duplicates many tens of times more nodes than an optimal sequence of MERGE and SPLIT transformations [Janssen and Corporaal 1997]. The heuristics described by Janssen and Corporaal can reduce the amount of duplication to very near optimal. But because almost all control-flow graphs generated by GHC are expected to be reducible, the cost of duplicating nodes is expected to be negligible, so GHC does the simplest thing that could possibly work.

## REFERENCES

- Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2007. *Compilers: Principles, Techniques, and Tools*. Pearson, Boston, 2nd edition.
- Frances E. Allen. 1970. Control flow analysis. *Proceedings of a Symposium on Compiler Optimization*, in *SIGPLAN Notices*, 5 (7):1–19.
- Brenda S. Baker. 1977 (January). An algorithm for structuring flowgraphs. *Journal of the ACM*, 24(1):98–120.
- Bytecode Alliance. 2022. Cranelift code generator. URL <https://github.com/bytecodealliance/wasmtime/tree/main/cranelift>.
- Jack W. Davidson and Christopher W. Fraser. 1984 (October). Code selection through object code optimization. *ACM Transactions on Programming Languages and Systems*, 6(4):505–526.
- Martin Erwig. 2001 (September). Inductive graphs and functional graph algorithms. *Journal of Functional Programming*, 11 (5):467–492.
- Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017 (June). Bringing the Web up to speed with WebAssembly. *Proceedings of the ACM SIGPLAN '17 Conference on Programming Language Design and Implementation*, in *SIGPLAN Notices*, 52(6):185–200.
- Matthew S Hecht and Jeffrey D Ullman. 1972. Flow graph reducibility. *SIAM Journal on Computing*, 1(2):188.
- Yuri Iozzelli. 2019 (April). Solving the structured control flow problem once and for all. URL <https://medium.com/leaningtech/solving-the-structured-control-flow-problem-once-and-for-all-5123117b1ee2>. Blog post, snapshotted in the Internet Archive on December 24, 2021.
- Johan Janssen and Henk Corporaal. 1997. Making graphs reducible with controlled node splitting. *ACM Transactions on Programming Languages and Systems*, 19(6):1031–1052.

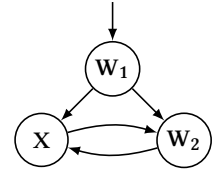


Fig. 7. Before SPLIT

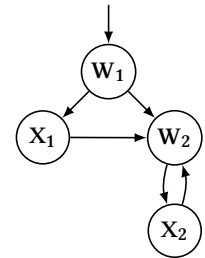


Fig. 8. After SPLIT

- Thomas Lengauer and Robert Endre Tarjan. 1979. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141.
- W. Wesley Peterson, Tadao Kasami, and Nobuki Tokura. 1973. On the capabilities of while, repeat, and exit statements. *Communications of the ACM*, 16(8):503–512.
- Simon L. Peyton Jones. 1992 (April). Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *Journal of Functional Programming*, 2(2):127–202.
- Simon L. Peyton Jones, Norman Ramsey, and Fermin Reig. 1999 (September). C--: A portable assembly language that supports garbage collection. In *International Conference on Principles and Practice of Declarative Programming, LNCS volume 1702*, pages 1–28. Springer Verlag. Invited paper.
- Norman Ramsey. 2022 (June). Beyond reloader: Recursive translation of unstructured control flow to structured control flow. Software artifact archived at Zenodo. <https://doi.org/10.5281/zenodo.6727752>
- Norman Ramsey and João Dias. 2005 (September). An applicative control-flow graph based on Huet’s zipper. In *ACM SIGPLAN Workshop on ML*, pages 101–122.
- Norman Ramsey, João Dias, and Simon L. Peyton Jones. 2010. Hoopl: A modular, reusable library for dataflow analysis and transformation. *Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell (Haskell 2010)*, in *SIGPLAN Notices*, 45(11): 121–134.
- Norman Ramsey, Simon L. Peyton Jones, and Christian Lindig. 2005 (February). The C-- language specification Version 2.0 (CVS revision 1.128). See <http://www.cs.tufts.edu/~nr/c-/code.html#spec>.
- Lyle Ramshaw. 1988. Eliminating go to’s while preserving program structure. *Journal of the ACM*, 35(4):893–920.
- Martin Sulzmann, Manuel MT Chakravarty, Simon Peyton Jones, and Kevin Donnelly. 2007. System F with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI)*, pages 53–66.
- Ben L. Titzer. 2022. A fast in-place interpreter for WebAssembly. *PACMPL*, 6(OOPSLA). To appear.
- Alon Zakai. 2011. Emscripten: An LLVM-to-JavaScript compiler. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA ’11*, page 301–312.