1 The Relooper Algorithm

Relooper is an algorithm that turns unstructured control flow into structured control flow. Unstructured control flow can contain arbitrary branch instructions to anywhere else in the code, whereas structured control flow is bounded within higher-level structures such as if and while blocks.

The Relooper algorithm is described in the Emscripten paper [1].

1.1 Input

The Relooper algorithm takes a so-called 'soup of blocks' as input. Each block is a section of the instructions that starts at a label and ends with a branch instructions (which may be either a conditional branch followed by an unconditional branch, or just an unconditional branch).

```
label:
...
non-branch instructions
...
if ... goto x      (optional)
goto y
```

The list of instructions generated in the intermediate representation is processed into a set of these blocks. They're called a 'soup' of blocks because they're no longer stored as a single continuous list, instead each block is separate and points to the blocks to which it can branch.

To avoid overloading the term 'block', we call these input blocks 'labels'.

1.2 Output

The Relooper algorithm generates a set of structured blocks, which are nested to represent the structured control flow.

There are three types of block:

- **Simple** blocks, which contain
  - An **Internal** label
  - A **Next block**
- **Loop** blocks, which contain
  - An **Inner** block
  - A **Next** block
- **Multiple** blocks, which contain
  - Some number of **Handled** blocks
  - A **Next** block

Simple blocks are the basic building block, and just contain one of the input labels (which contains the actual code to execute), and point to the block to which execution should pass next. When this is translated into target code, a Simple block gets translated directly into the code inside the label, and the Next block is put right after it.

Loop blocks represent any kind of loop in the code. The Inner block contains any labels that can branch back to the start of the loop (along some execution path, which may be multiple labels long). The Next block contains all the rest of the labels, from which execution will never be able to get back to the start of the loop.

Multiple blocks represent any kind of conditional execution, e.g. 'if' or 'switch' statements. The Handled blocks are any blocks to which execution can directly pass when we enter this block. By use of the label variable, described below, we decide which (if any) of the Handled blocks should get executed when we enter the Multiple block. Once the selected

Handled block has finished executing, we go to the Next block. It's also possible to not execute any of the Handled blocks, for example to represent the control flow of an 'if' statement without an 'else', in which case execution will go directly to the Next block.

1.3 Definitions

Labels are the input blocks, and **Blocks** are the output blocks, as described above.

When constructing a block from a set of labels, **Entries** are the subset of those labels that can be immediately reached when the block is executed. (I.e. those that are possible branch targets of the previous block.)

Each label has a set of **possible branch targets**. These are the labels to which this label directly branches.

Each label has a set of other labels which it can **Reach**. This is the transitive closure of the possible branch targets; that is, label X can reach label Y if, starting execution from label X, it is possible to end up executing label Y along some possible path of execution. We also call this the **Reachability** of a label.

1.4 The algorithm

The following is a broad overview of the steps of the Relooper algorithm. Some of the technical subtleties are described afterwards.

Given a set of labels, and a subset of those labels that are entries, try the following steps. Once one of the steps matches and creates a block, stop trying subsequent options.

1. For each label in the set, calculate which other labels it can reach.
2. If we have only one entry, and execution can't return to it (i.e. if the single entry can't reach itself), then create a Simple block.
   - Put the single entry as the Internal label.
   - Construct the Next block from all the other labels using this algorithm recursively. The entries for the Next block are all the possible branch targets of the Internal label.
3. If execution can return to all the entries, create a Loop block.
   - That is, take the union of the reachability of all the entries, and check whether all of the entries appear in that set.
   - Construct the Inner block from all the labels that can reach at least one of the entries. The entries for the Inner block are the same as the current entries.
   - Construct the Next block from all the other labels. The entries for the Next block are all the possible branch targets of labels in the Inner block that are labels in the Next block.
4. If there's more than one entry, try to create a Multiple block. (Note that creating a Multiple block may not be possible.)
   - For every entry, see if there are any labels it reaches that no other entry reaches. (This could be just the entry label itself.)

     If at least one entry has such labels that it uniquely reaches, we can create a Multiple block.
   - Create one Handled block for every entry that uniquely reaches some labels. Construct the Handled block from all the labels the entry uniquely reaches (including itself). The entry for that block is the entry we generated it from.
   - Construct the Next block from all the other labels. The entries for the Next block are all the entries we didn't convert into Handled blocks, plus any other possible branch targets out of the Handled blocks.
5. If creating a Multiple block failed, create a Loop block, in the same way as step 3.

- The Emscripten paper contains a proof of why this is possible [1, p. 10]. I won't repeat it here, but the essence is that we can prove that (1) the algorithm is successful whenever it completes; (2) the algorithm must complete, because the problem is simplified at every step when we create a block; and (3) the algorithm is always able to create a block from the labels it's given.

When Relooper is generating the output blocks, it processes and replaces the branch instructions in the input labels. The instructions in the output blocks don't contain any of the original branch instructions.

- When creating a loop, any branch instructions to the start of the loop are replaced with a `continue`, and any branch instructions to outside the loop (i.e. to the Next block) are replaced with `break` instructions. These store the ID of the loop they act on, because loops may be nested.
- When creating a Handled block, any branch instructions to the Next block are replaced with an `EndHandled` instruction. (Functionally this is identical to a `break`, but I chose to keep the two distinct from each other, so that `break` instructions can store the ID of the loop they jump out of, and `EndHandled` instructions can store the ID of the Multiple block.)
- In addition to replacing branch instructions as above, every branch instruction will set the label variable. The label variable is the mechanism for directing control flow in the generated code. Where there was a `goto X` instruction in the input, this is replaced with a `label = X` instruction.

  Whenever there is a choice of which block to enter, i.e. when entering a Multiple block, we insert instructions that check the value of the label variable, and execute the corresponding block. Essentially, this acts like a switch statement.

This will generate some overhead, however most of the set/check label instructions will be able to be optimised away, for example for Simple blocks where execution just passes straight to the next block unconditionally.

References

[1] Alon Zakai. *Emscripten: An LLVM-to-JavaScript Compiler*. Mozilla, 2013. URL: `https://raw.githubusercontent.com/emscripten-core/emscripten/main/docs/paper.pdf`.