Formal Replay of Translation Validation for Highly Optimised C
Work in Progress

Thomas Sewell

NICTA & UNSW, Sydney, Australia
`thomas.sewell@nicta.com.au`

Abstract

In previous work [6] we have implemented a translation validation mechanism for checking that a C compiler is adhering to the expected semantics of a verified program. We used this apparatus to check the compilation of the seL4 verified operating system kernel [2] by GCC 4.5.1, with some optimisations disabled. We obtained this result by carefully choosing a problem representation that worked well with certain highly optimised SMT solvers. This raises a question of correctness. While we are confident the result is correct, we still aim to replay this result with the most dependable tools available.

In this work we present a formalisation of the proof rules needed to replay the translation check within the theorem prover Isabelle/HOL. This is part of an ongoing effort to bring the entire translation validation result within a single trusted proof engine and derive a single correctness theorem. For the moment such a single correctness theorem is our gold standard of trustworthiness.

We had hoped to present the formal rule set in action through a worked example. Unfortunately while we have all the theory we need, the mechanisms for selecting and applying the rules and discharging the resulting proof obligations remain a work in progress, and our example proof is incomplete.

1 Introduction

In their study on compiler testing, Yang et al reported 325 previously unknown defects in 11 different C compilers [8]. Even the formally verified CompCert was found to exhibit 5 defects, albeit in its unverified front-end only. Ideally the C compiler should have a correctness proof, such as CompCert's, to eliminate internal defects, and additionally the source correctness proof should connect directly the correctness proof for the compiler.

In our previous work [6] we used a compiler-checking approach to establish the correctness of GCC's operation. The strength of the approach is that it connects directly upwards to the program proofs via the NICTA C semantics parser [7] and directly downwards to the semantics of the underlying hardware as seen through the Cambridge ARM semantics [1] via Myreen's decompilation apparatus [3, 4]. Formally, the various notions of refinement proved at different levels of this overall verification compose with each other, or rather, they would compose if they existed as theorems in a single logic. The weakness of this approach is that it requires a number of translations between tools to make use of their varying strengths and to connect to previously existing artefacts. The various artefacts and the toolchains that hosted them at the time of we completed our previous work are sketched in Figure 1 - the tools Z3 and SONOLAR here are the SMT solvers used in our experiments.

The existing SMT-based proof tool and some of the relevant Isabelle theories are available from `http://www.ssrg.nicta.com.au/software/TS/graph-refine/`.

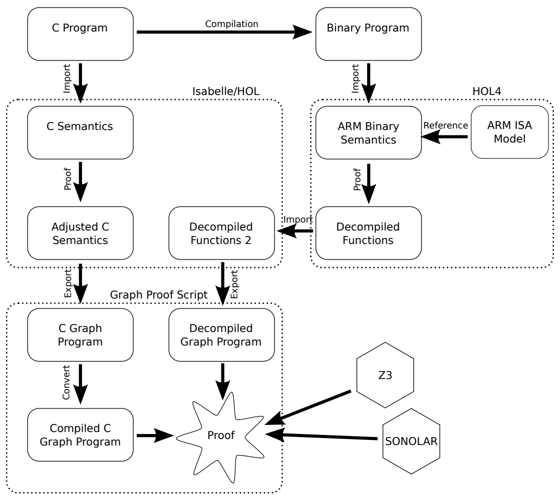


Figure 1: Artefacts in the correctness proof

Our current objective is to replay the proofs performed by our custom "graph proof script" within the theorem prover Isabelle/HOL [5]. The correctness of this argument then depends only on Isabelle's trusted core, its proof kernel, and not on any of the tools we have built to specifically address this problem. To perform this replay, we must formalise in Isabelle/HOL both the proof rules used by the graph script to structure its proofs, and also the logic by which the proof obligations are reduced to SMT queries.

To complete the overall theorem in Isabelle/HOL, we will additionally need to import Myreen's decompilation proofs from HOL4 into Isabelle/HOL. We will also need further guidance from our SMT solvers to address the largest and most difficult of the proof obligations generated in this proof framework. This is difficult, because the most effective solvers are the ones that do not produce proof traces. We leave both these problems to future work.

In the rest of this report, we introduce an example program and its compilation in Section 2, sketch a proof of compilation correctness in Section 3, define the problem formally in Section 4 and introduce the formalisation of the key rules of the proof in Section 5. In Section 6 we sketch our work in progress on reducing the remaining proof obligations to SMT-compatible form.

2 An Example Program

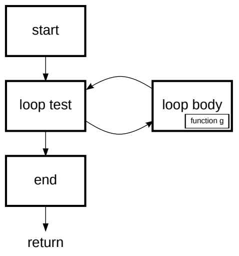
Consider the simple example program of two functions and one loop defined in Figure 2b. While this is a simple program, when we compile it with GCC 4.5.1 at optimisation level 2, it becomes far more complex. The control flow graph (CFG) of the program, together with the simplified CFG of the compiled binary, are also shown in Figure 2b.

As Figure 2b shows, the call to the function `g` has been inlined. The compiler has also introduced far more conditionals. The intent here is to unroll the central loop, with a single iteration of the loop starting at address `0x68` performing the actions of two iterations of the loop in the source. This combined iteration only checks `i < 100` once, which saves a lot of time. This works if `i` is always even at this point. The additional code complexity before the loop entry in the binary CFG helps ensure this.

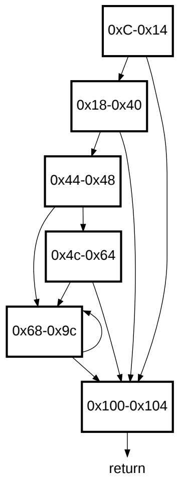
The first two checks of the binary CFG handle specifically the cases where the source loop

```
int
g (int i) {
  return i * 8 + (i & 15);
}

void
f (int *p, int x) {
  int i;

  for (i = x; i < 100; i ++) {
    p[i] = g (i);
  }
}
```

(a) Example source code



(b) Example C CFG



(c) Example binary CFG

is executed 0 or 1 times. The first execution of the source loop is also performed. The next key conditional checks whether the loop is to be executed an even number of times in total. If so, an additional copy of the loop body is executed, followed by a test for the case in which the loop was to be executed exactly 2 times and execution is now finished. The binary loop body then begins, executing the source loop actions twice and then checking whether to continue.

3 Informal Proof

We have introduced a function `f` and the structure of its compilation. The body of our work will be to check that the semantics of these programs match, and to prove this fact as completely as possible.

The explanation we have given so far takes the compiler's view, performing a series of transformations on the source structure to result in the binary structure. It would be possible to prove the correctness of the compilation in the same style, guessing which transformations had been applied and showing that each transformation was valid. The correctness proofs we will produce do not take this approach.

Instead, we approach this problem analytically. We consider some arbitrary collection of inputs to our function, and think of the function as a mathematical function which calculates a resulting value of memory as an expression over those inputs. We will show that the two expressions computed by the source and binary program are equal.

If the CFG of our function was acyclic, we could indeed convert the source and binary programs into expressions in the SMT language and have an SMT solver check the equality directly [1]. Since the CFG structure for `f` and its binary contain loops, we cannot simply express the effect of these programs as SMT expressions. Instead we must address the looping structure specifically.

Our proof works by induction on the sequence of visits to a particular point in the binary loop. We will prove a relationship to the sequence of visits to a matching point in the source

[1]This is a substantial simplification, since the conversion is somewhat involved and a number of side conditions must be checked also. For further details, see the full paper describing our previous work [6].

loop. To complicate this, the relationship between visits to the binary loop and visits to the source loop of `f` changes depending on which of the entry paths was taken in the binary loop. These two paths correspond to the cases where the loop in `f` would be executed an even or odd number of times. To avoid this complication, the first step of our proof is a case division on whether or not the instruction at binary address `0x4c` (see Figure 2b) is ever executed. We will sketch the proof of the case where it is visited, the even case, here. The odd case is essentially the same with different parameters.

The next step of our proof is to show by induction that a sequence of visits to the address `0x68` within the binary (the start of the loop basic block) correspond to a sequence of visits to the start of the loop body of the source CFG. Because two executions of the source loop are handled specially in the binary, the first such visit to the binary point actually corresponds to the third visit to the source point. Because the binary loop is unrolled, the subsequent visits will correspond to every second source visit. So the binary sequence matches the subsequence of the 3rd, 5th, 7th etc visits to the source loop.

We prove the binary sequence and the source subsequence correspond by induction. By correspondence, we mean that the given visits occur in the same cases, so, if the binary point is visited 3 times but not a 4th time, the source point must be visited 7 times, but not 9 times. The corresponding visits also have related variable values. So the contents of memory will be the same at corresponding visits and the values of the relevant registers can be expressed as functions of `i`, `x`, etc. We are omitting the details here: our automated proof process discovers this relationship and we are not interested in the specifics.

The induction proof must establish that, given related n-th visits of these visit sequences, the two loops either proceed to related $(n+1)$-th visits or they both exit. It must also show that the 1st visits to the visit sequences (which are the 3rd and 1st visits to the actual points of the CFGs) either both occur or both do not occur. These checks all concern only a bounded number of steps of the source and binary programs, and the relevant proof obligations can be converted to a bounded expression in the SMT language, and can be checked by an SMT solver.

Having proven that the visit sequences are related, we now consider three cases for the length of the visit sequences. The first case is that they are both infinite, which is possible for loops in C. It happens we can exclude this case since our source loop is bounded, but we don't need to use this reasoning. If both programs execute forever, then the compilation was valid[2].

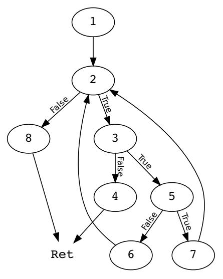
The second case is where both sequences are empty. This means that the binary point is not visited and the source point is visited at most 2 but not 3 times. In this case the number of possible paths through the two CFGs is bounded again, and we can once again derive expressions for the total effect of the two functions. The output memory equality we need to check can once again be exported to SMT and checked.

The third case is where the sequences contain some finite number of visits. If we call this number of visits n, then we know that the two executions reach related n-th visits, but that the $(n+1)$-th visits are not reached. This once again limits us to a single path through the CFGs, and we can derive expressions for the total effect of the functions, this time as expressions over the unknown but related values of the variables at the n-th visits. Once again we can prove the needed equality on the return values.

This is a sketch of a proof of correct compilation for the even case. Together with the odd case, which is similar, we have a proof that the compilation of this program was correct. The mechanism for finding and checking this proof already existed in our previous work[6] although we have extended it somewhat for this problem, including additional logic for finding

[2]In principle our program might have external effects that can be observed even when it does not terminate. For the moment we are ignoring this issue.

```
    struct node *
    find (struct tree *t, int k) {
1     struct node *p = t->trunk;
2     while (p) {
3       if (p->key == k)
4         return p;
5       else if (p->key < k)
6         p = p->right;
        else
7         p = p->left;
      }
8     return NULL;
    }
```



```
1: p := Mem[t + 4];

2: p == 0 ?

8: ret := 0
3: Mem[p] == k ?

4: ret := p;
5: Mem[p] < k ?

6: p := Mem[p + 4];
7: p := Mem[p + 8];
```

Figure 2: Example Conversion of Structure and Statements to Graph Language

the matching subsequences when one loop has been unrolled, and additional logic for finding and performing the kind of case split with which this proof began.

The challenge will be to formalise this proof fully in Isabelle/HOL.

4 Graph Language and Semantics

The first step in formalising this proof is to formally define the problem. We begin by formally defining a language in which to describe the source and binary encodings of `f`. The language we will use is a graph based program representation, which we call a “graph language”. This language was described in detail in our earlier work [6], we will provide an overview here. Note that the graph language is not the same as the CFGs we have mentioned already. Those CFGs are substantial simplifications of the graphs we use here, which encode the full program semantics and have a lot more nodes. The full graph structure of the program graph for `f` and its binary is shown in Appendix A.

The graph language provides a semantics somewhere between machine language and C. Like in C, statements can perform compound calculations. Like in machine language, however, control flow is governed by an explicit graph representation rather than by statement level syntax such as semicolons or `while` loops. All statements are numbered, and the steps between them are drawn as graph edges, giving us a labelled, directed graph. The advantage is that the context-dependent effects of `break`, `continue` and others are replaced by graph edges which simply specify the number of the next statement. The special label `Ret` represents return from the function. The other special label `Err` represents immediate failure, e.g. illegal pointer use or assertion failure.

The graph consists of three types of nodes. Conditional nodes are used to pick between execution paths, and correspond closely to decisions made by `if` and `while` statements in C. Basic nodes represent normal statements, or normal instructions in the binary, and update the value of variables with the result of some calculation (memory is represented as a variable, as are registers). Call nodes are used to represent function calls, which are distinguished from other statements. A number of restrictions enforced by the C parser are relevant here: function calls may be embedded in other expressions and statements only in very limited ways, statements with multiple effects are forbidden, and switch statements must always be convertible into a chain of if-else statements. An example conversion of a C function to graph representation is sketched in Figure 2.

The graph language was designed so its semantics would be straightforward to formalise in Isabelle/HOL or HOL4. The node types are introduced as a datatype:

```
datatype next_node = NextNode nat | Ret | Err

datatype node =
                Basic (next_node * (string * acc) list)
              | Cond (next_node * next_node * acc)
              | Call (next_node * string)
```

The types here have been amended for simplicity and the syntax is slightly fictional. The precise definitions are given in the **GraphLang** Isabelle/HOL theory that is provided as an appendix.

In this representation variables are indexed by name (a string) only. A `Basic` node is a tuple of a next program address and list of variables to update with expressions, a `Cond` node a pair of successors and a decision expression, and a `Call` node is a tuple of a successor, a function name to call, a list of argument expressions and a list of return variables to overwrite.

A graph function is a tuple of an input variable list, output variable list, graph body and entry point.

```
datatype graph_function = GraphFunction (string list * string list
        * (nat -> node option) * nat)
```

The semantics of the graph language is defined by a small-step relation. The configuration of the system at any step is represented by a stack of running programs. This formal model introduces a number of complexities into our proofs, but is nonetheless desirable because it allows us to model recursion without having to incorporate a user-supplied proof of recursion bound or termination. Each stack frame contains an executing address, the state mapping of variables to values, and the name of the currently executing function. The step relation also takes an environment parameter that maps function names to bodies.

```
type_synonym stack = (next_node * (string -> value) * string) list

constant exec_graph_step :: ((string -> graph_function option)
        -> (stack * stack) set)
```

The full semantics of the step relation are omitted for space reasons. The steps associated with the three node types are intuitive; there are also steps for returning from functions (normally and in error) and additional details for fetching the currently executing node.

We are proving refinement, a formal property which can be defined by appeal to the set of possibly-infinite traces generated by the small-step relation. Refinement means each trace of the binary program has a matching trace in the set of traces of the source program. Matching usually means that the matching traces must have the same termination behaviour as each other and, if terminating, return matching values. However, if the source program transitions to `Err` by performing undefined behaviour, any trace of the binary program matches.

We have a choice of semantics to apply when the source program would diverge, either by looping forever or recursing without bound. C programmers would usually expect the compiler to respect their intentions even if an infinite loop clearly serves no other purpose. We call this the precise semantics. However GCC provides a language extension for `const` and `pure` functions. The compiler may remove calls to these functions if their results are discarded. In principle this means that a non-terminating program might be optimised into a terminating one. We also allow an imprecise semantics, which allows this optimisation. It is certainly debatable whether these attributes should ever be used for a possibly-nonterminating function, but we support the option.

The source program `f` and its compiled implementation can both be represented in the graph language. The conversion of the C program semantics is straightforward. An experimental version of Myreen's decompiler can target the graph language directly, producing a graph language version of the binary.

5 Proof Rules

In our previous work [6] we introduced a type of proof scripts which have three rules, **Split**, **Restrict** and **Leaf** with which they capture a refinement argument. The proof script captures the part of the proof which must be discovered heuristically. Once the proof script is known, the problem reduces to a collection of decidable proof obligations. In this work we convert these rules into proof rules in Isabelle/HOL.

We have already discussed these rules, in some sense, in the informally sketched proof we gave already in Section 3. Recall the logic we used in multiple cases of the proof, where we observed that there were now only finitely many possible paths through the CFGs, and we can thus define the equality on the output memories as a concrete expression. We formalise this logic with the concept of a restriction on a trace. A restriction is placed on the number of visits to a given node in a graph, for instance, we might restrict the number of visits to the start of the loop body in `f` to be 0, 1 or 2.

We can restrict the total number of visits within a trace. We also use restrictions to identify different visits to the same node within a trace. In our sketched proof we considered a sequence of visits to a point p within the loop. We can identify the 3rd visit to p as the visit to p where the number of previous visits to p is restricted to the set $\{2\}$.

```
type_synonym trace = (nat -> stack option)
type_synonym restrs = (nat -> nat set)

constant restrs_condition :: (trace -> restrs -> nat -> bool)

constant visit :: (trace -> next_node
        -> restrs -> (string -> value) option)
constant restrs_eventually_condition :: (trace -> restrs -> bool)

constant pc :: (trace -> next_node -> restrs -> bool)

constant restrs_list :: ((nat * nat list) -> restrs)
```

Formally the constant `restrs_condition` defines if, at a given step in a trace, the restricted nodes have already been visited a matching number of times. From this we derive `visit`, which gives us the variable state at the first satisfying visit, if there is one, and `restrs_eventually_condition` which tells us if the restrictions hold eventually (at some point and then forever afterwards) on the trace. We define `pc`, the path condition of a visit, as the condition that a satisfying visit occurs.

The `restrs_list` constant defines a set of restrictions by explicitly listing them.

```
constant restr_trace_refine :: (bool
        -> (string -> graph_function option) -> string
```

```
          -> (string -> graph_function option) -> string
          -> restrs -> restrs -> output_relation
          -> trace -> trace -> bool)
```

The `restr_trace_refine` predicate captures our proof obligation: trace refinement given restrictions. The first argument specifies the precise or imprecise semantics mentioned previously. The next four arguments specify the graph functions (by name) and the execution environments they exist in. The restrictions are a collection of restrictions on the total trace which we may assume. The output relation defines the equalities that must hold on the values returned by the two programs. In this formalisation we also specify the two traces, allowing us to also name them in assumptions[3] We will formalise our key proof rules as introduction rules for this predicate.

The **Restrict** rule is the key rule which introduces a new restriction the visits to node n. The rule introduces the restriction that n is visited at least i times and less than j times overall. When we said in our informal proof that “the source point is visited at most 2 but not 3 times” we were appealing to the **Restrict** rule, with n the address of the source point, $i = 0$ and $j = 3$.

This rule adds to a restriction set listed by the `restr_list` operator. We add to a list with the list constructor #. The restriction is added in the sense that we must finish the proof by proving the same goal with the additional restriction.

The restriction is on visits to node n. We show n must be reached (`pc` is true) at a point where n has been encountered $i - 1$ times. Likewise we show n cannot be reached (`pc` is false) when it has already been encountered $j - 1$ times. Thus n is reached at least i times and less than j times overall.

theorem restr_trace_refine_Restr1:

```
 j ≠ 0
   ⟶ distinct (map fst rs)
   ⟶ wf_graph_function f ilen olen ⟶ Γ fn = Some f
   ⟶ (i ≠ 0 ⟶
      pc (NextNode n) (restrs_list ((n, [i − 1])
          # (restrs_visit rs (NextNode n) f))) tr)
   ⟶ ¬ pc (NextNode n) (restrs_list ((n, [j − 1])
          # (restrs_visit rs (NextNode n) f))) tr
   ⟶ restr_trace_refine prec Γ fn Γ′ fn′
          (restrs_list ((n, [i ..< j]) # rs)) rs′ orel tr tr′
   ⟶ restr_trace_refine prec Γ fn Γ′ fn′ (restrs_list rs) rs′ orel tr tr′
```

Formalising this rule led to the discovery of a number of side conditions that were only checked implicitly in our previous work. Arithmetic on naturals underflows in Isabelle/HOL, so we must check for the case of zero. We also require some simple wellformedness properties on our graph functions, such as that all graph arcs go to nodes defined within the graph. Another difference is that the previous work put both program graphs within a single numerical namespace. It had a single set of restrictions, where we now have two. We must therefore distinguish between this and a symmetric symmetric **Restrict2** rule which restricts the right-hand-side trace. These are expected variations between the formal and informal developments.

The adjustment performed by the constant `restrs_visit` was a more interesting omission. We check `pc` for our node of interest n relative to the restrictions we have already proven hold

[3]This works because we currently formalise our programs as deterministic. The C and binary semantics are deterministic until we begin reasoning about multithreading and I/O, which for the moment we are avoiding.

on the trace. The `restrs_visit` constant relaxes these restrictions, however, dropping any restrictions to nodes still reachable from n. We know these restrictions hold eventually, but we do not necessarily know they already hold when we visit n. Without this adjustment, we could get spurious contradictions for j and rule out possible traces. The `distinct` constraint above is needed for `restrs_visit` to have the correct effect. Our previous tool had no equivalent check, and could have admitted an unsound proof in this case. For technical reasons it would never have considered such a proof, and we retain confidence in our results. Nonetheless this clearly demonstrates the value of formalisation.

The **Leaf** rule is the terminal rule of proof scripts. Our informal proof ended once the number of possible paths through the CFGs was finite, and we could express our output equalities directly. The **Leaf** rule encodes this logic, checking the output relation of the traces directly under the assumption that enough restrictions are in place that the visit to the return points can be described concretely. It requires the return point of the binary trace to be reached, which essentially means that possibly-nonterminating loops have been handled. In the precise semantics, we need to know the source trace reaches its return point also.

theorem restr_trace_refine_Leaf:
  wf_graph_function f ilen olen ⟶ Γ fn = Some f
    ⟶ wf_graph_function f′ ilen′ olen′ ⟶ Γ′ fn′ = Some f′
    ⟶ pc Ret rs tr ⟶ (prec ⟶ pc Ret rs′ tr′)
    ⟶ output_rel orel (f, f′) (rs, rs′) (tr, tr′)
    ⟶ restr_trace_refine prec Γ fn Γ′ fn′ rs rs′ orel tr tr′

The most involved of these rules is the **Split** induction rule, which provides the mechanism for reasoning about loops. In the proof sketch in Section 3 we proved by induction a relation on subsequences of visits to a source and binary graph node. We then considered three cases, the case of infinitely many visits, which automatically implies refinement, the case where the subsequence does not begin, and the case where the subsequence contains exactly n visits for some positive n. The **Split** rule bundles together these two reasoning steps. The complication over our informal description is that the **Split** rule is designed for k-induction, where k previous visits $n \dots n+k-1$ are used to show the inductive step to visit $n+k$. The case we described previously is the special case where $k = 1$.

theorem restr_trace_refine_Split:
  let vres = λi. restrs_list ((sp, [start + i ∗ step]) # rs);
      vres′ = λi. restrs_list ((sp′, [start′ + i ∗ step′]) # rs′);
      vpc = λi. pc (NextNode sp) (vres i) tr;
      vpc′ = λi. pc (NextNode sp′) (vres′ i) tr′;
      visits = λi. (visit tr (NextNode sp) (vres i),
        visit tr′ (NextNode sp′) (vres′ i));
      rel = λi. vpc i ∧ vpc′ i ∧ vrel i (visits 0) (visits i)
   in (∀i. vpc (i + 1) ⟶ vpc i)
   ⟶ (∀i. i < k ⟶ vpc i ⟶ rel i)
   ⟶ (∀i. vpc (i + k) ⟶ (∀j < k. rel (i + j)) ⟶ rel (i + k))
   ⟶ k > 0 ⟶ step > 0 ⟶ step′ > 0
   ⟶ (¬ vpc k ⟶ restr_trace_refine prec Γ f Γ′ f′
        (restrs_list rs) (restrs_list rs′) orel tr tr′)
   ⟶ (∀i. ¬ vpc (i + k) ⟶ (∀j < k. rel (i + j))
      ⟶ restr_trace_refine prec Γ f Γ′ f′

(restrs_list rs) (restrs_list rs′) orel tr tr′)
⟶ restr_trace_refine prec Γ f Γ′ f′
(restrs_list rs) (restrs_list rs′) orel tr tr′

Here we abbreviate *vpci* as the path condition of the *i*-th subsequence visit in the binary program. The binary subsequence is defined by *start* and *step*. We abbreviate *reli* the condition that the *i*-th subsequence visits occur in both programs and match according to the given variable relation *vrel*. The rule requires that the subsequence visits happen in order, in particular that a visit to $i + 1$ implies a visit to i.

The initial condition of the induction is that the first k binary visits have matching visits if they occur. The inductive condition is that if binary visit $i + k$ occurs and there are matching visits in $\{i..i + k - 1\}$ there must be a matching source visit for $i + k$.

The two cases left to be addressed are the case where the binary subsequence does not reach k visits, and the case where $\{i..i + k - 1\}$ are the last k binary visits. No restrictions are added, but the rules are designed so that in each case sufficient information exists to immediately restrict the visits to the two split points using the **Restrict** rule.

Unlike the proof of **Restrict**, the proof of the induction rule did not result in any unexpected discoveries. The formal constraints that k and the subsequence step sizes must be positive, and that the binary path conditions must be monotonic, were unsurprising formal additions.

In addition to these rules we presented in the previous work, we also include three new rules. The **PCCases** rule is the path-condition case-split which we appealed to at the beginning of the informal proof in Section 3. We divide into subproofs depending on whether a given node is visited. This is logically trivial but necessary for our tool to produce **Split** rules handling unrolled loops like the one we have seen, with different entry paths for different cases.

theorem restr_trace_refine_PCCases1:
(pc nn rs1 tr
⟶ restr_trace_refine prec Γ fn Γ′ fn′ rs rs′ orel tr tr′)
⟶ (¬ pc nn rs1 tr
⟶ restr_trace_refine prec Γ fn Γ′ fn′ rs rs′ orel tr tr′)
⟶ restr_trace_refine prec Γ fn Γ′ fn′ rs rs′ orel tr tr′

The **Err** rule generates the assumption that `Err` is not visited, with any restrictions that are appropriate. The previous tool applied this second rule implicitly with various restriction choices when generating all proof obligations. Here we make this process explicit.

theorem restr_trace_refine_Err:
(¬ pc Err restrs tr′
⟶ restr_trace_refine prec Γ fn Γ′ fn′ rs rs′ orel tr tr′)
⟶ restr_trace_refine prec Γ fn Γ′ fn′ rs rs′ orel tr tr′

The final structural rule we add is the **Call** rule. We decompose our refinement proof at function boundaries, and this is the rule that allows us to appeal to the separate proof done for the called functions. The **Call** rule is technically intricate both to state and prove, and the version we have proven is still insufficiently general. We omit further details of this aspect for space reasons.

These proof rules can, for a given problem, be assembled into a single proof method and applied to a proof problem. In our example program `f`, the proof structure begins with a **PCCases** on the path into the loop, each possibility being addressed by a **Split** rule. Each split rule generates an initial ($< k$) and inductive ($\geq k$) case, a total of 4 cases, each of which is addressed by a pair of **Restrict** rules to fix the number of visits to the two loops followed by a **Leaf** rule. We have implemented an automatic mechanism for composing and applying this compound proof rule. When applied to our example problem, it discharges all `restr_trace_refine` goals, leaving 70 remaining proof obligations, 30 of them non-trivial.

6 Conclusions and Further Work

The rules we have described here handle the trace refinement logic, and reduce our problem to a series of proof obligations about specific `visit` instances. These instances can in turn be reduced to SMT-compatible expressions, and our next task is to perform this reduction in Isabelle/HOL also.

The complementary proof rules to the structuring rules we have just described are the visit rules, which allow us to evaluate the variable state at a visit to the variable state at the immediately previous visit, depending on whether the previous visit was to a `Basic`, `Cond` or `Call` node. Additional logic is needed for nodes with multiple predecessors, visits whose restrictions are impossible, etc.

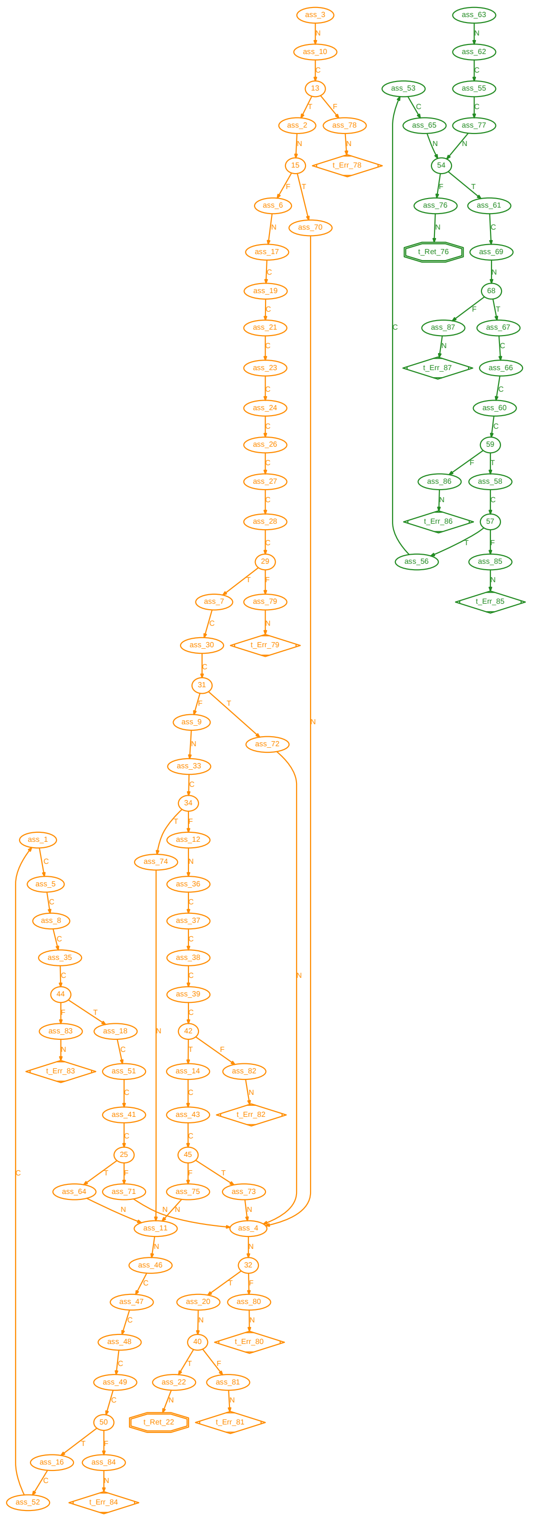
We have proven what we think is the essential rule set for this work, including all these visit rules. What we have not done is produced an automated proof method for applying these rules, which means they remain as yet untested. We hope in the near future to remedy this, and present a completed proof of the correct compilation of our program `f`.

Acknowledgements

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

A Full Problem Graph

This is a rendering of the overall structure of the refinement problem for `f`. The binary graph program in orange is produced by Myreen's decompiler, and the source graph program in green is derived from the Isabelle/HOL import of the C syntax of `f`. This shows the arcs and types of the nodes (`g` is inlined here and all nodes are conditionals or basic) but not any of the semantics; the full problem is more complex again.



References

[1] A. Fox and M. Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In M. Kaufmann and L. C. Paulson, editors, *1st Int. Conf. Interactive Theorem Proving*, volume 6172 of *LNCS*, pages 243–258, Edinburgh, UK, July 2010. Springer.

[2] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proc. 22nd SOSP*, pages 207–220, Big Sky, MT, USA, 2009. ACM.

[3] M. O. Myreen, M. J. C. Gordon, and K. Slind. Machine-code verification for multiple architectures - an application of decompilation into logic. In A. Cimatti and R. B. Jones, editors, *Formal Methods in Computer-Aided Design (FMCAD)*, pages 1–8. IEEE, 2008.

[4] M. O. Myreen, M. J. C. Gordon, and K. Slind. Decompilation into logic — improved. In G. Cabodi and S. Singh, editors, *Formal Methods in Computer-Aided Design (FMCAD)*, pages 78–81. IEEE, 2012.

[5] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[6] T. A. L. Sewell, M. O. Myreen, and G. Klein. Translation validation for a verified OS kernel. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 471–482. ACM, 2013.

[7] H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In M. Hofmann and M. Felleisen, editors, *Proc. 34th POPL*, pages 97–108, Nice, France, Jan. 2007. ACM.

[8] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In M. W. Hall and D. A. Padua, editors, *Proc. 32nd PLDI*, pages 283–294, San Jose, CA, USA, June 2011. ACM.