INFORMATION AND COMMUNICATION TECHNOLOGIES (ICT) PROGRAMME

Project FP7-ICT-2009-C-243881 CerCo

Report n. D3.4
Front-end Correctness Proofs

Version 1.0

Authors:
Brian Campbell, Ilias Garnier, James McKinna, Ian Stark

Project Acronym: CerCo
Project full title: Certified Complexity
Proposal/Contract no.: FP7-ICT-2009-C-243881 CerCo

Executive Summary

This document reports on **D3.4: Front-end Correctness Proofs**, the final deliverable of CerCo work package 3 *Verified Compiler - front end*. That work package was devoted to the formalisation and verification of the front-end of the CerCo cost lifting compiler, and Deliverable 3.4 is a machine-checked formal correctness proof of the compiler front-end, written in the Matita proof assistant.

The proof itself is contained in a series of Matita files, listed in Appendix A and culminating in the lemma `front_end_correct` and the theorem `correct` in the file `correctness.ma`. This document accompanies the deliverable and briefly reports on the work carried out in the development of that proof.

Abstract We report on the correctness proofs for the front-end of the CerCo cost lifting compiler. First, we identify the core result we wish to prove, which says that the we correctly predict the precise execution time for particular parts of the execution called *measurable* subtraces. Then we consider the three distinct parts of the task: showing that the *annotated source code* output by the compiler has equivalent behaviour to the original input (up to the annotations); showing that a measurable subtrace of the annotated source code corresponds to an equivalent measurable subtrace in the code produced by the front-end, including costs; and finally showing that the enriched *structured* execution traces required for cost correctness in the back-end can be constructed from the properties of the code produced by the front-end.

A key part of our work is that the intensional correctness results which show that we get consistent cost measurements throughout the intermediate languages of the compiler can be layered on top of normal forward simulation results, if we split those results into local call-structure preserving simulations. This split allowed us to concentrate on the **intensional** proofs by axiomatising some of the extensional simulation results that are very similar to existing compiler correctness results, such as CompCert.

This report is about the correctness results that are deliverable D3.4, which are about the formalised compiler described in D3.2, using the source language semantics from D3.1 and intermediate language semantics from D3.3. It builds on earlier work on the correctness of a toy compiler built to test the labelling approach in D2.1. Together with the companion deliverable about the correctness of the back-end, D4.4, we obtain results about the whole formalised compiler.

Contents

1 Introduction

The CerCo compiler produces a version of the source code containing annotations describing the timing behaviour of the object code, as well as the object code itself. It compiles C code, targeting microcontrollers implementing the Intel 8051 architecture. There are two versions: first, an initial prototype was implemented in OCaml [3], then a version was formalised in the Matita proof assistant [5, 8] and extracted to OCaml code to produce an executable compiler. In this document we present results from Deliverable 3.4, the formalised proofs in Matita about the front-end of the latter version of the compiler (culminating in the `front_end_correct` lemma), and describe how that fits into the verification of the whole compiler.

A key part of this work was to layer the *intensional* correctness results that show that the costs produced are correct on top of the proofs about the compiled code's *extensional* behaviour (that is, the functional correctness of the compiler). Unfortunately, the ambitious goal of completely verifying the entire compiler was not feasible within the time available, but

$$\begin{tikzcd}
 & & \mathsf{C} \arrow[d] & & \\
 & & \mathsf{Clight} \arrow[r] & \mathsf{Cminor} \arrow[r] & \mathsf{RTLabs} \arrow[d] \\
\mathsf{ASM} \arrow[d] & \mathsf{LIN} \arrow[l] & \mathsf{LTL} \arrow[l] & \mathsf{ERTL} \arrow[l] & \mathsf{RTL} \arrow[l] \\
\mathsf{8051} & & & &
\end{tikzcd}$$

Figure 1: Languages in the CerCo compiler

thanks to this separation of extensional and intensional proofs we are able to axiomatise some extensional simulation results which are similar to those in other compiler verification projects and concentrate on the novel intensional proofs. We were also able to add stack space costs to obtain a stronger result. The proofs were made more tractable by introducing compile-time checks for the 'sound and precise' cost labelling properties rather than proving that they are preserved throughout.

The overall statement of correctness says that the annotated program has the same behaviour as the input, and that for any suitably well-structured part of the execution (which we call *measurable*), the object code will execute the same behaviour taking precisely the time given by the cost annotations in the annotated source program.

In the next section we recall the structure of the compiler and make the overall statement more precise. Following that, in Section 3 we describe the statements we need to prove about the intermediate RTLabs programs for the back-end proofs. Section 4 covers the compiler passes which produce the annotated source program and Section 5 the rest of the transformations in the front-end. Then the compile-time checks for good cost labelling are detailed in Section 6 and the proofs that the structured traces required by the back-end exist are discussed in Section 7.

2 The compiler and its correctness statement

The uncertified prototype OCaml CerCo compiler was originally described in Deliverables 2.1 and 2.2. Its design was replicated in the formal Matita code, which was presented in Deliverables 3.2 and 4.2, for the front-end and back-end respectively.

The compiler uses a number of intermediate languages, as outlined the middle two lines of Figure 1. The upper line represents the front-end of the compiler, and the lower the back-end, finishing with Intel 8051 binary code. Not all of the front-end compiler passes introduce a new language, and Figure 2 presents a list of every pass involved.

The annotated source code is produced by the cost labelling phase. Note that there is a pass to replace C `switch` statements before labelling — we need to remove them because the simple form of labelling used in the formalised compiler is not quite capable of capturing their execution time costs, largely due to C's 'fall-through' behaviour where execution from one branch continues in the next unless there is an explicit `break`.

The cast removal phase which follows cost labelling simplifies expressions to prevent unnecessary arithmetic promotion, which is specified by the C standard but costly for an 8-bit target. The transformation to Cminor and subsequently RTLabs bear considerable resemblance

C (unformalised)
$\downarrow$ CIL parser (unformalised OCaml)
Clight
$\downarrow$ switch removal
$\downarrow$ labelling
$\downarrow$ cast removal
$\downarrow$ stack variable allocation and control structure simplification
Cminor
$\downarrow$ transform to RTL graph
RTLabs
$\downarrow$ check cost labelled properties of RTL graph
$\downarrow$ start of target specific back-end
$\vdots$

Figure 2: Front-end languages and compiler passes

to some passes of the CompCert compiler [4, 6], although we use a simpler Cminor where all loops use goto statements, and the RTLabs language retains a target-independent flavour. The back-end takes RTLabs code as input.

The whole compilation function returns the following information on success:

```
record compiler_output : Type[0] :=
 { c_labelled_object_code: labelled_object_code
 ; c_stack_cost: stack_cost_model
 ; c_max_stack: nat
 ; c_init_costlabel: costlabel
 ; c_labelled_clight: clight_program
 ; c_clight_cost_map: clight_cost_map
 }.
```

It consists of annotated 8051 object code, a mapping from function identifiers to the function's stack space usage, the space available for the stack after global variable allocation, a cost label covering the execution time for the initialisation of global variables and the call to the `main` function, the annotated source code, and finally a mapping from cost labels to actual execution time costs.

An OCaml pretty printer is used to provide a concrete version of the output code and annotated source code. In the case of the annotated source code, it also inserts the actual costs alongside the cost labels, and optionally adds a global cost variable and instrumentation to support further reasoning in external tools such as Frama-C.

2.1 Revisions to the prototype compiler

Our focus on intensional properties prompted us to consider whether we could incorporate stack space into the costs presented to the user. We only allocate one fixed-size frame per function, so modelling this was relatively simple. It is the only form of dynamic memory allocation provided by the compiler, so we were able to strengthen the statement of the goal to guarantee successful execution whenever the stack space obeys the `c_max_stack` bound calculated by subtracting the global variable requirements from the total memory available.

The cost labelling checks at the end of Figure 2 have been introduced to reduce the proof burden, and are described in Section 6.

The use of dependent types to capture simple intermediate language invariants makes every front-end pass a total function, except Clight to Cminor and the cost checks. Hence various well-formedness and type safety checks are performed only once between Clight and Cminor, and the invariants rule out any difficulties in the later stages. With the benefit of hindsight we would have included an initial checking phase to produce a 'well-formed' variant of Clight, conjecturing that this would simplify various parts of the proofs for the Clight stages which deal with potentially ill-formed code.

Following D2.2, we previously generated code for global variable initialisation in Cminor, for which we reserved a cost label to represent the execution time for initialisation. However, the back-end must also add an initial call to the main function, whose cost must also be accounted for, so we decided to move the initialisation code to the back-end and merge the costs.

2.2 Main correctness statement

Informally, our main intensional result links the time difference in a source code execution to the time difference in the object code, expressing the time for the source by summing the values for the cost labels in the trace, and the time for the target by a clock built in to the 8051 executable semantics.

The availability of precise timing information for 8501 implementations and the design of the compiler allow it to give exact time costs in terms of processor cycles, not just upper bounds. However, these exact results are only available if the subtrace we measure starts and ends at suitable points. In particular, pure computation with no observable effects may be reordered and moved past cost labels, so we cannot measure time between arbitrary statements in the program.

There is also a constraint on the subtraces that we measure due to the requirements of the correctness proof for the object code timing analysis. To be sure that the timings are assigned to the correct cost label, we need to know that each return from a function call must go to the correct return address. It is difficult to observe this property locally in the object code because it relies on much earlier stages in the compiler. To convey this information to the timing analysis extra structure is imposed on the subtraces, which is described in Section 3.

These restrictions are reflected in the subtraces that we give timing guarantees on; they must start at a cost label and end at the return of the enclosing function of the cost label[1]. A typical example of such a subtrace is the execution of an entire function from the cost label at the start of the function until it returns. We call such any such subtrace *measurable* if it (and the prefix of the trace from the start to the subtrace) can also be executed within the available stack space.

Now we can give the main intensional statement for the compiler. Given a *measurable* subtrace for a labelled Clight program, there is a subtrace of the 8051 object code program where the time differences match. Moreover, *observable* parts of the trace also match — these are the appearance of cost labels and function calls and returns.

More formally, the definition of this statement in Matita is

[1]We expect that this would generalise to more general subtraces by subtracting costs for unwanted measurable suffixes of a measurable subtrace.

```
definition simulates :=
  λp: compiler_output.
  let initial_status := initialise_status … (cm (c_labelled_object_code … p)) in
  ∀m1,m2.
   measurable Clight_pcs (c_labelled_clight … p) m1 m2
       (stack_sizes (c_stack_cost … p)) (c_max_stack … p) →
  ∀c1,c2.
   clock_after Clight_pcs (c_labelled_clight … p) m1 (c_clight_cost_map … p) = OK … c1 →
   clock_after Clight_pcs (c_labelled_clight … p) (m1+m2) (c_clight_cost_map … p) = OK … c2 →
  ∃n1,n2.
   observables Clight_pcs (c_labelled_clight … p) m1 m2 =
     observables (OC_preclassified_system (c_labelled_object_code … p))
          (c_labelled_object_code … p) n1 n2
  ∧
   clock ?? (execute (n1+n2) ? initial_status) =
     clock ?? (execute n1 ? initial_status) + (c2-c1).
```

where the `measurable`, `clock_after` and `observables` definitions are generic definitions for multiple languages; in this case the `Clight_pcs` record applies them to Clight programs.

There is a second part to the statement, which says that the initial processing of the input program to produce the cost labelled version does not affect the semantics of the program:

```
  ∀input_program,output.
  compile input_program = return output →
  not_wrong … (exec_inf … clight_fullexec input_program) →
  sim_with_labels
   (exec_inf … clight_fullexec input_program)
   (exec_inf … clight_fullexec (c_labelled_clight … output))
```

That is, any successful compilation produces a labelled program that has identical behaviour to the original, so long as there is no 'undefined behaviour'.

Note that this statement provides full functional correctness, including preservation of (non-)termination. The intensional result above does not do this directly — it does not guarantee the same result or same termination. There are two mitigating factors, however: first, to prove the intensional property you need local simulation results — these can be pieced together to form full behavioural equivalence, only time constraints have prevented us from doing so. Second, if we wish to confirm a result, termination, or non-termination we could add an observable witness, such as a function that is only called if the correct result is given. The intensional result guarantees that the observable witness is preserved, so the program must behave correctly.

These two results are combined in the the `correct` theorem in the file `correctness.ma`.

3 Correctness statement for the front-end

The essential parts of the intensional proof were outlined during work on a toy compiler in Task 2.1 [1, 2]. These are

1. functional correctness, in particular preserving the trace of cost labels,
2. the *soundness* and *precision* of the cost labelling on the object code, and

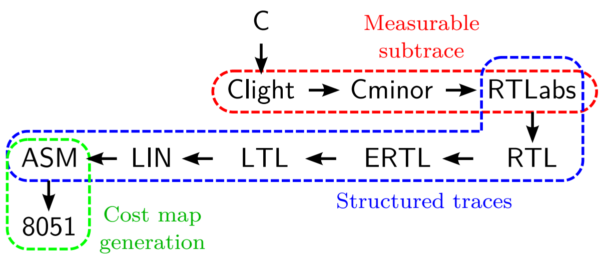

Figure 3: The compiler and proof outline

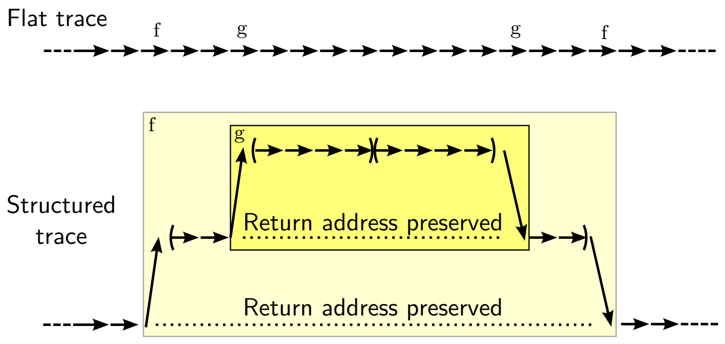

Figure 4: Nesting of functions in structured traces

3. the timing analysis on the object code produces a correct mapping from cost labels to time.

However, that toy development did not include function calls. For the full CerCo compiler we also need to maintain the invariant that functions return to the correct program location in the caller, as we mentioned in the previous section. During work on the back-end timing analysis (describe in more detail in the companion deliverable, D4.4) the notion of a *structured trace* was developed to enforce this return property, and also most of the cost labelling properties too.

Jointly, we generalised the structured traces to apply to any of the intermediate languages which have some idea of program counter. This means that they are introduced part way through the compiler, see Figure 3. Proving that a structured trace can be constructed at RTLabs has several virtues:

- This is the first language where every operation has its own unique, easily addressable, statement.
- Function calls and returns are still handled implicitly in the language and so the structural properties are ensured by the semantics.
- Many of the back-end languages from RTL onwards share a common core set of definitions, and using structured traces throughout increases this uniformity.

A structured trace is a mutually inductive data type which contains the steps from a normal program trace, but arranged into a nested structure which groups entire function calls together and aggregates individual steps between cost labels (or between the final cost label and the return from the function), see Figure 4. This captures the idea that the cost labels only represent costs *within* a function — calls to other functions are accounted for in the nested trace for their execution, and we can locally regard function calls as a single step.

These structured traces form the core part of the intermediate results that we must prove so that the back-end can complete the main intensional result stated above. In full, we provide the back-end with

1. A normal trace of the **prefix** of the program's execution before reaching the measurable subtrace. (This needs to be preserved so that we know that the stack space consumed is correct, and to set up the simulation results.)
2. The **structured trace** corresponding to the measurable subtrace.
3. An additional property about the structured trace that no 'program counter' is **repeated** between cost labels. Together with the structure in the trace, this takes over from showing that cost labelling is sound and precise.
4. A proof that the **observables** have been preserved.
5. A proof that the **stack limit** is still observed by the prefix and the structure trace. (This is largely a consequence of the preservation of observables.)

The `front_end_correct` lemma in the `correctness.ma` file provides a record containing these.

Following the outline in Figure 3, we will first deal with the transformations in Clight that produce the source program with cost labels, then show that measurable traces can be lifted to RTLabs, and finally show that we can construct the properties listed above ready for the back-end proofs.

4 Input code to cost labelled program

As explained on page 4, the costs of complex C `switch` statements cannot be represented with the simple labelling used in the formalised compiler. Our first pass replaces these statements with simpler C code, allowing our second pass to perform the cost labelling. We show that the behaviour of programs is unchanged by these passes using forward simulations[2].

4.1 Switch removal

We compile away `switch` statements into more basic Clight code. Note that this transformation does not necessarily deteriorate the efficiency of the generated code. For instance, compilers such as GCC introduce balanced trees of "if-then-else" constructs for small switches. However, our implementation strategy is much simpler. Let us consider the following input statement.

```
switch(e) {
case v1:
  stmt1;
case v2:
  stmt2;
```

[2]All of our languages are deterministic, which can be seen directly from their executable definitions. Thus we know that forward simulations are sufficient because the target cannot have any other behaviour.

```
default:
  stmt_default;
}
```

Note that stmt1, stmt2, ...stmt_default may contain `break` statements, which have the effect of exiting the switch statement. In the absence of break, the execution falls through each case sequentially. In our implementation, we produce an equivalent sequence of "if-then" chained by gotos:

```
fresh = e;
if(fresh == v1) {
  ⟦stmt1⟧;
  goto lbl_case2;
};
if(fresh == v2) {
  lbl_case2:
  ⟦stmt2⟧;
  goto lbl_case2;
};
⟦stmt_default⟧;
exit_label:
```

The proof had to tackle the following points:

- the source and target memories are not the same (due to the fresh variable),
- the flow of control is changed in a non-local way (e.g. **goto** instead of **break**).

In order to tackle the first point, we implemented a version of memory extensions similar to those of CompCert.

For the simulation we decided to prove a sufficient amount to give us confidence in the definitions and approach, but to curtail the proof because this pass does not contribute to the intensional correctness result. We tackled several simple cases, that do not interact with the switch removal per se, to show that the definitions were usable, and part of the switch case to check that the approach is reasonable. This comprises propagating the memory extension through each statement (except switch), as well as various invariants that are needed for the switch case (in particular, freshness hypotheses). The details of the evaluation process for the source switch statement and its target counterpart can be found in the file `switchRemoval.ma`, along more details on the transformation itself.

Proving the correctness of the second point would require reasoning on the semantics of `goto` statements. In the Clight semantics, this is implemented as a function-wide lookup of the target label. The invariant we would need is the fact that a global label lookup on a freshly created goto is equivalent to a local lookup. This would in turn require the propagation of some freshness hypotheses on labels. As discussed, we decided to omit this part of the correctness proof.

4.2 Cost labelling

The simulation for the cost labelling pass is the simplest in the front-end. The main argument is that any step of the source program is simulated by the same step of the labelled one, plus any extra steps for the added cost labels. The extra instructions do not change the memory or local environments, although we have to keep track of the extra instructions that appear in continuations, for example during the execution of a `while` loop.

We do not attempt to capture any cost properties of the labelling[3] in the simulation proof, which allows the proof to be oblivious to the choice of cost labels. Hence we do not have to reason about the threading of name generation through the labelling function, greatly reducing the amount of effort required.

5 Finding corresponding measurable subtraces

There follow the three main passes of the front-end:

1. simplification of casts in Clight code
2. Clight to Cminor translation, performing stack variable allocation and simplifying control structures
3. transformation to RTLabs control flow graph

We have taken a common approach to each pass: first we build (or axiomatise) forward simulation results that are similar to normal compiler proofs, but which are slightly more fine-grained so that we can see that the call structure and relative placement of cost labels is preserved.

Then we instantiate a general result which shows that we can find a *measurable* subtrace in the target of the pass that corresponds to the measurable subtrace in the source. By repeated application of this result we can find a measurable subtrace of the execution of the RTLabs code, suitable for the construction of a structured trace (see Section 7). This is essentially an extra layer on top of the simulation proofs that provides us with the additional information required for our intensional correctness proof.

5.1 Generic measurable subtrace lifting proof

Our generic proof is parametrised on a record containing small-step semantics for the source and target language, a classification of states (the same form of classification is used when defining structured traces), a simulation relation which respects the classification and cost labelling and four simulation results. The simulations are split by the starting state's classification and whether it is a cost label, which will allow us to observe that the call structure is preserved. They are:

1. a step from a 'normal' state (which is not classified as a call or return) which is not a cost label is simulated by zero or more 'normal' steps;
2. a step from a 'call' state followed by a cost label step is simulated by a step from a 'call' state, a corresponding label step, then zero or more 'normal' steps;
3. a step from a 'call' state not followed by a cost label similarly (note that this case cannot occur in a well-labelled program, but we do not have enough information locally to exploit this); and

[3]We describe how the cost properties are established in Section 6.

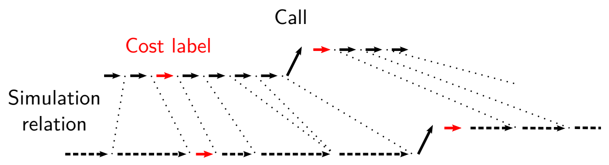

Figure 5: Tiling of simulation for a measurable subtrace

4. a cost label step is simulated by a cost label step.

Finally, we need to know that a successfully translated program will have an initial state in the simulation relation with the original program's initial state.

The back-end has similar requirements for lifting simulations to structured traces. Fortunately, our treatment of calls and returns can be slightly simpler because we have special call and return states that correspond to function entry and return that are separate from the actual instructions. This was originally inherited from our port of CompCert's Clight semantics, but proves useful here because we only need to consider adding extra steps *after* a call or return state, because the instruction step deals with extra steps that occur before. The back-end makes all of the call and return machinery explicit, and thus needs more complex statements about extra steps before and after each call and return.

To find the measurable subtrace in the target program's execution we walk along the original program's execution trace applying the appropriate simulation result by induction on the number of steps. While the number of steps taken varies, the overall structure is preserved, as illustrated in Figure 5. By preserving the structure we also maintain the same intensional observables. One delicate point is that the cost label following a call must remain directly afterwards[4] (both in the program code and in the execution trace), even if we introduce extra steps, for example to store parameters in memory in Cminor. Thus we have a version of the call simulation that deals with both the call and the cost label in one result.

In addition to the subtrace we are interested in measuring, we must prove that the earlier part of the trace is also preserved in order to use the simulation from the initial state. This proof also guarantees that we do not run out of stack space before the subtrace we are interested in. The lemmas for this prefix and the measurable subtrace are similar, following the pattern above. However, the measurable subtrace also requires us to rebuild the termination proof. This is defined recursively:

```
let rec will_return_aux C (depth:nat)
                             (trace:list (cs_state … C × trace)) on trace : bool :=
match trace with
[ nil ⇒ false
| cons h tl ⇒
  let ⟨s,tr⟩ := h in
  match cs_classify C s with
  [ cl_call ⇒ will_return_aux C (S depth) tl
  | cl_return ⇒
      match depth with
      [ O ⇒ match tl with [ nil ⇒ true | _ ⇒ false ]
      | S d ⇒ will_return_aux C d tl
```

[4]The prototype compiler allowed some straight-line code to appear before the cost label until a later stage of the compiler, but we must move the requirement forward to fit with the structured traces.

```
        ]
    | _ ⇒ will_return_aux C depth tl
    ]
  ].
```

The depth is the number of return states we need to see before we have returned to the original function (initially zero) and trace the measurable subtrace obtained from the running the semantics for the correct number of steps. This definition unfolds tail recursively for each step, and once the corresponding simulation result has been applied a new one for the target can be asserted by unfolding and applying the induction hypothesis on the shorter trace.

Combining the lemmas about the prefix and the measurable subtrace requires a little care because the states joining the two might not be related in the simulation. In particular, if the measurable subtrace starts from the cost label at the beginning of the function there may be some extra instructions in the target code to execute to complete function entry before the states are back in the relation. Hence we carefully phrased the lemmas to allow for such extra steps.

Together, these then gives us an overall result for any simulation fitting the requirements above (contained in the meas_sim record):

```
theorem measured_subtrace_preserved :
  ∀MS:meas_sim.
  ∀p1,p2,m,n,stack_cost,max.
  ms_compiled MS p1 p2 →
  measurable (ms_C1 MS) p1 m n stack_cost max →
  ∃m',n'.
    measurable (ms_C2 MS) p2 m' n' stack_cost max ∧
    observables (ms_C1 MS) p1 m n = observables (ms_C2 MS) p2 m' n'.
```

The stack space requirement that is embedded in measurable is a consequence of the preservation of observables, because it is determined by the functions called and returned from, which are observable.

5.2 Simulation results for each pass

We now consider the simulation results for the passes, each of which is used to instantiate the measured_subtrace_preserved theorem to construct the measurable subtrace for the next language.

5.2.1 Cast simplification

The parser used in CerCo introduces a lot of explicit type casts. If left as they are, these constructs can greatly hamper the quality of the generated code – especially as the architecture we consider is an 8-bit one. In Clight, casts are expressions. Hence, most of the work of this transformation proceeds on expressions. The transformation proceeds by recursively trying to coerce an expression to a particular integer type, which is in practice smaller than the original one. This functionality is implemented by two mutually recursive functions whose signature is the following.

```
let rec simplify_expr (e:expr) (target_sz:intsize) (target_sg:signedness)
  : Σresult:bool×expr.
    ∀ge,en,m. simplify_inv ge en m e (\snd result) target_sz target_sg (\fst result) := ...
```

```
and simplify_inside (e:expr) : Σresult:expr. conservation e result := ...
```

The simplify_inside acts as a wrapper for simplify_expr. Whenever simplify_inside encounters a Ecast expression, it tries to coerce the sub-expression to the desired type using simplify_expr, which tries to perform the actual coercion. In return, simplify_expr calls back simplify_inside in some particular positions, where we decided to be conservative in order to simplify the proofs. However, the current design allows to incrementally revert to a more aggressive version, by replacing recursive calls to simplify_inside by calls to simplify_expr *and* proving the corresponding invariants – where possible.

The simplify_inv invariant encodes either the conservation of the semantics during the transformation corresponding to the failure of the coercion (Inv_eq constructor), or the successful downcast of the considered expression to the target type (Inv_coerce_ok).

```
inductive simplify_inv
  (ge : genv) (en : env) (m : mem)
  (e1 : expr) (e2 : expr) (target_sz : intsize) (target_sg : signedness) : bool → Prop :=
| Inv_eq : ∀result_flag. ...
     simplify_inv ge en m e1 e2 target_sz target_sg result_flag
| Inv_coerce_ok : ∀src_sz,src_sg.
     typeof e1 = Tint src_sz src_sg →
     typeof e2 = Tint target_sz target_sg →
     smaller_integer_val src_sz target_sz src_sg (exec_expr ge en m e1) (exec_expr ge en m e2) →
     simplify_inv ge en m e1 e2 target_sz target_sg true.
```

The conservation invariant for simplify_inside simply states the conservation of the semantics, as in the Inv_eq constructor of the previous invariant.

```
definition conservation := λe,result. ∀ge,en,m.
          res_sim ? (exec_expr ge en m e) (exec_expr ge en m result)
       ∧ res_sim ? (exec_lvalue ge en m e) (exec_lvalue ge en m result)
       ∧ typeof e = typeof result.
```

This invariant is then easily lifted to statement evaluations. The main problem encountered with this particular pass was dealing with inconsistently typed programs, a canonical case being a particular integer constant of a certain size typed with another size. This prompted the need to introduce numerous type checks, making both the implementation and the proof more complex, even though more comprehensive checks are made in the next stage.

5.2.2 Clight to Cminor

This pass is the last one operating on the Clight language. Its input is a full Clight program, and its output is a Cminor program. Note that we do not use an equivalent of CompCert's C#minor language: we translate directly to a variant of Cminor. This presents the advantage of not requiring the special loop constructs, nor the explicit block structure. Another salient point of our approach is that a significant number of the properties needed for the simulation proof were directly encoded in dependently typed translation functions. In particular, freshness conditions and well-typedness conditions are included. The main effects of the transformation from Clight to Cminor are listed below.

- Variables are classified as being either globals, stack-allocated locals or potentially register-allocated locals. The value of register-allocated local variables is moved out of the modelled memory and stored in a dedicated environment.

- In Clight, each local variable has a dedicated memory block, whereas stack-allocated locals are bundled together on a function-by-function basis.
- Loops are converted to jumps.

The first two points require memory injections which are more flexible that those needed in the switch removal case. In the remainder of this section, we briefly discuss our implementation of memory injections, and then the simulation proof.

Memory injections. Our memory injections are modelled after the work of Blazy & Leroy. However, the corresponding paper is based on the first version of the CompCert memory model [7], whereas we use a much more concrete model, allowing byte-level manipulations (as in the later version of CompCert's memory model). We proved roughly 80 % of the required lemmas. Notably, some of the difficulties encountered were due to overly relaxed conditions on pointer validity (fixed during development). Some more side conditions had to be added to take care of possible overflows when converting from $\mathbb{Z}$ block bounds to 16 bit pointer offsets (in practice, such overflows only occur in edge cases that are easily ruled out – but this fact is not visible in memory injections). Concretely, some of the lemmas on the preservation of simulation of loads after writes were axiomatised, due to a lack of time.

Simulation proof. We proved the simulation result for expressions and a representative selection of statements. In particular we tackled `while` statements to ensure that we correctly translate loops because our approach differs from CompCert by converting directly to Cminor `goto`s rather than maintaining a notion of loop in Cminor. We also have a partial proof for function entry, covering the setup of the memory injection, but not function exit. Exits, and the remaining statements, have been axiomatised.

Careful management of the proof state was required because proof terms are embedded in Cminor code to show that invariants are respected. These proof terms appear in the proof state when inverting the translation functions, and they can be large and awkward. While generalising them away is usually sufficient, it can be difficult when they appear under a binder.

5.2.3 Cminor to RTLabs

The translation from Cminor to RTLabs is a fairly routine control flow graph (CFG) construction. As such, we chose to axiomatise the associated extensional simulation results. However, we did prove several properties of the generated programs:

- All statements are type correct with respect to the declared pseudo-register type environment.
- The CFG is closed, and has a distinguished entry node and a unique exit node.

These properties rely on similar properties about type safety and the presence of `goto`-labels for Cminor programs which are checked at the preceding stage. As a result, this transformation is total and any compilation failures must occur when the corresponding Clight source is available and a better error message can be generated.

The proof obligations for these properties include many instances of graph inclusion. We automated these proofs using a small amount of reflection, making the obligations much easier to handle. One drawback to enforcing invariants throughout is that temporarily breaking them can be awkward. For example, `return` statements were originally used as placeholders for `goto` destinations that had not yet been translated. However, this made establishing the single exit node property rather difficult, and a different placeholder was chosen instead. In other circumstances it is possible to prove a more complex invariant then simplify it at the end of the transformation.

6 Checking cost labelling properties

Ideally, we would provide proofs that the cost labelling pass always produces programs that are soundly and precisely labelled and that each subsequent pass preserves these properties. This would match our use of dependent types to eliminate impossible sources of errors during compilation, in particular retaining intermediate language type information.

However, given the limited amount of time available we realised that implementing a compile-time check for a sound and precise labelling of the RTLabs intermediate code would reduce the proof burden considerably. This is similar in spirit to the use of translation validation in certified compilation, which makes a similar trade-off between the potential for compile-time failure and the volume of proof required.

The check cannot be pushed into a later stage of the compiler because much of the information is embedded into the structured traces. However, if an alternative method was used to show that function returns in the compiled code are sufficiently well-behaved, then we could consider pushing the cost property checks into the timing analysis itself. We leave this as a possible area for future work.

6.1 Implementation and correctness

For a cost labelling to be sound and precise we need a cost label at the start of each function, after each branch and at least one in every loop. The first two parts are trivial to check by examining the code. In RTLabs the last part is specified by saying that there is a bound on the number of successive instruction nodes in the CFG that you can follow before you encounter a cost label, and checking this is more difficult.

The implementation progresses through the set of nodes in the graph, following successors until a cost label is found or a label-free cycle is discovered (in which case the property does not hold and we return an error). This is made easier by the prior knowledge that every successor of a branch instruction is a cost label, so we do not need to search each branch. When a label is found, we remove the chain of program counters from the set and continue from another node in the set until it is empty, at which point we know that there is a bound for every node in the graph.

Directly reasoning about the function that implements this procedure would be rather awkward, so an inductive specification of a single step of its behaviour was written and proved to match the implementation. This was then used to prove the implementation sound and complete.

While we have not attempted to prove that the cost labelled properties are established and preserved earlier in the compiler, we expect that the effort for the Cminor to RTLabs stage alone would be similar to the work outlined above, because it involves the change from requiring a cost label at particular positions to requiring cost labels to break loops in the

CFG. As there are another three passes to consider (including the labelling itself), we believe that using the check above is much simpler overall.

7 Existence of a structured trace

The *structured trace* idea introduced in Section 3 enriches the execution trace of a program by nesting function calls in a mixed-step style and embedding the cost labelling properties of the program. See Figure 4 on page 8 for an illustration of a structured trace. It was originally designed to support the proof of correctness for the timing analysis of the object code in the back-end, then generalised to provide a common structure to use from the end of the front-end to the object code.

To make the definition generic we abstract over the semantics of the language,

```
record abstract_status : Type[1] :=
  { as_status :> Type[0]
  ; as_execute : relation as_status
  ; as_pc : DeqSet
  ; as_pc_of : as_status → as_pc
  ; as_classify : as_status → status_class
  ; as_label_of_pc : as_pc → option costlabel
  ; as_after_return : (Σs:as_status. as_classify s = cl_call) → as_status → Prop
  ; as_result: as_status → option int
  ; as_call_ident : (Σs:as_status.as_classify s = cl_call) → ident
  ; as_tailcall_ident : (Σs:as_status.as_classify s = cl_tailcall) → ident
  }.
```

which requires a type of states, an execution relation[5], some notion of abstract program counter with decidable equality, the classification of states, and functions to extract the observable intensional information (cost labels and the identity of functions that are called). The `as_after_return` property links the state before a function call with the state after return, providing the evidence that execution returns to the correct place. The precise form varies between stages; in RTLabs it insists the CFG, the pointer to the CFG node to execute next, and some call stack information is preserved.

The structured traces are defined using three mutually inductive types. The core data structure is `trace_any_label`, which captures some straight-line execution until the next cost label or the return from the enclosing function. Any function calls are embedded as a single step, with its own trace nested inside and the before and after states linked by `as_after_return`; and states classified as a 'jump' (in particular branches) must be followed by a cost label.

The second type, `trace_label_label`, is a `trace_any_label` where the initial state is cost labelled. Thus a trace in this type identifies a series of steps whose cost is entirely accounted for by the label at the start.

Finally, `trace_label_return` is a sequence of `trace_label_label` values which end in the return from the function. These correspond to a measurable subtrace, and in particular include executions of an entire function call (and so are used for the nested calls in `trace_any_label`).

[5]All of our semantics are executable, but using a relation was simpler in the abstraction.

7.1 Construction

The construction of the structured trace replaces syntactic cost labelling properties, which place requirements on where labels appear in the program, with semantic properties that constrain the execution traces of the program. The construction begins by defining versions of the sound and precise labelling properties on states and global environments (for the code that appears in each of them) rather than whole programs, and showing that these are preserved by steps of the RTLabs semantics.

Then we show that each cost labelling property required by the definition of structured traces is locally satisfied. These proofs are broken up by the classification of states. Similarly, we prove a step-by-step stack preservation result, which states that the RTLabs semantics never changes the lower parts of the stack.

The core part of the construction of a structured trace is to use the proof of termination from the measurable trace (defined on page 12) to 'fold up' the execution into the nested form. The results outlined above fill in the proof obligations for the cost labelling properties and the stack preservation result shows that calls return to the correct location.

The structured trace alone is not sufficient to capture the property that the program is soundly labelled. While the structured trace guarantees termination, it still permits a loop to be executed a finite number of times without encountering a cost label. We eliminate this by proving that no 'program counter' repeats within any `trace_any_label` section by showing that it is incompatible with the property that there is a bound on the number of successor instructions you can follow in the CFG before you encounter a cost label (from Section 6.1).

7.1.1 Complete execution structured traces

The development of the construction above started relatively early, before the measurable subtrace preservation proofs. To be confident that the traces were well-formed at that time, we also developed a complete execution form that embeds the traces above. This includes non-terminating program executions, where an infinite number of the terminating structured traces are embedded. This construction confirmed that our definition of structured traces was consistent, although we later found that we did not require the whole execution version for the compiler correctness results.

To construct these we need to know whether each function call will eventually terminate, requiring the use of the excluded middle. This classical reasoning is local to the construction of whole program traces and is not necessary for our main results.

8 Conclusion

In combination with the work on the CerCo back-end and by concentrating on the novel intensional parts of the proof, we have shown that it is possible to construct certifying compilers that correctly report execution time and stack space costs. The layering of intensional correctness proofs on top of normal simulation results provides a useful separation of concerns, and could permit the reuse of existing results.

A Files

The following table gives a high-level overview of the Matita source files in Deliverable 3.4:

`compiler.ma`	Top-level compiler definitions, in particular `front_end`, and the whole compiler definition `compile`.
`correctness.ma`	Correctness results: `front_end_correct` and `correct`, respectively.
`Clight/*`	Clight: proofs for switch removal, cost labelling, cast simplification and conversion to Cminor.
`Cminor/*`	Cminor: axioms of conversion to RTLabs.
`RTLabs/*`	RTLabs: definitions and proofs for compile-time cost labelling checks, construction of structured traces.
`common/Measurable.ma`	Definitions for measurable subtraces.
`common/FEMeasurable.ma`	Generic measurable subtrace lifting proof.
`common/*`	Other common definitions relevant to many parts of the compiler and proof.
`utilities/*`	General purpose definitions used throughout, including extensions to the standard Matita library.

References

[1] Roberto M. Amadio, Nicolas Ayache, Yann Régis-Gianas, Kayvan Memarian, and Ronan Saillard. Compiler design and intermediate languages. Deliverable 2.1, Project FP7-ICT-2009-C-243881 CerCo.

[2] Nicolas Ayache, Roberto Amadio, and Yann Régis-Gianas. Certifying and reasoning on cost annotations in C programs. In Mariëlle Stoelinga and Ralf Pinger, editors, *Formal Methods for Industrial Critical Systems*, volume 7437 of *Lecture Notes in Computer Science*, pages 32–46. Springer Berlin / Heidelberg, 2012. 10.1007/978-3-642-32469-7_3.

[3] Nicolas Ayache, Roberto M. Amadio, and Yann Régis-Gianas. Prototype implementation. Deliverable 2.2, Project FP7-ICT-2009-C-243881 CerCo.

[4] Sandrine Blazy and Xavier Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3):263–288, 2009.

[5] Brian Campbell. CIC encoding: Front-end. Deliverable 3.2, Project FP7-ICT-2009-C-243881 CerCo.

[6] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[7] Xavier Leroy and Sandrine Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41(1):1–31, 2008.

[8] Dominic P. Mulligan and Claudio Sacerdoti Coen. CIC encoding: Back-end. Deliverable 4.2, Project FP7-ICT-2009-C-243881 CerCo.