CerCo

INFORMATION AND COMMUNICATION
TECHNOLOGIES
(ICT)
PROGRAMME

Project FP7-ICT-2009-C-243881 CerCo

Report n. D4.4
Back-end Correctness Proof

Version 1.0

Main Authors:
Jaap Boender, Dominic P. Mulligan, Mauro Piccolo,
Claudio Sacerdoti Coen and Paolo Tranquilli

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

Summary The deliverable D4.4 is composed of the following parts:

1. This summary.
2. The paper [1].
3. The paper [2].
4. The paper [3].
5. The paper [4].

This document and all the related matita developments can be downloaded at the page:

`http://cerco.cs.unibo.it/`

References

[1] Paolo Tranquilli and Claudio Sacerdoti Coen. Certification of the preservation of structure by a compiler's back-end pass. Internal report, 2013.

[2] Mauro Piccolo and Paolo Tranquilli. Certifying the back end pass of the cerco annotating compiler. Internal report, 2013.

[3] Dominic P. Mulligan and Claudio Sacerdoti Coen. On the correctness of an optimising assembler for the intel mcs-51 microprocessor. In Chris Hawblitzel and Dale Miller, editors, *CPP*, volume 7679 of *Lecture Notes in Computer Science*, pages 43–59. Springer, 2012.

[4] Jaap Boender and Claudio Sacerdoti Coen. On the correctness of a branch displacement algorithm. *CoRR*, abs/1209.5920, 2012.

Aim The aim of WP4 is is to build the trusted version of the compiler back-end, from the intermediate RTLabs language down to assembly. The development is made in matita, and it allows the trusted compiler to be extracted to OCaml.

The main planned contributions of deliverable D4.4 are formally checked proof of the semantics correspondence between the intermediate code and the target code, and of the preservation/modification of the control flow for complexity analysis.

Preservation of structure In [1] we present a genric approach to proving a forward simulation preserving the intensional structure of traces.

When a language starts to be able to meddle with return addresses that live in memory, the call structure is no more guaranteed to be preserved after the high-level, structured languages. This has little meaning as far as pure extensional semantic preservation is required—after all, if the source language meddles with the call structure there is no problem as long as the target language will follow. However in our approach we have cost labels spanning multiple calls, so that the cost of what follows a call is "paid" in advance. This has no hope of being correct if there is no guarantee that upon return from a call we land after the call itself.

In this part of the deliverable we will present our approach to this problem, which goes by including in semantic traces structural conditions, and giving generic proof obligations that enrich the classic step-by-step extensional preservation proof with the necessary hypotheses to ensure the preservation of the call and label structures. This approach can be applied on all passes starting from the RTLabs to RTL down to the assembly one.

The back-end correctness proof In [2], we give an outline of the actual correctness proof for the passes from RTLabs down to assembly. We skip the details of the extensional parts of each pass and we concentrate on two main aspects: how we deal with stack and how we ensure the conditions explained in [1] in the passes involving graph languages.

The assembler correctness proof In [3], we present a proof of correctness of our assembler to object code, given a correct policy for branch expansion (see next paragraph).

A branch expansion policy maker In [4] we finally present our algorithm for branch expansion, that is how generic assembly jumps are expanded to the different type of jumps available in the 8051 architecture (short, absolute and long). The correctness of this algorithm is proved, and is what required by the correctness of the whole assembler.

Certification of the Preservation of Structure by a Compiler's Back-end Pass*

Paolo Tranquilli and Claudio Sacerdoti Coen

Department of Computer Science and Engineering, University of Bologna,
Paolo.Tranquilli@unibo.it, Claudio.SacerdotiCoen@unibo.it

Abstract. The labelling approach is a technique to lift cost models for non-functional properties of programs from the object code to the source code. It is based on the preservation of the structure of the high level program in every intermediate language used by the compiler. Such structure is captured by observables that are added to the semantics and that needs to be preserved by the forward simulation proof of correctness of the compiler. Additional special observables are required for function calls. In this paper we present a generic forward simulation proof that preserves all these observables. The proof statement is based on a new mechanised semantics that traces the structure of execution when the language is unstructured. The generic semantics and simulation proof have been mechanised in the interactive theorem prover Matita.

1 Introduction

The *labelling approach* has been introduced in [5] as a technique to *lift* cost models for non-functional properties of programs from the object code to the source code. Examples of non-functional properties are execution time, amount of stack/heap space consumed and energy required for communication. The basic idea of the approach is that it is impossible to provide a *uniform* cost model for an high level language that is preserved *precisely* by a compiler. For instance, two instances of an assignment $x = y$ in the source code can be compiled very differently according to the place (registers vs stack) where x and y are stored at the moment of execution. Therefore a precise cost model must assign a different cost to every occurrence, and the exact cost can only be known after compilation.

According to the labelling approach, the compiler is free to compile and optimise the source code without any major restriction, but it must keep trace of what happens to basic blocks during the compilation. The cost model is then computed on the object code. It assigns a cost to every basic block. Finally, the compiler propagates back the cost model to the source level, assigning a cost to each basic block of the source code.

Implementing the labelling approach in a certified compiler allows to reason formally on the high level source code of a program to prove non-functional

* The project CerCo acknowledges the financial support of the Future and Emerging Technologies (FET) programme within the Seventh Framework Programme for Research of the European Commission, under FET-Open grant number: 243881

properties that are granted to be preserved by the compiler itself. The trusted code base is then reduced to 1) the interactive theorem prover (or its kernel) used in the certification of the compiler and 2) the software used to certify the property on the source language, that can be itself certified further reducing the trusted code base. In [5] the authors provide an example of a simple certified compiler that implements the labelling approach for the imperative `While` language [11], that does not have pointers and function calls.

The labelling approach has been shown to scale to more interesting scenarios. In particular in [2] it has been applied to a functional language and in [13] it has been shown that the approach can be slightly complicated to handle loop optimisations and, more generally, program optimisations that do not preserve the structure of basic blocks. On-going work also shows that the labelling approach is also compatible with the complex analyses required to obtain a cost model for object code on processors that implement advanced features like pipelining, superscalar architectures and caches.

In the European Project CerCo (Certified Complexity [1]) [1] we are certifying a labelling approach based compiler for a large subset of C to 8051 object code. The compiler is moderately optimising and implements a compilation chain that is largely inspired to that of CompCert [7,9]. Compared to work done in [5], the main novelty and source of difficulties is due to the presence of function calls. Surprisingly, the addition of function calls require a revisitation of the proof technique given in [5]. In particular, at the core of the labelling approach there is a forward simulation proof that, in the case of `While`, is only minimally more complex than the proof required for the preservation of the functional properties only. In the case of a programming language with function calls, instead, it turns out that the forward simulation proof for the back-end languages must grant a whole new set of invariants.

In this paper we present a formalisation in the Matita interactive theorem prover [3,4] of a generic version of the simulation proof required for unstructured languages. All back-end languages of the CerCo compiler are unstructured languages, so the proof covers half of the correctness of the compiler. The statement of the generic proof is based on a new semantics for imperative unstructured languages that is based on *structured traces* and that restores the preservation of structure in the observables of the semantics. The generic proof allows to almost completely split the part of the simulation that deals with functional properties only from the part that deals with the preservation of structure.

The plan of this paper is the following. In Section 2 we sketch the labelling method and the problems derived from the application to languages with function calls. In Section 3 we introduce a generic description of an unstructured imperative language and the corresponding structured traces (the novel semantics). In Section 4 we describe the forward simulation proof. Conclusions and future works are in Section 5

[1] `http://cerco.cs.unibo.it`

2 The labelling approach

We briefly sketch here a simplified version of the labelling approach as introduced in [5]. The simplification strengthens the sufficient conditions given in [5] to allow a simpler explanation. The simplified conditions given here are also used in the CerCo compiler to simplify the proof.

Let $\mathcal{P}$ be a programming language whose semantics is given in terms of observables: a run of a program yields a finite or infinite stream of observables. We also assume for the time being that function calls are not available in $\mathcal{P}$. We want to associate a cost model to a program P written in $\mathcal{P}$. The first step is to extend the syntax of $\mathcal{P}$ with a new construct `emit L` where L is a label distinct from all observables of $\mathcal{P}$. The semantics of `emit L` is the emission of the observable `L` that is meant to signal the beginning of a basic block.

There exists an automatic procedure that injects into the program P an `emit L` at the beginning of each basic block, using a fresh `L` for each block. In particular, the bodies of loops, both branches of `if-then-else`s and the targets of `goto`s must all start with an emission statement.

Let now C be a compiler from $\mathcal{P}$ to the object code $\mathcal{M}$, that is organised in passes. Let $\mathcal{Q}_i$ be the i-th intermediate language used by the compiler. We can easily extend every intermediate language (and its semantics) with an `emit L` statement as we did for $\mathcal{P}$. The same is possible for $\mathcal{M}$ too, with the additional difficulty that the syntax of object code is given as a sequence of bytes. The injection of an emission statement in the object code can be done using a map that maps two consecutive code addresses with the statement. The intended semantics is that, if $(pc_1, pc_2) \mapsto$ `emit L` then the observable `L` is emitted after the execution of the instruction stored at pc_1 and before the execution of the instruction stored at pc_2. The two program counters are necessary because the instruction stored at pc_1 can have multiple possible successors (e.g. in case of a conditional branch or an indirect call). Dually, the instruction stored at pc_2 can have multiple possible predecessors (e.g. if it is the target of a jump).

The compiler, to be functionally correct, must preserve the observational equivalence, i.e. executing the program after each compiler pass should yield the same stream of observables. After the injection of emission statements, observables now capture both functional and non-functional behaviours. This correctness property is called in the literature a forward simulation and is sufficient for correctness when the target language is deterministic [8]. We also require a stronger, non-functional preservation property: after each pass all basic blocks must start with an emission statement, and all labels `L` must be unique.

Now let M be the object code obtained for the program P. Let us suppose that we can statically inspect the code M and associate to each basic block a cost (e.g. the number of clock cycles required to execute all instructions in the basic block, or an upper bound to that time). Every basic block is labelled with an unique label `L`, thus we can actually associate the cost to `L`. Let call it k(`L`).

The function k is defined as the cost model for the object code control blocks. It can be equally used as well as the cost model for the source control blocks. Indeed, if the semantics of P is the stream $L_1 L_2 \ldots$, then, because of forward

simulation, the semantics of M is also $L_1 L_2 \ldots$ and its actual execution cost is $\Sigma_i k(L_i)$ because every instruction belongs to a control block and every control block is labelled. Thus it is correct to say that the execution cost of P is also $\Sigma_i k(L_i)$. In other words, we have obtained a cost model k for the blocks of the high level program P that is preserved by compilation.

How can the user profit from the high level cost model? Suppose, for instance, that he wants to prove that the WCET of his program is bounded by c. It is sufficient for him to prove that $\Sigma_i k(L_i) \leq c$, which is now a purely functional property of the code. He can therefore use any technique available to certify functional properties of the source code. What is suggested in [5] is to actually instrument the source code P by replacing every label emission statement `emit L` with the instruction `cost += k(L)` that increments a global fresh variable `cost`. The bound is now proved by establishing the program invariant `cost` $\leq c$, which can be done for example using the Frama-C [6] suite if the source code is some variant of C.

2.1 Labelling function calls

We now want to extend the labelling approach to support function calls. On the high level, *structured* programming language $\mathcal{P}$ there is not much to change. When a function is invoked, the current basic block is temporarily exited and the basic block the function starts with take control. When the function returns, the execution of the original basic block is resumed. Thus the only significant change is that basic blocks can now be nested. Let `E` be the label of the external block and `I` the label of a nested one. Since the external starts before the internal, the semantics observed will be `E I` and the cost associated to it on the source language will be $k(\texttt{E})+k(\texttt{I})$, i.e. the cost of executing all instructions in the block `E` first plus the cost of executing all the instructions in the block `I`. However, we know that some instructions in `E` are executed after the last instruction in `I`. This is actually irrelevant because we are here assuming that costs are additive, so that we can freely permute them[2]. Note that, in the present discussion, we are assuming that the function call terminates and yields back control to the basic block `E`. If the call diverges, the instrumentation `cost += k(E)` executed at the beginning of `E` is still valid, but just as an upper bound to the real execution cost: only precision is lost.

Let now consider what happens when we move down the compilation chain to an unstructured intermediate or final language. Here unstructured means that the only control operators are conditional and unconditional jumps, function calls and returns. Unlike a structured language, though, there is no guarantee that a function will return control just after the function call point. The semantics of the return statement, indeed, consists in fetching the return address from some internal structure (typically the control stack) and jumping directly

[2] The additivity assumption fails on modern processors that have stateful subsystems, like caches and pipelines. The extension of the labelling approach to those systems is therefore non trivial and under development in the CerCo project.

to it. The code can freely manipulate the control stack to make the procedure returns to whatever position. Indeed, it is also possible to break the well nesting of function calls/returns.

Is it the case that the code produced by a correct compiler must respect the additional property that every function returns just after its function call point? The answer is negative and the property is not implied by forward simulation proofs. For instance, imagine to modify a correct compiler pass by systematically adding one to the return address on the stack and by putting a `NOP` (or any other instruction that takes one byte) after every function call. The obtained code will be functionally indistinguishable, and the added instructions will all be dead code.

This lack of structure in the semantics badly interferes with the labelling approach. The reason is the following: when a basic block labelled with `E` contains a function call, it no longer makes any sense to associate to a label `E` the sum of the costs of all the instructions in the block. Indeed, there is no guarantee that the function will return into the block and that the instructions that will be executed after the return will be the ones we are paying for in the cost model.

How can we make the labelling approach work in this scenario? We only see two possible ways. The first one consists in injecting an emission statement after every function call: basic blocks no longer contain function calls, but are now terminated by them. This completely solves the problem and allows the compiler to break the structure of function calls/returns at will. However, the technique has several drawbacks. First of all, it greatly augments the number of cost labels that are injected in the source code and that become instrumentation statements. Thus, when reasoning on the source code to prove non-functional properties, the user (or the automation tool) will have to handle larger expressions. Second, the more labels are emitted, the more difficult it becomes to implement powerful optimisations respecting the code structure. Indeed, function calls are usually implemented in such a way that most registers are preserved by the call, so that the static analysis of the block is not interrupted by the call and an optimisation can involve both the code before and after the function call. Third, instrumenting the source code may require unpleasant modification of it. Take, for example, the code `f(g(x));`. We need to inject an emission statement/instrumentation instruction just after the execution of `g`. The only way to do that is to rewrite the code as `y = g(x); emit L; f(y);` for some fresh variable `y`. It is pretty clear how in certain situations the obtained code would be more obfuscated and then more difficult to manually reason on.

For the previous reasons, in this paper and in the CerCo project we adopt a different approach. We do not inject emission statements after every function call. However, we want to propagate a strong additional invariant in the forward simulation proof. The invariant is the propagation of the structure of the original high level code, even if the target language is unstructured. The structure we want to propagate, that will become more clear in the next section, comprises 1) the property that every function should return just after the function call point,

which in turns imply well nesting of function calls; 2) the property that every basic block starts with a code emission statement.

In the original labelling approach of [5], the second property was granted syntactically as a property of the generated code. In our revised approach, instead, we will impose the property on the runs: it will be possible to generate code that does not respect the syntactic property, as soon as all possible runs respect it. For instance, dead code will no longer be required to have all basic blocks correctly labelled. The switch is suggested from the fact that the first of the two properties — that related to function calls/returns — can only be defined as property of runs, not of the static code. The switch is beneficial to the proof because the original proof was made of two parts: the forward simulation proof and the proof that the static property was granted. In our revised approach the latter disappears and only the forward simulation is kept.

In order to capture the structure semantics so that it is preserved by a forward simulation argument, we need to make the structure observable in the semantics. This is the topic of the next section.

3 Structured traces

The program semantics adopted in the traditional labelling approach is based on labelled deductive systems. Given a set of observables $\mathcal{O}$ and a set of states $\mathcal{S}$, the semantics of one deterministic execution step is defined as a function $S \to S \times O^*$ where O^* is a (finite) stream of observables. The semantics is then lifted compositionally to multiple (finite or infinite) execution steps. Finally, the semantics of a a whole program execution is obtained by forgetting about the final state (if any), yielding a function $S \to O^*$ that given an initial status returns the finite or infinite stream of observables in output.

We present here a new definition of semantics where the structure of execution, as defined in the previous section, is now observable. The idea is to replace the stream of observables with a structured data type that makes explicit function call and returns and that grants some additional invariants by construction. The data structure, called *structured traces*, is defined inductively for terminating programs and coinductively for diverging ones. In the paper we focus only on the inductive structure, i.e. we assume that all programs that are given a semantics are total. The Matita formalisation also shows the coinductive definitions. The semantics of a program is then defined as a function that maps an initial state into a structured trace.

In order to have a definition that works on multiple intermediate languages, we abstract the type of structure traces over an abstract data type of abstract statuses:

```
record abstract_status := { S: Type[0];
 as_execute: S → S → Prop;     as_classify: S → classification;
 as_costed: S → Prop;      as_label: ∀ s. as_costed S s → label;
 as_call_ident: ∀ s. as_classify S s = cl_call → label;
```

```
as_after_return:
  (Σs:as_status. as_classify s = Some ? cl_call) → as_status → Prop }
```

The predicate `as_execute` s_1 s_2 holds if s_1 evolves into s_2 in one step; `as_classify` s c holds if the next instruction to be executed in s is classified according to $c \in$ {`cl_return`,`cl_jump`,`cl_call`,`cl_other`} (we omit tail-calls for simplicity); the predicate `as_costed` s holds if the next instruction to be executed in s is a cost emission statement (also classified as `cl_other`); finally (`as_after_return` s_1 s_2) holds if the next instruction to be executed in s_2 follows the function call to be executed in (the witness of the Σ-type) s_1. The two functions `as_label` and `as_cost_ident` are used to extract the cost label/function call target from states whose next instruction is a cost emission/function call statement.

The inductive type for structured traces is actually made by three multiple inductive types with the following semantics:

1. (`trace_label_return` s_1 s_2) is a trace that begins in the state s_1 (included) and ends just before the state s_2 (excluded) such that the instruction to be executed in s_1 is a label emission statement and the one to be executed in the state before s_2 is a return statement. Thus s_2 is the state after the return. The trace may contain other label emission statements. It captures the structure of the execution of function bodies: they must start with a cost emission statement and must end with a return; they are obtained by concatenating one or more basic blocks, all starting with a label emission (e.g. in case of loops).
2. (`trace_any_label` b s_1 s_2) is a trace that begins in the state s_1 (included) and ends just before the state s_2 (excluded) such that the instruction to be executed in s_2/in the state before s_2 is either a label emission statement or or a return, according to the boolean b. It must not contain any label emission statement. It captures the notion of a suffix of a basic block.
3. (`trace_label_label` b s_1 s_2) is the special case of (`trace_any_label` b s_1 s_2) such that the instruction to be executed in s_1 is a label emission statement. It captures the notion of a basic block.

$$\frac{\texttt{trace\_label\_label}\ \mathit{true}\ s_1\ s_2}{\texttt{trace\_label\_return}\ s_1\ s_2}\ (\textsc{tlr\_base})$$

$$\frac{\texttt{trace\_label\_label}\ \mathit{false}\ s_1\ s_2 \qquad \texttt{trace\_label\_return}\ s_2\ s_3}{\texttt{trace\_label\_return}\ s_1\ s_3}\ (\textsc{tlr\_step})$$

$$\frac{\texttt{trace\_any\_label}\ b\ s_1\ s_2 \qquad \texttt{as\_costed}\ s_1}{\texttt{trace\_label\_label}\ b\ s_1\ s_2}\ (\textsc{tll\_base})$$

$$\frac{\texttt{as\_execute}\ s_1\ s_2 \qquad \texttt{as\_classify}\ s_1 \in \{\texttt{cl\_jump},\texttt{cl\_other}\} \qquad \texttt{as\_costed}\ s_2}{\texttt{trace\_any\_label}\ \mathit{false}\ s_1\ s_2}\ (\textsc{tal\_base\_not\_return})$$

$$\frac{\texttt{as\_execute}\ s_1\ s_2 \qquad \texttt{as\_classify}\ s_1 = \texttt{cl\_return}}{\texttt{trace\_any\_label}\ \mathit{true}\ s_1\ s_2}\ (\textsc{tal\_base\_return})$$

$$\frac{\begin{array}{c}\texttt{as\_execute}\ s_1\ s_2 \qquad \texttt{as\_classify}\ s_1 = \texttt{cl\_call} \\ \texttt{as\_after\_return}\ s_1\ s_3 \qquad \texttt{trace\_label\_return}\ s_2\ s_3 \qquad \texttt{as\_costed}\ s_3\end{array}}{\texttt{trace\_any\_label}\ \mathit{false}\ s_1\ s_3}\ (\textsc{tal\_base\_call})$$

$$\frac{\begin{array}{c}\texttt{as\_execute}\ s_1\ s_2 \qquad \texttt{as\_classify}\ s_1 = \texttt{cl\_call} \\ \texttt{as\_after\_return}\ s_1\ s_3 \qquad \texttt{trace\_label\_return}\ s_2\ s_3 \\ \neg\texttt{as\_costed}\ s_3 \qquad \texttt{trace\_any\_label}\ b\ s_3\ s_4\end{array}}{\texttt{trace\_any\_label}\ b\ s_1\ s_4}\ (\textsc{tal\_step\_call})$$

$$\frac{\begin{array}{c}\texttt{as\_execute}\ s_1\ s_2 \qquad \neg\texttt{as\_costed}\ s_2 \\ \texttt{trace\_any\_label}\ b\ s_2\ s_3 \texttt{as\_classify}\ s_1 = \texttt{cl\_other}\end{array}}{\texttt{trace\_any\_label}\ b\ s_1\ s_3}\ (\textsc{tal\_step\_default})$$

A `trace_label_return` is isomorphic to a list of `trace_label_label`s that ends with a cost emission followed by a return terminated `trace_label_label`. The interesting cases are those of `trace_any_label` $b\ s_1\ s_2$. A `trace_any_label` is a sequence of steps built by a syntax directed definition on the classification of s_1. The constructors of the datatype impose several invariants that are meant to impose a structure to the otherwise unstructured execution. In particular, the following invariants are imposed:

1. the trace is never empty; it ends with a return iff b is true
2. a jump must always be the last instruction of the trace, and it must be followed by a cost emission statement; i.e. the target of a jump is always the beginning of a new basic block; as such it must start with a cost emission statement
3. a cost emission statement can never occur inside the trace, only in the status immediately after
4. the trace for a function call step is made of a subtrace for the function body of type `trace_label_return` $s_1\ s_2$, possibly followed by the rest of the trace for this basic block. The subtrace represents the function execution. Being an inductive datum, it grants totality of the function call. The status s_2 is the one that follows the return statement. The next instruction of s_2 must follow the function call instruction. As a consequence, function calls are also well nested.

The three mutual structural recursive functions `flatten_trace_label_return`, `flatten_trace_label_label` and `flatten_trance_any_label` allow to extract from a structured trace the list of states whose next instruction is a cost emission statement. We only show here the type of one of them:

```
flatten_trace_label_return:
 ∀S: abstract_status. ∀s_1, s_2.
  trace_label_return s_1 s_2 → list (as_cost_label S)
```

Cost prediction on structured traces The first main theorem of CerCo about traces (theorem `compute_max_trace_label_return_cost_ok_with_trace`) holds for the instantiation of the structured traces to the concrete status of object code programs. Simplifying a bit, it states that

$$\begin{array}{l}\forall s_1, s_2.\forall \tau : \texttt{trace\_label\_return}\ s_1\ s_2.\\ \quad \texttt{clock}\ s_2 = \texttt{clock}\ s_1 + \Sigma_{s\in(\texttt{flatten\_trace\_label\_return}\ \tau)}\ k(\mathcal{L}(s))\end{array} \quad (1)$$

where $\mathcal{L}$ maps a labelled state to its emitted label, and the cost model k is statically computed from the object code by associating to each label `L` the sum of the cost of the instructions in the basic block that starts at `L` and ends before the next labelled instruction. The theorem is proved by structural induction over the structured trace, and is based on the invariant that iff the function that computes the cost model has analysed the instruction to be executed at s_2 after the one to be executed at s_1, and if the structured trace starts with s_1, then eventually it will contain also s_2. When s_1 is not a function call, the result holds trivially because of the (`as_execute` s_1 s_2) condition obtained by inversion on the trace. The only non trivial case is the one of function calls: the cost model computation function does recursion on the first instruction that follows that function call; the `as_after_return` condition of the `tal_base_call` and `tal_step_call` grants exactly that the execution will eventually reach this state.

Structured traces similarity and cost prediction invariance A compiler pass maps source to object code and initial states to initial states. The source code and initial state uniquely determine the structured trace of a program, if it exists. The structured trace fails to exists iff the structural conditions are violated by the program execution (e.g. a function body does not start with a cost emission statement). Let us assume that the target structured trace exists.

What is the relation between the source and target structured traces? In general, the two traces can be arbitrarily different. However, we are interested only in those compiler passes that maps a trace τ_1 to a trace τ_2 such that

$$\texttt{flatten\_trace\_label\_return}\ \tau_1 = \texttt{flatten\_trace\_label\_return}\ \tau_2 \quad (2)$$

The reason is that the combination of 1 with 2 yields the corollary

$$\begin{array}{l}\forall s_1, s_2.\forall \tau : \texttt{trace\_label\_return}\ s_1\ s_2.\\ \quad \mathit{clock}\ s_2 - \mathit{clock}\ s_1\\ = \Sigma_{s\in(\texttt{flatten\_trace\_label\_return}\ \tau_1)}\ k(\mathcal{L}(s))\\ = \Sigma_{s\in(\texttt{flatten\_trace\_label\_return}\ \tau_2)}\ k(\mathcal{L}(s))\end{array} \quad (3)$$

This corollary states that the actual execution time of the program can be computed equally well on the source or target language. Thus it becomes possible to transfer the cost model from the target to the source code and reason on the source code only.

We are therefore interested in conditions stronger than 2. Therefore we introduce here a similarity relation between traces with the same structure. Theorem `tlr_rel_to_traces_same_flatten` in the Matita formalisation shows that 2 holds for every pair (τ_1, τ_2) of similar traces.

Intuitively, two traces are similar when one can be obtained from the other by erasing or inserting silent steps, i.e. states that are not `as_costed` and that are classified as `other`. Silent steps do not alter the structure of the traces. In particular, the relation maps function calls to function calls to the same function, label emission statements to emissions of the same label, concatenation of subtraces to concatenation of subtraces of the same length and starting with the same emission statement, etc.

In the formalisation the three similarity relations — one for each trace kind — are defined by structural recursion on the first trace and pattern matching over the second. Here we turn the definition into inference rules for the sake of readability. We also omit from trace constructors all arguments, but those that are traces or that are used in the premises of the rules.

$$\frac{\texttt{tll\_rel}\ tll_1\ tll_2}{\texttt{tlr\_rel}\ (\texttt{tlr\_base}\ tll_1)\ (\texttt{tlr\_base}\ tll_2)}$$

$$\frac{\texttt{tll\_rel}\ tll_1\ tll_2 \qquad \texttt{tlr\_rel}\ tlr_1\ tlr_2}{\texttt{tlr\_rel}\ (\texttt{tlr\_step}\ tll_1\ tlr_1)\ (\texttt{tlr\_step}\ tll_2\ tlr_2)}$$

$$\frac{\texttt{as\_label}\ H_1 = \texttt{as\_label}\ H_2 \qquad \texttt{tal\_rel}\ tal_1\ tal_2}{\texttt{tll\_rel}\ (\texttt{tll\_base}\ tal_1\ H_1)\ (\texttt{tll\_base}\ tal_2\ H_2)}$$

$$\frac{}{\texttt{tal\_rel}\ \texttt{tal\_base\_not\_return}\ (taa@\texttt{tal\_base\_not\_return}}$$

$$\frac{}{\texttt{tal\_rel}\ \texttt{tal\_base\_return}\ (taa@\texttt{tal\_base\_return}}$$

$$\frac{\texttt{tlr\_rel}\ tlr_1\ tlr_2 \qquad \texttt{as\_call\_ident}\ H_1 = \texttt{as\_call\_ident}\ H_2}{\texttt{tal\_rel}\ (\texttt{tal\_base\_call}\ H_1\ tlr_1)\ (taa@\texttt{tal\_base\_call}\ H_2\ tlr_2)}$$

$$\frac{\texttt{tlr\_rel}\ tlr_1\ tlr_2 \qquad \texttt{as\_call\_ident}\ H_1 = \texttt{as\_call\_ident}\ H_2 \qquad \texttt{tal\_collapsable}\ tal_2}{\texttt{tal\_rel}\ (\texttt{tal\_base\_call}\ tlr_1)\ (taa@\texttt{tal\_step\_call}\ tlr_2\ tal_2)}$$

$$\frac{\texttt{tlr\_rel}\ tlr_1\ tlr_2 \qquad \texttt{as\_call\_ident}\ H_1 = \texttt{as\_call\_ident}\ H_2 \qquad \texttt{tal\_collapsable}\ tal_1}{\texttt{tal\_rel}\ (\texttt{tal\_step\_call}\ tlr_1\ tal_1)\ (taa@\texttt{tal\_base\_call}\ tlr_2)}$$

$$\frac{\texttt{tlr\_rel}\ tlr_1\ tlr_2 \qquad \texttt{tal\_rel}\ tal_1\ tal_2 \qquad \texttt{as\_call\_ident}\ H_1 = \texttt{as\_call\_ident}\ H_2}{\texttt{tal\_rel}\ (\texttt{tal\_step\_call}\ tlr_1\ tal_1)\ (taa@\texttt{tal\_step\_call}\ tlr_2\ tal_2)}$$

$$\frac{\texttt{tal\_rel}\ tal_1\ tal_2}{\texttt{tal\_rel}\ (\texttt{tal\_step\_default}\ tal_1)\ tal_2}$$

In the preceding rules, a *taa* is an inhabitant of the `trace_any_any` s_1 s_2 inductive data type whose definition is not in the paper for lack of space. It is the type of valid prefixes (even empty ones) of `trace_any_label`s that do not contain any function call. Therefore it is possible to concatenate (using "@") a `trace_any_any` to the left of a `trace_any_label`. A `trace_any_any` captures a sequence of silent moves.

The `tal_collapsable` unary predicate over `trace_any_label`s holds when the argument does not contain any function call and it ends with a label (not a return). The intuition is that after a function call we can still perform a sequence of silent actions while remaining similar.

4 Forward simulation

We summarise here the results of the previous sections. Each intermediate unstructured language can be given a semantics based on structured traces, that single out those runs that respect a certain number of invariants. A cost model can be computed on the object code and it can be used to predict the execution costs of runs that produce structured traces. The cost model can be lifted from the target to the source code of a pass if the pass maps structured traces to similar structured traces. The latter property is called a *forward simulation*.

As for labelled transition systems, in order to establish the forward simulation we are interested in (preservation of observables), we are forced to prove a stronger notion of forward simulation that introduces an explicit relation between states. The classical notion of a 1-to-0-or-many forward simulation is the existence of a relation R over states such that if $s_1 R s_2$ and $s_1 \rightarrow^1 s_1'$ then there exists an s_2' such that $s_2 \rightarrow^* s_2'$ and $s_1' R s_2'$. In our context, we need to replace the one and multi step transition relations $\rightarrow^n$ with the existence of a structured trace between the two states, and we need to add the request that the two structured traces are similar. Thus what we would like to state is something like:
for all s_1, s_2, s_1' such that there is a τ_1 from s_1 to s_1' and $s_1 R s_2$ there exists an s_2' such that $s_1' R s_2'$ and a τ_2 from s_2 to s_2' such that τ_1 is similar to τ_2. We call this particular form of forward simulation *trace reconstruction*.

The statement just introduced, however, is too simplistic and not provable in the general case. To understand why, consider the case of a function call and the pass that fixes the parameter passing conventions. A function call in the source code takes in input an arbitrary number of pseudo-registers (the actual parameters to pass) and returns an arbitrary number of pseudo-registers (where the result is stored). A function call in the target language has no input nor output parameters. The pass must add explicit code before and after the function call to move the pseudo-registers content from/to the hardware registers or the stack in order to implement the parameter passing strategy. Similarly, each function body must be augmented with a preamble and a postamble to complete/initiate

the parameter passing strategy for the call/return phase. Therefore what used to be a call followed by the next instruction to execute after the function return, now becomes a sequence of instructions, followed by a call, followed by another sequence. The two states at the beginning of the first sequence and at the end of the second sequence are in relation with the status before/after the call in the source code, like in an usual forward simulation. How can we prove however the additional condition for function calls that asks that when the function returns the instruction immediately after the function call is called? To grant this invariant, there must be another relation between the address of the function call in the source and in the target code. This additional relation is to be used in particular to relate the two stacks.

Another example is given by preservation of code emission statements. A single code emission instruction can be simulated by a sequence of steps, followed by a code emission, followed by another sequence. Clearly the initial and final statuses of the sequence are to be in relation with the status before/after the code emission in the source code. In order to preserve the structured traces invariants, however, we must consider a second relation between states that traces the preservation of the code emission statement.

Therefore we now introduce an abstract notion of relation set between abstract statuses and an abstract notion of 1-to-0-or-many forward simulation conditions. These two definitions enjoy the following remarkable properties:

1. they are generic enough to accommodate all passes of the CerCo compiler
2. the conjunction of the 1-to-0-or-many forward simulation conditions are just slightly stricter than the statement of a 1-to-0-or-many forward simulation in the classical case. In particular, they only require the construction of very simple forms of structured traces made of silent states only.
3. they allow to prove our main result of the paper: the 1-to-0-or-many forward simulation conditions are sufficient to prove the trace reconstruction theorem

Point 3. is the important one. First of all it means that we have reduced the complex problem of trace reconstruction to a much simpler one that, moreover, can be solved with slight adaptations of the forward simulation proof that is performed for a compiler that only cares about functional properties. Therefore we have successfully splitted as much as possible the proof of preservation of functional properties from that of non-functional ones. Secondly, combined with the results in the previous section, it implies that the cost model can be computed on the object code and lifted to the source code to reason on non-functional properties, assuming that the 1-to-0-or-many forward simulation conditions are fulfilled for every compiler pass.

Relation sets We introduce now the four relations $\mathcal{S}, \mathcal{C}, \mathcal{L}, \mathcal{R}$ between abstract statuses that are used to correlate the corresponding statues before and after a compiler pass. The first two are abstract and must be instantiated by every pass. The remaining two are derived relations.

The $\mathcal{S}$ relation between states is the classical relation used in forward simulation proofs. It correlates the data of the status (e.g. registers, memory, etc.).

The $\mathcal{C}$ relation correlates call states. It allows to track the position in the target code of every call in the source code.

The $\mathcal{L}$ relation simply says that the two states are both label emitting states that emit the same label. It allows to track the position in the target code of every cost emitting statement in the source code.

Finally the $\mathcal{R}$ relation is the more complex one. Two states s_1 and s_2 are $\mathcal{R}$ correlated if every time s_1 is the successors of a call state that is $\mathcal{C}$-related to a call state s_2' in the target code, then s_2 is the successor of s_2'. We will require all pairs of states that follow a related call to be $\mathcal{R}$-related. This is the fundamental requirement to grant that the target trace is well structured, i.e. that function calls are well nested and always return where they are supposed to.

```
record status_rel (S1,S2 : abstract_status) : Type[1] := {
  𝒮: S1 → S2 → Prop;
  𝒞: (Σs.as_classifier S1 s cl_call) →
     (Σs.as_classifier S2 s cl_call) → Prop }.

definition ℒ S1 S2 st1 st2 := as_label S1 st1 = as_label S2 st2.

definition ℛ S1 S2 (R: status_rel S1 S2) s1_ret s2_ret
 ∀s1_pre,s2_pre.
  as_after_return s1_pre s1_ret → s1_pre ℛ s2_pre →
   as_after_return s2_pre s2_ret.
```

1-to-0-or-many forward simulation conditions

Condition 1 (Cases `cl_other` and `cl_jump`) *For all s_1, s_1', s_2 such that $s_1 \mathcal{S} s_1'$, and `as_execute` s_1 s_1', and `as_classify` s_1 = `cl_other` or `as_classify` s_1 = `cl_other` and `as_costed` s_1', there exists an s_2' and a `trace_any_any_free` s_2 s_2' called $taaf$ such that $s_1'(\mathcal{S} \cap \mathcal{L})s_2'$ and either $taaf$ is non empty, or one among s_1 and s_1' is `as_costed`.*

In the above condition, a `trace_any_any_free` s_1 s_2 is an inductive type of structured traces that do not contain function calls or cost emission statements. Differently from a `trace_any_any`, the instruction to be executed in the lookahead state s_2 may be a cost emission statement.

The intuition of the condition is that one step can be replaced with zero or more steps if it preserves the relation between the data and if the two final statuses are labelled in the same way. Moreover, we must take special care of the empty case to avoid collapsing two consecutive states that emit the same label to just one state, missing one of the two emissions.

Condition 2 (Case `cl_call`) *For all s_1, s_1', s_2 s.t. $s_1 \mathcal{S} s_1'$ and `as_execute` s_1 s_1' and `as_classify` s_1 = `cl_call`, there exists s_2', s_b, s_a, a `trace_any_any` s_2 s_b, and a `trace_any_any_free` s_a s_2' such that: s_a is classified as a `cl_call`, the `as_identifiers` of the two call states are the same, $s_1 \mathcal{C} s_b$, `as_execute` s_b s_a holds, $s_1' \mathcal{L} s_b$ and $s_1' \mathcal{S} s_2'$.*

The condition says that, to simulate a function call, we can perform a sequence of silent actions before and after the function call itself. The old and new call states must be $\mathcal{C}$-related, the old and new states at the beginning of the function execution must be $\mathcal{L}$-related and, finally, the two initial and final states must be $\mathcal{S}$-related as usual.

Condition 3 (Case `cl_return`) *For all s_1, s_1', s_2 s.t. $s_1 \mathcal{S} s_1'$, `as_execute` s_1 s_1' and `as_classify` s_1 = `cl_return`, there exists s_2', s_b, s_a, a `trace_any_any` s_2 s_b, a `trace_any_any_free` s_a s_2' called $taaf$ such that: s_a is classified as a `cl_return`, $s_1 \mathcal{C} s_b$, the predicate `as_execute` s_b s_a holds, $s_1' \mathcal{R} s_a$ and $s_1' (\mathcal{S} \cap \mathcal{L}) s_2'$ and either $taaf$ is non empty, or s_a is not `as_costed`.*

Similarly to the call condition, to simulate a return we can perform a sequence of silent actions before and after the return statement itself. The old and the new statements after the return must be $\mathcal{R}$-related, to grant that they returned to corresponding calls. The two initial and final states must be $\mathcal{S}$-related as usual and, moreover, they must exhibit the same labels. Finally, when the suffix is non empty we must take care of not inserting a new unmatched cost emission statement just after the return statement.

Main result: the 1-to-0-or-many forward simulation conditions are sufficient to trace reconstruction Let us assume that a relation set is given such that the 1-to-0-or-many forward simulation conditions are satisfied. Under this assumption we can prove the following three trace reconstruction theorems by mutual structural induction over the traces given in input between the s_1 and s_1' states.

In particular, the `status_simulation_produce_tlr` theorem applied to the `main` function of the program and equal s_{2_b} and s_2 states shows that, for every initial state in the source code that induces a structured trace in the source code, the compiled code produces a similar structured trace.

Theorem 1 (`status_simulation_produce_tlr`). *For every s_1, s_1', s_{2_b}, s_2 s.t. there is a `trace_label_return` s_1 s_1' called tlr_1 and a `trace_any_any` s_{2_b} s_2 and $s_1 \mathcal{L} s_{2_b}$ and $s_1 \mathcal{S} s_2$, there exists s_{2_m}, s_2' s.t. there is a `trace_label_return` s_{2_b} s_{2_m} called tlr_2 and there is a `trace_any_any_free` s_{2_m} s_2' called $taaf$ s.t. if $taaf$ is non empty then $\neg$(`as_costed` s_{2_m}), and `tlr_rel` tlr_1 tlr_2 and $s_1' (\mathcal{S} \cap \mathcal{L}) s_2'$ and $s_1' \mathcal{R} s_{2_m}$.*

The theorem states that a `trace_label_return` in the source code together with a precomputed preamble of silent states (the `trace_any_any`) in the target code induces a similar `trace_label_return` trace in the target code which can be followed by a sequence of silent states. Note that the statement does not require the produced `trace_label_return` trace to start with the precomputed preamble, even if this is likely to be the case in concrete implementations. The preamble in input is necessary for compositionality, e.g. because the 1-to-0-or-many forward simulation conditions allow in the case of function calls to execute a preamble of silent instructions just after the function call.

Theorem 2 (status_simulation_produce_tll). *For every* s_1, s_1', s_{2_b}, s_2 *s.t. there is a* `trace_label_label` b s_1 s_1' *called* tll_1 *and a* `trace_any_any` s_{2_b} s_2 *and* $s_1 \mathcal{L} s_{2_b}$ *and* $s_1 \mathcal{S} s_2$*, there exists* s_{2_m}, s_2' *s.t.*

- *if* b *(the trace ends with a return) then there exists* s_{2_m}, s_2' *and a trace* `trace_label_label` b s_{2_b} s_{2_m} *called* tll_2 *and a* `trace_any_any_free` s_{2_m} s_2' *called* taa_2 *s.t.* $s_1'(\mathcal{S} \cap \mathcal{L})s_2'$ *and* $s_1' \mathcal{R} s_{2_m}$ *and* `tll_rel` tll_1 tll_2 *and if* taa_2 *is non empty then* $\neg$(`as_costed` s_{2_m})
- *else there exists* s_2' *and a* `trace_label_label` b s_{2_b} s_2' *called* tll_2 *such that* $s_1'(\mathcal{S} \cap \mathcal{L})s_2'$ *and* `tll_rel` tll_1 tll_2*.*

The statement is similar to the previous one: a source `trace_label_label` and a given target preamble of silent states in the target code induce a similar `trace_label_label` in the target code, possibly followed by a sequence of silent moves that become the preamble for the next `trace_label_label` translation.

Theorem 3 (status_simulation_produce_tal). *For every* s_1, s_1', s_2 *s.t. there is a* `trace_any_label` b s_1 s_1' *called* tal_1 *and* $s_1 \mathcal{S} s_2$

- *if* b *(the trace ends with a return) then there exists* s_{2_m}, s_2' *and a trace* `trace_any_label` b s_2 s_{2_m} *called* tal_2 *and a* `trace_any_any_free` s_{2_m} s_2' *called* taa_2 *s.t.* $s_1'(\mathcal{S} \cap \mathcal{L})s_2'$ *and* $s_1' \mathcal{R} s_{2_m}$ *and* `tal_rel` tal_1 tal_2 *and if* taa_2 *is non empty then* $\neg$(`as_costed` s_{2_m})
- *else there exists* s_2' *and a* `trace_any_label` b s_2 s_2' *called* tal_2 *such that either* $s_1'(\mathcal{S} \cap \mathcal{L})s_2'$ *and* `tal_rel` tal_1 tal_2 *or* $s_1'(\mathcal{S} \cap \mathcal{L})s_2$ *and* `tal_collapsable` tal_1 *and* $\neg$(`as_costed` s_1)

The statement is also similar to the previous ones, but for the lack of the target code preamble.

5 Conclusions and future works

The labelling approach is a technique to implement compilers that induce on the source code a non uniform cost model determined from the object code produced. The cost model assigns a cost to each basic block of the program. The main theorem of the approach says that there is an exact correspondence between the sequence of basic blocks started in the source and object code, and that no instruction in the source or object code is executed outside a basic block. Thus the cost of object code execution can be computed precisely on the source.

In this paper we scale the labelling approach to cover a programming language with function calls. This introduces new difficulties only when the language is unstructured, i.e. it allows function calls to return anywhere in the code, destroying the hope of a static prediction of the cost of basic blocks. We restore static predictability by introducing a new semantics for unstructured programs that single outs well structured executions. The latter are represented by structured traces, a generalisation of streams of observables that capture several structural invariants of the execution, like well nesting of functions or the fact

that every basic block must start with a code emission statement. We show that structured traces are sufficiently structured to statically compute a precise cost model on the object code.

We introduce a similarity relation on structured traces that must hold between source and target traces. When the relation holds for every program, we prove that the cost model can be lifted from the object to the source code.

In order to prove that similarity holds, we present a generic proof of forward simulation that is aimed at pulling apart as much as possible the part of the simulation related to non-functional properties (preservation of structure) from that related to functional properties. In particular, we reduce the problem of preservation of structure to that of showing a 1-to-0-or-many forward simulation that only adds a few additional proof obligations to those of a traditional, function properties only, proof.

All results presented in the paper are part of a larger certification of a C compiler which is based on the labelling approach. The certification, done in Matita, is the main deliverable of the FET-Open Certified Complexity (CerCo).

The short term future work consists in the completion of the certification of the CerCo compiler exploiting the main theorem of this paper.

Related works CerCo is the first project that explicitly tries to induce a precise cost model on the source code in order to establish non-functional properties of programs on an high level language. Traditional certifications of compilers, like [9,10], only explicitly prove preservation of the functional properties.

Usually forward simulations take the following form: for each transition from s_1 to s_2 in the source code, there exists an equivalent sequence of transitions in the target code of length n. The number n of transition steps in the target code can just be the witness of the existential statement. An equivalent alternative when the proof of simulation is constructive consists in providing an explicit function, called *clock function* in the literature [12], that computes n from s_1. Every clock function constitutes then a cost model for the source code, in the spirit of what we are doing in CerCo. However, we believe our solution to be superior in the following respects: 1) the machinery of the labelling approach is insensible to the resource being measured. Indeed, any cost model computed on the object code can be lifted to the source code (e.g. stack space used, energy consumed, etc.). On the contrary, clock functions only talk about number of transition steps. In order to extend the approach with clock functions to other resources, additional functions must be introduced. Moreover, the additional functions would be handled differently in the proof. 2) the cost models induced by the labelling approach have a simple presentation. In particular, they associate a number to each basic block. More complex models can be induced when the approach is scaled to cover, for instance, loop optimisations [13], but the costs are still meant to be easy to understand and manipulate in an interactive theorem prover or in Frama-C. On the contrary, a clock function is a complex function of the state s_1 which, as a function, is an opaque object that is difficult to reify as source code in order to reason on it.

References

1. Amadio, R., Asperti, A., Ayache, N., Campbell, B., Mulligan, D., Pollack, R., Régis-Gianas, Y., Sacerdoti Coen, C., Stark, I.: Certified complexity. Procedia Computer Science 7, 175–177 (2011)
2. Amadio, R., Régis-Gianas, Y.: Certifying and reasoning on cost annotations of functional programs. In: Pea, R., Eekelen, M., Shkaravska, O. (eds.) Foundational and Practical Aspects of Resource Analysis, Lecture Notes in Computer Science, vol. 7177, pp. 72–89. Springer Berlin Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-32495-6_5, extended version to appear in Higher Order and Symbolic Computation, 2013
3. Asperti, A., Ricciotti, W., Sacerdoti Coen, C., Tassi, E.: The matita interactive theorem prover. In: Bjrner, N., Sofronie-Stokkermans, V. (eds.) Automated Deduction CADE-23, Lecture Notes in Computer Science, vol. 6803, pp. 64–69. Springer Berlin Heidelberg (2011), http://dx.doi.org/10.1007/978-3-642-22438-6_7
4. Asperti, A., Sacerdoti Coen, C., Tassi, E., Zacchiroli, S.: User interaction with the matita proof assistant. Journal of Automated Reasoning 39, 109–139 (2007), http://dx.doi.org/10.1007/s10817-007-9070-5
5. Ayache, N., Amadio, R., Régis-Gianas, Y.: Certifying and reasoning on cost annotations in C programs. In: Stoelinga, M., Pinger, R. (eds.) Formal Methods for Industrial Critical Systems, Lecture Notes in Computer Science, vol. 7437, pp. 32–46. Springer Berlin Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-32469-7_3
6. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-c: a software analysis perspective. In: Proceedings of the 10th international conference on Software Engineering and Formal Methods. pp. 233–247. SEFM'12, Springer-Verlag, Berlin, Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-33826-7_16
7. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM 52(7), 107–115 (2009), http://gallium.inria.fr/~xleroy/publi/compcert-CACM.pdf
8. Leroy, X.: A formally verified compiler back-end. Journal of Automated Reasoning 43(4), 363–446 (2009), http://gallium.inria.fr/~xleroy/publi/compcert-backend.pdf
9. Leroy, X., Blazy, S.: Formal verification of a C-like memory model and its uses for verifying program transformations. Journal of Automated Reasoning 41(1), 1–31 (2008)
10. Moore, J.S.: A mechanically verified language implementation. Journal of Automated Reasoning 5, 461–492 (1989), http://dx.doi.org/10.1007/BF00243133
11. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1999)
12. Ray, S., Moore, J.S.: Proof Styles in Operational Semantics. In: Hu, A.J., Martin, A.K. (eds.) Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2004). LNCS, vol. 3312, pp. 67–81. Springer, Austin, TX (Nov 2004)
13. Tranquilli, P.: Indexed labels for loop iteration dependent costs (Jan 2013), http://www.cs.unibo.it/~tranquil/content/docs/indlabels.pdf, to appear in Electronic Proceedings in Theoretical Computer Science, proceedings of Quantitative Aspects of Programming Languages and Systems (QAPL 2013)

Certifying the back-end pass of the CerCo annotating compiler

Mauro Piccolo, Claudio Sacerdoti Coen and Paolo Tranquilli

Department of Computer Science and Engineering, University of Bologna
{Mauro.Piccolo, Claudio.SacerdotiCoen, Paolo.Tranquilli}@unibo.it

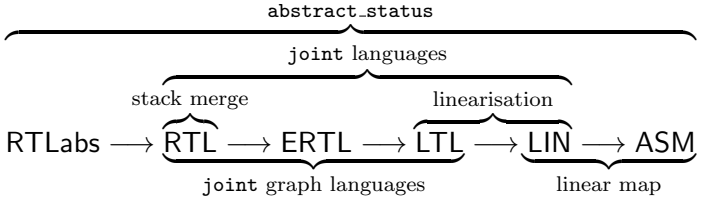
Abstract

We present the certifying effort of the back-end of the CerCo annotating compiler to 8051 assembly. Apart from the proofs needed for propagating the extensional part of execution traces, additional care must be taken in order to ensure a form of intensional correctness of the pass, necessary to prove that the lifting of computational costs is correct. We concentrate here on two aspects of the proof: how stack is dealt with during the pass, and how generic graph language passes can be proven correct from an extensional and intensional point of view.

1 The Back-End Correctness Proof at a Glance

The back-end part of the compiler takes an RTLabs program together with an initialising cost label and gives the assembly code to be fed to the assembler. From the semantic point of view, at this stage of the compiler, we are interested in propagating structured traces, as explained in [2].

A schema with all the back-end passes and the salient and common points of each one is the following:



First, here all language semantics share a common interface via `abstract_status`, which is at the base of the definition of structured traces. The generic shape of each pass of the back-end is thus the same: we get in input an unstructured prefix trace followed by a structured one which we want to measure in the source language, and we need to prove the existence of the corresponding unstructured and structured traces in the target language, with the two traces producing the same observables.

From RTL down to LIN we are in the common syntactic and semantic language we call `joint`. Inside the first of these languages, RTL, we pass from separate local memory spaces for each function call to a unique one. Even though parameters are passed on stack only in ERTL and variable allocation and spilling is done in LTL, we already have values on stack at this stage, and we ensure all the stack space that will eventually be needed is allocated at each call.

```
record ft_and_tlr (S : abstract_status) (prefix, subtrace : list intensional_event)
(fn : ident) (st1 : S) : Prop :=
{ st2 : S
; st3 : S
; ft : flat_trace S st1 st2
; tlr : trace_label_return S st2 st3
; tlr_unrpt : tlr_unrepeating … tlr
; ft_is_prefix : ft_observables … ft = prefix
; fn_is_current : ft_current_function … ft = Some ? fn
; tlr_is_subtrace : observables_trace_label_return … tlr fn = subtrace
}.
```

Figure 1: The `ft_and_tlr` record (which stands for `flat_trace` and `trace_label_return`), which abstracts the kind of invariant that all back-end passes need to preserve.

The next two passes live in the graph part of `joint`, and can benefit from a generic approach to graph translation and its proof of correctness, as described in section 3.

Next, a generic `joint` linearisation pass is performed to go from LTL to LIN. The proof is generic too, with reasonable and straightforward proof obligations asking that the common semantic operations of the two languages depend on program counters in memory only to control the flow. Program counters are indeed the only semantic entity that changes during this pass.

The last pass is, as far as function bodies are concerned, a simple linear map—every LIN instruction is mapped to a single ASM instruction, so that we may say that LIN is a subset of ASM. Less straightforward is that function bodies are merged together, and that pointers pass from formal data to actual one. Values in memory need therefore to be remapped to relate the states in the two languages.

1.1 A common invariant

This block of traces with properties is recurrent enough to merit the matita definition in Figure 1. We rely on the fact that all semantics from RTLabs down to object code can be expressed with the `abstract_status` record, by which all our definitions of traces are expressed. The parameters of the record fix respectively the intensional events encountered in the prefix (as stated by `ft_is_prefix`), the ones in the structured trace (`tlr_is_subtrace`) and the name of the function in which `tlr` takes place (`fn_is_current`), and the initial state. The additional property `tlr_unrpt` tells that program counters do not repeat locally (i.e. between two labels) in `tlr`, a property needed for the soundness of the cost static analysis.

1.2 The statement

The back-end correctness theorem proves the matita statement in Figure 2. The output of the a proof is a `ft_and_tlr` structure as outlined before, which constitutes the preconditions of the assembly proof. The `sigma` and `pol` parameters are passed to ASM's semantics, however they have significance only with respect to the ASM to object code correctness [1].

[1] `sigma` and `pol` are what allows to maps instructions between ASM and the produced object code. This information is gained during the jump expansion pass, cf. [1].

```
theorem back_end_correct :
∀observe,init_cost,p_rtlabs,p_asm,stack_m,max_stack,prefix,subtrace,fn.
back_end observe init_cost p_rtlabs =
    return ⟨p_asm, stack_m, max_stack ⟩→
back_end_preconditions p_rtlabs (stack_sizes stack_m) max_stack prefix subtrace fn →
∀sigma,pol.
ft_and_tlr (ASM_status p_asm sigma pol)
    prefix subtrace fn (initialise_status ? p_asm).
```

(a) The statement.

```
record back_end_preconditions (p_rtlabs : RTLabs_program)
(stacksizes : ident →option ℕ) (max_stack : ℕ)
(prefix, subtrace : list intensional_event) (fn : ident) : Prop :=
{ ra_init : RTLabs_status (make_global p_rtlabs)
; ra_init_ok : make_ext_initial_state p_rtlabs = return ra_init
; ra_max :
  le (maximum (update_stacksize_info stacksizes (mk_stacksize_info 0 0)
    (extract_call_ud_from_observables (prefix @ subtrace)))) max_stack
; ra_ft_tlr : ft_and_tlr (RTLabs_status (make_global p_rtlabs))
    prefix subtrace fn ra_init
}.
```

(b) The definition of preconditions for the correctness of the back-end.

Figure 2: The statement of the back-end correctness result.

Care must be taken in dealing with the stack. The back-end pass produces a *stack model*: the amount of stack needed by each function (`stack_m`), together with the maximal usable stack (`max_stack`, 2^{16} minus the size occupied by global variables). While programs in the front end do not fail for stack overflow, the stack information is lifted to source much in the same ways as time consumption information, so that we can actually use as a hypothesis on input source traces that they do not use more stack than allowed. This hypothesis is included in the `measurable` predicate over front-end traces, and we put it to use during the back-end pass, where we pass to a unique bounded stack space for all functions. More details will be presented in section 2.

The above explanation is why the back-end correctness results requires some additional preconditions with respect to a `ft_and_tlr` for RTLabs. This is accomplished by the `ra_max` field in the record `back_end_preconditions` (Figure 2b). The other fields hold the initial state `ra_init` of the program in RTLabs (with a proof that it is actually the initial state).

2 Dealing with the Stack

Setting apart the extensional details of the proofs, a delicate point is how we deal with stack.

A first minor issue is that the information we have about stack usage of each function call evolves along the back-end passes:

- in RTL we must allocate some stack for what where referenced local variables (including local arrays) in source code;
- in ERTL we add up the stack used to pass parameters that overflow the hardware registers available for the task;
- finally in LTL we add the space necessary for spilled pseudo-registers, and the stack usage of functions is finally fixed.

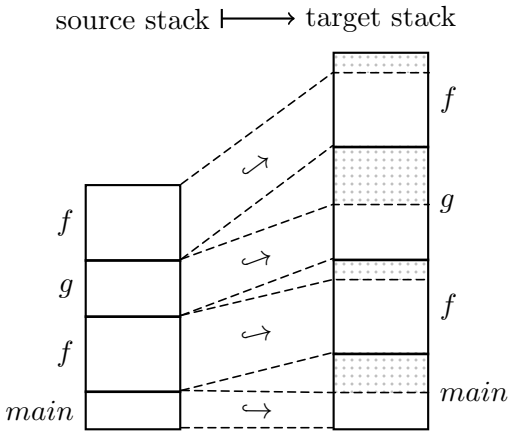
Another issue that we already mentioned above is that somewhere during the back-end pass we must pass from a semantics with no stack overflow error to one that can fail.

In the following we describe the approach we have taken to these issues.

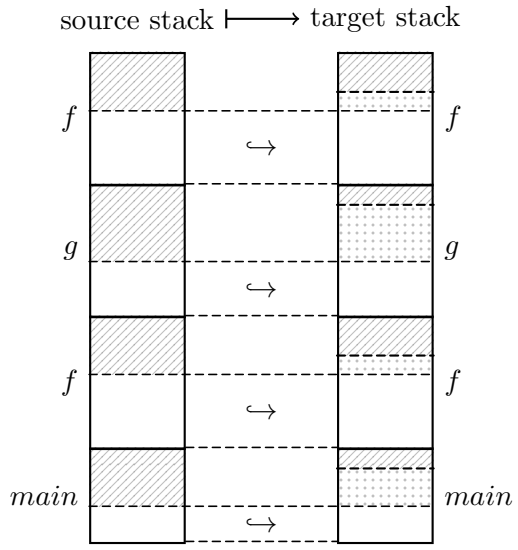
2.1 An evolving stack usage

The stack size we know each function must at least have is written in a field of `joint` internal functions (called `joint_if_stacksize`), and as already said evolves during the back-end pass. The *naïve* approach to defining semantics of these languages is to allocate the minimal necessary space for each function call, reading it from this syntactic field. The drawback is that then at each update of the stack size we must remap the data pointers in memory, moreover in a way that is dependent on the call stack. This is exemplified by the picture in Figure 3a.

We decided to take another approach, where stack sizes of each function call are a parameter to the semantics, and all passes are proved correct for the same stack sizes in source and target language, possibly with a hypothesis on these "semantic" stack sizes to be greater than the "syntactic" ones (but not all passes require it). The picture becomes thus the one in Figure 3b. The advantage is that now the data in memory does not change along the evolving of stack usage, apart from the first time we pass from separate independent stack spaces and a unique one, a situation we describe in the next subsection.



(a) How stack is remapped if we use tight stack spaces for each language. Dotted areas are the new parts of the local stacks.



(b) Constant stack usage taken as a semantic parameter across languages. Shaded areas are unused parts of the stack.

Figure 3: Comparison between using tight stack sizes across languages, i.e. reading how much stack is required from the function, and using the same stack sizes provided as a parameter to the semantics. We suppose the call stack is $main, f, g, f$.

2.2 From an unbounded to a bounded stack

The way the `ft_and_tlr` is propagated is almost everywhere by means of the proof of existence of a special relation between states that is a simulation with respect to execution. There are more details involved than in a usual extensional proof, regarding the intensional preservation. The details are contained in [2].

Here we concentrate on a particular aspect that eludes the generic treatment: the moment when we pass from an unbounded stack to a bounded one. This passage is delicate for two reasons. Firstly, while usually we can assure forward simulation of an execution step by simply depending on that step not producing an error, here is no more the case, as the stack overflow error must start happening somewhere in the back-end. Secondly, merging the stack spaces of the function calls requires mapping the pointer values contained in memory.

In order to tackle these subtleties, we firstly decided to isolate this passage, without mingling it with other pass-dependent semantic preservation proofs. Rather than performing the proof when passing from a language to another, we decided to split in three the semantics of a single language (RTL).

The first one, called `RTL_semantics_separate`, mimics how stack is modelled in RTLabs: each function call gets allocated an independent stack space. Passing from RTLabs to this semantics involves the usual simulation relation. A peculiarity is that data pointers in memory need not change.

The middle, second one, is called `RTL_semantics_separate_overflow`, and it has separate allocated stack spaces just as above. However the execution is defined so that if the call stack as a whole uses more space than (later) physically available (as provided by a parameter), an error is issued. The actual states are identical, and the simulation is an almost trivial 1-to-1,

where the stack correctness inherited from back_end_preconditions can be easily put to use even if the generic results for structure preserving simulation actually cannot be used here.

The third one, defined as RTL_semantics_unique, switches to a semantics with a single, bounded stack space, where moreover memory has already been granted to global variables too. From this semantics down to LIN, all data in memory excluding pieces of program counters remain the same, as specified in subsection 2.1. However in order to show a forwarding simulation from RTL_semantics_separate_overflow, one needs to remap data pointers. No further hypothesis on stack usage needs to be employed, as it is integrated in the fact that the execution step does not fail.

3 A Modular Approach to the Correctness of the Graph Passes

We will now outline how the proof can be carried out generally in the graph passes of the back-end that share the joint language interface.

An instance of the record sem_graph_params contains all common information about programs of a joint-language whose source code can be logically organized as a graph. Thus, every program of this kind will be an inhabitant of the type joint_program p, where p is an instance of sem_graph_params. We want to stress that such a record is in a sub-type relation with the record params, which was explained in Deliverable 4.3.

In order to establish the correctness of our cost-model in each compiler's back-end pass, where both source and target programs are joint-graph-programs, i.e. the source program src (resp. the target program out) is such that there is a p_in (resp. p_out) of type sem_graph_params such that src : joint_program p_in (resp. out : joint_program p_out), we would like to prove a statement similar to this shape.

```
theorem joint_correctness : ∀p_in,p_out : sem_graph_params.
∀prog : joint_program p_in.∀stack_size.
let trans_prog :=transform_program … prog in
∀init_in.
make_initial_state (mk_prog_params p_in prog stack_size) = OK ? init_in →
∃init_out.
make_initial_state (mk_prog_params p_out trans_prog stack_size) = OK ? init_out ∧
∃[1] R : status_rel (joint_abstract_status (mk_prog_params p_in prog stack_size))
         (joint_abstract_status (mk_prog_params p_out trans_prog stack_size)).
  status_simulation_with_init
    (joint_abstract_status (mk_prog_params p_in prog stack_size))
    (joint_abstract_status (mk_prog_params p_out trans_prog stack_size))
    R init_in init_out.
```

When proving this statement (for each concrete instance of language), we need to proceed by cases according the classification of each state in the source program (cl_jmp, cl_other, cl_call and cl_return) and then prove the suitable conditions explained in [2], according to the case we are currently considering (Condition 1 for cl_other and cl_jump case, Condition 2 for cl_call or Condition 3 for cl_return case). Roughly speaking, proving these conditions means producing traces of some kind in the target language that are in a suitable correspondence with an execution step in the source language.

Since traces carry both extensional and intensional information, the main disadvantage with this approach is that the extensional part of the proof (for example the preservation the semantic relation between states of the program under evaluation) and the intensional part

(for example, all proof obligation dealing with the presence of a cost label in some point of the code) are mixed together in a not very clear way.

Furthermore, some proof obligations concerning the relation among program counters (for example, the call relation between states which is one of the field of `status_rel` record) depend only on the way the translation from source to target program is done. In particular if the translation satisfies some suitable properties, then it is possible to define some "standard relations" that automatically satisfy these proof obligations, in an independent way from the specific source and target languages.

So our question is whether there is a way to use these observation in order to factorize all common aspects of all correctness proofs of each pass involving joint languages. The aim is having a layer on top of the one talking about traces, that ask the user wishing to prove the correctness of a single pass involving two joint-languages, to provide some semantic state relation that satisfies some conditions. These conditions never speak about traces and they are mostly extensional, with the intensional part being limited to some additional required properties of the translation. This layer will use the trace machinery in order to deliver the desired correctness proof of the pass.

In order to reach this goal, we have to analyse first whether there is a common way to perform language translation for each pass. After having defined and specified the translation machinery, we will explain how it is possible to use it in order to build such a layer. So this section is organized as follows: in the first part we explain the translation machinery while in the second part we explain such a layer.

3.1 Graph translation

Since a program is just a collection of functions, the compositional approach suggests us to define the translation of a program in terms of the way we translate each internal function. Thus, if we let `src_g_pars` and `dst_g_pars` being the graph parameters of respectively the source and the target program, the aim is writing a matita's function that takes as input an object of type `joint_closed_internal_function src_g_pars` together with additional information (that we will explain better later) and gives as output an object of type `joint_closed_internal_function dst_g_pars` with some properties, that corresponds to the result of the translation (for the definition of `joint_closed_internal_function`, see Deliverable 4.2 and 4.3) . The signature of the definition is the following one.

```
definition b_graph_translate :
  ∀src_g_pars,dst_g_pars : graph_params.
  ∀globals: list ident.
  ∀data : bound_b_graph_translate_data src_g_pars dst_g_pars globals.
  ∀def_in : joint_closed_internal_function src_g_pars globals.
  Σdef_out : joint_closed_internal_function dst_g_pars globals.
  ∃data',regs,f_lbls,f_regs.
    bind_new_instantiates ?? data' data regs ∧
    b_graph_translate_props … data' def_in def_out f_lbls f_regs.
```

Let us now discuss in detail what are the parameter to be provide in input and what is the output of the translation process.

3.1.1 Input requested by the translation process

Clearly, `b_graph_translate` takes as input the internal function of the source language we want to translate. But it also takes in input some useful information which will be used in order to dictate the translation process. These informations are all contained in an instance of the following record, where we skip some technical details.

```
record b_graph_translate_data
  (src, dst : graph_params)
  (globals : list ident) : Type[0] :=
{ init_ret : call_dest dst
; init_params : paramsT dst
; init_stack_size : ℕ
; added_prologue : list (joint_seq dst globals)
; new_regs : list register
; f_step : label →joint_step src globals →bind_step_block dst globals
; f_fin : label →joint_fin_step src →bind_fin_block dst globals
; good_f_step : …
; good_f_fin : …
; f_step_on_cost :
  ∀l,c.f_step l (COST_LABEL … c) =
    bret ? (step_block ??) ⟨[ ], λ_.COST_LABEL dst globals c, [ ] ⟩
; cost_in_f_step :
  ∀l,s,c.
  bind_new_P ??
    (λblock.∀l'.\snd (\fst block) l' = COST_LABEL dst globals c →
       s = COST_LABEL … c) (f_step l s)
}
```

Table 1 summarizes what each field means and how it is used in the translation process. We will say that an identifier of a pseudo-register (resp. a code label) is *fresh* when it never appears in the code of the source function.

3.1.2 Output given the translation process

Clearly `g_graph_translate` gives in output the translated internal function. Unfortunately, for what we are going to develop later, this information is insufficient because we need some information about the correspondence between the source internal function and the translated one. Such an information is given by the second component of the Σ-type returned by the translation process. We remind that the return type of `b_graph_translate` is the following.

```
Σdef_out : joint_closed_internal_function dst_g_pars globals.
 ∃data',regs,f_lbls,f_regs.
   bind_new_instantiates ?? data' data regs ∧
   b_graph_translate_props … data' def_in def_out f_lbls f_regs.
```

The correspondence between the source internal function and the translated one is made explicit by two functions: `f_lbls : code_point src_g_pars →list (code_point dst_g_pars)` and `f_regs : code_point src_g_pars →list (register)`. To understand their meaning, we need to stress the fact that the translation process translates every statement appearing in the code of the source internal function in a given code point l into a block of statements in the code of the translated function where the first instruction of the block has l as code

Field	Explanation
`init_ret`	It tells where the result for the translated function is stored.
`init_params`	It tells what is the translation of the formal parameters.
`init_stack_size`	It tells how to fill the field `joint_if_stacksize` of the translated function, which tells how much stack is required by the function at the given stage of compilation.
`added_prologue`	It is a list of sequential statements of the target language which is always added at the beginning of the translated function.
`new_regs`	It is a list of identifiers for fresh pseudo-registers that are generated when instantiating this record. They are typically used to store data that must be saved across the call (e.g. the return address popped from the internal stack).
`f_step`	It is a function that tells how to translate all *step statements* i.e. statements admitting a syntactical successor. Statements of this kind are sequential statements, a cost emission statement, a call statement or a conditional statement: in the two latter cases the syntactical successors are respectively the returning address (at the end of a function call) and the point where the flow will continue in case the guard is false.
`f_fin`	As above, but for *final statements* i.e. statements do not admitting a syntactical successor. Statements of this kind are return and unconditioned jump statements.
`good_f_step`	It tells that the translation function of all step statement is well formed in the sense that the block `f_step I` for a step `I` only uses code labels and register appearing in `I`, `new_regs` or generated by the appropriate identifier universe fields of the translated function.
`good_f_fin`	As above, but for `f_fin`.
`f_step_on_cost`, `cost_in_f_step`	They give a particular restriction on the translation of a cost-emission statement: it tells that the translation of a cost-emission has to be the identity, i.e. it should be translated as the same cost-emission statement. Furthermore, the translation never introduce new cost-emission statements which do not correspond to a cost emission in the source.

Table 1: The explanation of the various fields of the `b_graph_translate_data` record.

point identifier and all succeeding instructions of the block have fresh code points identifiers. Furthermore statements of this block may use fresh identifiers of pseudo-registers or (even worse) they may use some fresh code-point identifiers being generated by the translation (we will see later that this will be important when translating a call statement). We will use the above mentioned functions to retrieve this information. `f_lbls` takes in input an identifier l for a code point in the source code and it gives back the list of all fresh code point identifiers generated by the translation process of the statement in the source located in l. `f_regs` takes in input an identifier l for a code point in the source code and it gives back the list of all fresh register identifiers generated by the translation process of the statement in the source located in l.

The above mentioned properties about the functions `f_lbls` and `f_regs`, together with some other ones, are expressed and formalized by the following propositional record.

```
record b_graph_translate_props
  (src_g_pars, dst_g_pars : graph_params)
  (globals: list ident)
  (data : b_graph_translate_data src_g_pars dst_g_pars globals)
  (def_in : joint_closed_internal_function src_g_pars globals)
  (def_out : joint_closed_internal_function dst_g_pars globals)
  (f_lbls : label →list label)
  (f_regs : label →list register) : Prop :=
{ res_def_out_eq :
          joint_if_result … def_out = init_ret … data
; pars_def_out_eq :
          joint_if_params … def_out = init_params … data
; ss_def_out_eq :
          joint_if_stacksize … def_out = init_stack_size … data
; partition_lbls : partial_partition … f_lbls
; partition_regs : partial_partition … f_regs
; freshness_lbls :
  (∀l.All …
    (λlbl.¬in_universe … lbl (joint_if_luniverse … def_in) ∧
          in_universe … lbl (joint_if_luniverse … def_out)) (f_lbls l))
; freshness_regs :
  (∀l.All …
    (λreg.¬in_universe … reg (joint_if_runiverse … def_in) ∧
          in_universe … reg (joint_if_runiverse … def_out)) (f_regs l))
; freshness_data_regs :
  All … (λreg.¬in_universe … reg (joint_if_runiverse … def_in) ∧
              in_universe … reg (joint_if_runiverse … def_out))
       (new_regs … data)
; data_regs_disjoint : ∀l,r.r ∈f_regs l →r ∈new_regs … data →False
; multi_fetch_ok :
  ∀l,s.stmt_at … (joint_if_code … def_in) l = Some ? s →
  let lbls :=f_lbls l in let regs :=f_regs l in
  match s with
  [ sequential s' nxt ⇒
    let block :=
      if not_emptyb … (added_prologue … data) ∧
         eq_identifier … (joint_if_entry … def_in) l then
        bret … [ ]
      else
```

```
       f_step … data l s' in
    l -(block, l::lbls, regs)→ nxt in joint_if_code … def_out
  | final s' ⇒
    l -(f_fin … data l s', l::lbls, regs)→ it in joint_if_code … def_out
  | FCOND abs _ _ _ ⇒⨂abs
  ]
; prologue_ok :
  if not_emptyb … (added_prologue … data) then
    ∃lbl.¬in_universe … lbl (joint_if_luniverse … def_in) ∧
    joint_if_entry … def_out
      -( ⟨ [ ],λ_.COST_LABEL … (get_first_costlabel … def_in),
           added_prologue … data ⟩,
        f_lbls … lbl)→ joint_if_entry … def_in in joint_if_code … def_out
  else (joint_if_entry … def_out = joint_if_entry … def_in)
}.
```

Table 2 summarizes the meaning of each field of the record.

3.2 A general correctness proof

In order to prove our general result, we need to define the usual semantic (data) relation among states of the source and target language and call relation between states. We remind that two states are in call relation whenever a call statement is fetched at state's current program counter. These two relations have to satisfy some condition, already explained at the beginning of this deliverable (see Section ??). In this section we will give some general conditions that these two relations have to satisfy in order to obtain the desired simulation result. We begin our analysis from the latter relation (the call one) and then we show how to relate it with a semantic relation satisfying some conditions that allow us to prove our general result.

3.2.1 A standard calling relation

Two states are in call relation whenever it is possible to fetch a call statement at the program counter given by the two states. We will exploit the properties of the translation explained in previous subsection in order to define a standard calling relation. We remind that the translation of a given statement I is a block of statements $b = \langle I_1, \ldots I_n \rangle$ with $n \geq 1$. When I is a call, then we will require that there is a unique $k \in [1, n]$ such I_k has to be a call statement (we will see the formalization of this condition in the following subsection when we relate the calling relation with the semantic one). The idea of defining a standard calling relation is to compute the code-point identifier of this I_k, starting from the code-point identifier of the statement I in the code of the source internal function. We will see how to use the information provided by the translation process (in particular the function `f_lbls`) in order to obtain this result. We will see that, for technical reason, we will compute the code point of I starting from the code point of I_k.

In order to explain how this machinery work. we need to enter more in the detail of the translation process. Given a step-statement I, formally its translation is a triple $f\_step(s) = \langle pre, p, post \rangle$ such that pre is a list of sequential statements called *preamble*, p is a step-statement (we call it *pivot*) and $post$ is again a list of sequential statements called *postamble*. When $pre = [s_1, \ldots, s_n]$ and $post = [s'_1, \ldots, s'_m]$, the corresponding block being the translation

Field	Explanation
`res_def_out_eq`, `pars_def_out_eq`, `ss_def_out_eq`	The return value location, the formal parameters and the syntactic stack size are all as dictated by the corresponding `b_graph_translate_data` fields.
`partition_lbls`, `partition_regs`	`f_lbls` and `f_regs` are partial partitions, i.e. all lists in their image are repetition-free and disjoint for different input values.
`freshness_lbls`, `freshness_regs`, `freshness_data_regs`	All lists of code-point identifiers and register generated at a source code-point, and the registers in `new_regs` are fresh: they were not in the former source universe and they are in the new one. All identifiers of pseudo-register being element of the field `new_regs` of the record of type `b_graph_translate_data` provided in input are fresh.
`data_regs_disjoint`	All pseudo-register in `new_regs` never appear in `f_regs`.
`multi_fetch_ok`	Given a statement I and a code-point identifier l of the source internal function such that I is located in l, if the translation process translate I into a statement block $\langle I_1, \ldots, I_n \rangle$ then $f\_lbls(l) = [l_1, \ldots, l_{n-1}]$ in the translated source code we have that I_1 is located in l with $l1$ as syntactical successor, I_2 is located in l_1 with l_2 as syntactical successor, and so on with the last statement I_n located in l_{n-1} and it may have a syntactical successor depending whether I is a step-statement or not: in the former case we have that the syntactical successor of I_n is the syntactical successor of I, in the latter case, I_n is a final statement.
`prologue_ok`	If the field `added_prologue` of the record of type `b_graph_translate_data` provided in input is empty, then the code-point identifier of the first instruction of the translated function is the same of the one of the source internal function. Otherwise the two code points are different, with the first instruction of the translated function being a cost-emission statement followed by the instructions of `added_prologue`; the last instruction of `added_prologue` has an identifier l as syntactical successor and l is the same identifier as the one of the first instruction of the source internal function and in l we fetch a NOP instruction.

Table 2: The explanation of the various fields of the `b_graph_translate_props` record.

of I is $\langle s_1, \ldots, s_n, p, s'_1, \ldots, s'_m \rangle$. In case I is a final statement, than its translation does not have postamble, i.e. it is a pair $f\_fin(s) = \langle pre, p \rangle$ where the pivot p is a final statement.

Given a statement I at a given code point l in the source internal function and given the pivot statement p of the translation of I staying at code-point l' in the translated internal function, there is an easy way to relate l and l'. Notice that, in case the preamble is empty, for the property of the translation process we have then $l = l'$, while if the preamble is non-empty. then l' is $n-1$-th element of $f\_lbls(l)$, where $n \geq 0$ is the length of the preamble. The matita's definition computing the code points according to the above mentioned specification is the following one.

```
definition sigma_label : ∀p_in,p_out : sem_graph_params.
joint_program p_in →(ident →option ℕ) →
(∀globals.joint_closed_internal_function p_in globals
        →bound_b_graph_translate_data p_in p_out globals) →
(block →list register) →lbl_funct_type →regs_funct_type →
block →label →option label :=
λp_in,p_out,prog,stack_size,init,init_regs,f_lbls,f_regs,bl,searched.
! bl ← code_block_of_block bl ;
! <id,fn> ←res_to_opt … (fetch_internal_function …
                        (joint_globalenv p_in prog stack_size) bl);
! <res,s> ←find ?? (joint_if_code ?? fn)
  (λlbl.λ_.
    match preamble_length … prog stack_size init init_regs f_regs bl lbl with
            [ None ⇒false
            | Some n ⇒match nth_opt ? n (lbl::(f_lbls bl lbl)) with
                      [ None ⇒false
                      | Some x ⇒eq_identifier … searched x
                      ]
            ]);
return res.
```

This function takes in input all the information provided by the translation process (in particular the function f_lbls), a function location and a code-point identifier l; it fetches the internal function of the source language in the corresponding location. Then it unfolds the code of the fetched function looking for a label l' and a statement I located in l', such that, either $l = l'$ in case the preamble of the translation of I is empty or l' is the $n-1$-th where $n \geq 0$ is the length of the preamble of I. The function $find$ is the procedure realizing this search. If $preamble\_length$ is the function giving in output the length of the preamble and if nth_opt is the function giving the n-th element of a list, then this condition can be summarized as following: we are looking for a label l' such that l is the n-th element of the list $l :: f\_lbls(l)$, where n is the length of the preamble.

We can prove that, starting from a code point identifier of the translated internal function, whenever there exists a code-point identifier in the source internal function satisfying the above condition, then it is always unique. The properties `partition_lbls` and `freshness_lbls` provided by the translation process are crucial in the proof of this statement. We can wrap this function inside the definition of the desired relation between program counter states in the following way The conditional at the beginning is put to deal with the pre-main case, which is translated without following the standard translation process we explained in previous section.

```
definition sigma_pc_opt :
∀p_in,p_out : sem_graph_params.
```

```
joint_program p_in →(ident →option ℕ) →
(∀globals.joint_closed_internal_function p_in globals
        →bound_b_graph_translate_data p_in p_out globals) →
(block →list register) →lbl_funct_type →regs_funct_type →
program_counter →option program_counter :=
λp_in,p_out,prog,stack_size,init,init_regs,f_lbls,f_regs,pc.
let target_point :=point_of_pc p_out pc in
if eqZb (block_id (pc_block pc)) (-1) then
 return pc
else
 ! source_point ←sigma_label p_in p_out prog stack_size init init_regs
                 f_lbls f_regs (pc_block pc) target_point;
 return pc_of_point p_in (pc_block pc) source_point.

definition sigma_stored_pc :=
λp_in,p_out,prog,stack_size,init,init_regs,f_lbls,f_regs,pc.
match sigma_pc_opt p_in p_out prog stack_size init init_regs f_lbls f_regs pc with
     [None ⇒null_pc (pc_offset … pc) | Some x ⇒x].
```

The main result about the program counter relation we have defined is the following. If we fetch a statement I in at a given program counter pc in the source program, then there is a program counter pc' in the target program which is in relation with pc (i.e. $sigma\_stored\_pcpc' = pc$) and the fetched statement at pc' is the pivot statement of the translation. The formalization of this statement in matita is given in the following.

```
lemma fetch_statement_sigma_stored_pc :
∀p_in,p_out,prog,stack_sizes,
init,init_regs,f_lbls,f_regs,pc,f,fn,stmt.
b_graph_transform_program_props p_in p_out stack_sizes
  init prog init_regs f_lbls f_regs →
block_id … (pc_block pc) ≠-1 →
let trans_prog :=b_graph_transform_program p_in p_out init prog in
fetch_statement p_in … (joint_globalenv p_in prog stack_sizes) pc =
return ⟨f,fn,stmt⟩→
∃data.bind_instantiate ?? (init … fn) (init_regs (pc_block pc)) = return data ∧
match stmt with
[ sequential step nxt ⇒
   ∃step_block : step_block p_out (prog_names … trans_prog).
   bind_instantiate ?? (f_step … data (point_of_pc p_in pc) step)
               (f_regs (pc_block pc) (point_of_pc p_in pc)) = return step_block ∧
   ∃pc'.sigma_stored_pc p_in p_out prog stack_sizes init
                                      init_regs f_lbls f_regs pc' = pc ∧
   ∃fn',nxt'.
   fetch_statement p_out … (joint_globalenv p_out trans_prog stack_sizes) pc' =
   if not_emptyb … (added_prologue … data) ∧
      eq_identifier … (point_of_pc p_in pc) (joint_if_entry … fn)
   then OK ? <f,fn',sequential ?? (NOOP … ) nxt'>
   else OK ? <f,fn',
            sequential ?? ((\snd(\fst step_block)) (point_of_pc p_in pc')) nxt'>
| final fin ⇒
    ∃fin_block.bind_instantiate ?? (f_fin … data (point_of_pc p_in pc) fin)
                (f_regs (pc_block pc) (point_of_pc p_in pc)) = return fin_block ∧
    ∃pc'.sigma_stored_pc p_in p_out prog stack_sizes init
```

```
                                        init_regs f_lbls f_regs pc' = pc ∧
    ∃fn'.fetch_statement p_out …
       (joint_globalenv p_out trans_prog stack_sizes) pc'
       = return ⟨f,fn',final ?? (\snd fin_block) ⟩
| FCOND abs _ _ _ ⇒⨂abs
].
```

If we combine the statement above with the fact that the pivot statement of the translation of a call statement is always a call statement (which we will formalize better in the following section), then we can define our standard calling relation in the following way.

```
(λs1:Σs: (joint_abstract_status (mk_prog_params p_in ??)).as_classifier ? s cl_call.
 λs2:Σs:(joint_abstract_status (mk_prog_params p_out ??)).as_classifier ? s cl_call.
pc ? s1 = sigma_stored_pc
  p_in p_out prog stack_sizes init init_regs f_lbls f_regs (pc ? s2)).
```

We stress the fact that such a call relation will be always defined in this way for all joint-languages, in an independent way from the specific pass. The only condition we will ask is that the pass should use the translation process we explain in the previous section.

3.2.2 The semantic relation

The semantic relation between states is the classical relation used in forward simulation proofs. It correlates the data of the status (e.g. register, memory, *etc*). We remind that the notion of state in joint language is summarized in the following record.

```
record state_pc (semp : sem_state_params) : Type[0] :=
  { st_no_pc :>state semp
  ; pc : program_counter
  ; last_pop : program_counter
  }.
```

It consists of three fields: the field `st_no_pc` contains all data information of the state (the content of the registers, of the memory and so on), the field `pc` contains the current program counter, while the field `last_pop` is the address of the last popped calling address when executing a return instruction.

The type of the semantic relation between state is the following.

```
definition joint_state_pc_relation :=
λP_in,P_out : sem_graph_params.state_pc P_in →state_pc P_out →Prop.
```

We would like to state some conditions the semantic relation between states have to satisfy in order to get our simulation result. We would like that this relation have some flavour of compositionality. In particular we would like that it depends strictly on the contents of the field `st_no_pc`, i.e. the field that really contains data information of the state. So we need also a data relation, i.e. a relation of this type.

```
definition joint_state_relation :=
λP_in,P_out.program_counter →state P_in →state P_out →Prop.
```

Notice that the data relation cab depend on a specific program counter of the source. This is done to capture complex data relations like the ones in the ERTL to LTL pass, in which you need to know where data in pseudo-registers of ERTL are stored by the translation (either

in hardware register or in memory) and this information depends on the code point on the statement being translated.

The compositionality requirement is expressed by several conditions (which are part of a bigger record). Condition `fetch_ok_sigma_state_ok` postulates that two state that are in semantic relation should have their data field also in relation. Condition `fetch_ok_pc_ok` postulates that two states that are in semantic relation should have the same program counter. This is due to the way the translation is performed. In fact a statement I at a code point l in the source internal function is translated with a block of instructions in the translated internal function whose initial statement is at the same code point l. The condition `fetch_ok_sigma_last_pop_ok` postulates that two states that are in semantic relation have the last popped calling address in call relation. Finally `st_rel_def` postulates that given two states having the same program counter, the last pop fields in call relation and the data fields also in data relation, then they are in semantic relation.

Another important condition is that the pivot statement of the translation of a call statement is always a call statement. This is important in order to obtain the correctness of the call relation and return relation between state. We call this condition `call_is_call`.

The conditions we are going to present now are standard semantic commutation lemmas that are commonly used when proving the correctness of the operational semantics of many imperative languages. We introduce some notation. We will use $I, J \ldots$ to range over by statements. We will use $l_1, \ldots, l_n$ to range over by code point identifiers. We will use $r_1, \ldots, r_n$ to range over by register identifiers. We will use $s_1, s_1', s_1'', \ldots$ to range over by states of programs of the source language. We will use $s_2, s_2', s_2'', \ldots$ to range over by states of programs of the target language. We denote respectively with $\simeq_S$, $\simeq_C$ and $\simeq_L$ the semantic relation, the call relation and the cost-label relation between states. These relations have been introduced at the beginning of this Deliverable (see Section ??). If $instr = [I_1, \ldots, I_n]$ is a list of instructions, then we write $s_i \xrightarrow{instr} s_i'$ ($i \in [1, 2]$) when s_i' is the state being the result of the evaluation of the sequence of instructions $instr$ (performed in the order they appear in the list) starting from the initial state s_i. When $instr = [I]$ is a singleton, we use to omit square brackets and we write $s_i \xrightarrow{I} s_i'$. We will denote with π_i ($i \in [1, t]$) the projecting functions of t-uples. We will denote with f_step and f_fin the translating functions of respectively step-statements and final statements. We remind that f_step gives a triple as output (a list of instruction called preamble,an instruction called pivot and a list of instructions called postamble) while f_fin gives a pair as output (a list of instruction called preamble and an instruction called pivot). Furthermore, we denote with *prologue* the content of the field `added_prologue` of the record provided in input to the translation process.

Many commutation conditions can be depicted using diagrams. We will use them to give a pictorial flavour of the conditions we will ask in order to obtain the final correctness statement. Given the states s_1, s_1', s_2, s_2' and the instructions $I, J_1, \ldots, J_k$, the following diagram

$$\begin{array}{ccc} s_1 & \xrightarrow{\quad I \quad} & s_1' \\ \Big| \simeq_S & & \Big| \simeq_S \\ s_2 & \xrightarrow{[J_1,\ldots,J_k]} & s_2' \end{array}$$

depicts a situation in which the state $s_1 \xrightarrow{I} s_1'$, $s_2 \xrightarrow{I} s_2'$, $s_1 \simeq_S s_2$ and $s_1' \simeq_S s_2'$.

Commutation of pre-main instructions (`pre_main_ok`). In order to get the commutation of pre-main instructions (states whose function location of program counter is -1), we have to prove the following condition: for all s_1, s_1', s_2 such that $s_1 \xrightarrow{I} s_1'$ and $s_1 \simeq_S s_2$, then there exists an s_2' such that $s_2 \xrightarrow{J} s_2'$ and $s_1 \simeq_S s_2'$ i.e. such that the following diagram commutes.

$$\begin{array}{ccc} s_1 & \xrightarrow{I} & s_1' \\ \big| \simeq_S & & \big| \simeq_S \\ s_2 & \xrightarrow{J} & s_2' \end{array}$$

Commutation of conditional jump (`cond_commutation`). For all s_1, s_1' and s_2 such that $s_1 \xrightarrow{COND\ r\ l} s_1'$ and $s_1 \simeq_S s_2$ then

- there are s_2^{fin} and s_2' such that $s_2 \xrightarrow{\pi_1(f\_step(COND\ r\ l))} s_2^{fin}$, $s_2^{fin} \overset{\pi_2(f\_step(COND\ r\ l))'}{s}{}_2$ and $s_1' \simeq_S s_2'$, i.e. the following diagram commutes

$$\begin{array}{ccccc} s_1 & & \xrightarrow{COND\ l\ r} & & s_1' \\ \big| \simeq_S & & & & \big| \simeq_S \\ s_2 & \xrightarrow{\pi_1(f\_step(COND\ r\ l))} & s_2^{fin} & \xrightarrow{\pi_2(f\_step(COND\ r\ l))} & s_2' \end{array}$$

- $\pi_3(f\_step(COND\ r\ l))$ is empty, while $\pi_2(f\_step(COND\ r\ l)) = COND\ r'\ l'$ is a conditional jump such that $l = l'$.

Commutation of sequential statements (`seq_commutation`). In case of a sequential statement I, its translation $f\_step(I) = \langle pre, J, post \rangle$ is coerced into a list of sequential statements pre `@` $[J]$ `@` $post$. Then we can state the condition in the following way. For all s_1, s_1', s_2 such that $s_1 \xrightarrow{I} s_1'$ and $s_1 \simeq_S s_2$ then there is s_2' such that $s_2 \xrightarrow{f\_step(I)} s_s'$ and $s_1' \simeq_S s_2'$, i.e. such that the following diagram commutes.

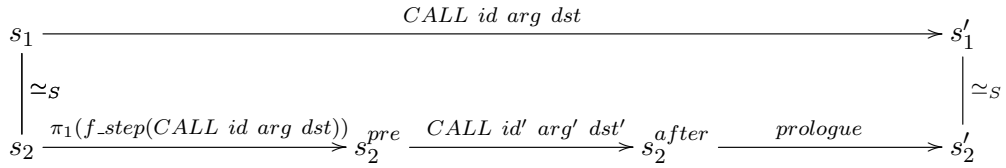
$$\begin{array}{ccc} s_1 & \xrightarrow{I} & s_1' \\ \big| \simeq_S & & \big| \simeq_S \\ s_2 & \xrightarrow{f\_step(I)} & s_2' \end{array}$$

Commutation of call statement (`call_commutation`). For all s_1, s_1', s_2 such that we have $s_1 \xrightarrow{CALL\ id\ arg\ dst} s_1'$, $s_1 \simeq_S s_2$ and the statement fetched in the target function at the program counter in call relation with the program counter of s_1 is $\pi_2(f\_step(CALL\ id\ arg\ dst)) = CALL\ id'\ arg'\ dst'$ for some id', ags', dst', then there are $s_2^{pre}, s_2^{after}, s_2'$ such that

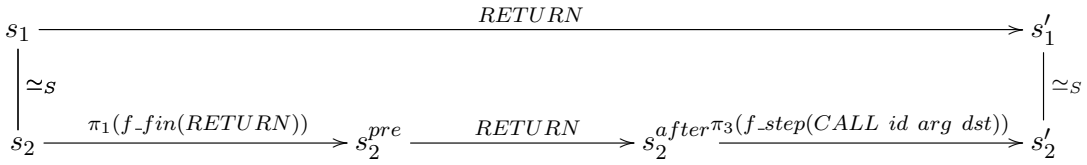
- $s_2 \xrightarrow{\pi_1(f\_step(CALL\ id\ arg\ dst))} s_2^{pre}$,
- $s_2^{pre} \xrightarrow{CALL\ id'\ \arg'\ dst'} s_2^{after}$,
- $s_2^{after} \xrightarrow{prologue} s_2'$,

- $s'_1 \simeq_L s_2^{after}$ and $s'_1 \simeq_S s'_2$.

The situation is depicted by the following diagram.



Commutation of return statement (`return_commutation`). For all s_1, s'_1, s_2 such that $s_1 \xrightarrow{RETURN} s'_1$, $s_1 \simeq_S s_2$, if $CALL\ id\ arg\ dst$ is the call statement that caused the function call ened by the current return (i.e. it is the statement whose code point identifier is the syntactical predecessor of the program counter of s'_1), then $\pi_2(f\_fin(RETURN)) = RETURN$, there are $s_2^{pre}, s_2^{after}, s'_2$ such that $s_2 \xrightarrow{\pi_1(f\_fin(RETURN))} s_2^{pre}$, $s_2^{pre} \xrightarrow{RETURN} s_2^{after}$, $s_2^{after} \xrightarrow{\pi_3(f\_step(CALL\ id\ arg\ dst))} s'_2$ and $s'_1 \simeq_S s'_2$. The following diagram depicts the above described requested situation.



3.2.3 Conclusion

After having provided a semantic relation among states that satisfies some conditions that correspond to commutation lemmas that are commonly proved in a forward simulation proof, it is possible to prove the general theorem. All these condition are summarized in a propositional record called `good_state_relation`. The statement we are able to prove have the following shape.

```
theorem joint_correctness : ∀p_in,p_out : sem_graph_params.
∀prog : joint_program p_in.∀stack_size : ident →option ℕ.
∀init : (∀globals.joint_closed_internal_function p_in globals →
        bound_b_graph_translate_data p_in p_out globals).
∀init_regs : block →list register.∀f_lbls : lbl_funct_type.
∀f_regs : regs_funct_type.∀st_no_pc_rel : joint_state_relation p_in p_out.
∀st_rel : joint_state_pc_relation p_in p_out.
good_state_relation p_in p_out prog stack_size init init_regs
 f_lbls f_regs st_no_pc_rel st_rel →
let trans_prog :=b_graph_transform_program … init prog in
∀init_in.
make_initial_state (mk_prog_params p_in prog stack_size) = OK ? init_in →
∃init_out.
make_initial_state (mk_prog_params p_out trans_prog stack_size) = OK ? init_out ∧
∃[1] R.
  status_simulation_with_init
    (joint_abstract_status (mk_prog_params p_in prog stack_size))
    (joint_abstract_status (mk_prog_params p_out trans_prog stack_size))
    R init_in init_out.
```

The module formalizing the formal machinery we described in this document consists of about 3000 lines of matita code. We stress the fact that this machinery proves general properties that do not depend on the specific back-end graph compiler pass.

References

[1] Jaap Boender and Claudio Sacerdoti Coen. On the correctness of a branch displacement algorithm. *CoRR*, abs/1209.5920, 2012.

[2] Paolo Tranquilli and Claudio Sacerdoti Coen. Certification of the preservation of structure by a compiler's back-end pass. Internal report, 2013.

On the correctness of an optimising assembler for the Intel MCS-51 microprocessor*

Dominic P. Mulligan and Claudio Sacerdoti Coen

Dipartimento di Scienze dell'Informazione,
Universitá degli Studi di Bologna

Abstract. We present a proof of correctness in Matita for an optimising assembler for the MCS-51 microcontroller. The efficient expansion of pseudoinstructions, namely jumps, into machine instructions is complex. We isolate the decision making over how jumps should be expanded from the expansion process itself as much as possible using 'policies', making the proof of correctness for the assembler more straightforward.
Our proof strategy contains a tracking facility for 'good addresses' and only programs that use good addresses have their semantics preserved under assembly, as we observe that it is impossible for an assembler to preserve the semantics of every assembly program. Our strategy offers increased flexibility over the traditional approach to proving the correctness of assemblers, wherein addresses in assembly are kept opaque and immutable. In particular, we may experiment with allowing the benign manipulation of addresses.

Keywords: Verified software, CerCo (Certified Complexity), MCS-51 microcontroller, Matita proof assistant

1 Introduction

We consider the formalisation of an assembler for the Intel MCS-51 8-bit microprocessor in the Matita proof assistant [1]. This formalisation forms a major component of the EU-funded CerCo ('Certified Complexity') project [3], concerning the construction and formalisation of a concrete complexity preserving compiler for a large subset of the C programming language.

The MCS-51 dates from the early 1980s and is commonly called the 8051/8052. Derivatives are still widely manufactured by a number of semiconductor foundries, with the processor being used especially in embedded systems.

The MCS-51 has a relative paucity of features compared to its more modern brethren, with the lack of any caching or pipelining features meaning that timing of execution is predictable, making the MCS-51 very attractive for CerCo's ends. However, the MCS-51's paucity of features—though an advantage in many

* The project CerCo acknowledges the financial support of the Future and Emerging Technologies (FET) programme within the Seventh Framework Programme for Research of the European Commission, under FET-Open grant number: 243881.

respects—also quickly becomes a hindrance, as the MCS-51 features a relatively minuscule series of memory spaces by modern standards. As a result our C compiler, to be able to successfully compile realistic programs for embedded devices, ought to produce 'tight' machine code.

To do this, we must solve the 'branch displacement' problem—deciding how best to expand pseudojumps to labels in assembly language to machine code jumps. The branch displacement problem arises when pseudojumps can be expanded in different ways to real machine instructions, but the different expansions are not equivalent (e.g. differ in size or speed) and not always correct (e.g. correctness is only up to global constraints over the compiled code). For instance, some jump instructions (short jumps) are very small and fast, but they can only reach destinations within a certain distance from the current instruction. When the destinations are too far away, larger and slower long jumps must be used. The use of a long jump may augment the distance between another pseudojump and its target, forcing another long jump use, in a cascade. The job of the optimising compiler (assembler) is to individually expand every pseudo-instruction in such a way that all global constraints are satisfied and that the compiled program is minimal in size and faster in concrete time complexity. This problem is known to be computationally hard for most CISC architectures (see [4]).

To simplify the CerCo C compiler we have chosen to implement an optimising assembler whose input language the compiler will target. Labels, conditional jumps to labels, a program preamble containing global data and a `MOV` instruction for moving this global data into the MCS-51's one 16-bit register all feature in our assembly language. We further simplify by ignoring linking, assuming that all our assembly programs are pre-linked.

Another complication we have addressed is that of the cost model. CerCo imposes a cost model on C programs or, more specifically, on simple blocks of instructions. This cost model is induced by the compilation process itself, and its non-compositional nature allows us to assign different costs to identical C statements depending on how they are compiled. In short, we aim to obtain a very precise costing for a program by embracing the compilation process, not ignoring it. At the assembler level, this is reflected by our need to induce a cost model on the assembly code as a function of the assembly program and the strategy used to solve the branch displacement problem. In particular, our optimising assembler should also return a map that assigns a cost (in clock cycles) to every instruction in the source program. We expect the induced cost to be preserved by the assembler: we will prove that the compiled code tightly simulates the source code by taking exactly the predicted amount of time.

Note that the temporal tightness of the simulation is a fundamental prerequisite of the correctness of the simulation because some functions of the MCS-51—timers and I/O—depend on the microprocessor's clock. If the pseudo- and concrete clock differ the result of an I/O operation may not be preserved.

Branch displacement algorithms must have a deep knowledge of the way the rest of the assembler works in order to build globally correct solutions. Proving their correctness is quite a complex task (see, for instance, the companion

paper [2]). Nevertheless, the correctness of the whole assembler only depends on the correctness of the branch displacement algorithm. Therefore, in the rest of the paper, we presuppose the existence of a correct policy, to be computed by a branch displacement algorithm if it exists. A policy is the decision over how any particular jump should be expanded; it is correct when the global constraints are satisfied. The assembler fails to assemble an assembly program if and only if a correct policy does not exist. This is stated in an elegant way in the dependent type of the assembler: the assembly function is total over a program, a policy and the proof that the policy is correct for that program.

A final complication in the proof is due to the kind of semantics associated to pseudo-assembly programs. Should assembly programs be allowed to freely manipulate addresses? The traditional answer is 'no': values stored in memory or registers are either concrete data or symbolic addresses. The latter can only be manipulated in very restricted ways and programs that do not do so are not assigned a semantics and cannot be reasoned about. All programs that have a semantics have it preserved by the assembler. We take an alternative approach, allowing programs to freely manipulate addresses non-symbolically but only granting a preservation of semantics to those programs that act in 'well-behaved' ways. In principle, this should allow some reasoning on the actual semantics of malign programs. In practice, we note how our approach facilitates more code reuse between the semantics of assembly code and object code.

The formalisation of the assembler and its correctness proof are given in Sect. 2. Sect. 3 presents the conclusions and relations with previous work.

Matita Matita is a proof assistant based on a variant of the Calculus of (Co)inductive Constructions [1]. It features dependent types that we exploit in the formalisation. The (simplified) syntax of the statements and definitions in the paper should be self-explanatory. Pairs are denoted with angular brackets, $\langle -, - \rangle$.

Matita features a liberal system of coercions. It is possible to define a uniform coercion $\lambda x.\langle x, ? \rangle$ from every type T to the dependent product $\Sigma x : T.P\ x$. The coercion opens a proof obligation that asks the user to prove that P holds for x. When a coercion must be applied to a complex term (a λ-abstraction, a local definition, or a case analysis), the system automatically propagates the coercion to the sub-terms For instance, to apply a coercion to force $\lambda x.M : A \to B$ to have type $\forall x : A.\Sigma y : B.P\ x\ y$, the system looks for a coercion from $M : B$ to $\Sigma y : B.P\ x\ y$ in a context augmented with $x : A$. This is significant when the coercion opens a proof obligation, as the user will be presented with multiple, but simpler proof obligations in the correct context. In this way, Matita supports the 'Russell' proof methodology developed by Sozeau in [12], with an implementation that is lighter and more tightly integrated with the system than that of Coq.

2 Certification of an optimising assembler

Our aim here is to explain the main ideas and steps of the certified proof of correctness for an optimising assembler for the MCS-51.

In Subsect. 2.1 we sketch an operational semantics (a realistic and efficient emulator) for the MCS-51. We also introduce a syntax for decoded instructions that will be reused for the assembly language.

In Subsect. 2.2 we describe the assembly language and its operational semantics. The latter is parametric in the cost model that will be induced by the assembler, reusing the semantics of the machine code on all 'real' instructions.

Branch displacement policies are introduced in Subsect. 2.3 where we also describe the assembler as a function over policies as previously described.

To prove our assembler correct we show that the object code given in output, together with a cost model for the source program, simulates the source program executed using that cost model. The proof can be divided into two main lemmas. The first is correctness with respect to fetching, described in Subsect. 2.4. Roughly it states that a step of fetching at the assembly level, returning the decoded instruction I, is simulated by n steps of fetching at the object level that returns instructions $J_1, \ldots, J_n$, where $J_1, \ldots, J_n$ is, amongst the possible expansions of I, the one picked by the policy. The second lemma states that $J_1, \ldots, J_n$ simulates I but only if I is well-behaved, i.e. manipulates addresses in 'good' ways. To keep track of well-behaved address manipulations we record where addresses are currently stored (in memory or an accumulator). We introduce a dynamic checking function that inspects this map to determine if the operation is well-behaved, with an affirmative answer being the pre-condition of the lemma. The second lemma is detailed in Subsect. 2.5 where we also establish correctness of our assembler as a composition of the two lemmas: programs that are well-behaved when executed under the cost model induced by the compiler are correctly simulated by the compiled code.

2.1 Machine code and its semantics

We implemented a realistic and efficient emulator for the MCS-51 microprocessor. An MCS-51 program is just a sequence of bytes stored in the read-only code memory of the processor, represented as a compact trie of bytes addressed by the program counter. The `Status` of the emulator is a record that contains the microprocessor's program counter, registers, stack pointer, clock, special function registers, data memory, and so on. The value of the code memory is a parameter of the record since it is not changed during execution.

The `Status` records is itself an instance of a more general datatype `PreStatus` that abstracts over the implementation of code memory in order to reuse the same datatype for the semantics of the assembly language in the next section.

The execution of a single instruction is performed by the `execute_1` function, parametric over the content `cm` of the code memory:

```
definition execute_1: ∀cm. Status cm → Status cm
```

The function `execute_1` closely matches the fetch-decode-execute cycle of the MCS-51 hardware, as described by a Siemen's manufacturer's data sheet [11]. Fetching and decoding are performed simultaneously: we first fetch, using the

program counter, from code memory the first byte of the instruction to be executed, decoding the resulting opcode, fetching more bytes as is necessary to decode the arguments. Decoded instructions are represented by the `instruction` data type which extends a data type of `preinstruction`s that will be reused for the assembly language.

```
inductive preinstruction (A: Type[0]): Type[0] :=
 | ADD: ⟦acc_a⟧ → ⟦registr; direct; indirect; data⟧ → preinstruction A
 | DEC: ⟦acc_a; registr; direct; indirect⟧ → preinstruction A
 | JB: ⟦bit_addr⟧ → A → preinstruction A
 | ...
inductive instruction: Type[0] :=
 | LCALL: ⟦addr16⟧ → instruction
 | AJMP: ⟦addr11⟧ → instruction
 | RealInstruction: preinstruction ⟦relative⟧ → instruction.
 | ...
```

The MCS-51 has many operand modes, but an unorthogonal instruction set: every opcode is only enable for a finite subset of the possible operand modes. Here we exploit dependent types and an implicit coercion to synthesise the type of arguments of opcodes from a vector of names of operand modes. For example, `ACC` has two operands, the first one constrained to be the `A` accumulator, and the second one to be a disjoint union of register, direct, indirect and data operand modes.

The parameterised type A of `preinstruction` represents the addressing mode allowed for conditional jumps; in the `RealInstruction` constructor we constraint it to be a relative offset. A different instantiation (labels) will be used in the next section for assembly programs.

Once decoded, execution proceeds by a case analysis on the decoded instruction, following the operation of the hardware. For example, the `DEC` preinstruction ('decrement') is executed as follows:

```
 | DEC addr ⇒
  let s := add_ticks1 s in
  let ⟨result, flags⟩ := sub_8_with_carry (get_arg_8 s true addr)
   (bitvector_of_nat 8 1) false in
     set_arg_8 s addr result
```

Here, `add_ticks1` models the incrementing of the internal clock of the microprocessor; it is a parameter of the semantics of `preinstruction`s that is fixed in the semantics of `instruction`s according to the manufacturer datasheet.

2.2 Assembly code and its semantics

An assembly program is a list of potentially labelled pseudoinstructions, bundled with a preamble consisting of a list of symbolic names for locations in data memory (i.e. global variables). All preinstructions are pseudoinstructions, but conditional jumps are now only allowed to use `Identifiers` (labels) as their target.

```
inductive pseudo_instruction: Type[0] :=
  | Instruction: preinstruction Identifier → pseudo_instruction
    ...
  | Jmp: Identifier → pseudo_instruction
  | Call: Identifier → pseudo_instruction
  | Mov: ⟦dptr⟧ → Identifier → pseudo_instruction.
```

The pseudoinstructions `Jmp`, `Call` and `Mov` are generalisations of machine code unconditional jumps, calls and move instructions respectively, all of whom act on labels, as opposed to concrete memory addresses. The object code calls and jumps that act on concrete memory addresses are ruled out of assembly programs not being included in the preinstructions (see previous Section).

Execution of pseudoinstructions is an endofunction on `PseudoStatus`. A `PseudoStatus` is an instance of `PreStatus` that differs from a `Status` only in the datatype used for code memory: a list of optionally labelled pseudoinstructions versus a trie of bytes. The `PreStatus` type is crucial for sharing the majority of the semantics of the two languages.

Emulation for pseudoinstructions is handled by `execute_1_pseudo_instruction`:

```
definition execute_1_pseudo_instruction:
 ∀cm. ∀costing:(∀ppc: Word. ppc < |snd cm| → nat × nat).
  ∀s:PseudoStatus cm. program_counter s < |snd cm| → PseudoStatus cm
```

The type of `execute_1_pseudo_instruction` is more involved than that of `execute_1`. The first difference is that execution is only defined when the program counter points to a valid instruction, i.e. it is smaller than the length `|snd cm|` of the program. The second difference is the abstraction over the cost model, abbreviated here as *costing*. The costing is a function that maps valid program counters to pairs of natural numbers representing the number of clock ticks used by the pseudoinstructions stored at those program counters. For conditional jumps the two numbers differ to represent different costs for the 'true branch' and the 'false branch'. In the next section we will see how the optimising assembler induces the only costing (induced by the branch displacement policy deciding how to expand pseudojumps) that is preserved by compilation.

Execution proceeds by first fetching from pseudo-code memory using the program counter—treated as an index into the pseudoinstruction list. This index is always guaranteed to be within the bounds of the pseudoinstruction list due to the dependent type placed on the function. No decoding is required. We then proceed by case analysis over the pseudoinstruction, reusing the code for object code for all instructions present in the MCS-51's instruction set. For all newly introduced pseudoinstructions, we simply translate labels to concrete addresses before behaving as a 'real' instruction.

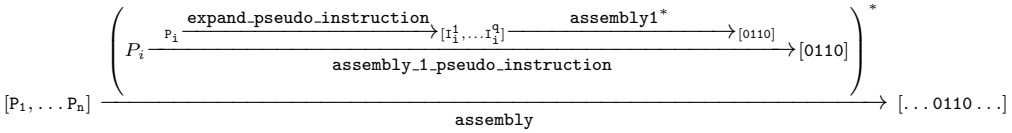
We do not perform any kind of symbolic execution, wherein data is the disjoint union of bytes and addresses, with addresses kept opaque and immutable. Labels are immediately translated before execution to concrete addresses, and registers and memory locations only ever contain bytes, never labels. As a consequence, we allow the programmer to mangle, change and generally adjust addresses as

they want, under the proviso that the translation process may not be able to preserve the semantics of programs that do this. This will be further discussed in Subsect. 2.5. The only limitation introduced by this approach is that the size of assembly programs is bounded by 2^{16}.

2.3 The assembler

The assembler takes in input an assembly program made of pseudoinstructions and a branch displacement policy for it. It returns both the object code (a list of bytes to be loaded in code memory for execution) and the costing for the source.

Conceptually the assembler works in two passes. The first pass expands every pseudoinstruction into a list of machine code instructions using the function `expand_pseudo_instruction`. The policy determines which expansion among the alternatives will be chosen for pseudo-jumps and pseudo-calls. Once the expansion is performed, the cost of the pseudoinstruction is defined as the cost of the expansion. The second pass encodes as a list of bytes the expanded instruction list by mapping the function `assembly1` across the list, and then flattening.



In order to understand the type for the policy, we briefly hint at the branch displacement problem for the MCS-51. A detailed description is found in [2]. The MCS-51 features three unconditional jump instructions: `LJMP` and `SJMP`—'long jump' and 'short jump' respectively—and an 11-bit oddity of the MCS-51, `AJMP`. Each of these three instructions expects arguments in different sizes and behaves in markedly different ways: `SJMP` may only perform a 'local jump' to an address closer then 2^7 bytes; `LJMP` may jump to any address in the MCS-51's memory space and `AJMP` may jump to any address in the current memory page. Memory pages partition the code memory into 2^8 disjoint areas. The size of each opcode is different, with long jumps being larger than the other two. Because of the presence of `AJMP`, an optimal global solution may be locally unoptimal, employing a long jump where a shorter one could be used to force later jumps to stay inside single memory pages.

Similarly, a conditional pseudojump must be translated potentially into a configuration of machine code instructions, depending on the distance to the jump's target. For example, to translate a jump to a label, a single conditional jump pseudoinstruction may be translated into a block of three real instructions as follows (here, `JZ` is 'jump if accumulator is zero'):

```
        JZ  label                    JZ    size of SJMP instruction
        ...          translates to   SJMP  size of LJMP instruction
label:  MOV A B          ⟹          LJMP  address of label
                                     ...
                                     MOV   A B
```

Naturally, if `label` is 'close enough', a conditional jump pseudoinstruction is mapped directly to a conditional jump machine instruction; the above translation only applies if `label` is not sufficiently local.

The cost returned by the assembler for a pseudoinstruction is set to be the cost of its expansion in clock cycles. For conditional jumps that are expanded as just shown, the costs of taking the true and false branches are different and both need to be returned.

The `expand_pseudo_instruction` function is driven by a policy in the choice of expansion of pseudoinstructions. The simplest idea is then to define policies as functions that maps jumps to their size. This simple idea, however, is impractical because short jumps require the offset of the target. For instance, suppose that at address `ppc` in the assembly program we found `Jmp l` such that l is associated to the pseudo-address `a` and the policy wants the `Jmp` to become a `SJMP` δ. To compute δ, we need to know what the addresses `ppc+1` and `a` will become in the assembled program to compute their difference. The address `a` will be associated to is a function of the expansion of all the pseudoinstructions between `ppc` and `a`, which is still to be performed when expanding the instruction at `ppc`.

To solve the issue, we define the policy `policy` as a map from a valid pseudo-address to the corresponding address in the assembled program. Therefore, δ in the example above can be computed simply as `policy(a) - policy(ppc + 1)`. Moreover, the `expand_pseudo_instruction` emits a `SJMP` only after verifying for each `Jmp` that $\delta < 128$. When this is not the case, the function emits an `AJMP` if possible, or an `LJMP` otherwise, therefore always picking the locally best solution. In order to accommodate those optimal solutions that require local sub-optimal choices, the policy may also return a Boolean used to force the translation of a `Jmp` into a `LJMP` even if $\delta < 128$. An essentially identical mechanism exists for call instructions and conditional jumps.

In order for the translation of a jump to be correct, the address associated to `a` by the policy and by the assembler must coincide. The latter is the sum of the size of all the expansions of the pseudo-instructions that precede the one at address `a`: the assembler just concatenates all expansions sequentially. To grant this property, we impose a correctness criterion over policies. A policy is correct when `policy(0) = 0` and for all valid pseudoaddresses `ppc`

$$\texttt{policy(ppc+1) = policy(ppc) + instruction\_size(ppc)} \leq 2^{16}$$

Here `instruction_size(ppc)` is the size in bytes of the expansion of the pseudoinstruction found at `pcc`, i.e. the length of `assembly_1_pseudo_instruction(ppc)`.

2.4 Correctness of the assembler with respect to fetching

We now begin the proof of correctness of the assembler. Correctness consists of two properties: firstly that the assembly process never fails when fed a correct policy and secondly the object code returned simulates the source code when the latter is executed according to the cost model also returned by the assembler. This second property can be further decomposed into two main properties: correctness with respect to fetching and decoding and correctness with respect to execution.

Informally, correctness with respect to fetching is the following statement: when we fetch an assembly pseudoinstruction `I` at address `ppc`, then we can fetch the expanded pseudoinstruction(s) `[J1, ..., Jn] = fetch_pseudo_instruction ... I ppc` from `policy ppc` in the code memory obtained by loading the assembled object code. This section reviews the main steps to prove correctness with respect to fetching. Subsect. 2.5 deals with correctness with respect to execution: the instructions `[J1, ..., Jn]` simulate the pseudoinstruction `I`.

The (slightly simplified) Russell type for the `assembly` function is:

```
definition assembly:
 ∀program: pseudo_assembly_program. ∀policy.
  Σassembled: list Byte × (BitVectorTrie nat 16).
   |program| ≤ 2^16 → policy is correct for program →
     policy (|program|) = |fst assembled| ≤ 2^16 ∧
     ∀ppc: pseudo_program_counter. ppc < 2^16 →
       let pseudo_instr := fetch from program at ppc in
       let assembled_i := assemble pseudo_instr in
         |assembled_i| ≤ 2^16 ∧
           ∀n: nat. n < |assembled_i| → ∃k: nat.
             nth assembled_i n = nth assembled (policy ppc + k).
```

In plain words, the type of `assembly` states the following. Given a correct policy for the program to be assembled, the assembler never fails and returns some object code and a costing function. Under the condition that the policy is 'correct' for the program and the program is fully addressable by a 16-bit word, the object code is also fully addressable by a 16-bit word. Moreover, the result of assembling the pseudoinstruction obtained fetching from the assembly address `ppc` is a list of bytes found in the generated object code starting from the object code address `policy(ppc)`.

Essentially the type above states that the `assembly` function correctly expands pseudoinstructions, and that the expanded instruction reside consecutively in memory. The fundamental hypothesis is correctness of the policy which allows us to prove the inductive step of the proof, which proceeds by induction over the assembly program. It is then straightforward to lift the property from lists of bytes (object code) to tries of bytes (i.e. code memories after loading). The `assembly_ok` lemma does the lifting.

We have established that every pseudoinstruction is compiled to a sequence of bytes that is found in memory at the expect place. This does not trivially imply that those bytes will be decoded in a correct way to recover the pseudoinstruction expansion. Indeed, we first need to prove a lemma that establishes that the `fetch` function is the left inverse of the `assembly1` function:

```
lemma fetch_assembly:
 ∀pc: Word.
 ∀i: instruction.
 ∀code_memory: BitVectorTrie Byte 16.
 ∀assembled: list Byte.
  assembled = assemble i →
```

```
 let len := |assembled| in
 let pc_plus_len := pc + len in
  encoding_check pc pc_plus_len assembled →
  let ⟨instr, pc', ticks⟩ := fetch pc in
   instr = i ∧ ticks = (ticks_of_instruction instr) ∧ pc' = pc_plus_len.
```

We read `fetch_assembly` as follows. Any time the encoding `assembled` of an instruction `i` is found in code memory starting at position `pc` (the hypothesis `encoding_check` ...), when we fetch at address `pc` retrieving the instruction `i`, the new program counter is `pc` plus the length of the encoding, and the cost of the fetched instruction is the one predicted for `i`. Or, in plainer words, assembling, storing and then immediately fetching gets you back to where you started.

Remembering that `assembly_1_pseudo_instruction` is the composition of `assembly1` with `expand_pseudo_instruction`, we can lift the previous result from instructions (already expanded) to pseudoinstructions (to be expanded):

```
lemma fetch_assembly_pseudo:
 ∀program: pseudo_assembly_program.
 ∀policy,ppc,code_memory.
 let ⟨preamble, instr_list⟩ := program in
 let pi := π1 (fetch_pseudo_instruction instr_list ppc) in
 let pc := policy ppc in
 let instructions := expand_pseudo_instruction policy ppc pi in
 let ⟨l, a⟩ := assembly_1_pseudoinstruction policy ppc pi in
 let pc_plus_len := pc + l in
  encoding_check code_memory pc pc_plus_len a →
   fetch_many code_memory pc_plus_len pc instructions.
```

Here, `l` is the number of machine code instructions the pseudoinstruction at hand has been expanded into. We assemble a single pseudoinstruction with `assembly_1_pseudoinstruction`, which internally calls `expand_pseudo_instruction`. The function `fetch_many` fetches multiple machine code instructions from code memory and performs some routine checks.

Intuitively, Lemma `fetch_assembly_pseudo` says that expanding a pseudoinstruction into n instructions, encoding the instructions and immediately fetching n instructions back yield exactly the expansion.

Combining `assembly_ok` with the previous lemma and a proof of correctness of loading object code in memory, we finally get correctness of the assembler with respect to fetching:

```
lemma fetch_assembly_pseudo2:
 ∀program. |snd program| ≤ 2^16 →
 ∀policy. policy is correct for program →
 ∀ppc. ppc < |snd program| →
  let ⟨assembled, costs'⟩ := π1 (assembly program policy) in
  let cmem := load_code_memory assembled in
  let ⟨pi, newppc⟩ := fetch_pseudo_instruction program ppc in
  let instructions := expand_pseudo_instruction policy ppc pi in
    fetch_many cmem (policy newppc) (policy ppc) instructions.
```

Here we use π_1 to project the existential witness from the Russell-typed function `assembly`. We read `fetch_assembly_pseudo2` as follows. Suppose we are given an assembly program which can be addressed by a 16-bit word and a policy that is correct for this program. Suppose we are able to successfully assemble an assembly program using `assembly` and produce a code memory, `cmem`. Then, fetching a pseudoinstruction from the pseudo-code memory stored in the interval $[ppc, newppc]$ corresponds to fetching a sequence of instructions from the real code memory, stored in the interval $[policy(ppc), policy(ppc + 1)]$. The correspondence is precise: the fetched instructions are exactly those obtained expanding the pseudoinstruction according to policy.

In order to complete the proof of correctness of the assembler, we need to prove that each pseudoinstruction is simulated by the execution of its expansion (correctness with respect to execution). In general this is not the case when instructions freely manipulate program addresses. Characterising well-behaved programs and proving correctness with respect to expansion is discussed next.

2.5 Correctness for 'well-behaved' assembly programs

Most assemblers can map a single pseudoinstruction to zero or more machine instructions, whose size (in bytes) is not independent of the expansion. The assembly process therefore always produces a map (which for us is just the policy) that associates to each assembly address `a` a code memory address `policy(a)` where the instructions that correspond to the pseudoinstruction at `a` are located. Ordinarily, the map is not just a linear function, but depends on the local choices and global optimisations performed.

During execution of assembly code, addresses can be stored in memory locations or in the registers. Moreover, arithmetical operations can be applied to addresses, for example to compare them or to shift a function pointer in order to implement C `switch` statements. In order to show that the object code simulates the assembly code we must compute the processor status that corresponds to the assembly status. In particular, those `a` in memory that are used as data should be preserved as `a`, but those used as addresses should be changed into `policy(a)`. Moreover, every arithmetic operation should commute with `policy` in order for the semantics to be preserved.

Following the previous observation, we can ask if it is possible at all for an assembler to preserve the semantics of an assembly program. The traditional approach to the verification of assemblers answers the question in the affirmative by restricting the semantics of assembly programs. In particular, the type of memory cells and registers is set to the disjoint union of data and symbolic addresses, and the semantics is always forced to consider all possible combinations of arguments (data vs. data, data vs. addresses, and so on), rejecting operations whose semantics cannot be preserved.

$$\texttt{Mem} : \texttt{Addr} \to \texttt{Bytes} + \texttt{Addr} \qquad [\![-]\!] : \texttt{Instr} \to \texttt{Mem} \to \texttt{option Mem}$$

$$[\![\texttt{MUL @A1 @A2}]\!]^{\texttt{M}} = \begin{cases} \texttt{Byte b1, Byte b2} & \to \texttt{Some(M with accumulator} := \texttt{b1} + \texttt{b2}) \\ -, \texttt{ Addr a} & \to \texttt{None} \\ \texttt{Addr a, } - & \to \texttt{None} \end{cases}$$

This approach has two main limitations. The first one is that it does not assign any semantics to interesting programs that could intentionally mangle addresses for malign (e.g. viruses) or benign (e.g. operating systems) purposes. The second is that it does not allow one to adequately share the semantics of assembly pseudoinstructions and object code instructions: only the `Byte-Byte` branch above can share the semantics with the object code `MUL`.

In this paper we have already taken a different approach from Sect. 2.2, where we have assigned a semantics to every assembly program by not distinguishing at all between data and symbolic addresses. Memory cells and registers always hold bytes, and symbolic labels are mapped to absolute addresses before execution. Consequently we do not expect that all assembly programs will have their semantics respected by object code. We call those programs that do *well-behaved*. Further, we can now reason over the semantics of programs that are not well-behaved, and that we can handle well-behavedness as an open predicate, recognising more and more good behaviours as required. Naturally, compilers that target our assembler will need to produce well-behaved programs, which is usually the case by construction.

The definition of well-behavedness we employ uses a map to keep track of the memory locations and registers that hold addresses during execution of an assembly program. The map acts as a sort of dynamic typing system sitting atop memory. This approach seems similar to one taken by Tuch *et al* [13] for reasoning about low-level C code.

The semantics of an assembly program is then augmented with a function that at each execution step updates the map, signalling an error when the program performs an ill-behaved operation. The actual computation is not performed by this mechanism, being already part of the assembly semantics.

$$\texttt{AddrMap} : \texttt{Addr} \to \{\mathit{Data}, \mathit{Addr}\} \qquad [\![-]\!] : \texttt{Instr} \to \texttt{AddrMap} \to \texttt{option AddrMap}$$

$$[\![\texttt{MUL @A1 @A2}]\!]^{\texttt{M}} = \begin{cases} \texttt{Data, Data} & \to \mathit{Some}(M\text{with accumulator} := \texttt{Data}) \\ -, \texttt{ Addr a} & \to \texttt{None} \\ \texttt{Addr a, } - & \to \texttt{None} \end{cases}$$

To prove semantic preservation we must associate an object code status to each assembly pseudostatus. This operation is driven by the current `AddrMap`: if at address `a` the assembly level memory holds `d`, then if `AddrMap(a) = Data` the object code memory will hold `d` (data is preserved), otherwise it will hold `policy(d)`. If all the operations accepted by the address update map are well-behaved, this is sufficient to show preservation of the semantics for those computation steps that are well-behaved, i.e. such that the map update does not fail.

We now apply the previous idea to the MCS-51, an 8-bit processor whose code memory is word addressed. All MCS-51 operations can therefore only

manipulate and store one half of the address at a time (lower or higher bits). For instance, a memory cell could contain just the lower 8 bits of an address `a`. The corresponding cell at object code level must therefore hold the lower 8 bits of `policy(a)`, which can be computed only if we can also retrieve the higher 8 bits of `a`. We achieve this by storing the missing half of an address in the `AddrMap` — called `internal_pseudo_address_map` in the formalisation.

```
definition address_entry := upper_lower × Byte.
definition internal_pseudo_address_map :=
  (BitVectorTrie address_entry 7) × (BitVectorTrie address_entry 7)
    × (option address_entry).
```

Here, `upper_lower` is an inductive type with two constructors: `Upper` and `Lower`. The map consists of three components to track addresses in lower and upper internal ram and also in the accumulator `A`. If an assembly address `a` holds `h` and if the current `internal_pseudo_address_map` maps `a` to $\langle$ `Upper, l`$\rangle$, then `h` is the upper part of the `h·l` address and `a` will hold the upper part of `policy(h·l)` in the object code status.

The relationship between assembly pseudostatus and object code status is computed by the following function which deterministically maps each pseudostatus into a corresponding status. It takes in input the policy and both the current pseudostatus and the current tracking map in order to identify those memory cells and registers that hold fragments of addresses to be mapped using `policy` as previously explained. It also calls the assembler to replace the code memory of the assembly status (i.e. the assembly program) with the object code produced by the assembler.

```
definition status_of_pseudo_status:
 internal_pseudo_address_map → ∀pap. ∀ps: PseudoStatus pap.
 ∀policy. Status (code_memory_of_pseudo_assembly_program pap policy)
```

The function that implements the tracking map update, previously denoted by $[\![-]\!]$, is called `next_internal_pseudo_address_map` in the formalisation. For the time being, we accept as good behaviours address copying amongst memory cells and the accumulator (`MOV` pseudoinstruction) and the use of the `CJNE` conditional jump that compares two addresses and jumps to a given label if the two labels are equal. Moreover, `RET` to return from a function call is well-behaved iff the lower and upper parts of the return address, fetched from the stack, are both marked as complementary parts of the same address (i.e. `h` is tracked as $\langle$`Upper,l`$\rangle$ and `l` is tracked as $\langle$`Lower,h`$\rangle$. These three operations are sufficient to implement the backend of the CerCo compiler. Other good behaviours could be recognised in the future, for instance in order to implement the C branch statement efficiently.

```
definition next_internal_pseudo_address_map: internal_pseudo_address_map →
 ∀cm. (Identifier → PseudoStatus cm → Word) → ∀s: PseudoStatus cm.
   program_counter s < 2^16 → option internal_pseudo_address_map
```

We now state the (simplified) statement of correctness of our compiler, whose proofs combines correctness with respect to fetching and correctness with respect

to execution. It states that the well-behaved execution of a single assembly pseudoinstruction according to the cost model induced by compilation is correctly simulated by the execution of (possibly) many machine code instructions.

```
theorem main_thm:
 ∀M, M': internal_pseudo_address_map.
 ∀program: pseudo_assembly_program.
 ∀program_in_bounds: |program| ≤ 2^16.
 ∀policy. policy is correct for program.
 ∀ps: PseudoStatus program. ps < |program|.
  next_internal_pseudo_address_map M program ...= Some M' →
   ∃n. execute n (status_of_pseudo_status M ps policy) =
    status_of_pseudo_status M'
     (execute_1_pseudo_instruction program (ticks_of program policy) ps)
     policy.
```

The statement is standard for forward simulation, but restricted to `PseudoStatuses ps` whose tracking map is `M` and who are well-behaved according to `internal_pseudo_address_map M`. The `ticks_of program policy` function returns the costing computed by assembling the `program` using the given `policy`. An obvious corollary of `main_thm` is the correct simulation of n well-behaved steps by some number of steps m, where each step must be well-behaved with respect to the tracking map returned by the previous step.

3 Conclusions

We are proving the correctness of an assembler for MCS-51 assembly language. Our assembly language features labels, arbitrary conditional and unconditional jumps to labels, global data and instructions for moving this data into the MCS-51's single 16-bit register. Expanding these pseudoinstructions into machine code instructions is not trivial, and the proof that the assembly process is 'correct', in that the semantics of a subset of assembly programs are not changed is complex.

The formalisation is a component of CerCo which aims to produce a verified concrete complexity preserving compiler for a large subset of the C language. The verified assembler, complete with the underlying formalisation of the semantics of MCS-51 machine code, will form the bedrock layer upon which the rest of CerCo will build its verified compiler platform.

We may compare our work to an 'industrial grade' assembler for the MCS-51: SDCC [10], the only open source C compiler that targets the MCS-51 instruction set. It appears that all pseudojumps in SDCC assembly are expanded to `LJMP` instructions, the worst possible jump expansion policy from an efficiency point of view. Note that this policy is the only possible policy *in theory* that makes every assembly program well-behaved, preserving its the semantics during the assembly process. This comes at the expense of assembler completeness as the generated program may be too large for code memory, there being a trade-off between the completeness of the assembler and the efficiency of the assembled program. The

definition and proof of a terminating, correct jump expansion policy is described elsewhere [2].

Verified assemblers could also be applied to the verification of operating system kernels and other formalised compilers. For instance the verified seL4 kernel [5], CompCert [7] and CompCertTSO [14] all explicitly assume the existence of trustworthy assemblers. The fact that an optimising assembler cannot preserve the semantics of all assembly programs may have consequences for these projects.

Our formalisation exploits dependent types in different ways and for multiple purposes. The first purpose is to reduce potential errors in the formalisation of the microprocessor. Dependent types are used to constrain the size of bitvectors and tries that represent memory quantities and memory areas respectively. They are also used to simulate polymorphic variants in Matita, in order to provide precise typings to various functions expecting only a subset of all possible addressing modes that the MCS-51 offers. Polymorphic variants nicely capture the absolutely unorthogonal instruction set of the MCS-51 where every opcode must accept its own subset of the 11 addressing mode of the processor.

The second purpose is to single out sources of incompleteness. By abstracting our functions over the dependent type of correct policies, we were able to manifest the fact that the compiler never refuses to compile a program where a correct policy exists. This also allowed to simplify the initial proof by dropping lemmas establishing that one function fails if and only if some previous function does so.

Finally, dependent types, together with Matita's liberal system of coercions, allow us to simulate almost entirely in user space the proof methodology 'Russell' of Sozeau [12]. Not every proof has been carried out in this way: we only used this style to prove that a function satisfies a specification that only involves that function in a significant way. It would not be natural to see the proof that fetch and assembly commute as the specification of one of the two functions.

Related work We are not the first to consider the correctness of an assembler for a non-trivial assembly language. The most impressive piece of work in this domain is Piton [8], a stack of verified components, written and verified in ACL2, ranging from a proprietary FM9001 microprocessor verified at the gate level, to assemblers and compilers for two high-level languages—Lisp and μGypsy [9]. Klein and Nipkow also provide a compiler, virtual machine and operational semantics for the Jinja [6] language and prove that their compiler is semantics and type preserving.

Though other verified assemblers exist what sets our work apart from that above is our attempt to optimise the generated machine code. This complicates a formalisation as an attempt at the best possible selection of machine instructions must be made—especially important on devices with limited code memory. Care must be taken to ensure that the time properties of an assembly program are not modified by assembly lest we affect the semantics of any program employing the MCS-51's I/O facilities. This is only possible by inducing a cost model on the source code from the optimisation strategy and input program.

Resources Our source files are available at http://cerco.cs.unibo.it. We assumed several properties of 'library functions', e.g. modular arithmetic and datastructure manipulation. We axiomatised various small functions needed to complete the main theorems, as well as some 'routine' proof obligations of the theorems themselves, in focusing on the main meat of the theorems. We believe that the proof strategy is sound and that all axioms can be closed, up to minor bugs that should have local fixes that do not affect the global proof strategy.

The complete development is spread across 29 files with around 20,000 lines of Matita source. Relevant files are: `AssemblyProof.ma`, `AssemblyProofSplit.ma` and `AssemblyProofSplitSplit.ma`, consisting of approximately 4500 lines of Matita source. Numerous other lines of proofs are spread all over the development because of dependent types and the Russell proof style, which does not allow one to separate the code from the proofs. The low ratio between source lines and the number of lines of proof is unusual, but justified by the fact that the pseudo-assembly and the assembly language share most constructs and large swathes of the semantics are shared.

References

1. Asperti, A., Sacerdoti Coen, C., Tassi, E., Zacchiroli, S.: User interaction with the Matita proof assistant. Automated Reasoning 39, 109–139 (2007)
2. Boender, J., Sacerdoti Coen, C.: On the correctness of a branch displacement algorithm. http://arxiv.org/abs/1209.5920 (2012)
3. The CerCo FET-Open project. http://cerco.cs.unibo.it/ (2011)
4. Branch displacement optimisation. http://groups.google.com/group/alt.lang.asm/msg/d31192d442accad3 (2006)
5. Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an operating system kernel. In: SOSP (2009)
6. Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine and compiler. ACM Transactions on Programming Languages and Systems 28(4), 619–695 (2006)
7. Leroy, X.: A formally verified compiler back-end. Automated Reasoning 43(4), 363–446 (2009)
8. Moore, J.S.: Piton: A mechanically verified assembly language, Automated Reasoning Series, vol. 3. Springer (1996)
9. Moore, J.S.: A grand challenge proposal for formal methods (2005)
10. Small device C compiler 3.0.0. http://sdcc.sourceforge.net/ (2011)
11. Siemens Semiconductor Group 8051 derivative instruction set. http://www.win.tue.nl/~aeb/comp/8051/instruction-set.pdf (2011)
12. Sozeau, M.: Subset coercions in Coq. In: Types. pp. 237–252 (2006)
13. Tuch, H., Klein, G., Norrish, M.: Types, bytes, and separation logic. In: POPL. pp. 97–108 (2007)
14. Ševčík, J., Vafeiadis, V., Zappa Nardelli, F., Jagannathan, S., Sewell, P.: Relaxed-memory concurrency and verified compilation. In: POPL (2011)

On the correctness of a branch displacement algorithm*

Jaap Boender and Claudio Sacerdoti Coen

Dipartimento di Scienze dell'Informazione,
Università degli Studi di Bologna

Abstract The branch displacement problem is a well-known problem in assembler design. It revolves around the feature, present in several processor families, of having different instructions, of different sizes, for jumps of different displacements. The problem, which is provably NP-hard, is then to select the instructions such that one ends up with the smallest possible program.
During our research with the CerCo project on formally verifying a C compiler, we have implemented and proven correct an algorithm for this problem. In this paper, we discuss the problem, possible solutions, our specific solutions and the proofs.

Keywords: formal verification, assembler, branch displacement optimisation

1 Introduction

The problem of branch displacement optimisation, also known as jump encoding, is a well-known problem in assembler design [4]. It is caused by the fact that in many architecture sets, the encoding (and therefore size) of some instructions depends on the distance to their operand (the instruction 'span'). The branch displacement optimisation problem consists of encoding these span-dependent instructions in such a way that the resulting program is as small as possible.

This problem is the subject of the present paper. After introducing the problem in more detail, we will discuss the solutions used by other compilers, present the algorithm we use in the CerCo assembler, and discuss its verification, that is the proofs of termination and correctness using the Matita proof assistant [1].

The research presented in this paper has been executed within the CerCo project which aims at formally verifying a C compiler with cost annotations. The target architecture for this project is the MCS-51, whose instruction set contains span-dependent instructions. Furthermore, its maximum addressable memory size is very small (64 Kb), which makes it important to generate programs that are as small as possible.

* Research supported by the CerCo project, within the Future and Emerging Technologies (FET) programme of the Seventh Framework Programme for Research of the European Commission, under FET-Open grant number 243881

With this optimisation, however, comes increased complexity and hence increased possibility for error. We must make sure that the branch instructions are encoded correctly, otherwise the assembled program will behave unpredictably.

2 The branch displacement optimisation problem

In most modern instruction sets that have them, the only span-dependent instructions are branch instructions. Taking the ubiquitous x86-64 instruction set as an example, we find that it contains eleven different forms of the unconditional branch instruction, all with different ranges, instruction sizes and semantics (only six are valid in 64-bit mode, for example). Some examples are shown in Figure 1.

Instruction	Size (bytes)	Displacement range
Short jump	2	-128 to 127 bytes
Relative near jump	5	-2^{32} to $2^{32} - 1$ bytes
Absolute near jump	6	one segment (64-bit address)
Far jump	8	entire memory

Figure 1. List of x86 branch instructions

The chosen target architecture of the CerCo project is the Intel MCS-51, which features three types of branch instructions (or jump instructions; the two terms are used interchangeably), as shown in Figure 2.

Instruction	Size (bytes)	Execution time (cycles)	Displacement range
SJMP ('short jump')	2	2	-128 to 127 bytes
AJMP ('absolute jump')	2	2	one segment (11-bit address)
LJMP ('long jump')	3	3	entire memory

Figure 2. List of MCS-51 branch instructions

Conditional branch instructions are only available in short form, which means that a conditional branch outside the short address range has to be encoded using three branch instructions (for instructions whose logical negation is available, it can be done with two branch instructions, but for some instructions this is not available); the call instruction is only available in absolute and long forms.

Note that even though the MCS-51 architecture is much less advanced and simpler than the x86-64 architecture, the basic types of branch instruction remain the same: a short jump with a limited range, an intra-segment jump and a jump that can reach the entire available memory.

Generally, in code fed to the assembler as input, the only difference between branch instructions is semantics, not span. This means that a distinction is made between an unconditional branch and the several kinds of conditional branch, but not between their short, absolute or long variants.

The algorithm used by the assembler to encode these branch instructions into the different machine instructions is known as the *branch displacement algorithm*. The optimisation problem consists of finding as small an encoding as possible, thus minimising program length and execution time.

This problem is known to be NP-complete [8,11], which could make finding an optimal solution very time-consuming.

The canonical solution, as shown by Szymanski [11] or more recently by Dickson [2] for the x86 instruction set, is to use a fixed point algorithm that starts with the shortest possible encoding (all branch instruction encoded as short jumps, which is likely not a correct solution) and then iterates over the program to re-encode those branch instructions whose target is outside their range.

Adding absolute jumps

In both papers mentioned above, the encoding of a jump is only dependent on the distance between the jump and its target: below a certain value a short jump can be used; above this value the jump must be encoded as a long jump.

Here, termination of the smallest fixed point algorithm is easy to prove. All branch instructions start out encoded as short jumps, which means that the distance between any branch instruction and its target is as short as possible. If, in this situation, there is a branch instruction b whose span is not within the range for a short jump, we can be sure that we can never reach a situation where the span of j is so small that it can be encoded as a short jump. This argument continues to hold throughout the subsequent iterations of the algorithm: short jumps can change into long jumps, but not *vice versa*, as spans only increase. Hence, the algorithm either terminates early when a fixed point is reached or when all short jumps have been changed into long jumps.

Also, we can be certain that we have reached an optimal solution: a short jump is only changed into a long jump if it is absolutely necessary.

However, neither of these claims (termination nor optimality) hold when we add the absolute jump, as with absolute jumps, the encoding of a branch instruction no longer depends only on the distance between the branch instruction and its target: in order for an absolute jump to be possible, they need to be in the same segment (for the MCS-51, this means that the first 5 bytes of their addresses have to be equal). It is therefore entirely possible for two branch instructions with the same span to be encoded in different ways (absolute if the branch instruction and its target are in the same segment, long if this is not the case).

This invalidates our earlier termination argument: a branch instruction, once encoded as a long jump, can be re-encoded during a later iteration as an absolute jump. Consider the program shown in Figure 3. At the start of the first iteration,

```
    jmp X
    ⋮
L0:
    ⋮
    jmp L0
```

Figure 3. Example of a program where a long jump becomes absolute

both the branch to `X` and the branch to $\texttt{L}_0$ are encoded as small jumps. Let us assume that in this case, the placement of $\texttt{L}_0$ and the branch to it are such that $\texttt{L}_0$ is just outside the segment that contains this branch. Let us also assume that the distance between $\texttt{L}_0$ and the branch to it are too large for the branch instruction to be encoded as a short jump.

All this means that in the second iteration, the branch to $\texttt{L}_0$ will be encoded as a long jump. If we assume that the branch to `X` is encoded as a long jump as well, the size of the branch instruction will increase and $\texttt{L}_0$ will be 'propelled' into the same segment as its branch instruction, because every subsequent instruction will move one byte forward. Hence, in the third iteration, the branch to $\texttt{L}_0$ can be encoded as an absolute jump. At first glance, there is nothing that prevents us from constructing a configuration where two branch instructions interact in such a way as to iterate indefinitely between long and absolute encodings.

This situation mirrors the explanation by Szymanski [11] of why the branch displacement optimisation problem is NP-complete. In this explanation, a condition for NP-completeness is the fact that programs be allowed to contain *pathological* jumps. These are branch instructions that can normally not be encoded as a short(er) jump, but gain this property when some other branch instructions are encoded as a long(er) jump. This is exactly what happens in Figure 3. By encoding the first branch instruction as a long jump, another branch instruction switches from long to absolute (which is shorter).

In addition, our previous optimality argument no longer holds. Consider the program shown in Figure 4. Suppose that the distance between $\texttt{L}_0$ and $\texttt{L}_1$ is such that if `jmp X` is encoded as a short jump, there is a segment border just after $\texttt{L}_1$. Let us also assume that the three branches to $\texttt{L}_1$ are all in the same segment, but far enough away from $\texttt{L}_1$ that they cannot be encoded as short jumps.

Then, if `jmp X` were to be encoded as a short jump, which is clearly possible, all of the branches to $\texttt{L}_1$ would have to be encoded as long jumps. However, if `jmp X` were to be encoded as a long jump, and therefore increase in size, $\texttt{L}_1$ would be 'propelled' across the segment border, so that the three branches to $\texttt{L}_1$ could be encoded as absolute jumps. Depending on the relative sizes of long and absolute jumps, this solution might actually be smaller than the one reached by the smallest fixed point algorithm.

```
L0: jmp X
X:
    ⋮
L1:
    ⋮
    jmp L1
    ⋮
    jmp L1
    ⋮
    jmp L1
    ⋮
```

Figure 4. Example of a program where the fixed-point algorithm is not optimal

3 Our algorithm

3.1 Design decisions

Given the NP-completeness of the problem, to arrive at an optimal solution (using, for example, a constraint solver) will potentially take a great amount of time.

The SDCC compiler [9], which has a backend targetting the MCS-51 instruction set, simply encodes every branch instruction as a long jump without taking the distance into account. While certainly correct (the long jump can reach any destination in memory) and a very fast solution to compute, it results in a less than optimal solution.

On the other hand, the `gcc` compiler suite [3], while compiling C on the x86 architecture, uses a greatest fix point algorithm. In other words, it starts with all branch instructions encoded as the largest jumps available, and then tries to reduce the size of branch instructions as much as possible.

Such an algorithm has the advantage that any intermediate result it returns is correct: the solution where every branch instruction is encoded as a large jump is always possible, and the algorithm only reduces those branch instructions whose destination address is in range for a shorter jump. The algorithm can thus be stopped after a determined number of steps without sacrificing correctness.

The result, however, is not necessarily optimal. Even if the algorithm is run until it terminates naturally, the fixed point reached is the *greatest* fixed point, not the least fixed point. Furthermore, `gcc` (at least for the x86 architecture) only uses short and long jumps. This makes the algorithm more efficient, as shown in the previous section, but also results in a less optimal solution.

In the CerCo assembler, we opted at first for a least fixed point algorithm, taking absolute jumps into account.

Here, we ran into a problem with proving termination, as explained in the previous section: if we only take short and long jumps into account, the jump encoding can only switch from short to long, but never in the other direction. When we add absolute jumps, however, it is theoretically possible for a branch instruction to switch from absolute to long and back, as previously explained.

Proving termination then becomes difficult, because there is nothing that precludes a branch instruction from oscillating back and forth between absolute and long jumps indefinitely.

In order to keep the algorithm in the same complexity class and more easily prove termination, we decided to explicitly enforce the 'branch instructions must always grow longer' requirement: if a branch instruction is encoded as a long jump in one iteration, it will also be encoded as a long jump in all the following iterations. This means that the encoding of any branch instruction can change at most two times: once from short to absolute (or long), and once from absolute to long.

There is one complicating factor. Suppose that a branch instruction is encoded in step n as an absolute jump, but in step $n+1$ it is determined that (because of changes elsewhere) it can now be encoded as a short jump. Due to the requirement that the branch instructions must always grow longer, this means that the branch encoding will be encoded as an absolute jump in step $n+1$ as well.

This is not necessarily correct. A branch instruction that can be encoded as a short jump cannot always also be encoded as an absolute jump, as a short jump can bridge segments, whereas an absolute jump cannot. Therefore, in this situation we have decided to encode the branch instruction as a long jump, which is always correct.

The resulting algorithm, while not optimal, is at least as good as the ones from SDCC and `gcc`, and potentially better. Its complexity remains the same (there are at most $2n$ iterations, where n is the number of branch instructions in the program).

3.2 The algorithm in detail

The branch displacement algorithm forms part of the translation from pseudo-code to assembler. More specifically, it is used by the function that translates pseudo-addresses (natural numbers indicating the position of the instruction in the program) to actual addresses in memory.

Our original intention was to have two different functions, one function $\texttt{policy} : \mathbb{N} \to \{\texttt{short\_jump}, \texttt{absolute\_jump}, \texttt{long\_jump}\}$ to associate jumps to their intended encoding, and a function $\sigma : \mathbb{N} \to \texttt{Word}$ to associate pseudo-addresses to machine addresses. σ would use `policy` to determine the size of jump instructions.

This turned out to be suboptimal from the algorithmic point of view and impossible to prove correct.

From the algorithmic point of view, in order to create the `policy` function, we must necessarily have a translation from pseudo-addresses to machine ad-

dresses (i.e. a σ function): in order to judge the distance between a jump and its destination, we must know their memory locations. Conversely, in order to create the σ function, we need to have the `policy` function, otherwise we do not know the sizes of the jump instructions in the program.

Much the same problem appears when we try to prove the algorithm correct: the correctness of `policy` depends on the correctness of σ, and the correctness of σ depends on the correctness of `policy`.

We solved this problem by integrating the `policy` and σ algorithms. We now have a function $\sigma : \mathbb{N} \to \texttt{Word} \times \texttt{bool}$ which associates a pseudo-address to a machine address. The boolean denotes a forced long jump; as noted in the previous section, if during the fixed point computation an absolute jump changes to be potentially re-encoded as a short jump, the result is actually a long jump. It might therefore be the case that jumps are encoded as long jumps without this actually being necessary, and this information needs to be passed to the code generating function.

The assembler function encodes the jumps by checking the distance between source and destination according to σ, so it could select an absolute jump in a situation where there should be a long jump. The boolean is there to prevent this from happening by indicating the locations where a long jump should be encoded, even if a shorter jump is possible. This has no effect on correctness, since a long jump is applicable in any situation.

function F($labels$,old_sigma,$instr$,ppc,acc)
  $\langle added, pc, sigma\rangle \leftarrow acc$
  if $instr$ is a backward jump to j **then**
    $length \leftarrow \text{jump\_size}(pc, sigma_1(labels(j)))$
  else if $instr$ is a forward jump to j **then**
    $length \leftarrow \text{jump\_size}(pc, old\_sigma_1(labels(j)) + added)$
  else
    $length \leftarrow$ `short_jump`
  end if
  $old\_length \leftarrow \text{old\_sigma}_1(ppc)$
  $new\_length \leftarrow \max(old\_length, length)$
  $old\_size \leftarrow \text{old\_sigma}_2(ppc)$
  $new\_size \leftarrow \text{instruction\_size}(instr, new\_length)$
  $new\_added \leftarrow added + (new\_size - old\_size)$
  $new\_sigma_1(ppc + 1) \leftarrow pc + new\_size$
  $new\_sigma_2(ppc) \leftarrow new\_length$
return $\langle new\_added, pc + new\_size, new\_sigma\rangle$
end function

Figure 5. The heart of the algorithm

The algorithm, shown in Figure 5, works by folding the function F over the entire program, thus gradually constructing *sigma*. This constitutes one step in

the fixed point calculation; successive steps repeat the fold until a fixed point is reached.

Parameters of the function F are:

- a function *labels* that associates a label to its pseudo-address;
- *old_sigma*, the σ function returned by the previous iteration of the fixed point calculation;
- *instr*, the instruction currently under consideration;
- *ppc*, the pseudo-address of *instr*;
- *acc*, the fold accumulator, which contains *pc* (the highest memory address reached so far), *added* (the number of bytes added to the program size with respect to the previous iteration), and of course *sigma*, the σ function under construction.

The first two are parameters that remain the same through one iteration, the final three are standard parameters for a fold function (including *ppc*, which is simply the number of instructions of the program already processed).

The σ functions used by F are not of the same type as the final σ function: they are of type $\sigma : \mathbb{N} \to \mathbb{N} \times \{\texttt{short\_jump}, \texttt{absolute\_jump}, \texttt{long\_jump}\}$; a function that associates a pseudo-address with a memory address and a jump length. We do this to be able to ease the comparison of jump lengths between iterations. In the algorithm, we use the notation $sigma_1(x)$ to denote the memory address corresponding to x, and $sigma_2(x)$ to denote the jump length corresponding to x.

Note that the σ function used for label lookup varies depending on whether the label is behind our current position or ahead of it. For backward branches, where the label is behind our current position, we can use *sigma* for lookup, since its memory address is already known. However, for forward branches, the memory address of the address of the label is not yet known, so we must use *old_sigma*.

We cannot use *old_sigma* without change: it might be the case that we have already increased the size of some branch instructions before, making the program longer and moving every instruction forward. We must compensate for this by adding the size increase of the program to the label's memory address according to *old_sigma*, so that branch instruction spans do not get compromised.

Note also that we add the pc to *sigma* at location $ppc + 1$, whereas we add the jump length at location *ppc*. We do this so that $sigma(ppc)$ will always return a pair with the start address of the instruction at *ppc* and the length of its branch instruction (if any); the end address of the program can be found at $sigma(n + 1)$, where n is the number of instructions in the program.

4 The proof

In this section, we present the correctness proof for the algorithm in more detail. The main correctness statement is as follows (slightly simplified, here):

```
definition sigma_policy_specification :=
 λprogram: pseudo_assembly_program.
 λsigma: Word → Word.
 λpolicy: Word → bool.
  sigma (zero ...) = zero ... ∧
  ∀ppc: Word.∀ppc_ok.
  let ⟨preamble, instr_list⟩ := program in
  let pc := sigma ppc in
  let instruction :=
   \fst (fetch_pseudo_instruction instr_list ppc ppc_ok) in
  let next_pc := \fst (sigma (add ? ppc (bitvector_of_nat ? 1))) in
   (nat_of_bitvector ... ppc ≤ |instr_list| →
    next_pc = add ? pc (bitvector_of_nat ...
    (instruction_size ... sigma policy ppc instruction)))
   ∧
   ((nat_of_bitvector ... ppc < |instr_list| →
    nat_of_bitvector ... pc < nat_of_bitvector ... next_pc)
   ∨ (nat_of_bitvector ... ppc = |instr_list| → next_pc = (zero ...))).
```

Informally, this means that when fetching a pseudo-instruction at *ppc*, the translation by σ of $ppc + 1$ is the same as $\sigma(ppc)$ plus the size of the instruction at *ppc*. That is, an instruction is placed consecutively after the previous one, and there are no overlaps.

Instructions are also stocked in order: the memory address of the instruction at *ppc* should be smaller than the memory address of the instruction at $ppc + 1$. There is one exeception to this rule: the instruction at the very end of the program, whose successor address can be zero (this is the case where the program size is exactly equal to the amount of memory).

Finally, we enforce that the program starts at address 0, i.e. $\sigma(0) = 0$.

Since our computation is a least fixed point computation, we must prove termination in order to prove correctness: if the algorithm is halted after a number of steps without reaching a fixed point, the solution is not guaranteed to be correct. More specifically, branch instructions might be encoded which do not coincide with the span between their location and their destination.

Proof of termination rests on the fact that the encoding of branch instructions can only grow larger, which means that we must reach a fixed point after at most $2n$ iterations, with n the number of branch instructions in the program. This worst case is reached if at every iteration, we change the encoding of exactly one branch instruction; since the encoding of any branch instructions can change first from short to absolute and then from absolute to long, there can be at most $2n$ changes.

The proof has been carried out using the "Russell" style from [10]. We have proven some invariants of the F function from the previous section; these invariants are then used to prove properties that hold for every iteration of the fixed point computation; and finally, we can prove some properties of the fixed point.

4.1 Fold invariants

These are the invariants that hold during the fold of F over the program, and that will later on be used to prove the properties of the iteration.

Note that during the fixed point computation, the σ function is implemented as a trie for ease of access; computing $\sigma(x)$ is achieved by looking up the value of x in the trie. Actually, during the fold, the value we pass along is a pair $\mathbb{N} \times \mathtt{ppc_pc_map}$. The first component is the number of bytes added to the program so far with respect to the previous iteration, and the second component, ppc_pc_map, is a pair consisting of the current size of the program and our σ function.

```
definition out_of_program_none :=
 λprefix:list labelled_instruction.λsigma:ppc_pc_map.
 ∀i.i < 2^16 → (i > |prefix| ↔
  bvt_lookup_opt ... (bitvector_of_nat ? i) (\snd sigma) = None ?).
```

This invariant states that any pseudo-address not yet examined is not present in the lookup trie.

```
definition not_jump_default :=
 λprefix:list labelled_instruction.λsigma:ppc_pc_map.
 ∀i.i < |prefix| →
  ¬ is_jump (\snd (nth i ? prefix ⟨None ?, Comment []⟩)) →
  \snd (bvt_lookup ... (bitvector_of_nat ? i) (\snd sigma)
   ⟨0,short_jump⟩) = short_jump.
```

This invariant states that when we try to look up the jump length of a pseudo-address where there is no branch instruction, we will get the default value, a short jump.

```
definition jump_increase :=
 λprefix:list labelled_instruction.λop:ppc_pc_map.λp:ppc_pc_map.
 ∀i.i ≤ |prefix| →
 let ⟨opc,oj⟩ :=
  bvt_lookup ... (bitvector_of_nat ? i) (\snd op) ⟨0,short_jump⟩ in
 let ⟨pc,j⟩ :=
  bvt_lookup ... (bitvector_of_nat ? i) (\snd p) ⟨0,short_jump⟩ in
  jmpleq oj j.
```

This invariant states that between iterations (with *op* being the previous iteration, and p the current one), jump lengths either remain equal or increase. It is needed for proving termination.

```
definition sigma_compact_unsafe :=
 λprogram:list labelled_instruction.λlabels:label_map.λsigma:ppc_pc_map.
 ∀n.n < |program| →
  match bvt_lookup_opt … (bitvector_of_nat ? n) (\snd sigma) with
  [ None ⇒ False
  | Some x ⇒ let ⟨pc,j⟩ := x in
    match bvt_lookup_opt … (bitvector_of_nat ? (S n)) (\snd sigma) with
    [ None ⇒ False
    | Some x1 ⇒ let ⟨pc1,j1⟩ := x1 in
       pc1 = pc + instruction_size_jmplen j
        (\snd (nth n ? program ⟨None ?, Comment []⟩)))
    ]
  ].
```

This is a temporary formulation of the main property (`sigma_policy_specification`); its main difference from the final version is that it uses `instruction_size_jmplen` to compute the instruction size. This function uses j to compute the span of branch instructions (i.e. it uses the σ function under construction), instead of looking at the distance between source and destination. This is because σ is still under construction; later on we will prove that after the final iteration, `sigma_compact_unsafe` is equivalent to the main property.

```
definition sigma_safe :=
 λprefix:list labelled_instruction.λlabels:label_map.λadded:ℕ.
 λold_sigma:ppc_pc_map.λsigma:ppc_pc_map.
 ∀i.i < |prefix| → let ⟨pc,j⟩ :=
  bvt_lookup … (bitvector_of_nat ? i) (\snd sigma) ⟨0,short_jump⟩ in
  let pc_plus_jmp_length := bitvector_of_nat ? (\fst (bvt_lookup …
   (bitvector_of_nat ? (S i)) (\snd sigma) ⟨0,short_jump⟩)) in
  let ⟨label,instr⟩ := nth i ? prefix ⟨None ?, Comment [ ]⟩ in
   ∀dest.is_jump_to instr dest →
    let paddr := lookup_def … labels dest 0 in
    let addr := bitvector_of_nat ? (if leb i paddr (* forward jump *)
    then \fst (bvt_lookup … (bitvector_of_nat ? paddr) (\snd old_sigma)
     ⟨0,short_jump⟩) + added
    else \fst (bvt_lookup … (bitvector_of_nat ? paddr) (\snd sigma)
     ⟨0,short_jump⟩)) in
    match j with
    [ short_jump ⇒ ¬is_call instr ∧
       \fst (short_jump_cond pc_plus_jmp_length addr) = true
    | absolute_jump ⇒ ¬is_relative_jump instr ∧
       \fst (absolute_jump_cond pc_plus_jmp_length addr) = true ∧
       \fst (short_jump_cond pc_plus_jmp_length addr) = false
    | long_jump ⇒ \fst (short_jump_cond pc_plus_jmp_length addr) = false
       ∧ \fst (absolute_jump_cond pc_plus_jmp_length addr) = false
    ].
```

This is a more direct safety property: it states that branch instructions are encoded properly, and that no wrong branch instructions are chosen.

Note that we compute the distance using the memory address of the instruction plus its size: this follows the behaviour of the MCS-51 microprocessor, which increases the program counter directly after fetching, and only then executes the branch instruction (by changing the program counter again).

```
\fst (bvt_lookup ... (bitvector_of_nat ? 0) (\snd policy)
 ⟨0,short_jump⟩) = 0)
\fst policy = \fst (bvt_lookup ...
 (bitvector_of_nat ? (|prefix|)) (\snd policy) ⟨0,short_jump⟩)
```

These two properties give the values of σ for the start and end of the program; $\sigma(0) = 0$ and $\sigma(n)$, where n is the number of instructions up until now, is equal to the maximum memory address so far.

```
(added = 0 → policy_pc_equal prefix old_sigma policy))
(policy_jump_equal prefix old_sigma policy → added = 0))
```

And finally, two properties that deal with what happens when the previous iteration does not change with respect to the current one. *added* is a variable that keeps track of the number of bytes we have added to the program size by changing the encoding of branch instructions. If *added* is 0, the program has not changed and vice versa.

We need to use two different formulations, because the fact that *added* is 0 does not guarantee that no branch instructions have changed. For instance, it is possible that we have replaced a short jump with an absolute jump, which does not change the size of the branch instruction.

Therefore `policy_pc_equal` states that $old\_sigma_1(x) = sigma_1(x)$, whereas `policy_jump_equal` states that $old\_sigma_2(x) = sigma_2(x)$. This formulation is sufficient to prove termination and compactness.

Proving these invariants is simple, usually by induction on the prefix length.

4.2 Iteration invariants

These are invariants that hold after the completion of an iteration. The main difference between these invariants and the fold invariants is that after the completion of the fold, we check whether the program size does not supersede 64 Kb, the maximum memory size the MCS-51 can address.

The type of an iteration therefore becomes an option type: `None` in case the program becomes larger than 64 Kb, or `Some` σ otherwise. We also no longer use a natural number to pass along the number of bytes added to the program size, but a boolean that indicates whether we have changed something during the iteration or not.

If an iteration returns `None`, we have the following invariant:

```
definition nec_plus_ultra :=
 λprogram:list labelled_instruction.λp:ppc_pc_map.
 ¬(∀i.i < |program| →
  is_jump (\snd (nth i ? program ⟨None ?, Comment []⟩))) →
  \snd (bvt_lookup … (bitvector_of_nat 16 i) (\snd p) ⟨0,short_jump⟩) =
   long_jump).
```

This invariant is applied to *old_sigma*; if our program becomes too large for memory, the previous iteration cannot have every branch instruction encoded as a long jump. This is needed later in the proof of termination.

If the iteration returns `Some` σ, the invariants `out_of_program_none`, `not_jump_default`, `jump_increase`, and the two invariants that deal with $\sigma(0)$ and $\sigma(n)$ are retained without change.

Instead of using `sigma_compact_unsafe`, we can now use the proper invariant:

```
definition sigma_compact :=
 λprogram:list labelled_instruction.λlabels:label_map.λsigma:ppc_pc_map.
 ∀n.n < |program| →
  match bvt_lookup_opt … (bitvector_of_nat ? n) (\snd sigma) with
  [ None ⇒ False
  | Some x ⇒ let ⟨pc,j⟩ := x in
    match bvt_lookup_opt … (bitvector_of_nat ? (S n)) (\snd sigma) with
    [ None ⇒ False
    | Some x1 ⇒ let ⟨pc1,j1⟩ := x1 in
      pc1 = pc + instruction_size
       (λid.bitvector_of_nat ? (lookup_def ?? labels id 0))
       (λppc.bitvector_of_nat ?
        (\fst (bvt_lookup … ppc (\snd sigma) ⟨0,short_jump⟩))))
       (λppc.jmpeqb long_jump (\snd (bvt_lookup … ppc
        (\snd sigma) ⟨0,short_jump⟩))) (bitvector_of_nat ? n)
       (\snd (nth n ? program ⟨None ?, Comment []⟩)))
    ]
  ].
```

This is almost the same invariant as `sigma_compact_unsafe`, but differs in that it computes the sizes of branch instructions by looking at the distance between position and destination using σ.

In actual use, the invariant is qualified: σ is compact if there have been no changes (i.e. the boolean passed along is `true`). This is to reflect the fact that we are doing a least fixed point computation: the result is only correct when we have reached the fixed point.

There is another, trivial, invariant if the iteration returns `Some` σ:

```
\fst p < 2^16
```

The invariants that are taken directly from the fold invariants are trivial to prove.

The proof of `nec_plus_ultra` works as follows: if we return `None`, then the program size must be greater than 64 Kb. However, since the previous iteration

did not return `None` (because otherwise we would terminate immediately), the program size in the previous iteration must have been smaller than 64 Kb.

Suppose that all the branch instructions in the previous iteration are encoded as long jumps. This means that all branch instructions in this iteration are long jumps as well, and therefore that both iterations are equal in the encoding of their branch instructions. Per the invariant, this means that $added = 0$, and therefore that all addresses in both iterations are equal. But if all addresses are equal, the program sizes must be equal too, which means that the program size in the current iteration must be smaller than 64 Kb. This contradicts the earlier hypothesis, hence not all branch instructions in the previous iteration are encoded as long jumps.

The proof of `sigma_compact` follows from `sigma_compact_unsafe` and the fact that we have reached a fixed point, i.e. the previous iteration and the current iteration are the same. This means that the results of `instruction_size_jmplen` and `instruction_size` are the same.

4.3 Final properties

These are the invariants that hold after $2n$ iterations, where n is the program size (we use the program size for convenience; we could also use the number of branch instructions, but this is more complex). Here, we only need `out_of_program_none`, `sigma_compact` and the fact that $\sigma(0) = 0$.

Termination can now be proved using the fact that there is a $k \leq 2n$, with n the length of the program, such that iteration k is equal to iteration $k+1$. There are two possibilities: either there is a $k < 2n$ such that this property holds, or every iteration up to $2n$ is different. In the latter case, since the only changes between the iterations can be from shorter jumps to longer jumps, in iteration $2n$ every branch instruction must be encoded as a long jump. In this case, iteration $2n$ is equal to iteration $2n + 1$ and the fixpoint is reached.

5 Conclusion

In the previous sections we have discussed the branch displacement optimisation problem, presented an optimised solution, and discussed the proof of termination and correctness for this algorithm, as formalised in Matita.

The algorithm we have presented is fast and correct, but not optimal; a true optimal solution would need techniques like constraint solvers. While outside the scope of the present research, it would be interesting to see if enough heuristics could be found to make such a solution practical for implementing in an existing compiler; this would be especially useful for embedded systems, where it is important to have as small solution as possible.

In itself the algorithm is already useful, as it results in a smaller solution than the simple 'every branch instruction is long' used up until now—and with only 64 Kb of memory, every byte counts. It also results in a smaller solution than

the greatest fixed point algorithm that `gcc` uses. It does this without sacrificing speed or correctness.

This algorithm is part of a greater whole, the CerCo project, which aims to completely formalise and verify a concrete cost preserving compiler for a large subset of the C programming language. More information on the formalisation of the assembler, of which the present work is a part, can be found in a companion publication [7].

5.1 Related work

As far as we are aware, this is the first formal discussion of the branch displacement optimisation algorithm.

The CompCert project is another verified compiler project. Their backend [5] generates assembly code for (amongst others) subsets of the PowerPC and x86 (32-bit) architectures. At the assembly code stage, there is no distinction between the span-dependent jump instructions, so a branch displacement optimisation algorithm is not needed.

An offshoot of the CompCert project is the CompCertTSO project, who add thread concurrency and synchronisation to the CompCert compiler [12]. This compiler also generates assembly code and therefore does not include a branch displacement algorithm.

Finally, there is also the Piton stack [6], which not only includes the formal verification of a compiler, but also of the machine architecture targeted by that compiler, a bespoke microprocessor called the FM9001. However, this architecture does not have different jump sizes (branching is simulated by assigning values to the program counter), so the branch displacement problem is irrelevant.

5.2 Formal development

All Matita files related to this development can be found on the CerCo website, `http://cerco.cs.unibo.it`. The specific part that contains the branch displacement algorithm is in the `ASM` subdirectory, in the files `PolicyFront.ma`, `PolicyStep.ma` and `Policy.ma`.

References

1. Asperti, A., Sacerdoti Coen, C., Tassi, E., Zacchiroli, S.: User interaction with the Matita proof assistant. Automated Reasoning 39, 109–139 (2007)
2. Dickson, N.G.: A simple, linear-time algorithm for x86 jump encoding. CoRR abs/0812.4973 (2008)
3. Gnu compiler collection 4.7.0. http://gcc.gnu.org/ (2012)
4. Hyde, R.: Branch displacement optimisation. http://groups.google.com/group/alt.lang.asm/msg/d31192d442accad3 (2006)
5. Leroy, X.: A formally verified compiler back-end. Journal of Automated Reasoning 43, 363–446 (2009), http://dx.doi.org/10.1007/s10817-009-9155-4, 10.1007/s10817-009-9155-4
6. Moore, J.S.: Piton: A mechanically verified assembly language, Automated Reasoning Series, vol. 3. Springer (1996)
7. Mulligan, D.P., Sacerdoti Coen, C.: On the correctness of an optimising assembler for the Intel MCS-51 microprocessor (2012), submitted
8. Robertson, E.L.: Code generation and storage allocation for machines with span-dependent instructions. ACM Trans. Program. Lang. Syst. 1(1), 71–83 (Jan 1979), http://doi.acm.org/10.1145/357062.357067
9. Small device C compiler 3.1.0. http://sdcc.sourceforge.net/ (2011)
10. Sozeau, M.: Subset coercions in Coq. In: TYPES. pp. 237–252 (2006)
11. Szymanski, T.G.: Assembling code for machines with span-dependent instructions. Commun. ACM 21(4), 300–308 (Apr 1978), http://doi.acm.org/10.1145/359460.359474
12. Ševčik, J., Vafeiadis, V., Zappa Nardelli, F., Jagannathan, S., Sewell, P.: Relaxed-memory concurrency and verified compilation. SIGPLAN Not. 46(1), 43–54 (Jan 2011), http://doi.acm.org/10.1145/1925844.1926393