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 [**?**]. This formalisation forms a major component of the EU-funded CerCo ('Certified Complexity') project [**?**], 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 [**?**]).

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 pre-requisite 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 [**?**]). 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 [**?**]. 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 [**?**], 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 [**?**]. 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.

$$\left(\begin{tikzcd} P_i \arrow[r, "\texttt{expand\_pseudo\_instruction}"] \arrow[rr, bend right, "\texttt{assembly\_1\_pseudo\_instruction}"'] & {[I_i^1,\ldots I_i^q]} \arrow[r, "\texttt{assembly1}^*"] & {[0110]} \end{tikzcd}\right)^*$$

$$\begin{tikzcd} {[P_1,\ldots P_n]} \arrow[r, "\texttt{assembly}"'] & {[\ldots 0110\ldots]} \end{tikzcd}$$

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 [?]. 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 $[\mathit{ppc}, \mathit{newppc}]$ corresponds to fetching a sequence of instructions from the real code memory, stored in the interval $[\mathit{policy}(\mathit{ppc}), \mathit{policy}(\mathit{ppc}+1)]$. 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.

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

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

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* [**?**] 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.

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

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

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 ⟨ `Upper, l`⟩, 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 ⟨`Upper,l`⟩ and `l` is tracked as ⟨`Lower,h`⟩. 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 [**?**], 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 [?].

Verified assemblers could also be applied to the verification of operating system kernels and other formalised compilers. For instance the verified seL4 kernel [?], CompCert [?] and CompCertTSO [?] 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 [?]. 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 [?], 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 [?]. Klein and Nipkow also provide a compiler, virtual machine and operational semantics for the Jinja [?] 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.