A Reuse-Based Multi-Stage Compiler Verification for Language IMP

Pasquale Noce
Senior Staff Engineer at HID Global, Italy
pasquale dot noce dot lavoro at gmail dot com
pasquale dot noce at hidglobal dot com

March 17, 2025

Abstract

After introducing the didactic imperative programming language IMP, Nipkow and Klein's book on formal programming language semantics (version of March 2021) specifies compilation of IMP commands into a lower-level language based on a stack machine, and expounds a formal verification of that compiler. Exercise 8.4 asks the reader to adjust such proof for a new compilation target, consisting of a machine language that (i) accesses memory locations through their addresses instead of variable names, and (ii) maintains a stack in memory via a stack pointer rather than relying upon a built-in stack. A natural strategy to maximize reuse of the original proof is keeping the original language as an assembly one and splitting compilation into multiple steps, namely a source-to-assembly step matching the original compilation process followed by an assembly-to-machine step. In this way, proving assembly code-machine code equivalence is the only extant task.

A previous paper by the present author introduces a reasoning toolbox that allows for a compiler correctness proof shorter than the book's one, as such promising to constitute a further enhanced reference for the formal verification of real-world compilers. This paper in turn shows that such toolbox can be reused to accomplish the aforesaid task as well, which demonstrates that the proposed approach also promotes proof reuse in multi-stage compiler verifications.

Contents

1 Compiler formalization

```
theory Compiler
  imports
    HOL−IMP.Big-Step
    HOL−IMP.Star
begin
```

This paper is dedicated to Gaia and Greta, my sweet nieces, who fill my life with love and happiness.

After introducing the didactic imperative programming language IMP, [5] specifies compilation of IMP commands into a lower-level language based on a stack machine, and expounds a formal verification of that compiler. Exercise 8.4 asks the reader to adjust such proof for a new compilation target, consisting of a machine language that (i) accesses memory locations through their addresses instead of variable names, and (ii) maintains a stack in memory via a stack pointer rather than relying upon a built-in stack. A natural strategy to maximize reuse of the original proof is keeping the original language as an assembly one and splitting compilation into multiple steps, namely a source-to-assembly step matching the original compilation process followed by an assembly-to-machine step. In this way, proving assembly code-machine code equivalence is the only extant task.

[7] introduces a reasoning toolbox that allows for a compiler correctness proof shorter than the book's one, as such promising to constitute a further enhanced reference for the formal verification of real-world compilers. This paper in turn shows that such toolbox can be reused to accomplish the aforesaid task as well, which demonstrates that the proposed approach also promotes proof reuse in multi-stage compiler verifications.

The formal proof development presented in this paper consists of two theory files, as follows.

- The former theory, briefly referred to as "the *Compiler* theory", is derived from the *HOL−IMP.Compiler* one included in the Isabelle2021-1 distribution [4].
  However, the signature of function *bcomp* is modified in the same way as in [7].

- The latter theory, briefly referred to as “the *Compiler2* theory”, is derived from the *Compiler2* one developed in [7].
  However, unlike [7], the original language IMP is considered here, without extending it with non-deterministic choice. Hence, the additional case pertaining to non-deterministic choice in the proof of lemma *ccomp-correct* is not present any longer.

Both theory files are split into the same subsections as the respective original theories, and only the most salient differences with respect to the original theories are commented in both of them.

For further information about the formal definitions and proofs contained in this paper, see Isabelle documentation, particularly [6], [3], [1], and [2].

1.1 List setup

declare [[*coercion-enabled*]]
declare [[*coercion int* :: *nat* ⇒ *int*]]
declare [[*syntax-ambiguity-warning* = *false*]]

abbreviation (**output**)
isize xs ≡ *int* (*length xs*)

notation *isize* (‹*size*›)

primrec (*nonexhaustive*) *inth* :: *'a list* ⇒ *int* ⇒ *'a* (**infixl** ‹!!› *100*) **where**
(*x* # *xs*) !! *i* = (*if i* = *0 then x else xs* !! (*i* − *1*))

lemma *inth-append* [*simp*]:
0 ≤ *i* ⟹
(*xs* @ *ys*) !! *i* = (*if i* < *size xs then xs* !! *i else ys* !! (*i* − *size xs*))
⟨*proof*⟩

1.2 Instructions and stack machine

Here below, both the syntax and the semantics of the instruction set are defined. As a deterministic language is considered here, as opposed to the non-deterministic one addressed in [7], instruction semantics can be defined via a simple non-recursive function *iexec* (identical to the one used in [5], since the instruction set is the same). However, an inductive predicate *iexec-pred*, resembling the *iexec* one used in [7] and denoted by the same infix symbol ↦, is also defined. Though notation (*ins*, *cf*) ↦ *cf'* is just an alias for *cf'* = *iexec ins cf*, it is used in place of the latter in the definition of predicate *exec1*, which formalizes single-step program execution. The reason is that the compiler correctness proof developed in the *Compiler2* theory of [7] depends on the introduction and elimination rules deriving from predicate *iexec*'s inductive definition. Thus, the use of predicate *iexec-pred* is a trick enabling Isabelle's classical reasoner to keep using such rules, which restricts the changes to be made to the proofs in the *Compiler2* theory to those required by the change of the compilation target.

The instructions defined by type *instr*, which refer to memory locations via variable names, will keep being used as an assembly language. In order to have a machine language rather referring to memory locations via their addresses, modeled as integers, an additional type *m-instr* of machine instructions, in one-to-one correspondence with assembly instructions, is introduced. The underlying idea is to reuse the proofs that source code and compiled (assembly) code simulate each other built in [4] and [7], so that the only extant task is proving that assembly code and machine code in turn simulate each other. This is nothing but an application of the *divide et impera* strategy of considering multiple compilation stages mentioned in [5], section 8.5.

In other words, the solution developed in what follows does not require any change to the original compiler completeness and correctness proofs. This result is achieved by splitting compilation into multiple steps, namely a source-to-assembly step matching the original compilation process, to which the aforesaid proofs still apply, followed by an assembly-to-machine step. In this way, to establish source code-machine code equivalence, the assembly code-machine code one is all that is left to be proven. In addition to proof reuse, this approach provides the following further advantages.

- There is no need to reason about the composition and decomposition of machine code sequences, which would also involve the composition and decomposition of the respective mappings between used variables and their addresses (as opposed to what happens with assembly code sequences).

- There is no need to change the original compilation functions, modeling the source-to-assembly compilation step in the current context. In fact, the outputs of these functions are assembly programs, namely lists of assembly instructions, which are in one-to-one correspondence with machine ones. Thus, the assembly-to-machine compilation step can easily be modeled as a mapping of such a list into a machine instruction one, where each referenced variable can be assigned an unambiguous address based on the position of the first/last instruction referencing it within the assembly program.

datatype *instr* =
  LOADI int | *LOAD vname* | *ADD* | *STORE vname* |
  JMP int | *JMPLESS int* | *JMPGE int*

```
type-synonym stack = val list
type-synonym config = int × state × stack

abbreviation hd2 xs ≡ hd (tl xs)
abbreviation tl2 xs ≡ tl (tl xs)

fun iexec :: instr ⇒ config ⇒ config where
iexec ins (i, s, stk) = (case ins of
  LOADI n ⇒ (i + 1, s, n # stk) |
  LOAD x ⇒ (i + 1, s, s x # stk) |
  ADD ⇒ (i + 1, s, (hd2 stk + hd stk) # tl2 stk) |
  STORE x ⇒ (i + 1, s(x := hd stk), tl stk) |
  JMP n ⇒ (i + 1 + n, s, stk) |
  JMPLESS n ⇒ (if hd2 stk < hd stk then i + 1 + n else i + 1, s, tl2 stk) |
  JMPGE n ⇒ (if hd2 stk ≥ hd stk then i + 1 + n else i + 1, s, tl2 stk))

inductive iexec-pred :: instr × config ⇒ config ⇒ bool
  (infix ‹↦› 55) where
(ins, cf) ↦ iexec ins cf

definition exec1 :: instr list ⇒ config ⇒ config ⇒ bool
  (‹(-/ ⊢/ -/ →/ -)› 55) where
P ⊢ cf → cf′ ≡ (P !! fst cf, cf) ↦ cf′ ∧ 0 ≤ fst cf ∧ fst cf < size P

abbreviation exec :: instr list ⇒ config ⇒ config ⇒ bool
  (‹(-/ ⊢/ -/ →*/ -)› 55) where
exec P ≡ star (exec1 P)


declare iexec-pred.intros [intro]

inductive-cases LoadIE [elim!]: (LOADI i, pc, s, stk) ↦ cf
inductive-cases LoadE [elim!]: (LOAD x, pc, s, stk) ↦ cf
inductive-cases AddE [elim!]: (ADD, pc, s, stk) ↦ cf
inductive-cases StoreE [elim!]: (STORE x, pc, s, stk) ↦ cf
inductive-cases JmpE [elim!]: (JMP i, pc, s, stk) ↦ cf
inductive-cases JmpLessE [elim!]: (JMPLESS i, pc, s, stk) ↦ cf
inductive-cases JmpGeE [elim!]: (JMPGE i, pc, s, stk) ↦ cf

lemmas exec-induct = star.induct [of exec1 P, split-format(complete)]

lemma iexec-simp:
 (ins, cf) ↦ cf′ = (cf′ = iexec ins cf)
⟨proof⟩

lemma exec1I [intro, code-pred-intro]:
⟦c′ = iexec (P !! i) (i, s, stk); 0 ≤ i; i < size P⟧ ⟹
   P ⊢ (i, s, stk) → c′
⟨proof⟩
```

```
type-synonym addr = int

datatype m-instr =
  M-LOADI int | M-LOAD addr | M-ADD | M-STORE addr |
  M-JMP int | M-JMPLESS int | M-JMPGE int
```

Here below are the recursive definitions of functions *vars*, which takes an assembly program as input and returns a list without repetitions of the referenced variables, and *addr-of*, which in turn takes a list of variables *xs* and a variable *x* as inputs and returns the address *a* of *x*. If *x* is included in *xs*, *a* is set to the one-based right offset of the leftmost occurrence of *x* in *xs*, otherwise *a* is set to zero.

Therefore, for any assembly program *P*, function *addr-of* (*vars P*) maps each variable occurring within *P* to a distinct positive address, and any other, unused variable to a default, invalid address (zero).

```
primrec vars :: instr list ⇒ vname list where
vars [] = [] |
vars (ins # P) = (case ins of
  LOAD x ⇒ if x ∈ set (vars P) then [] else [x] |
  STORE x ⇒ if x ∈ set (vars P) then [] else [x] |
  - ⇒ []) @ vars P

primrec addr-of :: vname list ⇒ vname ⇒ addr where
addr-of [] - = 0 |
addr-of (x # xs) y = (if x = y then size xs + 1 else addr-of xs y)
```

Functions *vars* and *addr-of* can be used to translate an assembly program into a machine program, which is done by the subsequent functions *to-m-instr* and *to-m-prog*. The former takes a list of variables *xs* and an assembly instruction *ins* as inputs and returns the corresponding machine instruction, which refers to address *addr-of xs x* whenever *ins* references variable *x*. Then, the latter function turns each instruction contained in the input assembly program *P* into the corresponding machine one, using function *to-m-instr* (*vars P*) for such mapping. Hence, each variable *x* occurring within *P* is turned into the address *addr-of* (*vars P*) *x*, as expected.

In addition, the types *m-state* and *m-config* of machine states and configurations are also defined here below. The former one encompasses any function mapping addresses to values. The latter one reflects the fact that the third element of a machine configuration has to be a pointer to a stack maintained by the machine state, rather than a list-encoded stack as keeps happening with assembly configurations. This can be achieved using a natural number *sp* as third element, standing for the current size of the machine stack. Hence, if it is nonempty, the address of its topmost element matches $-sp$, given that the machine stack will be modeled by making it start from address -1 and grow downward.

```
fun to-m-instr :: vname list ⇒ instr ⇒ m-instr where
to-m-instr xs ins = (case ins of
  LOADI n ⇒ M-LOADI n |
  LOAD x ⇒ M-LOAD (addr-of xs x) |
  ADD ⇒ M-ADD |
  STORE x ⇒ M-STORE (addr-of xs x) |
  JMP n ⇒ M-JMP n |
  JMPLESS n ⇒ M-JMPLESS n |
  JMPGE n ⇒ M-JMPGE n)

fun to-m-prog :: instr list ⇒ m-instr list where
to-m-prog P = map (to-m-instr (vars P)) P

type-synonym m-state = addr ⇒ val
type-synonym m-config = int × m-state × nat
```

Next are the definitions of functions *to-state* and *to-m-state*, which turn a machine program state *ms* into an equivalent assembly program state *s* and vice versa, based on an input list of variables *xs*. Here, *equivalent* means that for each variable *x* in *xs*, *s* assigns *x* the same value that *ms* assigns to *x*'s address *addr-of xs x*.

Function *to-m-state xs s* maps any positive address *a* up to *size xs* to value *s x*, where *x* is the variable occurring within *xs* at the zero-based left offset *size xs* − *a*, and any other, unused address to a default, dummy value (zero). The resulting machine program state is equivalent to *s* since the zero-based left offset *size xs* − *a* points to the same variable *x* within *xs* as the one-based right offset *a*. As long as *xs* does not contain any repetition, as happens with the outputs of function *vars*, *x* is indeed the variable such that *addr-of xs x* = *a*, by virtue of the definition of function *addr-of*. To perform the reverse conversion, function *to-state xs ms* merely needs to map any variable *x* to *ms* (*addr-of xs x*).

Hence, for any assembly program *P*, function *to-state* (*vars P*) converts each state of the resulting machine program *to-m-prog P* into an equivalent state of *P*, while *to-m-state* (*vars P*) performs conversions the other way around.

```
fun to-state :: vname list ⇒ m-state ⇒ state where
to-state xs ms x = ms (addr-of xs x)

fun to-m-state :: vname list ⇒ state ⇒ m-state where
```

to-m-state xs s a = (*if 0* < *a* ∧ *a* ≤ *size xs then s* (*xs* !! (*size xs* − *a*)) *else 0*)

Likewise, functions *add-stack* and *add-m-stack* are defined to convert machine stacks into assembly ones and vice versa. Function *add-stack* takes a stack pointer and a machine state *ms* as inputs, and returns a list-encoded stack mirroring the machine one maintained by *ms*. Conversely, function *add-m-stack* takes a stack pointer, a list-encoded stack *stk*, and a machine state *ms* as inputs, and returns the machine state obtained by extending *ms* with a machine stack mirroring *stk*.

primrec *add-stack* :: *nat* ⇒ *m-state* ⇒ *stack* **where**
add-stack 0 - = [] |
add-stack (*Suc n*) *ms* = *ms* (−*Suc n*) # *add-stack n ms*

primrec *add-m-stack* :: *nat* ⇒ *stack* ⇒ *m-state* ⇒ *m-state* **where**
add-m-stack 0 - ms = *ms* |
add-m-stack (*Suc n*) *stk ms* = (*add-m-stack n* (*tl stk*) *ms*)(−*Suc n* := *hd stk*)

Here below, the semantics of machine instructions and the execution of machine programs are defined. Such definitions resemble their assembly counterparts, but no inductive predicate like *iexec-pred* is needed here. In fact, *iexec-pred* is employed to enable Isabelle's classical reasoner to use the resulting introduction and elimination rules in the compiler correctness proof contained in the *Compiler2* theory, which in the current context shows that source code simulates assembly code. As all that is required here is to establish the further, missing link between assembly code and machine code, the compiler correctness proof can keep referring to assembly code – indeed, it does not demand any change at all. Consequently, no machine counterpart of inductive predicate *iexec-pred* is needed in the definition of machine instruction semantics.

As usual, any two machine configurations *mcf* and *mcf'* may be linked by a single-step execution of a machine program *MP* only if *mcf*'s program counter points to some instruction *mins* within *MP*. However, *mcf'* is not required to match, but just to be *equivalent* to the machine configuration produced by the execution of *mins* in *mcf*; namely, program counters and stack pointers have to be equal, but machine states just have to match up to the machine stack's top. Moreover, *mcf*'s machine stack has to be large enough to store the operands, if any, required for executing *mins*. As shown in what follows, these conditions are necessary for the lemmas establishing single-step assembly code-machine code equivalence to hold.

primrec *m-msp* :: *m-instr* ⇒ *nat* **where**
m-msp (*M-LOADI n*) = *0* |

```
m-msp (M-LOAD a) = 0 |
m-msp M-ADD = 2 |
m-msp (M-STORE a) = 1 |
m-msp (M-JMP n) = 0 |
m-msp (M-JMPLESS n) = 2 |
m-msp (M-JMPGE n) = 2

definition msp :: instr list ⇒ int ⇒ nat where
msp P i ≡ m-msp (to-m-instr [] (P !! i))


fun m-iexec :: m-instr ⇒ m-config ⇒ m-config where
m-iexec mins (i, ms, sp) = (case mins of
  M-LOADI n ⇒ (i + 1, ms(−1 − sp := n), sp + 1) |
  M-LOAD a ⇒ (i + 1, ms(−1 − sp := ms a), sp + 1) |
  M-ADD ⇒ (i + 1, ms(1 − sp := ms (1 − sp) + ms (−sp)), sp − 1) |
  M-STORE a ⇒ (i + 1, ms(a := ms (−sp)), sp − 1) |
  M-JMP n ⇒ (i + 1 + n, ms, sp) |
  M-JMPLESS n ⇒
    (if ms (1 − sp) < ms (−sp) then i + 1 + n else i + 1, ms, sp − 2) |
  M-JMPGE n ⇒
    (if ms (1 − sp) ≥ ms (−sp) then i + 1 + n else i + 1, ms, sp − 2))

fun m-config-equiv :: m-config ⇒ m-config ⇒ bool (infix ‹≅› 55) where
(i, ms, sp) ≅ (i', ms', sp') =
  (i = i' ∧ sp = sp' ∧ (∀ a ≥ −sp. ms a = ms' a))

definition m-exec1 :: m-instr list ⇒ m-config ⇒ m-config ⇒ bool
  (‹(-/ ⊢/ -/ →/ -)› [59, 0, 59] 60) where
MP ⊢ mcf → mcf' ≡
  mcf' ≅ m-iexec (MP !! fst mcf) mcf ∧ 0 ≤ fst mcf ∧ fst mcf < size MP ∧
    m-msp (MP !! fst mcf) ≤ snd (snd mcf)

abbreviation m-exec :: m-instr list ⇒ m-config ⇒ m-config ⇒ bool
  (‹(-/ ⊢/ -/ →*/ -)› [59, 0, 59] 60) where
m-exec MP ≡ star (m-exec1 MP)
```

Here below is the proof of lemma *exec1-m-exec1*, which states that, under proper assumptions, single-step assembly code executions are simulated by machine code ones. The assumptions are that the initial stack pointer is not less than the number of the operands taken by the instruction to be run, and not greater than the size of the initial assembly stack. Unfortunately, the resulting stack pointer is not guaranteed to keep fulfilling the former assumption for the next instruction; indeed, an arbitrary instruction list is generally not so well-behaved. So, in order to prove that assembly programs are simulated by machine ones, it needs to be proven that any machine program produced by compiling a source one is actually well-behaved in this respect; namely, that a starting machine configuration with stack pointer zero, as well as any intermediate configuration reached thereafter, meet the aforesaid assumptions when executing every such program. This issue will be addressed in the *Compiler2* theory.

At first glance, the need for the assumption causing this issue might appear to result from the lower bound on the initial machine stack size introduced in *m-exec1*'s definition. If that were really the case, the aforesaid issue could be solved by merely dropping this condition (leaving aside its necessity for the twin lemma *m-exec1-exec1* to hold, discussed later on). Nonetheless, a more in-depth investigation shows that the incriminated assumption would be required all the same: were it dropped, a counterexample for lemma *exec1-m-exec1* would arise for P !! $pc = ADD$, $sp = 1$ (addition rather pops *two* operands from the machine stack), and $hd\ stk \neq 0$. In fact, the initial configuration in *exec1-m-exec1*'s conclusion would map addresses 0 and -1 to values 0 and *hd stk*. Hence, the configuration correspondingly output by function *m-iexec M-ADD* would map address 0 to *hd stk*, whereas the final configuration in *exec1-m-exec1*'s conclusion would map it to 0. Being $sp' = 0$, this state of affairs would not satisfy *m-exec1*'s definition, which would rather require the machine states of those configurations to match at every address from 0 upward.

Lemma *exec1-m-exec1* would fail to hold if ≅ were replaced with = within *m-exec1*'s definition. In fact, function *to-m-state* invariably returns machine states mapping any nonpositive address to zero, and function *add-m-stack* leaves unchanged any value below the machine stack's top. Thus, upon any machine instruction *mins* that pops a value $i \neq 0$ from the stack's top address a, the configuration obtained by applying function *m-iexec mins* to the initial configuration in *exec1-m-exec1*'s conclusion maps a to i, whereas the final configuration maps a to 0. As a result, the machine states of those configurations match only up to the machine stack's top, exactly as required using ≅ in *m-exec1*'s definition.

lemma *inth-map* [*simp*]:
$[\![0 \leq i;\ i < size\ xs]\!] \Longrightarrow (map\ f\ xs)\ !!\ i = f\ (xs\ !!\ i)$
⟨*proof*⟩

lemma *inth-set* [*simp*]:
$[\![0 \leq i;\ i < size\ xs]\!] \Longrightarrow xs\ !!\ i \in set\ xs$
⟨*proof*⟩

lemma *vars-dist*:
distinct (*vars P*)
⟨*proof*⟩

lemma *vars-load*:
$[\![0 \leq i;\ i < size\ P;\ P\ !!\ i = LOAD\ x]\!] \Longrightarrow x \in set\ (vars\ P)$

⟨*proof*⟩

lemma *vars-store*:
$\llbracket 0 \le i;\ i < \mathit{size}\ P;\ P\ !!\ i = \mathit{STORE}\ x\rrbracket \Longrightarrow x \in \mathit{set}\ (\mathit{vars}\ P)$
⟨*proof*⟩

lemma *addr-of-max*:
$\mathit{addr\text{-}of}\ xs\ x \le \mathit{size}\ xs$
⟨*proof*⟩

lemma *addr-of-neq*:
$1 + \mathit{size}\ xs \neq \mathit{addr\text{-}of}\ xs\ x$
⟨*proof*⟩

lemma *addr-of-correct*:
$x \in \mathit{set}\ xs \Longrightarrow xs\ !!\ (\mathit{size}\ xs - \mathit{addr\text{-}of}\ xs\ x) = x$
⟨*proof*⟩

lemma *addr-of-nneg*:
$0 \le \mathit{addr\text{-}of}\ xs\ x$
⟨*proof*⟩

lemma *addr-of-set*:
$x \in \mathit{set}\ xs \Longrightarrow 0 < \mathit{addr\text{-}of}\ xs\ x$
⟨*proof*⟩

lemma *addr-of-unique*:
$\llbracket \mathit{distinct}\ xs;\ 0 < a;\ a \le \mathit{size}\ xs\rrbracket \Longrightarrow \mathit{addr\text{-}of}\ xs\ (xs\ !!\ (\mathit{size}\ xs - a)) = a$
⟨*proof*⟩

lemma *add-m-stack-nneg*:
$0 \le a \Longrightarrow \mathit{add\text{-}m\text{-}stack}\ n\ \mathit{stk}\ \mathit{ms}\ a = \mathit{ms}\ a$
⟨*proof*⟩

lemma *add-m-stack-hd*:
$0 < n \Longrightarrow \mathit{add\text{-}m\text{-}stack}\ n\ \mathit{stk}\ \mathit{ms}\ (-n) = \mathit{hd}\ \mathit{stk}$
⟨*proof*⟩

lemma *add-m-stack-hd2*:
$1 < n \Longrightarrow \mathit{add\text{-}m\text{-}stack}\ n\ \mathit{stk}\ \mathit{ms}\ (1 - \mathit{int}\ n) = \mathit{hd2}\ \mathit{stk}$
⟨*proof*⟩

lemma *add-m-stack-nth*:
$\llbracket -n \le a;\ n \le \mathit{length}\ \mathit{stk}\rrbracket \Longrightarrow$
$\quad \mathit{add\text{-}m\text{-}stack}\ n\ \mathit{stk}\ \mathit{ms}\ a = (\mathit{if}\ 0 \le a\ \mathit{then}\ \mathit{ms}\ a\ \mathit{else}\ \mathit{stk}\ !\ (\mathit{nat}\ (n + a)))$
⟨*proof*⟩

lemma *exec1-m-exec1* [*simplified Let-def*]:
$\llbracket P \vdash (pc,\ s,\ \mathit{stk}) \rightarrow (pc',\ s',\ \mathit{stk}');\ \mathit{msp}\ P\ pc \le sp;\ sp \le \mathit{length}\ \mathit{stk}\rrbracket \Longrightarrow$

$\quad \textit{let } sp' = sp + \textit{length } stk' - \textit{length } stk \textit{ in to-m-prog } P \vdash$
$\quad\quad (pc,\ \textit{add-m-stack } sp\ stk\ (\textit{to-m-state } (\textit{vars } P)\ s),\ sp) \rightarrow$
$\quad\quad (pc',\ \textit{add-m-stack } sp'\ stk'\ (\textit{to-m-state } (\textit{vars } P)\ s'),\ sp')$
$\langle \textit{proof} \rangle$

Here below is the proof of lemma *m-exec1-exec1*, which reverses the previous one and states that single-step machine code executions are simulated by assembly code ones. As opposed to lemma *exec1-m-exec1*, the present one does not require any assumption apart from having two arbitrary machine configurations linked by a single-step program execution. Hence, this time there is no obstacle to proving lemma *m-exec-exec*, which generalizes *m-exec1-exec1* to multiple-step program executions, as a direct consequence of *m-exec1-exec1* via induction over the reflexive transitive closure of binary predicate *m-exec1* (*to-m-prog P*), where P is the given, arbitrary assembly program.

If the condition that the initial machine stack be large enough to store the operands of the current instruction were removed from *m-exec1*'s definition, lemma *m-exec1-exec1* would not hold. A counterexample would be the case where $P\ !!\ pc = \textit{ADD}$, $sp = 1$, and $stk = []$. Being $sp' = 0$, the final assembly stack in *m-exec1-exec1*'s conclusion would be empty, whereas according to *exec1*'s definition, the assembly stack resulting from the execution of an addition cannot be empty.

lemma *addr-of-nset*:
$\ x \notin \textit{set } xs \Longrightarrow \textit{addr-of } xs\ x = \textit{0}$
$\langle \textit{proof} \rangle$

lemma *addr-of-inj*:
$\ \textit{inj-on } (\textit{addr-of } xs)\ (\textit{set } xs)$
$\langle \textit{proof} \rangle$

lemma *addr-of-neq2*:
$[\![x \in \textit{set } xs;\ x' \neq x]\!] \Longrightarrow \textit{addr-of } xs\ x' \neq \textit{addr-of } xs\ x$
$\langle \textit{proof} \rangle$

lemma *to-state-eq*:
$\forall a \geq \textit{0}.\ ms'\ a = ms\ a \Longrightarrow \textit{to-state } xs\ ms' = \textit{to-state } xs\ ms$
$\langle \textit{proof} \rangle$

lemma *to-state-upd*:
$[\![\forall a \geq \textit{0}.\ ms'\ a = (\textit{if } a = \textit{addr-of } xs\ x \textit{ then } i \textit{ else } ms\ a);\ x \in \textit{set } xs]\!] \Longrightarrow$
$\quad \textit{to-state } xs\ ms' = (\textit{to-state } xs\ ms)(x := i)$
$\langle \textit{proof} \rangle$

lemma *add-stack-eq*:
$[\![\forall a \in \{-m..<\textit{0}\}.\ ms'\ a = ms\ a;\ m = n]\!] \Longrightarrow \textit{add-stack } m\ ms' = \textit{add-stack } n\ ms$

⟨*proof*⟩

lemma *add-stack-eq2*:
$[\![\forall a \in \{-n..{<}0\}.\ ms'\ a = (\mathit{if}\ a = -n\ \mathit{then}\ i\ \mathit{else}\ ms\ a);\ 0 < n]\!] \Longrightarrow$
$\quad \mathit{add\text{-}stack}\ n\ ms' = i\ \#\ \mathit{add\text{-}stack}\ (n - 1)\ ms$
⟨*proof*⟩

lemma *add-stack-hd*:
$0 < n \Longrightarrow hd\ (\mathit{add\text{-}stack}\ n\ ms) = ms\ (-n)$
⟨*proof*⟩

lemma *add-stack-hd2*:
$1 < n \Longrightarrow hd2\ (\mathit{add\text{-}stack}\ n\ ms) = ms\ (1 - \mathit{int}\ n)$
⟨*proof*⟩

lemma *add-stack-nnil*:
$0 < n \Longrightarrow \mathit{add\text{-}stack}\ n\ ms \neq []$
⟨*proof*⟩

lemma *add-stack-nnil2*:
$1 < n \Longrightarrow tl\ (\mathit{add\text{-}stack}\ n\ ms) \neq []$
⟨*proof*⟩

lemma *add-stack-tl*:
$tl\ (\mathit{add\text{-}stack}\ n\ ms) = \mathit{add\text{-}stack}\ (n - 1)\ ms$
⟨*proof*⟩

lemma *m-exec1-exec1* [*simplified*]:
$\mathit{to\text{-}m\text{-}prog}\ P \vdash (pc,\ ms,\ sp) \longrightarrow (pc',\ ms',\ sp') \Longrightarrow$
$\quad P \vdash (pc,\ \mathit{to\text{-}state}\ (\mathit{vars}\ P)\ ms,\ \mathit{add\text{-}stack}\ sp\ ms\ @\ stk) \rightarrow$
$\qquad (pc',\ \mathit{to\text{-}state}\ (\mathit{vars}\ P)\ ms',\ \mathit{add\text{-}stack}\ sp'\ ms'\ @\ stk)$
⟨*proof*⟩

lemma *m-exec-exec*:
$\mathit{to\text{-}m\text{-}prog}\ P \vdash (pc,\ ms,\ sp) \longrightarrow\!*\ (pc',\ ms',\ sp') \Longrightarrow$
$\quad P \vdash (pc,\ \mathit{to\text{-}state}\ (\mathit{vars}\ P)\ ms,\ \mathit{add\text{-}stack}\ sp\ ms\ @\ stk) \rightarrow\!*$
$\qquad (pc',\ \mathit{to\text{-}state}\ (\mathit{vars}\ P)\ ms',\ \mathit{add\text{-}stack}\ sp'\ ms'\ @\ stk)$
⟨*proof*⟩

1.3 Verification infrastructure

lemma *iexec-shift* [*simp*]:
$((n + i',\ s',\ stk') = \mathit{iexec}\ \mathit{ins}\ (n + i,\ s,\ stk)) =$
$\quad ((i',\ s',\ stk') = \mathit{iexec}\ \mathit{ins}\ (i,\ s,\ stk))$
⟨*proof*⟩

lemma *exec1-appendR*:
$P \vdash c \rightarrow c' \Longrightarrow P\ @\ P' \vdash c \rightarrow c'$
⟨*proof*⟩

lemma *exec-appendR*:
$P \vdash c \rightarrow\ast c' \Longrightarrow P\ @\ P' \vdash c \rightarrow\ast c'$
⟨*proof*⟩

lemma *exec1-appendL*:
fixes $i\ i' :: int$
shows $P \vdash (i,\ s,\ stk) \rightarrow (i',\ s',\ stk') \Longrightarrow$
$P'\ @\ P \vdash (size\ P' + i,\ s,\ stk) \rightarrow (size\ P' + i',\ s',\ stk')$
⟨*proof*⟩

lemma *exec-appendL*:
fixes $i\ i' :: int$
shows $P \vdash (i,\ s,\ stk) \rightarrow\ast (i',\ s',\ stk') \Longrightarrow$
$P'\ @\ P \vdash (size\ P' + i,\ s,\ stk) \rightarrow\ast (size\ P' + i',\ s',\ stk')$
⟨*proof*⟩

lemma *exec-Cons-1* [*intro*]:
$P \vdash (0,\ s,\ stk) \rightarrow\ast (j,\ t,\ stk') \Longrightarrow$
$ins\ \#\ P \vdash (1,\ s,\ stk) \rightarrow\ast (1 + j,\ t,\ stk')$
⟨*proof*⟩

lemma *exec-appendL-if* [*intro*]:
fixes $i\ i'\ j :: int$
shows $[\![size\ P' \leq i;\ P \vdash (i - size\ P',\ s,\ stk) \rightarrow\ast (j,\ s',\ stk');$
$i' = size\ P' + j]\!] \Longrightarrow$
$P'\ @\ P \vdash (i,\ s,\ stk) \rightarrow\ast (i',\ s',\ stk')$
⟨*proof*⟩

lemma *exec-append-trans* [*intro*]:
fixes $i'\ i''\ j'' :: int$
shows $[\![P \vdash (0,\ s,\ stk) \rightarrow\ast (i',\ s',\ stk');\ size\ P \leq i';$
$P' \vdash (i' - size\ P,\ s',\ stk') \rightarrow\ast (i'',\ s'',\ stk'');\ j'' = size\ P + i'']\!] \Longrightarrow$
$P\ @\ P' \vdash (0,\ s,\ stk) \rightarrow\ast (j'',\ s'',\ stk'')$
⟨*proof*⟩

declare *Let-def* [*simp*]

1.4 Compilation

As mentioned previously, the definitions of the functions modeling source-to-assembly compilation, reported here below, need not be changed. Particularly, function *ccomp* can be used to define some abbreviations for functions *to-m-prog*, *to-state*, and *to-m-state*, in which their first parameter (an assembly program for *to-m-prog*, a list of variables for the other two functions) is replaced with a command. In fact, the compiler completeness and correctness properties apply to machine programs resulting from the compilation of source programs, that is, of commands. Consequently, such abbreviations, defined here below as well, can be used to express those properties in a more concise form.

primrec *acomp* :: *aexp* ⇒ *instr list* **where**
acomp (*N i*) = [*LOADI i*] |
acomp (*V x*) = [*LOAD x*] |
acomp (*Plus* a_1 a_2) = *acomp* a_1 @ *acomp* a_2 @ [*ADD*]

fun *bcomp* :: *bexp* × *bool* × *int* ⇒ *instr list* **where**
bcomp (*Bc v*, *f*, *i*) = (*if v* = *f then* [*JMP i*] *else* []) |
bcomp (*Not b*, *f*, *i*) = *bcomp* (*b*, ¬ *f*, *i*) |
bcomp (*And* b_1 b_2, *f*, *i*) =
  (*let* cb_2 = *bcomp* (b_2, *f*, *i*);
     cb_1 = *bcomp* (b_1, *False*, *size* cb_2 + (*if f then 0 else i*))
   in cb_1 @ cb_2) |
bcomp (*Less* a_1 a_2, *f*, *i*) =
  acomp a_1 @ *acomp* a_2 @ (*if f then* [*JMPLESS i*] *else* [*JMPGE i*])

primrec *ccomp* :: *com* ⇒ *instr list* **where**
ccomp SKIP = [] |
ccomp (*x* ::= *a*) = *acomp a* @ [*STORE x*] |
ccomp (c_1;; c_2) = *ccomp* c_1 @ *ccomp* c_2 |
ccomp (*IF b THEN* c_1 *ELSE* c_2) =
  (*let* cc_1 = *ccomp* c_1; cc_2 = *ccomp* c_2; *cb* = *bcomp* (*b*, *False*, *size* cc_1 + *1*)
   in cb @ cc_1 @ *JMP* (*size* cc_2) # cc_2) |
ccomp (*WHILE b DO c*) =
  (*let cc* = *ccomp c*; *cb* = *bcomp* (*b*, *False*, *size cc* + *1*)
   in cb @ *cc* @ [*JMP* (− (*size cb* + *size cc* + *1*))])

abbreviation *m-ccomp* :: *com* ⇒ *m-instr list* **where**
m-ccomp c ≡ *to-m-prog* (*ccomp c*)

abbreviation *m-state* :: *com* ⇒ *state* ⇒ *m-state* **where**
m-state c ≡ *to-m-state* (*vars* (*ccomp c*))

abbreviation *state* :: *com* ⇒ *m-state* ⇒ *state* **where**
state c ≡ *to-state* (*vars* (*ccomp c*))

lemma *acomp-correct* [*intro*]:
 acomp a ⊢ (*0*, *s*, *stk*) →∗ (*size* (*acomp a*), *s*, *aval a s* # *stk*)
⟨*proof*⟩

lemma *bcomp-correct* [*intro*]:
  fixes *i* :: *int*
  shows *0* ≤ *i* ⟹ *bcomp* (*b*, *f*, *i*) ⊢ (*0*, *s*, *stk*) →∗
    (*size* (*bcomp* (*b*, *f*, *i*)) + (*if f* = *bval b s then i else 0*), *s*, *stk*)
⟨*proof*⟩

1.5 Preservation of semantics

Like [4], this theory ends with the proof of theorem *ccomp-bigstep*, which states that source programs are simulated by assembly ones, as proving that assembly programs are in turn simulated by machine ones is still a pending task. This missing link will be established in the *Compiler2* theory. Such a state of affairs might appear as nothing but an extravagant choice: if the original development detailed in [5] addresses the "easy" direction of the program bisimulation proof in the *Compiler* theory, why moving its machine code add-on to the *Compiler2* theory? The bad news here are that the move has occurred as proving that assembly programs are simulated by machine ones is no longer "easy". Indeed, this task demands the further reasoning tools used in the *Compiler2* theory to cope with the reverse, "hard" direction of the program bisimulation proof. On the other hand, the good news are that such tools, in the form introduced in [7], are sufficiently general and powerful to also accomplish that task, as will be shown shortly.

theorem *ccomp-bigstep*:
$(c,\ s) \Rightarrow t \Longrightarrow ccomp\ c \vdash (0,\ s,\ stk) \rightarrow* (size\ (ccomp\ c),\ t,\ stk)$
⟨*proof*⟩

declare *Let-def* [*simp del*]

lemma *impCE2* [*elim*!]:
$[\![P \longrightarrow Q;\ \neg\ P \Longrightarrow R;\ P \Longrightarrow Q \Longrightarrow R]\!] \Longrightarrow R$
⟨*proof*⟩

lemma *Suc-lessI2* [*intro*!]:
$[\![m < n;\ m \neq n - 1]\!] \Longrightarrow Suc\ m < n$
⟨*proof*⟩

end

2 Compiler verification

theory *Compiler2*
  imports *Compiler*
begin

The reasoning toolbox introduced in the *Compiler2* theory of [7] to cope with the "hard" direction of the bisimulation proof can be outlined as follows.

First, predicate *execl-all* is defined to capture the notion of a *complete small-step* program execution – an *assembly* program execution in the current context –, where such an execution is modeled as a list of program configurations. This predicate has the property that, for any complete execution of program P @ P' @ P'' making the program counter point to the beginning of program P' in some step, there exists a sub-execution being also a complete execution of P'. Under the further assumption that any complete execution of P' fulfills a given predicate Q, this implies the existence of a sub-execution fulfilling Q (as established by lemma *execl-all-sub* in [7]).

The compilation of arithmetic/boolean expressions and commands, modeled by functions *acomp*, *bcomp*, and *ccomp*, produces programs matching pattern P @ P' @ P'', where sub-programs P, P', P'' may either be empty or result from the compilation of nested expressions or commands (possibly with the insertion of further instructions). Moreover, simulation of compiled programs by source ones can be formalized as the statement that any complete small-step execution of a compiled program meets a proper well-behavedness predicate *cpred*. By proving this statement via structural induction over commands, the resulting subgoals assume its validity for any nested command. If as many suitable well-behavedness predicates, *apred* and *bpred*, have been proven to hold for any complete execution of a compiled arithmetic/boolean expression, the above *execl-all*'s property entails that the complete execution targeted in each subgoal is comprised of pieces satisfying *apred*, *bpred*, or *cpred*, which enables to conclude that the whole execution satisfies *cpred*.

Can this machinery come in handy to generalize single-step assembly code simulation by machine code, established by lemma *exec1-m-exec1*, to full program executions? Actually, the gap to be filled in is showing that assembly program execution unfolds in such a way, that a machine stack pointer starting from zero complies with *exec1-m-exec1*'s assumptions in each intermediate step. The key insight, which provides the previous question with an affirmative answer, is that this property can as well be formalized as a well-behavedness predicate *mpred*, so that the pending task takes again the form of proving that such a predicate holds for any complete small-step execution of an assembly program.

Following this insight, the present theory extends the *Compiler2* theory of [7] by reusing its reasoning toolbox to additionally prove that any such program execution is indeed well-behaved in this respect, too.

2.1 Preliminary definitions and lemmas

To define predicate *mpred*, the value taken by the machine stack pointer in every program execution step needs to be expressed as a function of just the initial configuration and the current one, so that a quantification over each intermediate configuration can occur in the definition's right-hand side. On the other hand, within *exec1-m-exec1*'s conclusion, the stack pointer sp' resulting from single-step execution is $sp + \mathit{length}\ stk' - \mathit{length}\ stk$, where *stk* and *sp* are the assembly stack and the stack pointer prior to single-step execution and stk' is the ensuing assembly stack. Thus, the aforesaid function must be such that, by replacing *sp* with its value into the previous expression, *sp′*'s value is obtained. If $\mathit{sp} = \mathit{length}\ \mathit{stk} - \mathit{length}\ \mathit{stk}_0$, where stk_0 is the initial assembly stack, that expression gives $\mathit{sp}' = \mathit{length}\ \mathit{stk} - \mathit{length}\ \mathit{stk}_0 + \mathit{length}\ \mathit{stk}' - \mathit{length}\ \mathit{stk}$, and the right-hand side matches $\mathit{length}\ \mathit{stk}' - \mathit{length}\ \mathit{stk}_0$ by library lemma *add-diff-assoc2* provided that $\mathit{length}\ \mathit{stk}_0 \leq \mathit{length}\ \mathit{stk}$.

Thus, to meet *exec1-m-exec1*'s former assumption for an assembly program *P*, each intermediate configuration (*pc*, *s*, *stk*) in a list *cfs* must be such that (i) $\mathit{length}\ \mathit{stk} - \mathit{length}\ \mathit{stk}_0$ is not less than the number of the operands taken by *P*'s instruction at offset *pc*, and (ii) $\mathit{length}\ \mathit{stk}_0 \leq \mathit{length}\ \mathit{stk}$. Since the subgoals arising from structural induction will assume this to hold for pieces of a given complete execution, it is convenient to make *mpred* take two offsets *m* and *n* as further inputs besides *P* and *cfs*. This enables the quantification to only span the configurations within *cfs* whose offsets are comprised in the interval $\{m..{<}n\}$ (the upper bound is excluded as intermediate configurations alone are relevant). Unlike *apred*, *bpred*, and *cpred*, *mpred* expresses a well-behavedness condition applying indiscriminately to arithmetic/boolean expressions and commands, which is the reason why a single predicate suffices, as long as it takes a list of assembly instructions as input instead of a specific source code token.

```
fun execl :: instr list ⇒ config list ⇒ bool (infix ‹⊨› 55) where
P ⊨ cf # cf′ # cfs = (P ⊢ cf → cf′ ∧ P ⊨ cf′ # cfs) |
P ⊨ - = True

definition execl-all :: instr list ⇒ config list ⇒ bool (‹(-/ ⊨/ -□)› 55) where
P ⊨ cfs□ ≡ P ⊨ cfs ∧ cfs ≠ [] ∧
  fst (cfs ! 0) = 0 ∧ fst (cfs ! (length cfs − 1)) ∉ {0..<size P}

definition apred :: aexp ⇒ config ⇒ config ⇒ bool where
apred ≡ λa (pc, s, stk) (pc′, s′, stk′).
  pc′ = pc + size (acomp a) ∧ s′ = s ∧ stk′ = aval a s # stk

definition bpred :: bexp × bool × int ⇒ config ⇒ config ⇒ bool where
bpred ≡ λ(b, f, i) (pc, s, stk) (pc′, s′, stk′).
  pc′ = pc + size (bcomp (b, f, i)) + (if bval b s = f then i else 0) ∧
    s′ = s ∧ stk′ = stk

definition cpred :: com ⇒ config ⇒ config ⇒ bool where
cpred ≡ λc (pc, s, stk) (pc′, s′, stk′).
  pc′ = pc + size (ccomp c) ∧ (c, s) ⇒ s′ ∧ stk′ = stk

definition mpred :: instr list ⇒ config list ⇒ nat ⇒ nat ⇒ bool where
mpred P cfs m n ≡ case cfs ! 0 of (-, -, stk₀) ⇒
  ∀ k ∈ {m..<n}. case cfs ! k of (pc, -, stk) ⇒
```

$\quad \mathit{msp}\ P\ \mathit{pc} \le \mathit{length}\ \mathit{stk} - \mathit{length}\ \mathit{stk}_0 \wedge \mathit{length}\ \mathit{stk}_0 \le \mathit{length}\ \mathit{stk}$

abbreviation $\mathit{off} :: \mathit{instr}\ \mathit{list} \Rightarrow \mathit{config} \Rightarrow \mathit{config}$ **where**
$\mathit{off}\ P\ \mathit{cf} \equiv (\mathit{fst}\ \mathit{cf} - \mathit{size}\ P,\ \mathit{snd}\ \mathit{cf})$

By slightly extending their conclusions, the lemmas used to prove compiler correctness automatically for constructors *N*, *V*, *Bc*, and *SKIP* can be reused for the new well-behavedness proof as well. Actually, it is sufficient to additionally infer that (i) the given complete execution consists of one or two steps and (ii) in the latter case, the initial program counter is zero, so that the first inequality within *mpred*'s definition matches the trivial one $0 \le 0$.

lemma *iexec-offset* [*intro*]:
$(\mathit{ins},\ \mathit{pc},\ s,\ \mathit{stk}) \mapsto (\mathit{pc}',\ s',\ \mathit{stk}') \Longrightarrow$
$\quad (\mathit{ins},\ \mathit{pc} - i,\ s,\ \mathit{stk}) \mapsto (\mathit{pc}' - i,\ s',\ \mathit{stk}')$
$\langle \mathit{proof} \rangle$

lemma *execl-next*:
$[\![P \models \mathit{cfs};\ k < \mathit{length}\ \mathit{cfs};\ k \neq \mathit{length}\ \mathit{cfs} - \mathit{1}]\!] \Longrightarrow$
$\quad (P\ !!\ \mathit{fst}\ (\mathit{cfs}\ !\ k),\ \mathit{cfs}\ !\ k) \mapsto \mathit{cfs}\ !\ \mathit{Suc}\ k \wedge$
$\quad\quad \mathit{0} \le \mathit{fst}\ (\mathit{cfs}\ !\ k) \wedge \mathit{fst}\ (\mathit{cfs}\ !\ k) < \mathit{size}\ P$
$\langle \mathit{proof} \rangle$

lemma *execl-last*:
$[\![P \models \mathit{cfs};\ k < \mathit{length}\ \mathit{cfs};\ \mathit{fst}\ (\mathit{cfs}\ !\ k) \notin \{\mathit{0}..{<}\mathit{size}\ P\}]\!] \Longrightarrow$
$\quad \mathit{length}\ \mathit{cfs} - \mathit{1} = k$
$\langle \mathit{proof} \rangle$

lemma *execl-take*:
$P \models \mathit{cfs} \Longrightarrow P \models \mathit{take}\ n\ \mathit{cfs}$
$\langle \mathit{proof} \rangle$

lemma *execl-drop*:
$P \models \mathit{cfs} \Longrightarrow P \models \mathit{drop}\ n\ \mathit{cfs}$
$\langle \mathit{proof} \rangle$

lemma *execl-all-N* [*simplified*, *dest*]:
$[\mathit{LOADI}\ i] \models \mathit{cfs}\Box \Longrightarrow \mathit{apred}\ (N\ i)\ (\mathit{cfs}\ !\ \mathit{0})\ (\mathit{cfs}\ !\ (\mathit{length}\ \mathit{cfs} - \mathit{1})) \wedge$
$\quad \mathit{length}\ \mathit{cfs} = \mathit{2} \wedge \mathit{fst}\ (\mathit{cfs}\ !\ \mathit{0}) = \mathit{0}$
$\langle \mathit{proof} \rangle$

lemma *execl-all-V* [*simplified*, *dest*]:
$[\mathit{LOAD}\ x] \models \mathit{cfs}\Box \Longrightarrow \mathit{apred}\ (V\ x)\ (\mathit{cfs}\ !\ \mathit{0})\ (\mathit{cfs}\ !\ (\mathit{length}\ \mathit{cfs} - \mathit{1})) \wedge$
$\quad \mathit{length}\ \mathit{cfs} = \mathit{2} \wedge \mathit{fst}\ (\mathit{cfs}\ !\ \mathit{0}) = \mathit{0}$
$\langle \mathit{proof} \rangle$

lemma *execl-all-Bc* [*simplified*, *dest*]:

$[\![\mathit{if}\ v = f\ \mathit{then}\ [\mathit{JMP}\ i]\ \mathit{else}\ []\ \models \mathit{cfs}\square;\ \mathit{0} \le i]\!] \Longrightarrow$
$\quad \mathit{bpred}\ (\mathit{Bc}\ v,\ f,\ i)\ (\mathit{cfs}\ !\ \mathit{0})\ (\mathit{cfs}\ !\ (\mathit{length}\ \mathit{cfs} - \mathit{1}))\ \wedge$
$\quad \mathit{length}\ \mathit{cfs} = (\mathit{if}\ v = f\ \mathit{then}\ \mathit{2}\ \mathit{else}\ \mathit{1})\ \wedge\ \mathit{fst}\ (\mathit{cfs}\ !\ \mathit{0}) = \mathit{0}$
$\langle \mathit{proof} \rangle$

lemma *execl-all-SKIP* [*simplified*, *dest*]:
$[]\ \models \mathit{cfs}\square \Longrightarrow \mathit{cpred}\ \mathit{SKIP}\ (\mathit{cfs}\ !\ \mathit{0})\ (\mathit{cfs}\ !\ (\mathit{length}\ \mathit{cfs} - \mathit{1}))\ \wedge\ \mathit{length}\ \mathit{cfs} = \mathit{1}$
$\langle \mathit{proof} \rangle$

In [7], part of the proof of lemma *execl-all-sub* is devoted to establishing the fundamental property of predicate *execl-all* stated above: for any complete execution of program $P\ @\ P'\ @\ P''$ making the program counter point to the beginning of P' in its k-th step, there exists a sub-execution starting from the k-th step and being a complete execution of P'.

Here below, this property is proven as a lemma in its own respect, named *execl-all*, so that besides *execl-all-sub*, it can be reused to prove a further lemma *execl-all-sub-m*. This new lemma establishes that, if (i) *execl-all-sub*'s assumptions hold, (ii) any complete execution of P' fulfills predicate *mpred*, and (iii) the initial assembly stack is not longer than the one in the k-th step, then there exists a sub-execution starting from the k-th step and fulfilling both predicates Q and *mpred*. Within the new well-behavedness proof, this lemma will play the same role as *execl-all-sub* in the compiler correctness proof; namely, for each structural induction subgoal, it will entail that the respective complete execution is comprised of pieces fulfilling *mpred*. As with *execl-all-sub*, Q can be instantiated to *apred*, *bpred*, or *cpred*; indeed, knowing that sub-executions satisfy these predicates in addition to *mpred* is necessary to show that the whole execution satisfies *mpred*. For example, to draw the conclusion that the assembly code *acomp a* @ [*STORE x*] for an assignment meets *mpred*, one needs to know that *acomp a*'s sub-execution also meets *apred*, so that the assembly stack contains an element more than the initial stack when instruction *STORE x* is executed.

lemma *execl-sub-aux*:
$[\![\bigwedge m\ n.\ \forall k \in \{m..{<}n\}.\ Q\ P'\ (((\mathit{pc},\ s,\ \mathit{stk})\ \#\ \mathit{cfs})\ !\ k) \Longrightarrow P' \models$
$\quad \mathit{map}\ (\mathit{off}\ P)\ (\mathit{case}\ m\ \mathit{of}\ \mathit{0} \Rightarrow (\mathit{pc},\ s,\ \mathit{stk})\ \#\ \mathit{take}\ n\ \mathit{cfs}\ |\ \mathit{Suc}\ m \Rightarrow F\ \mathit{cfs}\ m\ n);$
$\quad \forall k \in \{m..{<}n{+}m{+}\mathit{length}\ \mathit{cfs}'\}.\ Q\ P'\ ((\mathit{cfs}'\ @\ (\mathit{pc},\ s,\ \mathit{stk})\ \#\ \mathit{cfs})\ !\ (k{-}m))]\!] \Longrightarrow$
$\quad P' \models (\mathit{pc} - \mathit{size}\ P,\ s,\ \mathit{stk})\ \#\ \mathit{map}\ (\mathit{off}\ P)\ (\mathit{take}\ n\ \mathit{cfs})$
$\quad (\mathbf{is}\ [\![\bigwedge\text{- -}.\ \forall k \in \text{-}.\ Q\ P'\ (?F\ k) \Longrightarrow \text{-};\ \forall k \in ?A.\ Q\ P'\ (?G\ k)]\!] \Longrightarrow \text{-})$
$\langle \mathit{proof} \rangle$

lemma *execl-sub*:
$[\![P\ @\ P'\ @\ P'' \models \mathit{cfs};\ \forall k \in \{m..{<}n\}.$
$\quad \mathit{size}\ P \le \mathit{fst}\ (\mathit{cfs}\ !\ k)\ \wedge\ \mathit{fst}\ (\mathit{cfs}\ !\ k) - \mathit{size}\ P < \mathit{size}\ P']\!] \Longrightarrow$
$\quad P' \models \mathit{map}\ (\mathit{off}\ P)\ (\mathit{drop}\ m\ (\mathit{take}\ (\mathit{Suc}\ n)\ \mathit{cfs}))$
$\quad (\mathbf{is}\ [\![\text{-};\ \forall k \in \text{-}.\ ?P\ P'\ \mathit{cfs}\ k]\!] \Longrightarrow P' \models \mathit{map}\ \text{-}\ (?F\ \mathit{cfs}\ m\ (\mathit{Suc}\ n)))$

⟨*proof*⟩

lemma *execl-all*:
assumes
A: *P* @ *P′ x* @ *P″* ⊨ *cfs*□ **and**
B: *k* < *length cfs* **and**
C: *fst* (*cfs* ! *k*) = *size P*
shows ∃ *k′* ∈ {*k*..<*length cfs*}. *P′ x* ⊨ *map* (*off P*) (*drop k* (*take* (*Suc k′*) *cfs*))□
(**is** ∃ *k′* ∈ -. - ⊨ *?F k′*□)
⟨*proof*⟩

lemma *execl-all-sub* [*rule-format*]:
assumes
A: *P* @ *P′ x* @ *P″* ⊨ *cfs*□ **and**
B: *k* < *length cfs* **and**
C: *fst* (*cfs* ! *k*) = *size P* **and**
D: ∀ *cfs*. *P′ x* ⊨ *cfs*□ ⟶ *Q x* (*cfs* ! *0*) (*cfs* ! (*length cfs* − *1*))
shows ∃ *k′* < *length cfs*. *Q x* (*off P* (*cfs* ! *k*)) (*off P* (*cfs* ! *k′*))
⟨*proof*⟩

lemma *execl-all-sub2*:
assumes
A: *P x* @ *P′ x′* @ *P″* ⊨ *cfs*□
(**is** *?P* ⊨ -□) **and**
B: ⋀*cfs*. *P x* ⊨ *cfs*□ ⟹ (λ(*pc*, *s*, *stk*) (*pc′*, *s′*, *stk′*).
pc′ = *pc* + *size* (*P x*) + *I s* ∧ *Q s s′* ∧ *stk′* = *F s stk*)
(*cfs* ! *0*) (*cfs* ! (*length cfs* − *1*))
(**is** ⋀*cfs*. - ⟹ *?Q x* (*cfs* ! *0*) (*cfs* ! (*length cfs* − *1*))) **and**
C: ⋀*cfs*. *P′ x′* ⊨ *cfs*□ ⟹ (λ(*pc*, *s*, *stk*) (*pc′*, *s′*, *stk′*).
pc′ = *pc* + *size* (*P′ x′*) + *I′ s* ∧ *Q′ s s′* ∧ *stk′* = *F′ s stk*)
(*cfs* ! *0*) (*cfs* ! (*length cfs* − *1*))
(**is** ⋀*cfs*. - ⟹ *?Q′ x′* (*cfs* ! *0*) (*cfs* ! (*length cfs* − *1*))) **and**
D: *I* (*fst* (*snd* (*cfs* ! *0*))) = *0*
shows ∃ *k* < *length cfs*. ∃ *t*. (λ(*pc*, *s*, *stk*) (*pc′*, *s′*, *stk′*).
pc = *0* ∧ *pc′* = *size* (*P x*) + *size* (*P′ x′*) + *I′ t* ∧ *Q s t* ∧ *Q′ t s′* ∧
stk′ = *F′ t* (*F s stk*)) (*cfs* ! *0*) (*cfs* ! *k*)
⟨*proof*⟩

lemma *execl-all-sub-m* [*rule-format*]:
assumes
A: *P* @ *P′ x* @ *P″* ⊨ *cfs*□ **and**
B: *k* < *length cfs* **and**
C: *fst* (*cfs* ! *k*) = *size P* **and**
D: *length* (*snd* (*snd* (*cfs* ! *0*))) ≤ *length* (*snd* (*snd* (*cfs* ! *k*))) **and**
E: ∀ *cfs*. *P′ x* ⊨ *cfs*□ ⟶ *Q x* (*cfs* ! *0*) (*cfs* ! (*length cfs* − *1*)) **and**
F: ∀ *cfs*. *P′ x* ⊨ *cfs*□ ⟶ *mpred* (*P′ x*) *cfs 0* (*length cfs* − *1*)
shows ∃ *k′* < *length cfs*. *Q x* (*off P* (*cfs* ! *k*)) (*off P* (*cfs* ! *k′*)) ∧
mpred (*P* @ *P′ x* @ *P″*) *cfs k k′*
⟨*proof*⟩

The lemmas here below establish the properties of predicate *mpred* required for the new well-behavedness proof. In more detail:

- Lemma *mpred-merge* states that, if two consecutive sublists of a list of configurations are both well-behaved, then such is the merged sublist. This lemma is the means enabling to infer that a complete execution made of well-behaved pieces is itself well-behaved.

- Lemma *mpred-drop* states that, under proper assumptions, if a sublist of a suffix of a list of configurations is well-behaved, then such is the matching sublist of the whole list. In the subgoal of the well-behavedness proof for loops where an iteration has been run, this lemma can be used to deduce the well-behavedness of the whole execution from that of the sub-execution following that iteration.

- Lemma *mpred-execl-m-exec* states that, if a nonempty small-step assembly code execution is well-behaved, then the machine configurations corresponding to the initial and final assembly ones are linked by a machine code execution. Namely, this lemma proves that the well-behavedness property expressed by predicate *mpred* is sufficient to fulfill the assumptions of lemma *exec1-m-exec1* in each intermediate step. Once any complete small-step assembly program execution is proven to satisfy *mpred*, this lemma can then be used to achieve the final goal of establishing that source programs are simulated by machine ones.

```
lemma mpred-merge:
⟦mpred P cfs k m; mpred P cfs m n⟧ ⟹ mpred P cfs k n
⟨proof⟩

lemma mpred-drop:
  assumes
    A: k ≤ length cfs and
    B: length (snd (snd (cfs ! 0))) ≤ length (snd (snd (cfs ! k)))
  shows mpred P (drop k cfs) m n ⟹ mpred P cfs (k + m) (k + n)
⟨proof⟩

lemma mpred-execl-m-exec [simplified Let-def]:
⟦cfs ≠ []; P ⊨ cfs; mpred P cfs 0 (length cfs − 1)⟧ ⟹
    case (cfs ! 0, cfs ! (length cfs − 1)) of ((pc, s, stk), (pc′, s′, stk′)) ⇒
      let sp′ = length stk′ − length stk in to-m-prog P ⊢
        (pc, to-m-state (vars P) s, 0) →*
        (pc′, add-m-stack sp′ stk′ (to-m-state (vars P) s′), sp′)
⟨proof⟩
```

2.2 Main theorems

Here below is the proof that every complete small-step execution of an assembly program fulfills predicate *cpred* (lemma *ccomp-correct*), which is reused as is from [7], followed by the proof that every such execution satisfies predicate *mpred* as well (lemma *ccomp-correct-m*), which closely resembles the former one.

lemma *acomp-acomp*:
$[\![\mathit{acomp}\ a_1\ @\ \mathit{acomp}\ a_2\ @\ P \models \mathit{cfs}\square;$
$\quad \bigwedge \mathit{cfs}.\ \mathit{acomp}\ a_1 \models \mathit{cfs}\square \Longrightarrow \mathit{apred}\ a_1\ (\mathit{cfs}\ !\ \mathit{0})\ (\mathit{cfs}\ !\ (\mathit{length}\ \mathit{cfs} - \mathit{1}));$
$\quad \bigwedge \mathit{cfs}.\ \mathit{acomp}\ a_2 \models \mathit{cfs}\square \Longrightarrow \mathit{apred}\ a_2\ (\mathit{cfs}\ !\ \mathit{0})\ (\mathit{cfs}\ !\ (\mathit{length}\ \mathit{cfs} - \mathit{1}))]\!] \Longrightarrow$
$\ \mathit{case}\ \mathit{cfs}\ !\ \mathit{0}\ \mathit{of}\ (\mathit{pc},\ s,\ \mathit{stk}) \Rightarrow \mathit{pc} = \mathit{0} \land (\exists\, k < \mathit{length}\ \mathit{cfs}.\ \mathit{cfs}\ !\ k =$
$\quad (\mathit{size}\ (\mathit{acomp}\ a_1\ @\ \mathit{acomp}\ a_2),\ s,\ \mathit{aval}\ a_2\ s\ \#\ \mathit{aval}\ a_1\ s\ \#\ \mathit{stk}))$
$\langle \mathit{proof} \rangle$

lemma *bcomp-bcomp*:
$[\![\mathit{bcomp}\ (b_1,\ f_1,\ i_1)\ @\ \mathit{bcomp}\ (b_2,\ f_2,\ i_2) \models \mathit{cfs}\square;$
$\quad \bigwedge \mathit{cfs}.\ \mathit{bcomp}\ (b_1,\ f_1,\ i_1) \models \mathit{cfs}\square \Longrightarrow$
$\qquad \mathit{bpred}\ (b_1,\ f_1,\ i_1)\ (\mathit{cfs}\ !\ \mathit{0})\ (\mathit{cfs}\ !\ (\mathit{length}\ \mathit{cfs} - \mathit{1}));$
$\quad \bigwedge \mathit{cfs}.\ \mathit{bcomp}\ (b_2,\ f_2,\ i_2) \models \mathit{cfs}\square \Longrightarrow$
$\qquad \mathit{bpred}\ (b_2,\ f_2,\ i_2)\ (\mathit{cfs}\ !\ \mathit{0})\ (\mathit{cfs}\ !\ (\mathit{length}\ \mathit{cfs} - \mathit{1}))]\!] \Longrightarrow$
$\ \mathit{case}\ \mathit{cfs}\ !\ \mathit{0}\ \mathit{of}\ (\mathit{pc},\ s,\ \mathit{stk}) \Rightarrow \mathit{pc} = \mathit{0} \land (\mathit{bval}\ b_1\ s \neq f_1 \longrightarrow$
$\quad (\exists\, k < \mathit{length}\ \mathit{cfs}.\ \mathit{cfs}\ !\ k = (\mathit{size}\ (\mathit{bcomp}\ (b_1,\ f_1,\ i_1)\ @\ \mathit{bcomp}\ (b_2,\ f_2,\ i_2)) +$
$\qquad (\mathit{if}\ \mathit{bval}\ b_2\ s = f_2\ \mathit{then}\ i_2\ \mathit{else}\ \mathit{0}),\ s,\ \mathit{stk})))$
$\langle \mathit{proof} \rangle$

lemma *acomp-correct* [*simplified*, *intro*]:
$\ \mathit{acomp}\ a \models \mathit{cfs}\square \Longrightarrow \mathit{apred}\ a\ (\mathit{cfs}\ !\ \mathit{0})\ (\mathit{cfs}\ !\ (\mathit{length}\ \mathit{cfs} - \mathit{1}))$
$\langle \mathit{proof} \rangle$

lemma *bcomp-correct* [*simplified*, *intro*]:
$[\![\mathit{bcomp}\ x \models \mathit{cfs}\square;\ \mathit{0} \leq \mathit{snd}\ (\mathit{snd}\ x)]\!] \Longrightarrow \mathit{bpred}\ x\ (\mathit{cfs}\ !\ \mathit{0})\ (\mathit{cfs}\ !\ (\mathit{length}\ \mathit{cfs} - \mathit{1}))$
$\langle \mathit{proof} \rangle$

lemma *bcomp-ccomp*:
$[\![\mathit{bcomp}\ (b,\ f,\ i)\ @\ \mathit{ccomp}\ c\ @\ P \models \mathit{cfs}\square;\ \mathit{0} \leq i;$
$\quad \bigwedge \mathit{cfs}.\ \mathit{ccomp}\ c \models \mathit{cfs}\square \Longrightarrow \mathit{cpred}\ c\ (\mathit{cfs}\ !\ \mathit{0})\ (\mathit{cfs}\ !\ (\mathit{length}\ \mathit{cfs} - \mathit{1}))]\!] \Longrightarrow$
$\ \mathit{case}\ \mathit{cfs}\ !\ \mathit{0}\ \mathit{of}\ (\mathit{pc},\ s,\ \mathit{stk}) \Rightarrow \mathit{pc} = \mathit{0} \land (\mathit{bval}\ b\ s \neq f \longrightarrow$
$\quad (\exists\, k < \mathit{length}\ \mathit{cfs}.\ \mathit{case}\ \mathit{cfs}\ !\ k\ \mathit{of}\ (\mathit{pc}',\ s',\ \mathit{stk}') \Rightarrow$
$\qquad \mathit{pc}' = \mathit{size}\ (\mathit{bcomp}\ (b,\ f,\ i)\ @\ \mathit{ccomp}\ c) \land (c,\ s) \Rightarrow s' \land \mathit{stk}' = \mathit{stk}))$
$\langle \mathit{proof} \rangle$

lemma *ccomp-ccomp*:
$[\![\mathit{ccomp}\ c_1\ @\ \mathit{ccomp}\ c_2 \models \mathit{cfs}\square;$
$\quad \bigwedge \mathit{cfs}.\ \mathit{ccomp}\ c_1 \models \mathit{cfs}\square \Longrightarrow \mathit{cpred}\ c_1\ (\mathit{cfs}\ !\ \mathit{0})\ (\mathit{cfs}\ !\ (\mathit{length}\ \mathit{cfs} - \mathit{1}));$
$\quad \bigwedge \mathit{cfs}.\ \mathit{ccomp}\ c_2 \models \mathit{cfs}\square \Longrightarrow \mathit{cpred}\ c_2\ (\mathit{cfs}\ !\ \mathit{0})\ (\mathit{cfs}\ !\ (\mathit{length}\ \mathit{cfs} - \mathit{1}))]\!] \Longrightarrow$

case cfs ! 0 of (pc, s, stk) ⇒ pc = 0 ∧ (∃ k < length cfs. ∃ t.
case cfs ! k of (pc′, s′, stk′) ⇒ pc′ = size (ccomp c_1 *@ ccomp* c_2*) ∧*
(c_1*, s) ⇒ t ∧ (*c_2*, t) ⇒ s′ ∧ stk′ = stk)*
⟨*proof*⟩

lemma *while-correct* [*simplified*, *intro*]:
⟦*bcomp (b, False, size (ccomp c) + 1) @ ccomp c @*
[JMP (− (size (bcomp (b, False, size (ccomp c) + 1) @ ccomp c) + 1))]
⊨ cfs□;
⋀*cfs. ccomp c ⊨ cfs*□ ⟹ *cpred c (cfs ! 0) (cfs ! (length cfs − 1))*⟧ ⟹
cpred (WHILE b DO c) (cfs ! 0) (cfs ! (length cfs − Suc 0))
(**is** ⟦*?cb @ ?cc @ [JMP (− ?n)] ⊨ -*□; ⋀-. - ⟹ -⟧ ⟹ *?Q cfs*)
⟨*proof*⟩

lemma *ccomp-correct* [*simplified*, *intro*]:
ccomp c ⊨ cfs□ ⟹ *cpred c (cfs ! 0) (cfs ! (length cfs − 1))*
⟨*proof*⟩

lemma *acomp-acomp-m*:
assumes
A: *acomp* a_1 *@ acomp* a_2 *@ P ⊨ cfs*□
(**is** *?P ⊨ -*□) **and**
B: ⋀*cfs. acomp* a_1 *⊨ cfs*□ ⟹ *mpred (acomp* a_1*) cfs 0 (length cfs − 1)* **and**
C: ⋀*cfs. acomp* a_2 *⊨ cfs*□ ⟹ *mpred (acomp* a_2*) cfs 0 (length cfs − 1)*
shows *case cfs ! 0 of (pc, s, stk) ⇒ ∃ k < length cfs.*
cfs ! k = (size (acomp a_1 *@ acomp* a_2*), s, aval* a_2 *s # aval* a_1 *s # stk) ∧*
mpred ?P cfs 0 k
⟨*proof*⟩

lemma *bcomp-bcomp-m* [*simplified*, *intro*]:
assumes *A*: *bcomp (*b_1*,* f_1*,* i_1*) @ bcomp (*b_2*,* f_2*,* i_2*) ⊨ cfs*□
(**is** *bcomp ?*x_1 *@ bcomp ?*x_2 *⊨ -*□)
assumes
B: ⋀*cfs. bcomp ?*x_1 *⊨ cfs*□ ⟹ *mpred (bcomp ?*x_1*) cfs 0 (length cfs − 1)* **and**
C: ⋀*cfs. bcomp ?*x_2 *⊨ cfs*□ ⟹ *mpred (bcomp ?*x_2*) cfs 0 (length cfs − 1)* **and**
D: *size (bcomp ?*x_2*) ≤* i_1 **and**
E: *0 ≤* i_2
shows *mpred (bcomp ?*x_1 *@ bcomp ?*x_2*) cfs 0 (length cfs − 1)*
(**is** *mpred ?P - - -*)
⟨*proof*⟩

lemma *acomp-correct-m* [*simplified*, *intro*]:
acomp a ⊨ cfs□ ⟹ *mpred (acomp a) cfs 0 (length cfs − 1)*
⟨*proof*⟩

lemma *bcomp-correct-m* [*simplified*, *intro*]:
⟦*bcomp x ⊨ cfs*□; *0 ≤ snd (snd x)*⟧ ⟹ *mpred (bcomp x) cfs 0 (length cfs − 1)*
⟨*proof*⟩

lemma *bcomp-ccomp-m*:
  assumes *A*: *bcomp* (*b*, *f*, *i*) @ *ccomp c* @ *P* ⊨ *cfs*□
    (**is** *bcomp ?x* @ *?cc* @ - ⊨ -□)
  assumes
    B: ⋀*cfs*. *?cc* ⊨ *cfs*□ ⟹ *mpred ?cc cfs 0* (*length cfs* − *1*) **and**
    C: *0* ≤ *i*
  shows *case cfs* ! *0 of* (*pc*, *s*, *stk*) ⇒ ∃ *k* < *length cfs*. ∃ *s*′.
    cfs ! *k* = (*size* (*bcomp ?x*) + (*if bval b s* = *f then i else size ?cc*), *s*′, *stk*) ∧
    mpred (*bcomp ?x* @ *?cc* @ *P*) *cfs 0 k*
⟨*proof*⟩

lemma *ccomp-ccomp-m* [*simplified*, *intro*]:
  assumes
    A: *ccomp* c_1 @ *ccomp* c_2 ⊨ *cfs*□
      (**is** *?P* ⊨ -□) **and**
    B: ⋀*cfs*. *ccomp* c_1 ⊨ *cfs*□ ⟹ *mpred* (*ccomp* c_1) *cfs 0* (*length cfs* − *1*) **and**
    C: ⋀*cfs*. *ccomp* c_2 ⊨ *cfs*□ ⟹ *mpred* (*ccomp* c_2) *cfs 0* (*length cfs* − *1*)
  shows *mpred ?P cfs 0* (*length cfs* − *1*)
⟨*proof*⟩

lemma *while-correct-m* [*simplified*, *simplified Let-def*, *intro*]:
 ⟦*bcomp* (*b*, *False*, *size* (*ccomp c*) + *1*) @ *ccomp c* @
    [*JMP* (− (*size* (*bcomp* (*b*, *False*, *size* (*ccomp c*) + *1*) @ *ccomp c*) + *1*))]
      ⊨ *cfs*□;
    ⋀*cfs*. *ccomp c* ⊨ *cfs*□ ⟹ *mpred* (*ccomp c*) *cfs 0* (*length cfs* − *1*)⟧ ⟹
  mpred (*ccomp* (*WHILE b DO c*)) *cfs 0* (*length cfs* − *Suc 0*)
  (**is** ⟦*?cb* @ *?cc* @ - ⊨ -□; ⋀-. - ⟹ -⟧ ⟹ -)
⟨*proof*⟩

lemma *ccomp-correct-m*:
 ccomp c ⊨ *cfs*□ ⟹ *mpred* (*ccomp c*) *cfs 0* (*length cfs* − *1*)
⟨*proof*⟩

Here below are the proofs of theorems *m-ccomp-bigstep* and *m-ccomp-exec*, which establish that machine programs simulate source ones and vice versa. The former theorem is inferred from theorem *ccomp-bigstep* and lemmas *mpred-execl-m-exec*, *ccomp-correct-m*, the latter one from lemma *m-exec-exec* and theorem *ccomp-exec*, in turn derived from lemma *ccomp-correct*.

lemma *exec-execl* [*dest*!]:
 P ⊢ *cf* →∗ *cf*′ ⟹ ∃ *cfs*. *P* ⊨ *cfs* ∧ *cfs* ≠ [] ∧ *hd cfs* = *cf* ∧ *last cfs* = *cf*′
⟨*proof*⟩

theorem *m-ccomp-bigstep*:
 (*c*, *s*) ⇒ *s*′ ⟹

m-ccomp c ⊢ (*0*, *m-state c s*, *0*) →∗ (*size* (*m-ccomp c*), *m-state c s′*, *0*)
⟨*proof*⟩

theorem *ccomp-exec*:
ccomp c ⊢ (*0*, *s*, *stk*) →∗ (*size* (*ccomp c*), *s′*, *stk′*) ⟹ (*c*, *s*) ⇒ *s′* ∧ *stk′* = *stk*
⟨*proof*⟩

theorem *m-ccomp-exec*:
m-ccomp c ⊢ (*0*, *ms*, *0*) →∗ (*size* (*m-ccomp c*), *ms′*, *sp*) ⟹
(*c*, *state c ms*) ⇒ *state c ms′* ∧ *sp* = *0*
⟨*proof*⟩

end

References

[1] A. Krauss. *Defining Recursive Functions in Isabelle/HOL.* https://isabelle.in.tum.de/website-Isabelle2021-1/dist/Isabelle2021-1/doc/functions.pdf.

[2] T. Nipkow. *A Tutorial Introduction to Structured Isar Proofs.* https://isabelle.in.tum.de/website-Isabelle2011/dist/Isabelle2011/doc/isar-overview.pdf.

[3] T. Nipkow. *Programming and Proving in Isabelle/HOL*, Dec. 2021. https://isabelle.in.tum.de/website-Isabelle2021-1/dist/Isabelle2021-1/doc/prog-prove.pdf.

[4] T. Nipkow and G. Klein. Theory HOL-IMP.Compiler (included in the Isabelle2021-1 distribution). https://isabelle.in.tum.de/website-Isabelle2021-1/dist/library/HOL/HOL-IMP/Compiler.html.

[5] T. Nipkow and G. Klein. *Concrete Semantics with Isabelle/HOL.* Springer-Verlag, Mar. 2021. (Current version: http://www.concrete-semantics.org/concrete-semantics.pdf).

[6] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, Dec. 2021. https://isabelle.in.tum.de/website-Isabelle2021-1/dist/Isabelle2021-1/doc/tutorial.pdf.

[7] P. Noce. A Shorter Compiler Correctness Proof for Language IMP. *Archive of Formal Proofs*, June 2021. https://isa-afp.org/entries/IMP_Compiler.html, Formal proof development.