A Shorter Compiler Correctness Proof for Language IMP

Pasquale Noce
Software 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

This paper presents a compiler correctness proof for the didactic imperative programming language IMP, introduced in Nipkow and Klein's book on formal programming language semantics (version of March 2021), whose size is just two thirds of the book's proof in the number of formal text lines. As such, it promises to constitute a further enhanced reference for the formal verification of compilers meant for larger, real-world programming languages.

The presented proof does not depend on language determinism, so that the proposed approach can be applied to non-deterministic languages as well. As a confirmation, this paper extends IMP with an additional non-deterministic choice command, and proves compiler correctness, viz. the simulation of compiled code execution by source code, for such extended language.

Contents

1 Compiler formalization

theory *Compiler*
imports
HOL−IMP.BExp

HOL−IMP.Star
begin

1.1 Introduction

This paper presents a compiler correctness proof for the didactic imperative programming language IMP, introduced in [5], shorter than the proof described in [5] and included in the Isabelle2021 distribution [1]. Actually, the size of the presented proof is just two thirds of the book's proof in the number of formal text lines, and as such it promises to constitute a further enhanced reference for the formal verification of compilers meant for larger, real-world programming languages.

Given compiler *completeness*, viz. the simulation of source code execution by compiled code, "in a deterministic language like IMP", compiler correctness "reduces to preserving termination: if the machine program terminates, so must the source program", even though proving this "is not much easier" ([5], section 8.4). However, the presented proof does not depend on language determinism, so that the proposed approach is applicable to non-deterministic languages as well.

As a confirmation, this paper extends IMP with an additional command c_1 *OR* c_2, standing for the non-deterministic choice between commands c_1 and c_2, and proves compiler *correctness*, viz. the simulation of compiled code execution by source code, for such extended language. Of course, the aforesaid comparison between proof sizes does not consider the lines in the proof of lemma *ccomp-correct* (which proves compiler correctness for commands) pertaining to non-deterministic choice, since this command is not included in the original language IMP. Anyway, non-deterministic choice turns out to extend that proof just by a modest number of lines.

For further information about the formal definitions and proofs contained in this paper, see Isabelle documentation, particularly [6], [4], [2], and [3].

1.2 Definitions

Here below are the definitions of IMP commands, extended with non-deterministic choice, as well as of their big-step semantics.

As in the original theory file [1], program counter's values are modeled using type *int* rather than *nat*. As a result, the same declarations and definitions used in [1] to deal with this modeling choice are adopted here as well.

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

```
datatype com =
  SKIP |
  Assign vname aexp  (‹- ::= -› [1000, 61] 61) |
  Seq com com  (‹-;;/ -› [60, 61] 60) |
  If bexp com com  (‹(IF -/ THEN -/ ELSE -)› [0, 0, 61] 61) |
  Or com com  (‹(- OR -)› [60, 61] 61) |
  While bexp com  (‹(WHILE -/ DO -)› [0, 61] 61)
```

```
inductive big-step :: com × state ⇒ state ⇒ bool (infix ‹⇒› 55) where
Skip:  (SKIP, s) ⇒ s |
Assign:  (x ::= a, s) ⇒ s(x := aval a s) |
Seq:  ⟦(c1, s1) ⇒ s2; (c2, s2) ⇒ s3⟧ ⟹ (c1;; c2, s1) ⇒ s3 |
IfTrue:  ⟦bval b s; (c1, s) ⇒ t⟧ ⟹ (IF b THEN c1 ELSE c2, s) ⇒ t |
IfFalse:  ⟦¬ bval b s; (c2, s) ⇒ t⟧ ⟹ (IF b THEN c1 ELSE c2, s) ⇒ t |
Or1:  (c1, s) ⇒ t ⟹ (c1 OR c2, s) ⇒ t |
Or2:  (c2, s) ⇒ t ⟹ (c1 OR c2, s) ⇒ t |
WhileFalse:  ¬ bval b s ⟹ (WHILE b DO c, s) ⇒ s |
WhileTrue:  ⟦bval b s1; (c, s1) ⇒ s2; (WHILE b DO c, s2) ⇒ s3⟧ ⟹
  (WHILE b DO c, s1) ⇒ s3
```

```
declare big-step.intros [intro]
```

```
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))
by (induction xs arbitrary: i, auto simp: algebra-simps)
```

Next, the instruction set and its semantics are defined. Particularly, to allow for the compilation of non-deterministic choice commands, the instruction set is extended with an additional instruction *JMPND* performing a non-deterministic jump – viz. as a result of its execution, the program counter unconditionally either jumps by the specified offset, or just moves to the next instruction.

As instruction execution can be non-deterministic, an inductively defined predicate *iexec*, rather than a simple non-recursive function as the one used in [1], must be introduced to define instruction semantics.

```
datatype instr =
  LOADI int | LOAD vname | ADD | STORE vname |
```

```
  JMP int | JMPLESS int | JMPGE int | JMPND int

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

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

inductive iexec :: instr × config ⇒ config ⇒ bool (infix ‹↦› 55) where
LoadI: (LOADI i, pc, s, stk) ↦ (pc + 1, s, i # stk) |
Load: (LOAD x, pc, s, stk) ↦ (pc + 1, s, s x # stk) |
Add: (ADD, pc, s, stk) ↦ (pc + 1, s, (hd2 stk + hd stk) # tl2 stk) |
Store: (STORE x, pc, s, stk) ↦ (pc + 1, s(x := hd stk), tl stk) |
Jmp: (JMP i, pc, s, stk) ↦ (pc + i + 1, s, stk) |
JmpLessY: hd2 stk < hd stk ⟹
  (JMPLESS i, pc, s, stk) ↦ (pc + i + 1, s, tl2 stk) |
JmpLessN: hd stk ≤ hd2 stk ⟹
  (JMPLESS i, pc, s, stk) ↦ (pc + 1, s, tl2 stk) |
JmpGeY: hd stk ≤ hd2 stk ⟹
  (JMPGE i, pc, s, stk) ↦ (pc + i + 1, s, tl2 stk) |
JmpGeN: hd2 stk < hd stk ⟹
  (JMPGE i, pc, s, stk) ↦ (pc + 1, s, tl2 stk) |
JmpNdY: (JMPND i, pc, s, stk) ↦ (pc + i + 1, s, stk) |
JmpNdN: (JMPND i, pc, s, stk) ↦ (pc + 1, s, stk)

declare iexec.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
inductive-cases JmpNdE [elim!]: (JMPND i, pc, s, stk) ↦ 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)
```

Next, compilation is formalized for arithmetic and boolean expressions (functions *acomp* and *bcomp*), as well as for commands (function *ccomp*). Particularly, as opposed to what happens in [1], here *bcomp* takes a single input, viz. a 3-tuple comprised of a boolean expression, a flag, and a jump offset. In this way, all three functions accept a single input, which enables to streamline the compiler correctness proof developed in what follows.

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 (c_1 *OR* c_2) =
  (*let* cc_1 = *ccomp* c_1; cc_2 = *ccomp* c_2
  in JMPND (*size* cc_1 + *1*) # 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*))])

Finally, two lemmas are proven automatically (both seem not to be included in the standard library, though being quite basic) and registered for use by automatic proof tactics. In more detail:

- The former lemma is an elimination rule similar to *impCE*, with the difference that it retains the antecedent of the implication in the premise where the consequent is assumed to hold. This rule permits to have both assumptions ¬ *bval b s* and *bval b s* in the respective cases resulting from the execution of boolean expression *b* in state *s*.
- The latter one is an introduction rule similar to *Suc-lessI*, with the difference that its second assumption is more convenient for proving statements of the form *Suc m* < *n* arising from the compiler correctness proof developed in what follows.

lemma *impCE2* [*elim*!]:
$[\![P \longrightarrow Q; \neg P \Longrightarrow R; P \Longrightarrow Q \Longrightarrow R]\!] \Longrightarrow R$
by *blast*

lemma *Suc-lessI2* [*intro*!]:
$[\![m < n; m \neq n - 1]\!] \Longrightarrow Suc\ m < n$
by *simp*

end

2 Compiler correctness

theory *Compiler2*
  imports *Compiler*
begin

2.1 Preliminary definitions and lemmas

Now everything is ready for the compiler correctness proof. First, two predicates are introduced, *execl* and *execl-all*, both taking as inputs a program, i.e. a list of instructions, *P* and a list of program configurations *cfs*, and respectively denoted using notations $P \models cfs$ and $P \models cfs\Box$. In more detail:

- $P \models cfs$ means that program *P* *may* transform each configuration within *cfs* into the subsequent one, if any (word *may* reflects the fact that programs can be non-deterministic in this case study).
  Thus, *execl* formalizes the notion of a *small-step* program execution.

- $P \models cfs\Box$ reinforces $P \models cfs$ by additionally requiring that *cfs* be nonempty, the initial program counter be zero (viz. execution starts from the first instruction in *P*), and the final program counter falls outside *P* (viz. execution terminates).
  Thus, *execl-all* formalizes the notion of a *complete small-step* program execution, so that assumptions $acomp\ a \models cfs\Box$, $bcomp\ x \models cfs\Box$, $ccomp\ c \models cfs\Box$ will be used in the compiler correctness proofs for arithmetic/boolean expressions and commands.

Moreover, predicates *apred*, *bpred*, and *cpred* are defined to capture the link between the initial and the final configuration upon the execution of an arithmetic expression, a boolean expression, and a whole program, respectively, and abbreviation *off* is introduced as a commodity to shorten the subsequent formal text.

fun *execl* :: *instr list* ⇒ *config list* ⇒ *bool* (**infix** ‹⊨› *55*) **where**

$P \models cf \# cf' \# cfs = (P \vdash cf \rightarrow cf' \wedge P \models cf' \# cfs)$ |
$P \models \text{-} = True$

definition *execl-all* :: *instr list* ⇒ *config list* ⇒ *bool* (‹(-/ ⊨/ -□)› *55*) **where**
$P \models cfs\Box \equiv P \models cfs \wedge cfs \neq [] \wedge$
$fst\ (cfs\ !\ 0) = 0 \wedge fst\ (cfs\ !\ (length\ cfs - 1)) \notin \{0..<size\ P\}$

definition *apred* :: *aexp* ⇒ *config* ⇒ *config* ⇒ *bool* **where**
$apred \equiv \lambda a\ (pc, s, stk)\ (pc', s', stk').$
$pc' = pc + size\ (acomp\ a) \wedge s' = s \wedge stk' = aval\ a\ s \# stk$

definition *bpred* :: *bexp* × *bool* × *int* ⇒ *config* ⇒ *config* ⇒ *bool* **where**
$bpred \equiv \lambda(b, f, i)\ (pc, s, stk)\ (pc', s', stk').$
$pc' = pc + size\ (bcomp\ (b, f, i)) + (if\ bval\ b\ s = f\ then\ i\ else\ 0) \wedge$
$s' = s \wedge stk' = stk$

definition *cpred* :: *com* ⇒ *config* ⇒ *config* ⇒ *bool* **where**
$cpred \equiv \lambda c\ (pc, s, stk)\ (pc', s', stk').$
$pc' = pc + size\ (ccomp\ c) \wedge (c, s) \Rightarrow s' \wedge stk' = stk$

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

Next, some lemmas about *execl* and *execl-all* are proven. In more detail, given a program P and a list of configurations *cfs* such that $P \models cfs$:

- Lemma *execl-next* states that for any configuration in *cfs* but the last one, the subsequent configuration must result from the execution of the referenced instruction of P in that configuration.
  Thus, *execl-next* permits to reproduce the execution of a single instruction.

- Lemma *execl-last* states that a configuration in *cfs* whose program counter falls outside P must be the last one in *cfs*.
  Thus, *execl-last* permits to infer the completion of program execution.

- Lemma *execl-drop* states that $P \models drop\ n\ cfs$ for any natural number n, and will be used to prove compiler correctness for loops by induction over the length of the list of configurations *cfs*.

Furthermore, some other lemmas enabling to prove compiler correctness automatically for constructors *N*, *V* (arithmetic expressions), *Bc* (boolean expressions) and *SKIP* (commands) are also proven.

lemma *iexec-offset-aux*:
$(i :: int) + 1 - j = i - j + 1 \wedge i + j - k + 1 = i - k + j + 1$

by *simp*

lemma *iexec-offset* [*intro*]:
(*ins*, *pc*, *s*, *stk*) ↦ (*pc*′, *s*′, *stk*′) ⟹
(*ins*, *pc* − *i*, *s*, *stk*) ↦ (*pc*′ − *i*, *s*′, *stk*′)
by (*erule iexec.cases*, *auto simp*: *iexec-offset-aux*)

lemma *execl-next*:
⟦*P* ⊨ *cfs*; *k* < *length cfs*; *k* ≠ *length cfs* − *1*⟧ ⟹
(*P* !! *fst* (*cfs* ! *k*), *cfs* ! *k*) ↦ *cfs* ! *Suc k*
by (*induction cfs arbitrary*: *k rule*: *execl.induct*, *auto*
simp: *nth-Cons exec1-def split*: *nat.split*)

lemma *execl-last*:
⟦*P* ⊨ *cfs*; *k* < *length cfs*; *fst* (*cfs* ! *k*) ∉ {*0*..<*size P*}⟧ ⟹
length cfs − *1* = *k*
by (*induction cfs arbitrary*: *k rule*: *execl.induct*, *auto*
simp: *nth-Cons exec1-def split*: *nat.split-asm*)

lemma *execl-drop*:
P ⊨ *cfs* ⟹ *P* ⊨ *drop n cfs*
by (*induction cfs arbitrary*: *n rule*: *execl.induct*,
simp-all add: *drop-Cons split*: *nat.split*)

lemma *execl-all-N* [*simplified*, *intro*]:
[*LOADI i*] ⊨ *cfs*□ ⟹ *apred* (*N i*) (*cfs* ! *0*) (*cfs* ! (*length cfs* − *1*))
by (*clarsimp simp*: *execl-all-def apred-def*, *cases cfs* ! *0*,
subgoal-tac length cfs − *1* = *1*, *frule-tac* [!] *execl-next*,
((*rule ccontr*)?, *fastforce*, (*rule execl-last*)?)+)

lemma *execl-all-V* [*simplified*, *intro*]:
[*LOAD x*] ⊨ *cfs*□ ⟹ *apred* (*V x*) (*cfs* ! *0*) (*cfs* ! (*length cfs* − *1*))
by (*clarsimp simp*: *execl-all-def apred-def*, *cases cfs* ! *0*,
subgoal-tac length cfs − *1* = *1*, *frule-tac* [!] *execl-next*,
((*rule ccontr*)?, *fastforce*, (*rule execl-last*)?)+)

lemma *execl-all-Bc* [*simplified*, *intro*]:
⟦*if v* = *f then* [*JMP i*] *else* [] ⊨ *cfs*□; *0* ≤ *i*⟧ ⟹
bpred (*Bc v*, *f*, *i*) (*cfs* ! *0*) (*cfs* ! (*length cfs* − *1*))
by (*clarsimp simp*: *execl-all-def bpred-def split*: *if-split-asm*, *cases cfs* ! *0*,
subgoal-tac length cfs − *1* = *1*, *frule-tac* [*1*−*2*] *execl-next*,
((*rule ccontr*)?, *force*, (*rule execl-last*)?)+, *rule execl.cases* [*of* ([], *cfs*)],
auto simp: *exec1-def*)

lemma *execl-all-SKIP* [*simplified*, *intro*]:
[] ⊨ *cfs*□ ⟹ *cpred SKIP* (*cfs* ! *0*) (*cfs* ! (*length cfs* − *1*))
by (*rule execl.cases* [*of* ([], *cfs*)], *auto simp*: *execl-all-def exec1-def cpred-def*)

Next, lemma *execl-all-sub* is proven. It states that, if P @ P' x @ $P'' \models$ *cfs*□, configuration *cf* within *cfs* refers to the start of program P' x, and the initial and the final configuration in every complete execution of P' x satisfy predicate Q x, then there exists a configuration *cf'* in *cfs* such that *cf* and *cf'* satisfy Q x.
Thus, this lemma permits to reproduce the execution of a subprogram, particularly:

- a compiled arithmetic expression a, where Q = *apred* and $x = a$,
- a compiled boolean expression b, where Q = *bpred* and $x = (b, f, i)$ (given a flag f and a jump offset i), and
- a compiled command c, where Q = *cpred* and $x = c$.

Furthermore, lemma *execl-all-sub2* is derived from *execl-all-sub* to enable a shorter symbolical execution of two consecutive subprograms.

lemma *execl-sub-aux*:
⟦⋀*m n*. ∀ *k* ∈ {*m*..<*n*}. *Q P'* (((*pc*, *s*, *stk*) # *cfs*) ! *k*) ⟹ *P'* ⊨
  map (*off P*) (*case m of 0* ⇒ (*pc*, *s*, *stk*) # *take n cfs* | *Suc m* ⇒ *F cfs m n*);
  ∀ *k* ∈ {*m*..<*n*+*m*+*length cfs'*}. *Q P'* ((*cfs'* @ (*pc*, *s*, *stk*) # *cfs*) ! (*k*−*m*))⟧ ⟹
 P' ⊨ (*pc* − *size P*, *s*, *stk*) # *map* (*off P*) (*take n cfs*)
 (**is** ⟦⋀- -. ∀ *k* ∈ -. *Q P'* (*?F k*) ⟹ -; ∀ *k* ∈ *?A*. *Q P'* (*?G k*)⟧ ⟹ -)
by (*subgoal-tac* ∀ *k* ∈ {*0*..<*n*}. *Q P'* (*?F k*), *fastforce*, *subgoal-tac*
∀ *k* ∈ {*0*..<*n*}. *k* + *m* + *length cfs'* ∈ *?A* ∧ *?F k* = *?G* (*k* + *m* + *length cfs'*),
fastforce, *simp add*: *nth-append*)

lemma *execl-sub*:
⟦*P* @ *P'* @ *P''* ⊨ *cfs*; ∀ *k* ∈ {*m*..<*n*}.
   size P ≤ *fst* (*cfs* ! *k*) ∧ *fst* (*cfs* ! *k*) − *size P* < *size P'*⟧ ⟹
 P' ⊨ *map* (*off P*) (*drop m* (*take* (*Suc n*) *cfs*))
 (**is** ⟦-; ∀ *k* ∈ -. *?P P' cfs k*⟧ ⟹ *P'* ⊨ *map* - (*?F cfs m* (*Suc n*)))
proof (*induction cfs arbitrary*: *m n rule*: *execl.induct* [*of* - *P'*], *auto*
simp: *take-Cons drop-Cons exec1-def split*: *nat.split*, *force*, *force*, *force*,
erule execl-sub-aux [**where** *m* = *0*], *subst append-Cons* [*of* - []], *simp*,
erule execl-sub-aux [**where** *m* = *Suc 0* **and** *cfs'* = []], *simp*)
 fix *P' pc s stk cfs m n*
 let *?cf* = (*pc*, *s*, *stk*) :: *config*
 assume ⋀*m n*. ∀ *k* ∈ {*m*..<*n*}. *?P P'* (*?cf* # *cfs*) *k* ⟹ *P'* ⊨
  map (*off P*) (*case m of 0* ⇒ *?cf* # *take n cfs* | *Suc m* ⇒ *?F cfs m n*)
 moreover assume ∀ *k* ∈ {*Suc* (*Suc m*)..<*Suc n*}. *?P P' cfs* (*k* − *Suc* (*Suc 0*))
 hence ∀ *k* ∈ {*Suc m*..<*n*}. *?P P'* (*?cf* # *cfs*) *k*
  by *force*
 ultimately show *P'* ⊨ *map* (*off P*) (*?F cfs m n*)
  by *fastforce*
qed

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 −
let *?P* = λ*k*. *size P* ≤ *fst* (*cfs* ! *k*) ∧ *fst* (*cfs* ! *k*) − *size P* < *size* (*P′ x*)
let *?A* = {*k′*. *k′* ∈ {*k*..<*length cfs*} ∧ ¬ *?P k′*}
have *E*: *Min ?A* ∈ *?A*
using *A* **and** *B* **by** (*rule-tac Min-in*, *simp-all add*: *execl-all-def*,
rule-tac exI [*of* - *length cfs* − *1*], *auto*)
hence *map* (*off P*) (*drop k* (*take* (*Suc* (*Min ?A*)) *cfs*)) ! *0* = *off P* (*cfs* ! *k*)
(**is** *?cfs* ! - = -)
by *auto*
moreover have *length cfs* ≤ *Suc* (*Min ?A*) ⟶ *Min ?A* = *length cfs* − *1*
using *E* **by** *auto*
with *A* **and** *B* **and** *E* **have** *F*: *?cfs* ! (*length ?cfs* − *1*) = *off P* (*cfs* ! *Min ?A*)
by (*subst nth-map*, *auto simp*: *min-def execl-all-def*, *arith*)
hence *?cfs* ≠ [] ∧ *fst* (*?cfs* ! (*length ?cfs* − *1*)) ∉ {*0*..<*size* (*P′ x*)}
using *E* **by** (*auto simp*: *min-def*)
moreover have ¬ (∃ *k′*. *k′* ∈ {*k′*. *k′* ∈ {*k*..<*Min ?A*} ∧ ¬ *?P k′*})
by (*rule notI*, *erule exE*, *frule rev-subsetD* [*of* - - *?A*], *rule subsetI*,
insert E, *simp*, *subgoal-tac finite ?A*, *drule Min-le*, *force*+)
hence *P′ x* ⊨ *?cfs*
using *A* **by** (*subst* (*asm*) *execl-all-def*, *rule-tac execl-sub*, *blast*+)
ultimately have *Q x* (*?cfs* ! *0*) (*?cfs* ! (*length ?cfs* − *1*))
using *C* **and** *D* **by** (*auto simp*: *execl-all-def*)
thus *?thesis*
using *E* **and** *F* **by** (*rule-tac exI* [*of* - *Min ?A*], *auto*)
qed

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*)

by (*subgoal-tac* [] @ *?P* ⊨ *cfs*□, *drule execl-all-sub* [**where** $k = 0$ **and** $Q = ?Q$],
insert A B, (*clarsimp simp*: *execl-all-def*)+, *insert A C D*, *drule execl-all-sub*
[**where** $Q = ?Q'$], *simp*+, *clarify*, *rule exI*, *rule conjI*, *simp*, *rule exI*, *auto*)

2.2 Main theorem

It is time to prove compiler correctness. First, lemmas *acomp-acomp*, *bcomp-bcomp* are derived from *execl-all-sub2* to reproduce the execution of two consecutive compiled arithmetic expressions (possibly generated by both *acomp* and *bcomp*) and boolean expressions (possibly generated by *bcomp*), respectively. Subsequently, the correctness of *acomp* and *bcomp* is proven in lemmas *acomp-correct*, *bcomp-correct*.

lemma *acomp-acomp*:
⟦*acomp* a_1 @ *acomp* a_2 @ *P* ⊨ *cfs*□;
⋀*cfs*. *acomp* a_1 ⊨ *cfs*□ ⟹ *apred* a_1 (*cfs* ! *0*) (*cfs* ! (*length cfs* − *1*));
⋀*cfs*. *acomp* a_2 ⊨ *cfs*□ ⟹ *apred* a_2 (*cfs* ! *0*) (*cfs* ! (*length cfs* − *1*))⟧ ⟹
case cfs ! *0 of* (*pc*, *s*, *stk*) ⇒ *pc* = *0* ∧ (∃ *k* < *length cfs*. *cfs* ! *k* =
(*size* (*acomp* a_1 @ *acomp* a_2), *s*, *aval* a_2 *s* # *aval* a_1 *s* # *stk*))
by (*drule execl-all-sub2* [**where** $I = \lambda s.\ 0$ **and** $I' = \lambda s.\ 0$ **and** $Q = \lambda s\ s'.\ s' = s$
and $Q' = \lambda s\ s'.\ s' = s$ **and** $F = \lambda s\ stk.$ *aval* a_1 *s* # *stk*
and $F' = \lambda s\ stk.$ *aval* a_2 *s* # *stk*], *auto simp*: *apred-def*)

lemma *bcomp-bcomp*:
⟦*bcomp* (b_1, f_1, i_1) @ *bcomp* (b_2, f_2, i_2) ⊨ *cfs*□;
⋀*cfs*. *bcomp* (b_1, f_1, i_1) ⊨ *cfs*□ ⟹
bpred (b_1, f_1, i_1) (*cfs* ! *0*) (*cfs* ! (*length cfs* − *1*));
⋀*cfs*. *bcomp* (b_2, f_2, i_2) ⊨ *cfs*□ ⟹
bpred (b_2, f_2, i_2) (*cfs* ! *0*) (*cfs* ! (*length cfs* − *1*))⟧ ⟹
case cfs ! *0 of* (*pc*, *s*, *stk*) ⇒ *pc* = *0* ∧ (*bval* b_1 *s* ≠ f_1 ⟶
(∃ *k* < *length cfs*. *cfs* ! *k* = (*size* (*bcomp* (b_1, f_1, i_1) @ *bcomp* (b_2, f_2, i_2)) +
(*if bval* b_2 *s* = f_2 *then* i_2 *else 0*), *s*, *stk*)))
by (*clarify*, *rule conjI*, *simp add*: *execl-all-def*, *rule impI*, *subst* (*asm*) *append-Nil2*
[*symmetric*], *drule execl-all-sub2* [**where** $I = \lambda s.$ *if bval* b_1 *s* = f_1 *then* i_1 *else 0*
and $I' = \lambda s.$ *if bval* b_2 *s* = f_2 *then* i_2 *else 0* **and** $Q = \lambda s\ s'.\ s' = s$
and $Q' = \lambda s\ s'.\ s' = s$ **and** $F = \lambda s\ stk.\ stk$ **and** $F' = \lambda s\ stk.\ stk$],
auto simp: *bpred-def*)

lemma *acomp-correct* [*simplified*, *intro*]:
acomp a ⊨ *cfs*□ ⟹ *apred a* (*cfs* ! *0*) (*cfs* ! (*length cfs* − *1*))
proof (*induction a arbitrary*: *cfs*, *simp-all*, *frule-tac* [*3*] *acomp-acomp*, *auto*)
fix a_1 a_2 *cfs s stk k*
assume *A*: *acomp* a_1 @ *acomp* a_2 @ [*ADD*] ⊨ *cfs*□
(**is** *?ca*$_1$ @ *?ca*$_2$ @ *?i* ⊨ -□)
assume *B*: *k* < *length cfs* **and**
C: *cfs* ! *k* = (*size ?ca*$_1$ + *size ?ca*$_2$, *s*, *aval* a_2 *s* # *aval* a_1 *s* # *stk*)
hence *cfs* ! *Suc k* = (*size* (*?ca*$_1$ @ *?ca*$_2$ @ *?i*), *s*, *aval* (*Plus* a_1 a_2) *s* # *stk*)
using *A* **by** (*insert execl-next* [*of ?ca*$_1$ @ *?ca*$_2$ @ *?i cfs k*],

simp add: *execl-all-def*, *drule-tac impI*, *auto*)
thus *apred* (*Plus* a_1 a_2) (*0*, *s*, *stk*) (*cfs* ! (*length cfs* − *Suc 0*))
using *A* **and** *B* **and** *C* **by** (*insert execl-last* [*of ?ca*$_1$ @ *?ca*$_2$ @ *?i cfs Suc k*],
simp add: *execl-all-def apred-def*, *drule-tac impI*, *auto*)
qed

lemma *bcomp-correct* [*simplified*, *intro*]:
⟦*bcomp x* ⊨ *cfs*□; *0* ≤ *snd* (*snd x*)⟧ ⟹ *bpred x* (*cfs* ! *0*) (*cfs* ! (*length cfs* − *1*))
proof (*induction x arbitrary*: *cfs rule*: *bcomp.induct*, *simp-all add*: *Let-def*,
frule-tac [*4*] *acomp-acomp*, *frule-tac* [*3*] *bcomp-bcomp*, *auto*, *force simp*: *bpred-def*)
fix b_1 b_2 *f i cfs s stk*
assume *A*: *bcomp* (b_1, *False*, *size* (*bcomp* (b_2, *f*, *i*)) + (*if f then 0 else i*)) @
bcomp (b_2, *f*, *i*) ⊨ *cfs*□
(**is** *bcomp* (-, -, *?n* + *?i*) @ *?cb* ⊨ -□)
moreover assume *B*: *cfs* ! *0* = (*0*, *s*, *stk*) **and**
⋀*cb cfs*. ⟦*cb* = *?cb*; *bcomp* (b_1, *False*, *?n* + *?i*) ⊨ *cfs*□⟧ ⟹
bpred (b_1, *False*, *?n* + *?i*) (*cfs* ! *0*) (*cfs* ! (*length cfs* − *Suc 0*))
ultimately have ∃ *k* < *length cfs*. *bpred* (b_1, *False*, *?n* + *?i*)
(*off* [] (*cfs* ! *0*)) (*off* [] (*cfs* ! *k*))
by (*rule-tac execl-all-sub*, *auto simp*: *execl-all-def*)
moreover assume *C*: ¬ *bval* b_1 *s*
ultimately obtain *k* **where** *D*: *k* < *length cfs* **and**
E: *cfs* ! *k* = (*size* (*bcomp* (b_1, *False*, *?n* + *?i*)) + *?n* + *?i*, *s*, *stk*)
using *B* **by** (*auto simp*: *bpred-def*)
assume *0* ≤ *i*
thus *bpred* (*And* b_1 b_2, *f*, *i*) (*0*, *s*, *stk*) (*cfs* ! (*length cfs* − *Suc 0*))
using *A* **and** *C* **and** *D* **and** *E* **by** (*insert execl-last*, *auto simp*:
execl-all-def bpred-def Let-def)
next
fix b_1 b_2 *f i cfs s stk k*
assume *A*: *bcomp* (b_1, *False*, *size* (*bcomp* (b_2, *f*, *i*)) + (*if f then 0 else i*)) @
bcomp (b_2, *f*, *i*) ⊨ *cfs*□
(**is** *?cb*$_1$ @ *?cb*$_2$ ⊨ -□)
assume *k* < *length cfs* **and** *0* ≤ *i* **and** *bval* b_1 *s* **and**
cfs ! *k* = (*size ?cb*$_1$ + *size ?cb*$_2$ + (*if bval* b_2 *s* = *f then i else 0*), *s*, *stk*)
thus *bpred* (*And* b_1 b_2, *f*, *i*) (*0*, *s*, *stk*) (*cfs* ! (*length cfs* − *Suc 0*))
using *A* **by** (*insert execl-last*, *auto simp*: *execl-all-def bpred-def Let-def*)
next
fix a_1 a_2 *f i cfs s stk k*
assume *A*: *acomp* a_1 @ *acomp* a_2 @
(*if f then* [*JMPLESS i*] *else* [*JMPGE i*]) ⊨ *cfs*□
(**is** *?ca*$_1$ @ *?ca*$_2$ @ *?i* ⊨ -□)
assume *B*: *k* < *length cfs* **and**
C: *cfs* ! *k* = (*size ?ca*$_1$ + *size ?ca*$_2$, *s*, *aval* a_2 *s* # *aval* a_1 *s* # *stk*)
hence *D*: *cfs* ! *Suc k* =
(*size* (*?ca*$_1$ @ *?ca*$_2$ @ *?i*) + (*if bval* (*Less* a_1 a_2) *s* = *f then i else 0*), *s*, *stk*)
using *A* **by** (*insert execl-next* [*of ?ca*$_1$ @ *?ca*$_2$ @ *?i cfs k*],
simp add: *execl-all-def*, *drule-tac impI*, *auto split*: *if-split-asm*)
assume *0* ≤ *i*

with *A* **and** *B* **and** *C* **and** *D* **have** *length cfs* − *1* = *Suc k*
by (*rule-tac execl-last*, *auto simp*: *execl-all-def*, *simp split*: *if-split-asm*)
thus *bpred* (*Less* a_1 a_2, *f*, *i*) (*0*, *s*, *stk*) (*cfs* ! (*length cfs* − *Suc 0*))
using *D* **by** (*simp add*: *bpred-def*)
qed

Next, lemmas *bcomp-ccomp*, *ccomp-ccomp* are derived to reproduce the execution of a compiled boolean expression followed by a compiled command and of two consecutive compiled commands, respectively (possibly generated by *ccomp*). Then, compiler correctness for loops and for all commands is proven in lemmas *while-correct* and *ccomp-correct*, respectively by induction over the length of the list of configurations and by structural induction over commands.

lemma *bcomp-ccomp*:
⟦*bcomp* (*b*, *f*, *i*) @ *ccomp c* @ *P* ⊨ *cfs*□; $0 \le i$;
⋀*cfs*. *ccomp c* ⊨ *cfs*□ ⟹ *cpred c* (*cfs* ! *0*) (*cfs* ! (*length cfs* − *1*))⟧ ⟹
case cfs ! *0 of* (*pc*, *s*, *stk*) ⇒ *pc* = *0* ∧ (*bval b s* ≠ *f* ⟶
(∃ *k* < *length cfs*. *case cfs* ! *k of* (*pc*′, *s*′, *stk*′) ⇒
pc′ = *size* (*bcomp* (*b*, *f*, *i*) @ *ccomp c*) ∧ (*c*, *s*) ⇒ *s*′ ∧ *stk*′ = *stk*))
by (*clarify*, *rule conjI*, *simp add*: *execl-all-def*, *rule impI*, *drule execl-all-sub2*
[**where** *I* = λ*s*. *if bval b s* = *f then i else 0* **and** *I*′ = λ*s*. *0*
and *Q* = λ*s s*′. *s*′ = *s* **and** *Q*′ = λ*s s*′. (*c*, *s*) ⇒ *s*′ **and** *F* = λ*s stk*. *stk*
and *F*′ = λ*s stk*. *stk*], *insert bcomp-correct* [*of* (*b*, *f*, *i*)],
auto simp: *bpred-def cpred-def*)

lemma *ccomp-ccomp*:
⟦*ccomp* c_1 @ *ccomp* c_2 ⊨ *cfs*□;
⋀*cfs*. *ccomp* c_1 ⊨ *cfs*□ ⟹ *cpred* c_1 (*cfs* ! *0*) (*cfs* ! (*length cfs* − *1*));
⋀*cfs*. *ccomp* c_2 ⊨ *cfs*□ ⟹ *cpred* c_2 (*cfs* ! *0*) (*cfs* ! (*length cfs* − *1*))⟧ ⟹
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*)
by (*subst* (*asm*) *append-Nil2* [*symmetric*], *drule execl-all-sub2* [**where** *I* = λ*s*. *0*
and *I*′ = λ*s*. *0* **and** *Q* = λ*s s*′. (c_1, *s*) ⇒ *s*′ **and** *Q*′ = λ*s s*′. (c_2, *s*) ⇒ *s*′
and *F* = λ*s stk*. *stk* **and** *F*′ = λ*s stk*. *stk*], *auto simp*: *cpred-def*, *force*)

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 (*induction cfs rule*: *length-induct*, *frule bcomp-ccomp*, *auto*)
fix *cfs s stk*
assume *A*: *?cb* @ *?cc* @ [*JMP* (− *size ?cb* − *size ?cc* − *1*)] ⊨ *cfs*□

hence ∃ *k* < *length cfs*. *bpred* (*b*, *False*, *size* (*ccomp c*) + *1*)
(*off* [] (*cfs* ! *0*)) (*off* [] (*cfs* ! *k*))
by (*rule-tac execl-all-sub*, *auto simp*: *execl-all-def*)
moreover assume *B*: ¬ *bval b s* **and** *cfs* ! *0* = (*0*, *s*, *stk*)
ultimately obtain *k* **where** *k* < *length cfs* **and** *cfs* ! *k* = (*?n*, *s*, *stk*)
by (*auto simp*: *bpred-def*)
thus *cpred* (*WHILE b DO c*) (*0*, *s*, *stk*) (*cfs* ! (*length cfs* − *Suc 0*))
using *A* **and** *B* **by** (*insert execl-last*, *auto simp*: *execl-all-def cpred-def Let-def*)
next
fix *cfs s s′ stk k*
assume *A*: *?cb* @ *?cc* @ [*JMP* (− *size ?cb* − *size ?cc* − *1*)] ⊨ *cfs*□
(**is** *?P* ⊨ -□)
assume *B*: *k* < *length cfs* **and** *cfs* ! *k* = (*size ?cb* + *size ?cc*, *s′*, *stk*)
moreover from *this* **have** *C*: *k* ≠ *length cfs* − *1*
using *A* **by** (*rule-tac notI*, *simp add*: *execl-all-def*)
ultimately have *D*: *cfs* ! *Suc k* = (*0*, *s′*, *stk*)
using *A* **by** (*insert execl-next* [*of ?P cfs k*], *auto simp*: *execl-all-def*)
moreover have *E*: *Suc k* + (*length* (*drop* (*Suc k*) *cfs*) − *1*) = *length cfs* − *1*
(**is** *Suc k* + (*length ?cfs* − *1*) = -)
using *B* **and** *C* **by** *simp*
ultimately have *?P* ⊨ *?cfs*□
using *A* **and** *B* **and** *C* **by** (*auto simp*: *execl-all-def intro*: *execl-drop*)
moreover assume ∀ *cfs′*. *length cfs′* < *length cfs* ⟶ *?P* ⊨ *cfs′*□ ⟶ *?Q cfs′*
hence *length ?cfs* < *length cfs* ⟶ *?P* ⊨ *?cfs*□ ⟶ *?Q ?cfs* **..**
ultimately have *cpred* (*WHILE b DO c*) (*cfs* ! *Suc k*) (*cfs* ! (*length cfs* − *1*))
using *B* **and** *C* **and** *E* **by** *simp*
moreover assume *bval b s* **and** (*c*, *s*) ⇒ *s′*
ultimately show *cpred* (*WHILE b DO c*) (*0*, *s*, *stk*) (*cfs* ! (*length cfs* − *Suc 0*))
using *D* **by** (*auto simp*: *cpred-def*)
qed

lemma *ccomp-correct*:
ccomp c ⊨ *cfs*□ ⟹ *cpred c* (*cfs* ! *0*) (*cfs* ! (*length cfs* − *1*))
proof (*induction c arbitrary*: *cfs*, *simp-all add*: *Let-def*, *frule-tac* [*4*] *bcomp-ccomp*,
frule-tac [*3*] *ccomp-ccomp*, *auto*)
fix *a x cfs*
assume *A*: *acomp a* @ [*STORE x*] ⊨ *cfs*□
hence ∃ *k* < *length cfs*. *apred a* (*off* [] (*cfs* ! *0*)) (*off* [] (*cfs* ! *k*))
by (*rule-tac execl-all-sub*, *auto simp*: *execl-all-def*)
moreover obtain *s stk* **where** *B*: *cfs* ! *0* = (*0*, *s*, *stk*)
using *A* **by** (*cases cfs* ! *0*, *simp add*: *execl-all-def*)
ultimately obtain *k* **where** *C*: *k* < *length cfs* **and**
D: *cfs* ! *k* = (*size* (*acomp a*), *s*, *aval a s* # *stk*)
by (*auto simp*: *apred-def*)
hence *cfs* ! *Suc k* = (*size* (*acomp a*) + *1*, *s*(*x* := *aval a s*), *stk*)
using *A* **by** (*insert execl-next* [*of acomp a* @ [*STORE x*] *cfs k*],
simp add: *execl-all-def*, *drule-tac impI*, *auto*)
thus *cpred* (*x* ::= *a*) (*cfs* ! *0*) (*cfs* ! (*length cfs* − *Suc 0*))
using *A* **and** *B* **and** *C* **and** *D* **by** (*insert execl-last* [*of acomp a* @ [*STORE x*]

cfs Suc k], *simp add*: *execl-all-def cpred-def*, *drule-tac impI*, *auto*)
next
fix c_1 c_2 *cfs s s′ t stk k*
assume *ccomp* c_1 @ *ccomp* c_2 ⊨ *cfs*□ **and** *k* < *length cfs* **and**
cfs ! *k* = (*size* (*ccomp* c_1) + *size* (*ccomp* c_2), *s′*, *stk*)
moreover assume (c_1, *s*) ⇒ *t* **and** (c_2, *t*) ⇒ *s′*
ultimately show *cpred* (c_1;; c_2) (*0*, *s*, *stk*) (*cfs* ! (*length cfs* − *Suc 0*))
by (*insert execl-last*, *auto simp*: *execl-all-def cpred-def*)
next
fix *b* c_1 c_2 *cfs s stk*
assume *A*: *bcomp* (*b*, *False*, *size* (*ccomp* c_1) + *1*) @ *ccomp* c_1 @
JMP (*size* (*ccomp* c_2)) # *ccomp* c_2 ⊨ *cfs*□
(**is** *bcomp ?x* @ *?cc*$_1$ @ - # *?cc*$_2$ ⊨ -□)
let *?P* = *bcomp ?x* @ *?cc*$_1$ @ [*JMP* (*size ?cc*$_2$)]
have ∃ *k* < *length cfs*. *bpred ?x* (*off* [] (*cfs* ! *0*)) (*off* [] (*cfs* ! *k*))
using *A* **by** (*rule-tac execl-all-sub*, *auto simp*: *execl-all-def*)
moreover assume *B*: ¬ *bval b s* **and** *cfs* ! *0* = (*0*, *s*, *stk*)
ultimately obtain *k* **where** *C*: *k* < *length cfs* **and** *D*: *cfs* ! *k* = (*size ?P*, *s*, *stk*)
by (*force simp*: *bpred-def*)
assume ⋀*cfs*. *?cc*$_2$ ⊨ *cfs*□ ⟹ *cpred* c_2 (*cfs* ! *0*) (*cfs* ! (*length cfs* − *Suc 0*))
hence ∃ *k′* < *length cfs*. *cpred* c_2 (*off ?P* (*cfs* ! *k*)) (*off ?P* (*cfs* ! *k′*))
using *A* **and** *C* **and** *D* **by** (*rule-tac execl-all-sub* [**where** *P″* = []], *auto*)
then obtain *k′* **where** *k′* < *length cfs* **and** *case cfs* ! *k′* *of* (*pc′*, *s′*, *stk′*) ⇒
pc′ = *size* (*?P* @ *?cc*$_2$) ∧ (c_2, *s*) ⇒ *s′* ∧ *stk′* = *stk*
using *D* **by** (*fastforce simp*: *cpred-def*)
thus *cpred* (*IF b THEN* c_1 *ELSE* c_2) (*0*, *s*, *stk*) (*cfs* ! (*length cfs* − *Suc 0*))
using *A* **and** *B* **by** (*insert execl-last*, *auto simp*: *execl-all-def cpred-def Let-def*)
next
fix *b* c_1 c_2 *cfs s s′ stk k*
assume *A*: *bcomp* (*b*, *False*, *size* (*ccomp* c_1) + *1*) @ *ccomp* c_1 @
JMP (*size* (*ccomp* c_2)) # *ccomp* c_2 ⊨ *cfs*□
(**is** *?cb* @ *?cc*$_1$ @ *?i* # *?cc*$_2$ ⊨ -□)
assume *B*: *k* < *length cfs* **and** *C*: *cfs* ! *k* = (*size ?cb* + *size ?cc*$_1$, *s′*, *stk*)
hence *D*: *cfs* ! *Suc k* = (*size* (*?cb* @ *?cc*$_1$ @ *?i* # *?cc*$_2$), *s′*, *stk*)
(**is** - = (*size ?P*, -, -))
using *A* **by** (*insert execl-next* [*of ?P cfs k*], *simp add*: *execl-all-def*,
drule-tac impI, *auto*)
assume *bval b s* **and** (c_1, *s*) ⇒ *s′*
thus *cpred* (*IF b THEN* c_1 *ELSE* c_2) (*0*, *s*, *stk*) (*cfs* ! (*length cfs* − *Suc 0*))
using *A* **and** *B* **and** *C* **and** *D* **by** (*insert execl-last* [*of ?P cfs Suc k*],
simp add: *execl-all-def cpred-def Let-def*, *drule-tac impI*, *auto*)
next
fix c_1 c_2 *cfs*
assume *A*: *JMPND* (*size* (*ccomp* c_1) + *1*) # *ccomp* c_1 @
JMP (*size* (*ccomp* c_2)) # *ccomp* c_2 ⊨ *cfs*□
(**is** *JMPND* (*?n*$_1$ + *1*) # *?cc*$_1$ @ *JMP ?n*$_2$ # *?cc*$_2$ ⊨ -□)
let *?P* = *JMPND* (*?n*$_1$ + *1*) # *?cc*$_1$ @ [*JMP ?n*$_2$]
assume
B: ⋀*cfs*. *?cc*$_1$ ⊨ *cfs*□ ⟹ *cpred* c_1 (*cfs* ! *0*) (*cfs* ! (*length cfs* − *Suc 0*)) **and**

  C: ⋀*cfs*. *?cc*$_2$ ⊨ *cfs*□ ⟹ *cpred* c_2 (*cfs* ! *0*) (*cfs* ! (*length cfs* − *Suc 0*))
 from *A* **obtain** *s stk* **where** *D*: *cfs* ! *0* = (*0*, *s*, *stk*)
  by (*cases cfs* ! *0*, *simp add*: *execl-all-def*)
 with *A* **have** *cfs* ! *1* = (*1*, *s*, *stk*) ∨ *cfs* ! *1* = (*?n*$_1$ + *2*, *s*, *stk*)
  by (*insert execl-next* [*of ?P* @ *?cc*$_2$ *cfs 0*], *simp add*: *execl-all-def*,
   drule-tac impI, *auto*)
 moreover {
  assume *E*: *cfs* ! *1* = (*1*, *s*, *stk*)
  hence ∃ *k* < *length cfs*. *cpred* c_1 (*off* [*hd ?P*] (*cfs* ! *1*)) (*off* [*hd ?P*] (*cfs* ! *k*))
   using *A* **and** *B* **by** (*rule-tac execl-all-sub*, *auto simp*: *execl-all-def*)
  then obtain *k* **where** *k* < *length cfs* **and** *case cfs* ! *k of* (*pc*′, *s*′, *stk*′) ⇒
   pc′ = *?n*$_1$ + *1* ∧ (c_1, *s*) ⇒ *s*′ ∧ *stk*′ = *stk*
   using *E* **by** (*fastforce simp*: *cpred-def*)
  moreover from *this* **have** *case cfs* ! *Suc k of* (*pc*′, *s*′, *stk*′) ⇒
   pc′ = *?n*$_1$ + *?n*$_2$ + *2* ∧ (c_1, *s*) ⇒ *s*′ ∧ *stk*′ = *stk*
   using *A* **by** (*insert execl-next* [*of ?P* @ *?cc*$_2$ *cfs k*], *simp add*: *execl-all-def*,
    drule-tac impI, *auto*)
  ultimately have *cpred* (c_1 *OR* c_2) (*cfs* ! *0*) (*cfs* ! (*length cfs* − *Suc 0*))
   using *A* **and** *D* **by** (*insert execl-last* [*of ?P* @ *?cc*$_2$ *cfs Suc k*],
    simp add: *execl-all-def cpred-def split-def Let-def*, *drule-tac impI*, *auto*)
 }
 moreover {
  assume *E*: *cfs* ! *1* = (*?n*$_1$ + *2*, *s*, *stk*)
  with *A* **and** *C* **have** ∃ *k*<*length cfs*. *cpred* c_2 (*off ?P* (*cfs*!*1*)) (*off ?P* (*cfs*!*k*))
   by (*rule-tac execl-all-sub* [**where** *P*′′ = []], *auto simp*: *execl-all-def*)
  then obtain *k* **where** *k* < *length cfs* **and** *case cfs* ! *k of* (*pc*′, *s*′, *stk*′) ⇒
   pc′ = *?n*$_1$ + *?n*$_2$ + *2* ∧ (c_2, *s*) ⇒ *s*′ ∧ *stk*′ = *stk*
   using *E* **by** (*fastforce simp*: *cpred-def*)
  with *A* **and** *D* **have** *cpred* (c_1 *OR* c_2) (*cfs* ! *0*) (*cfs* ! (*length cfs* − *Suc 0*))
   by (*insert execl-last*, *auto simp*: *execl-all-def cpred-def Let-def*)
 }
 ultimately show *cpred* (c_1 *OR* c_2) (*cfs* ! *0*) (*cfs* ! (*length cfs* − *Suc 0*)) **..**
qed

Finally, the main compiler correctness theorem, expressed using predicate *exec*, is proven. First, *P* ⊢ *cf* →∗ *cf*′ is shown to imply the existence of a nonempty list of configurations *cfs* such that *P* ⊨ *cfs*, whose initial and final configurations match *cf* and *cf*′, respectively. Then, the main theorem is derived as a straightforward consequence of this lemma and of the previous lemma *ccomp-correct*.

lemma *exec-execl* [*dest*!]:
 P ⊢ *cf* →∗ *cf*′ ⟹ ∃ *cfs*. *P* ⊨ *cfs* ∧ *cfs* ≠ [] ∧ *hd cfs* = *cf* ∧ *last cfs* = *cf*′
by (*erule star.induct*, *force*, *erule exE*, *rule list.exhaust*, *blast*,
 simp del: *last.simps*, *rule exI*, *subst execl.simps*(*1*), *simp*)

theorem *ccomp-exec*:

$ccomp\ c \vdash (0,\ s,\ stk) \rightarrow* (size\ (ccomp\ c),\ s',\ stk') \Longrightarrow (c,\ s) \Rightarrow s' \wedge stk' = stk$
by (*insert ccomp-correct*, *force simp*: *hd-conv-nth last-conv-nth execl-all-def cpred-def*)

end

References

[1] G. Klein. Theory HOL-IMP.Compiler2 (included in the Isabelle2021 distribution). https://isabelle.in.tum.de/website-Isabelle2021/dist/library/HOL/HOL-IMP/Compiler2.html.

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

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

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

[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*, Feb. 2021. https://isabelle.in.tum.de/website-Isabelle2021/dist/Isabelle2021/doc/tutorial.pdf.