Extension of Stateful Intransitive Noninterference with Inputs, Outputs, and Nondeterminism in Language IMP

Pasquale Noce
Senior Staff Firmware 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

In a previous paper of mine, the notion of termination-sensitive information flow security with respect to a level-based interference relation, as studied by Volpano, Smith, and Irvine and formalized in Nipkow and Klein's book on formal programming language semantics (in the version of February 2023), is generalized to the notion of termination-sensitive information flow correctness with respect to an interference function mapping program states to (generally) intransitive interference relations.

This paper extends both the aforesaid information flow correctness criterion and the related static type system to the case of an imperative programming language supporting inputs, outputs, and nondeterminism. Regarding inputs and nondeterminism, Volpano, Smith, and Irvine observe that their soundness theorem no longer holds if their core language is extended with these features. This paper shows that the difficulty can be solved by extending the inductive definition of the language's operational semantics, which enables to apply a suitably extended information flow correctness criterion based on stateful intransitive noninterference, as well as an extended static type system enforcing this criterion, to such an extended programming language. Although an extension with inputs, outputs, and nondeterminism of the didactic programming language IMP employed in the book is used for this purpose, the introduced concepts apply to larger, real-world imperative programming languages as well.

Contents

1 Extension of language IMP with inputs, outputs, and nondeterminism

```
theory Small-Step
  imports
    HOL−IMP.BExp
    HOL−IMP.Star
begin
```

In a previous paper of mine [10], the notion of termination-sensitive information flow security with respect to a level-based interference relation, as studied in [12], [11] and formalized in [8], is generalized to the notion of termination-sensitive information flow correctness with respect to an interference function mapping program states to (generally) intransitive interference relations. Moreover, a static type system is specified and is proven to be capable of enforcing such information flow correctness policies.

The present paper extends both the aforesaid information flow correctness criterion and the related static type system to the case of an imperative programming language supporting inputs, outputs, and nondeterminism. Regarding inputs and nondeterminism, [12], section 7.1, observes that "if we try to extend the core language with a primitive random number generator *rand( )* and allow an assignment such as *z := rand( )* to be well typed when z is low, then the soundness theorem no longer holds", and from this infers that "new security models [...] should be explored as potential notions of type soundness for new type systems that deal with nondeterministic programs". The present paper shows that this difficulty can be solved by extending the inductive definition of the programming language's operational semantics so as to reflect the fact that, even though the input instruction *z := rand( )* may set z to an arbitrary input value, the same program state is produced whenever the input value is the same. As shown in this paper, this enables to apply a suitably extended information flow correctness criterion based on stateful intransitive noninterference, as well as an extended static type system enforcing this criterion, to such an extended programming language.

The didactic imperative programming language IMP employed in [8], extended with an input instruction, an output instruction, and a control structure allowing for nondeterministic choice, will be used for this purpose. Yet, in the same way as in my previous paper [10], the introduced concepts are applicable to larger, real-world imperative programming languages, too, by just affording the additional type system complexity arising from richer language constructs.

For further information about the formal definitions and proofs contained in this paper, refer to Isabelle documentation, particularly [9], [4], [2], [3], and [1].

As mentioned above, the first task to be tackled, which is the subject of this section, consists of extending the original syntax, big-step operational semantics, and small-step operational semantics of language IMP, as formalized in [6], [5], and [7], respectively.

1.1 Extended syntax

The starting point is extending the original syntax of language IMP with the following additional constructs.

- An input instruction *IN* x, which sets variable x to an input value.
- An output instruction *OUT* x, which outputs the current value of variable x.
- A control structure c_1 *OR* c_2, which allows for a nondeterministic choice between commands c_1 and c_2.

```
declare [[syntax-ambiguity-warning = false]]

datatype com =
  SKIP |
  Assign vname aexp  (‹- ::= -› [1000, 61] 70) |
  Input vname  (‹(IN -)› [61] 70) |
  Output vname  (‹(OUT -)› [61] 70) |
  Seq com com  (‹-;;/ -› [61, 61] 70) |
  Or com com  (‹(- OR -)› [61, 61] 70) |
  If bexp com com  (‹(IF -/ THEN -/ ELSE -)› [0, 0, 61] 70) |
  While bexp com  (‹(WHILE -/ DO -)› [0, 61] 70)
```

1.2 Extended big-step semantics

The original big-step semantics of language IMP associates a pair formed by a command and an initial *program execution stage*, consisting of a program state, with a corresponding final program execution stage, consisting of a program state as well. The extended big-step semantics defined here below extends such program execution stage notion by considering, in addition to a program state, the following additional parameters.

- A *stream of input values*, consisting of a function f mapping each pair formed by a variable and a natural number with an integer value, where $f\ x\ n$ is the input value assigned to variable x by an input instruction *IN* x after n previous such assignments to x.

- A *trace of inputs*, consisting of a list *vs* of pairs formed by a variable and an integer value, to which a further element (x, i) is appended as a result of the execution of an input instruction *IN* x, where i is the input value assigned to variable x.

- A *trace of outputs*, consisting of a list *ws* of pairs formed by a variable and an integer value, to which a further element (x, i) is appended as a result of the execution of an output instruction *OUT* x, where i is the current value of variable x being output.

Unlike the other components of a program execution stage, the stream of input values is an *invariant* of the big-step semantics, and then also of the small-step semantics defined subsequently, in that any two program execution stages associated with each other by either semantics share the same stream of input values.

```
type-synonym stream = vname ⇒ nat ⇒ val
type-synonym inputs = (vname × val) list
type-synonym outputs = (vname × val) list
```

type-synonym *stage* = *state* × *stream* × *inputs* × *outputs*

inductive *big-step* :: *com* × *stage* ⇒ *stage* ⇒ *bool*
 (**infix** ‹⇒› *55*) **where**
Skip:
 (*SKIP*, *p*) ⇒ *p* |
Assign:
 (*x* ::= *a*, *s*, *p*) ⇒ (*s*(*x* := *aval a s*), *p*) |
Input:
 n = *length* [*p*←*vs*. *fst p* = *x*] ⟹ (*IN x*, *s*, *f*, *vs*, *ws*) ⇒
  (*s*(*x* := *f x n*), *f*, *vs* @ [(*x*, *f x n*)], *ws*) |
Output:
 (*OUT x*, *s*, *f*, *vs*, *ws*) ⇒ (*s*, *f*, *vs*, *ws* @ [(*x*, *s x*)]) |
Seq:
 ⟦(c_1, p_1) ⇒ p_2; (c_2, p_2) ⇒ p_3⟧ ⟹ (c_1;; c_2, p_1) ⇒ p_3 |
Or1:
 (c_1, *p*) ⇒ *p'* ⟹ (c_1 *OR* c_2, *p*) ⇒ *p'* |
Or2:
 (c_2, *p*) ⇒ *p'* ⟹ (c_1 *OR* c_2, *p*) ⇒ *p'* |
IfTrue:
 ⟦*bval b s*; (c_1, *s*, *p*) ⇒ *p'*⟧ ⟹
  (*IF b THEN* c_1 *ELSE* c_2, *s*, *p*) ⇒ *p'* |
IfFalse:
 ⟦¬ *bval b s*; (c_2, *s*, *p*) ⇒ *p'*⟧ ⟹
  (*IF b THEN* c_1 *ELSE* c_2, *s*, *p*) ⇒ *p'* |
WhileFalse:
 ¬ *bval b s* ⟹ (*WHILE b DO c*, *s*, *p*) ⇒ (*s*, *p*) |
WhileTrue:
 ⟦*bval b* s_1; (*c*, s_1, p_1) ⇒ (s_2, p_2);
  (*WHILE b DO c*, s_2, p_2) ⇒ (s_3, p_3)⟧ ⟹
 (*WHILE b DO c*, s_1, p_1) ⇒ (s_3, p_3)

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

inductive-cases *SkipE* [*elim*!]: (*SKIP*, *p*) ⇒ *p'*

inductive-cases *AssignE* [*elim*!]: (*x* ::= *a*, *p*) ⇒ *p'*

inductive-cases *InputE* [*elim*!]: (*IN x*, *p*) ⇒ *p'*

inductive-cases *OutputE* [*elim*!]: (*OUT x*, *p*) ⇒ *p'*

inductive-cases *SeqE* [*elim*!]: (c_1;; c_2, *p*) ⇒ *p'*

inductive-cases *OrE* [*elim*!]: (c_1 *OR* c_2, *p*) ⇒ *p'*

inductive-cases *IfE* [*elim*!]: (*IF b THEN* c_1 *ELSE* c_2, *p*) ⇒ *p'*

inductive-cases *WhileE* [*elim*]: (*WHILE b DO c*, *p*) ⇒ *p′*

1.3 Extended small-step semantics

The original small-step semantics of language IMP associates a pair formed by a command and a program execution stage, which consists of a program state, with another such pair, formed by a command to be executed next and a resulting program execution stage, which consists of a program state as well. The extended small-step semantics defined here below rather uses the same extended program execution stage notion as the extended big-step semantics specified above, and is defined accordingly.

inductive *small-step* :: *com* × *stage* ⇒ *com* × *stage* ⇒ *bool*
  (**infix** ‹→› *55*) **where**
Assign:
 (*x* ::= *a*, *s*, *p*) → (*SKIP*, *s*(*x* := *aval a s*), *p*) |
Input:
 n = *length* [*p*←*vs*. *fst p* = *x*] ⟹ (*IN x*, *s*, *f*, *vs*, *ws*) →
   (*SKIP*, *s*(*x* := *f x n*), *f*, *vs* @ [(*x*, *f x n*)], *ws*) |
Output:
 (*OUT x*, *s*, *f*, *vs*, *ws*) → (*SKIP*, *s*, *f*, *vs*, *ws* @ [(*x*, *s x*)]) |
Seq1:
 (*SKIP*;; c_2, *p*) → (c_2, *p*) |
Seq2:
 (c_1, *p*) → ($c_1{}'$, *p′*) ⟹ (c_1;; c_2, *p*) → ($c_1{}'$;; c_2, *p′*) |
Or1:
 (c_1 *OR* c_2, *p*) → (c_1, *p*) |
Or2:
 (c_1 *OR* c_2, *p*) → (c_2, *p*) |
IfTrue:
 bval b s ⟹ (*IF b THEN* c_1 *ELSE* c_2, *s*, *p*) → (c_1, *s*, *p*) |
IfFalse:
 ¬ *bval b s* ⟹ (*IF b THEN* c_1 *ELSE* c_2, *s*, *p*) → (c_2, *s*, *p*) |
WhileFalse:
 ¬ *bval b s* ⟹ (*WHILE b DO c*, *s*, *p*) → (*SKIP*, *s*, *p*) |
WhileTrue:
 bval b s ⟹ (*WHILE b DO c*, *s*, *p*) → (*c*;; *WHILE b DO c*, *s*, *p*)

declare *small-step.intros* [*simp*, *intro*]

inductive-cases *skipE* [*elim*!]: (*SKIP*, *p*) → *cf*

inductive-cases *assignE* [*elim*!]: (*x* ::= *a*, *p*) → *cf*

inductive-cases *inputE* [*elim*!]: (*IN x*, *p*) → *cf*

inductive-cases *outputE* [*elim*!]: (*OUT x*, *p*) → *cf*

inductive-cases *seqE* [*elim*!]: $(c_1;;\ c_2,\ p) \rightarrow cf$

inductive-cases *orE* [*elim*!]: $(c_1\ OR\ c_2,\ p) \rightarrow cf$

inductive-cases *ifE* [*elim*!]: $(IF\ b\ THEN\ c_1\ ELSE\ c_2,\ p) \rightarrow cf$

inductive-cases *whileE* [*elim*]: (*WHILE b DO c*, *p*) → *cf*

abbreviation *small-steps* :: *com* × *stage* ⇒ *com* × *stage* ⇒ *bool*
(**infix** ‹→∗› *55*) **where**
cf →∗ *cf′* ≡ *star small-step cf cf′*

function *small-stepsl* ::
com × *stage* ⇒ (*com* × *stage*) *list* ⇒ *com* × *stage* ⇒ *bool*
(‹(- →∗′{-′} -)› [*51*, *51*] *55*)
where
cf →∗{[]} *cf′* = (*cf* = *cf′*) |
cf →∗{*cfs* @ [*cf′*]} *cf″* = (*cf* →∗{*cfs*} *cf′* ∧ *cf′* → *cf″*)

by (*atomize-elim*, *auto intro*: *rev-cases*)
termination by *lexicographic-order*

1.4 Equivalence of big-step and small-step semantics

lemma *star-seq2*:
$(c_1,\ p) \rightarrow* (c_1',\ p') \Longrightarrow (c_1;;\ c_2,\ p) \rightarrow* (c_1';;\ c_2,\ p')$
proof (*induction rule*: *star-induct*)
  case *refl*
  thus *?case*
    by *simp*
next
  case *step*
  thus *?case*
    by (*blast intro*: *star.step*)
qed

lemma *seq-comp*:
$[\![(c_1,\ p_1) \rightarrow* (SKIP,\ p_2);\ (c_2,\ p_2) \rightarrow* (SKIP,\ p_3)]\!] \Longrightarrow$
  $(c_1;;\ c_2,\ p_1) \rightarrow* (SKIP,\ p_3)$
by (*blast intro*: *star.step star-seq2 star-trans*)

lemma *big-to-small*:
cf ⇒ *p* ⟹ *cf* →∗ (*SKIP*, *p*)
proof (*induction rule*: *big-step.induct*)
  fix $c_1\ c_2\ p_1\ p_2\ p_3$
  assume $(c_1,\ p_1) \rightarrow* (SKIP,\ p_2)$ **and** $(c_2,\ p_2) \rightarrow* (SKIP,\ p_3)$

```
  thus (c1;; c2, p1) →∗ (SKIP, p3)
    by (rule seq-comp)
next
  fix c1 c2 p p′
  assume (c1, p) →∗ (SKIP, p′)
  thus (c1 OR c2, p) →∗ (SKIP, p′)
    by (blast intro: star.step)
next
  fix c1 c2 p p′
  assume (c2, p) →∗ (SKIP, p′)
  thus (c1 OR c2, p) →∗ (SKIP, p′)
    by (blast intro: star.step)
next
  fix b c1 c2 s p p′
  assume bval b s
  hence (IF b THEN c1 ELSE c2, s, p) → (c1, s, p)
    by simp
  moreover assume (c1, s, p) →∗ (SKIP, p′)
  ultimately show
   (IF b THEN c1 ELSE c2, s, p) →∗ (SKIP, p′)
    by (simp add: star.step)
next
  fix b c1 c2 s p p′
  assume ¬ bval b s
  hence (IF b THEN c1 ELSE c2, s, p) → (c2, s, p)
    by simp
  moreover assume (c2, s, p) →∗ (SKIP, p′)
  ultimately show
   (IF b THEN c1 ELSE c2, s, p) →∗ (SKIP, p′)
    by (simp add: star.step)
next
  fix b c s1 s2 s3 p1 p2 p3
  assume bval b s1
  hence (WHILE b DO c, s1, p1) →∗ (c;; WHILE b DO c, s1, p1)
    by simp
  moreover assume
   (c, s1, p1) →∗ (SKIP, s2, p2) and
   (WHILE b DO c, s2, p2) →∗ (SKIP, s3, p3)
  hence (c;; WHILE b DO c, s1, p1) →∗ (SKIP, s3, p3)
    by (rule seq-comp)
  ultimately show (WHILE b DO c, s1, p1) →∗ (SKIP, s3, p3)
    by (blast intro: star-trans)
qed fastforce+

lemma small1-big-continue:
⟦cf → cf′; cf′ ⇒ p⟧ ⟹ cf ⇒ p
by (induction arbitrary: p rule: small-step.induct, force+)

lemma small-to-big:
```

```
  cf →∗ (SKIP, p) ⟹ cf ⇒ p
by (induction cf (SKIP, p) rule: star.induct,
  auto intro: small1-big-continue)

lemma big-iff-small:
  cf ⇒ p = cf →∗ (SKIP, p)
by (blast intro: big-to-small small-to-big)

end
```

2 Underlying concepts and formal definitions

```
theory Definitions
  imports Small-Step
begin
```

2.1 Global context definitions

As compared with my previous paper [10]:

- Type *flow*, which models any potential program execution flow as a list of instructions, occurring in their order of execution, is extended with two additional instructions, namely an input instruction *IN x* and an output instruction *OUT x* standing for the respective additional commands of the considered extension of language IMP.
- Function *run-flow*, which used to map a pair formed by such a program execution flow *cs* and a starting program state *s* to the resulting program state, here takes two additional parameters, namely a starting trace of inputs *vs* and a stream of input values *f*, since they are required as well for computing the resulting program state according to the semantics of the considered extension of language IMP.

```
declare [[syntax-ambiguity-warning = false]]
```

```
datatype com-flow =
  Assign vname aexp  (‹- ::= -› [1000, 61] 70) |
  Input vname  (‹(IN -)› [61] 70) |
  Output vname  (‹(OUT -)› [61] 70) |
  Observe vname set  (‹⟨-⟩› [61] 70)

type-synonym flow = com-flow list
type-synonym tag = vname × nat
type-synonym config = state set × vname set
type-synonym scope = config set × bool
```

type-synonym *state-upd* = *vname* × *val option*

definition *eq-streams* ::
stream ⇒ *stream* ⇒ *inputs* ⇒ *inputs* ⇒ *tag set* ⇒ *bool*
(‹(- = - ′(⊆ -, -, -′))› [*51*, *51*] *50*) **where**
f = *f′* (⊆ *vs*, *vs′*, *T*) ≡ ∀ (*x*, *n*) ∈ *T*.
f x (*length* [*p*←*vs*. *fst p* = *x*] + *n*) =
f′ x (*length* [*p*←*vs′*. *fst p* = *x*] + *n*)

abbreviation *eq-states* :: *state* ⇒ *state* ⇒ *vname set* ⇒ *bool*
(‹(- = - ′(⊆ -′))› [*51*, *51*] *50*) **where**
s = *t* (⊆ *X*) ≡ ∀ *x* ∈ *X*. *s x* = *t x*

abbreviation *univ-states* :: *state set* ⇒ *vname set* ⇒ *state set*
(‹(*Univ* - ′(⊆ -′))› [*51*] *75*) **where**
Univ A (⊆ *X*) ≡ {*s*. ∃ *t* ∈ *A*. *s* = *t* (⊆ *X*)}

abbreviation *univ-vars-if* :: *state set* ⇒ *vname set* ⇒ *vname set*
(‹(*Univ??* - -)› [*51*, *75*] *75*) **where**
Univ?? A X ≡ *if A* = {} *then UNIV else X*

abbreviation *tl2 xs* ≡ *tl* (*tl xs*)

primrec *avars* :: *aexp* ⇒ *vname set* **where**
avars (*N i*) = {} |
avars (*V x*) = {*x*} |
avars (*Plus* a_1 a_2) = *avars* a_1 ∪ *avars* a_2

primrec *bvars* :: *bexp* ⇒ *vname set* **where**
bvars (*Bc v*) = {} |
bvars (*Not b*) = *bvars b* |
bvars (*And* b_1 b_2) = *bvars* b_1 ∪ *bvars* b_2 |
bvars (*Less* a_1 a_2) = *avars* a_1 ∪ *avars* a_2

fun *no-upd* :: *flow* ⇒ *vname set* ⇒ *bool* **where**
no-upd (*x* ::= - # *cs*) *X* = (*x* ∉ *X* ∧ *no-upd cs X*) |
no-upd (*IN x* # *cs*) *X* = (*x* ∉ *X* ∧ *no-upd cs X*) |
no-upd (*OUT x* # *cs*) *X* = (*x* ∉ *X* ∧ *no-upd cs X*) |
no-upd (- # *cs*) *X* = *no-upd cs X* |
no-upd - - = *True*

fun *flow-aux* :: *com list* ⇒ *flow* **where**
flow-aux (*x* ::= *a* # *cs*) = (*x* ::= *a*) # *flow-aux cs* |
flow-aux (*IN x* # *cs*) = *IN x* # *flow-aux cs* |
flow-aux (*OUT x* # *cs*) = *OUT x* # *flow-aux cs* |
flow-aux (*IF b THEN - ELSE -* # *cs*) = ⟨*bvars b*⟩ # *flow-aux cs* |

flow-aux (*WHILE b DO* - # *cs*) = ⟨*bvars b*⟩ # *flow-aux cs* |
flow-aux (*c*;; - # *cs*) = *flow-aux* (*c* # *cs*) |
flow-aux (- # *cs*) = *flow-aux cs* |
flow-aux [] = []

definition *flow* :: (*com* × *stage*) *list* ⇒ *flow* **where**
flow cfs = *flow-aux* (*map fst cfs*)

function *in-flow* :: *flow* ⇒ *inputs* ⇒ *stream* ⇒ *inputs* **where**
in-flow (*cs* @ [- ::= -]) *vs f* = *in-flow cs vs f* |
in-flow (*cs* @ [*IN x*]) *vs f* = *in-flow cs vs f* @ (*let*
  n = *length* [*p*←*vs*. *fst p* = *x*] + *length* [*c*←*cs*. *c* = *IN x*]
  in [(*x*, *f x n*)]) |
in-flow (*cs* @ [*OUT* -]) *vs f* = *in-flow cs vs f* |
in-flow (*cs* @ [⟨-⟩]) *vs f* = *in-flow cs vs f* |
in-flow [] - - = []

proof *atomize-elim*
  fix *p* :: *flow* × *inputs* × *stream*
  show
   (∃ *cs x a vs f*. *p* = (*cs* @ [*x* ::= *a*], *vs*, *f*)) ∨
    (∃ *cs x vs f*. *p* = (*cs* @ [*IN x*], *vs*, *f*)) ∨
    (∃ *cs x vs f*. *p* = (*cs* @ [*OUT x*], *vs*, *f*)) ∨
    (∃ *cs X vs f*. *p* = (*cs* @ [⟨*X*⟩], *vs*, *f*)) ∨
    (∃ *vs f*. *p* = ([], *vs*, *f*))
    by (*cases p*, *metis com-flow.exhaust rev-exhaust*)
qed *auto*

termination by *lexicographic-order*

function *run-flow* :: *flow* ⇒ *inputs* ⇒ *state* ⇒ *stream* ⇒ *state* **where**
run-flow (*cs* @ [*x* ::= *a*]) *vs s f* = (*let t* = *run-flow cs vs s f*
  in t(*x* := *aval a t*)) |
run-flow (*cs* @ [*IN x*]) *vs s f* = (*let t* = *run-flow cs vs s f*;
  n = *length* [*p*←*vs*. *fst p* = *x*] + *length* [*c*←*cs*. *c* = *IN x*]
  in t(*x* := *f x n*)) |
run-flow (*cs* @ [*OUT* -]) *vs s f* = *run-flow cs vs s f* |
run-flow (*cs* @ [⟨-⟩]) *vs s f* = *run-flow cs vs s f* |
run-flow [] *vs s* - = *s*

proof *atomize-elim*
  fix *p* :: *flow* × *inputs* × *state* × *stream*
  show
   (∃ *cs x a vs s f*. *p* = (*cs* @ [*x* ::= *a*], *vs*, *s*, *f*)) ∨
    (∃ *cs x vs s f*. *p* = (*cs* @ [*IN x*], *vs*, *s*, *f*)) ∨
    (∃ *cs x vs s f*. *p* = (*cs* @ [*OUT x*], *vs*, *s*, *f*)) ∨
    (∃ *cs X vs s f*. *p* = (*cs* @ [⟨*X*⟩], *vs*, *s*, *f*)) ∨

(∃ *vs s f*. *p* = ([], *vs*, *s*, *f*))
by (*cases p*, *metis com-flow.exhaust rev-exhaust*)
qed *auto*

termination by *lexicographic-order*

function *out-flow* :: *flow* ⇒ *inputs* ⇒ *state* ⇒ *stream* ⇒ *outputs* **where**
out-flow (*cs* @ [- ::= -]) *vs s f* = *out-flow cs vs s f* |
out-flow (*cs* @ [*IN* -]) *vs s f* = *out-flow cs vs s f* |
out-flow (*cs* @ [*OUT x*]) *vs s f* = (*let t* = *run-flow cs vs s f*
in out-flow cs vs s f @ [(*x*, *t x*)]) |
out-flow (*cs* @ [⟨-⟩]) *vs s f* = *out-flow cs vs s f* |
out-flow [] - - - = []

proof *atomize-elim*
fix *p* :: *flow* × *inputs* × *state* × *stream*
show
(∃ *cs x a vs s f*. *p* = (*cs* @ [*x* ::= *a*], *vs*, *s*, *f*)) ∨
(∃ *cs x vs s f*. *p* = (*cs* @ [*IN x*], *vs*, *s*, *f*)) ∨
(∃ *cs x vs s f*. *p* = (*cs* @ [*OUT x*], *vs*, *s*, *f*)) ∨
(∃ *cs X vs s f*. *p* = (*cs* @ [⟨*X*⟩], *vs*, *s*, *f*)) ∨
(∃ *vs s f*. *p* = ([], *vs*, *s*, *f*))
by (*cases p*, *metis com-flow.exhaust rev-exhaust*)
qed *auto*

termination by *lexicographic-order*

2.2 Local context definitions

locale *noninterf* =
fixes
interf :: *state* ⇒ *'d* ⇒ *'d* ⇒ *bool*
(‹(-: - ⇝ -)› [*51*, *51*, *51*] *50*) **and**
dom :: *vname* ⇒ *'d* **and**
state :: *vname set*
assumes
interf-state: *s* = *t* (⊆ *state*) ⟹ *interf s* = *interf t*

context *noninterf*
begin

As in my previous paper [10], function *sources* is defined along with an auxiliary function *sources-aux* by means of mutual recursion. According to this definition, the set of variables *sources cs vs s f x*, where:

- *cs* is a program execution flow,

- *vs* is a trace of inputs,
- *s* is a program state,
- *f* is a stream of input values, and
- *x* is a variable,

contains a variable y if there exist a descending sequence of left sublists cs_{n+1}, cs_n @ $[c_n]$, ..., cs_1 @ $[c_1]$ of *cs* and a sequence of variables y_{n+1}, ..., y_1, where $n \geq 1$, $cs_{n+1} = cs$, $y_{n+1} = x$, and $y_1 = y$, satisfying the following conditions.

- For each positive integer $i \leq n$, the instruction c_i is an assignment $y_{i+1} ::= a_i$ such that:
  - $y_i \in avars\ a_i$,
  - $run\text{-}flow\ cs_i\ vs\ s\ f$: $dom\ y_i \rightsquigarrow dom\ y_{i+1}$, and
  - the right sublist of cs_{i+1} complementary to cs_i @ $[c_i]$ does not comprise any assignment or input instruction setting variable y_{i+1} (as the assignment c_i would otherwise be irrelevant),

  or else an observation $\langle X_i \rangle$ such that:
  - $y_i \in X_i$ and
  - $run\text{-}flow\ cs_i\ vs\ s\ f$: $dom\ y_i \rightsquigarrow dom\ y_{i+1}$.
- The program execution flow cs_1 does not comprise any assignment or input instruction setting variable y.

In addition, *sources cs vs s f x* contains variable x also if the program execution flow *cs* does not comprise any assignment or input instruction setting variable x.

function
  $sources :: flow \Rightarrow inputs \Rightarrow state \Rightarrow stream \Rightarrow vname \Rightarrow vname\ set$ **and**
  $sources\text{-}aux :: flow \Rightarrow inputs \Rightarrow state \Rightarrow stream \Rightarrow vname \Rightarrow vname\ set$
where

$sources\ (cs\ @\ [c])\ vs\ s\ f\ x = (case\ c\ of$
  $z ::= a \Rightarrow if\ z = x$
    $then\ sources\text{-}aux\ cs\ vs\ s\ f\ x \cup \bigcup\ \{sources\ cs\ vs\ s\ f\ y \mid y.$
      $run\text{-}flow\ cs\ vs\ s\ f\text{: } dom\ y \rightsquigarrow dom\ x \wedge y \in avars\ a\}$
    $else\ sources\ cs\ vs\ s\ f\ x \mid$
  $IN\ z \Rightarrow if\ z = x$

```
    then sources-aux cs vs s f x
    else sources cs vs s f x |
  ⟨X⟩ ⇒
    sources cs vs s f x ∪ ⋃ {sources cs vs s f y | y.
      run-flow cs vs s f: dom y ⇝ dom x ∧ y ∈ X} |
  - ⇒
    sources cs vs s f x) |

sources [] - - - x = {x} |

sources-aux (cs @ [c]) vs s f x = (case c of
  ⟨X⟩ ⇒
    sources-aux cs vs s f x ∪ ⋃ {sources cs vs s f y | y.
      run-flow cs vs s f: dom y ⇝ dom x ∧ y ∈ X} |
  - ⇒
    sources-aux cs vs s f x) |

sources-aux [] - - - - = {}

proof atomize-elim
  fix a :: flow × inputs × state × stream × vname +
    flow × inputs × state × stream × vname
  show
   (∃ cs c vs s f x. a = Inl (cs @ [c], vs, s, f, x)) ∨
    (∃ vs s f x. a = Inl ([], vs, s, f, x)) ∨
    (∃ cs c vs s f x. a = Inr (cs @ [c], vs, s, f, x)) ∨
    (∃ vs s f x. a = Inr ([], vs, s, f, x))
    by (metis obj-sumE prod-cases3 rev-exhaust)
qed auto

termination by lexicographic-order

lemmas sources-induct = sources-sources-aux.induct
```

Function *sources-out*, defined here below, takes the same parameters *cs*, *vs*, *s*, *f*, and *x* as function *sources*, and returns the set of the variables whose values in the program state *s* are allowed to affect the outputs of variable *x* possibly occurring as a result of the execution of flow *cs* if it starts from the initial state *s* and the initial trace of inputs *vs*, and takes place according to the stream of input values *f*.

In more detail, the set of variables *sources-out cs vs s f x* is defined as the union of any set of variables *sources* cs_i *vs s f* x_i, where cs_i @ $[c_i]$ is any left sublist of *cs* such that the instruction c_i is an output instruction $OUT\ x$, in which case $x_i = x$, or else an observation $\langle X_i \rangle$ such that:

- $x_i \in X_i$ and

- *run-flow* cs_i *vs s f*: *dom* $x_i \rightsquigarrow$ *dom x*.

```
function
  sources-out :: flow ⇒ inputs ⇒ state ⇒ stream ⇒ vname ⇒ vname set
where

sources-out (cs @ [c]) vs s f x = (case c of
  OUT z ⇒
    sources-out cs vs s f x ∪ (if z = x then sources cs vs s f x else {}) |
  ⟨X⟩ ⇒
    sources-out cs vs s f x ∪ ⋃ {sources cs vs s f y | y.
      run-flow cs vs s f: dom y ⇝ dom x ∧ y ∈ X} |
  - ⇒
    sources-out cs vs s f x) |

sources-out [] - - - - = {}

by (atomize-elim, auto intro: rev-cases)
termination by lexicographic-order
```

Function *tags*, defined here below, takes the same parameters *cs*, *vs*, *s*, *f*, and x as the previous functions, and returns the set of the *tags*, namely of the pairs (y, m) where y is a variable and m is a natural number, such that the m-th input instruction *IN y* within flow *cs* is allowed to affect the value of variable x resulting from the execution of *cs* if it starts from the initial state s and the initial trace of inputs *vs*, and takes place according to the stream of input values *f*.

In more detail, the set of tags *tags cs vs s f x* contains a tag (y, m) just in case there exist a descending sequence of left sublists cs_{n+1}, cs_n @ $[c_n]$, ..., cs_1 @ $[c_1]$ of *cs* and a sequence of variables y_{n+1}, ..., y_1, where $n \geq 1$, $cs_{n+1} = cs$, $y_{n+1} = x$, $y_1 = y$, and $y = x$ if $n = 1$, satisfying the following conditions.

- For each integer i, if any, such that $1 < i \leq n$, the instruction c_i is an assignment y_{i+1} ::= a_i such that:
  - $y_i \in$ *avars* a_i,
  - *run-flow* cs_i *vs s f*: *dom* $y_i \rightsquigarrow$ *dom* y_{i+1}, and
  - the right sublist of cs_{i+1} complementary to cs_i @ $[c_i]$ does not comprise any assignment or input instruction setting variable y_{i+1} (as the assignment c_i would otherwise be irrelevant),

  or else an observation $\langle X_i \rangle$ such that:

  - $y_i \in X_i$ and
  - *run-flow* cs_i *vs s f*: *dom* $y_i \rightsquigarrow$ *dom* y_{i+1}.

- The instruction c_1 is the m-th input instruction *IN y* within flow *cs*.

- The right sublist of cs_2 complementary to cs_1 @ [c_1] does not comprise any assignment or input instruction setting variable y (as the input instruction c_1 would otherwise be irrelevant).

```
function
  tags :: flow ⇒ inputs ⇒ state ⇒ stream ⇒ vname ⇒ tag set and
  tags-aux :: flow ⇒ inputs ⇒ state ⇒ stream ⇒ vname ⇒ tag set
where

tags (cs @ [c]) vs s f x = (case c of
  z ::= a ⇒ if z = x
    then tags-aux cs vs s f x ∪ ⋃ {tags cs vs s f y | y.
      run-flow cs vs s f: dom y ⇝ dom x ∧ y ∈ avars a}
    else tags cs vs s f x |
  IN z ⇒ if z = x
    then insert (x, length [c←cs. c = IN x]) (tags-aux cs vs s f x)
    else tags cs vs s f x |
  ⟨X⟩ ⇒
    tags cs vs s f x ∪ ⋃ {tags cs vs s f y | y.
      run-flow cs vs s f: dom y ⇝ dom x ∧ y ∈ X} |
  - ⇒
    tags cs vs s f x) |

tags [] - - - - = {} |

tags-aux (cs @ [c]) vs s f x = (case c of
  ⟨X⟩ ⇒
    tags-aux cs vs s f x ∪ ⋃ {tags cs vs s f y | y.
      run-flow cs vs s f: dom y ⇝ dom x ∧ y ∈ X} |
  - ⇒
    tags-aux cs vs s f x) |

tags-aux [] - - - - = {}

proof atomize-elim
  fix a :: flow × inputs × state × stream × vname +
    flow × inputs × state × stream × vname
  show
   (∃ cs c vs s f x. a = Inl (cs @ [c], vs, s, f, x)) ∨
    (∃ vs s f x. a = Inl ([], vs, s, f, x)) ∨
    (∃ cs c vs s f x. a = Inr (cs @ [c], vs, s, f, x)) ∨
    (∃ vs s f x. a = Inr ([], vs, s, f, x))
```

```
  by (metis obj-sumE prod-cases3 rev-exhaust)
qed auto

termination by lexicographic-order

lemmas tags-induct = tags-tags-aux.induct
```

Finally, function *tags-out*, defined here below, takes the same parameters *cs*, *vs*, *s*, *f*, and x as the previous functions, and returns the set of the tags (y, m) such that the m-th input instruction *IN y* within flow *cs* is allowed to affect the outputs of variable x possibly occurring as a result of the execution of flow *cs* if it starts from the initial state s and the initial trace of inputs *vs*, and takes place according to the stream of input values *f*.

In more detail, the set of tags *tags-out cs vs s f x* is defined as the union of any set of tags *tags* cs_i *vs s f* x_i, where cs_i @ $[c_i]$ is any left sublist of *cs* such that the instruction c_i is an output instruction *OUT x*, in which case $x_i = x$, or else an observation $\langle X_i \rangle$ such that:

- $x_i \in X_i$ and
- *run-flow* cs_i *vs s f*: *dom* $x_i \rightsquigarrow$ *dom x*.

```
function
  tags-out :: flow ⇒ inputs ⇒ state ⇒ stream ⇒ vname ⇒ tag set
where

tags-out (cs @ [c]) vs s f x = (case c of
  OUT z ⇒
    tags-out cs vs s f x ∪ (if z = x then tags cs vs s f x else {}) |
  ⟨X⟩ ⇒
    tags-out cs vs s f x ∪ ⋃ {tags cs vs s f y | y.
      run-flow cs vs s f: dom y ⇝ dom x ∧ y ∈ X} |
  - ⇒
    tags-out cs vs s f x) |

tags-out [] - - - - = {}

by (atomize-elim, auto intro: rev-cases)
termination by lexicographic-order
```

Predicate *correct*, defined here below, formalizes the extended termination-sensitive information flow correctness criterion. As in my previous paper [10], its parameters consist of a program c, a set of program states A, and a set of variables X.

In more detail, for any state s agreeing with a state in A on the value of each state variable contained in X, let the small-step semantics turn:

- the command c and the program execution stage $(s,\ f,\ vs,\ ws)$ into a command c_1 and a program execution stage $(s_1,\ f,\ vs_1,\ ws_1)$, and
- the command c_1 and the program execution stage $(s_1,\ f,\ vs_1,\ ws_1)$ into a command c_2 and a program execution stage $(s_2,\ f,\ vs_2,\ ws_2)$.

Furthermore, let:

- cs be the program execution flow leading from $(c_1,\ s_1,\ f,\ vs_1,\ ws_1)$ to $(c_2,\ s_2,\ f,\ vs_2,\ ws_2)$, and
- $(t_1,\ f',\ vs_1',\ ws_1')$ be any program execution stage,

and assume that the following conditions hold.

- S is a nonempty subset of the set of the variables x such that state t_1 agrees with s_1 on the value of each variable contained in $sources\ cs\ vs_1\ s_1\ f\ x$.
- For each variable x contained in S, and each tag $(y,\ n)$ contained in $tags\ cs\ vs_1\ s_1\ f\ x$, the stream of input values f' agrees with f on the input value assigned to variable y by an input instruction $IN\ y$ after n previous such assignments to y following any one tracked by the starting trace of inputs vs_1' and vs_1, respectively.

Then, the information flow is correct only if the small-step semantics turns the command c_1 and the program execution stage $(t_1,\ f',\ vs_1',\ ws_1')$ into a command c_2' and a program execution stage $(t_2,\ f',\ vs_2',\ ws_2')$ satisfying the following correctness conditions.

- $c_2' = SKIP$ just in case $c_2 = SKIP$; namely, program execution terminates just in case it terminates as a result of the execution of flow cs, so that the two program executions cannot be distinguished based on program termination.
- The resulting sequence of input requests $IN\ x$ being prompted, where x is any variable contained in S, matches the one triggered by the execution of flow cs, so that the two program executions cannot be distinguished based on those sequences.
- States t_2 and s_2 agree on the value of each variable contained in S, so that the two program executions cannot be distinguished based on the resulting program states.

Likewise, if the above assumptions hold for functions *sources-out* and *tags-out* in place of functions *sources* and *tags*, respectively, then the information flow correctness requires the first two correctness conditions listed above to hold as well, plus the following one.

- The resulting sequence of outputs of any variable contained in S matches the one produced by the execution of flow cs, so that the two program executions cannot be distinguished based on those sequences.

abbreviation *ok-flow-1* **where**

$$\begin{array}{l}
\mathit{ok\text{-}flow\text{-}1}\ c_1\ c_2\ c_2'\ s_1\ s_2\ t_1\ t_2\ f\ f'\ vs_1\ vs_1'\ vs_2\ vs_2'\ ws_1'\ ws_2'\ cs \equiv \\
\quad \forall S \subseteq \{x.\ s_1 = t_1\ (\subseteq \mathit{sources}\ cs\ vs_1\ s_1\ f\ x)\}. \\
\quad\quad S \neq \{\} \longrightarrow \\
\quad\quad f = f'\ (\subseteq vs_1,\ vs_1',\ \bigcup\ \{\mathit{tags}\ cs\ vs_1\ s_1\ f\ x \mid x.\ x \in S\}) \longrightarrow \\
\quad\quad\quad (c_1,\ t_1,\ f',\ vs_1',\ ws_1') \rightarrow\!*\ (c_2',\ t_2,\ f',\ vs_2',\ ws_2') \wedge \\
\quad\quad\quad (c_2 = \mathit{SKIP}) = (c_2' = \mathit{SKIP}) \wedge \\
\quad\quad\quad \mathit{map}\ \mathit{fst}\ [p \leftarrow \mathit{drop}\ (\mathit{length}\ vs_1)\ vs_2.\ \mathit{fst}\ p \in S] = \\
\quad\quad\quad\quad \mathit{map}\ \mathit{fst}\ [p \leftarrow \mathit{drop}\ (\mathit{length}\ vs_1')\ vs_2'.\ \mathit{fst}\ p \in S] \wedge \\
\quad\quad\quad s_2 = t_2\ (\subseteq S)
\end{array}$$

abbreviation *ok-flow-2* **where**

$$\begin{array}{l}
\mathit{ok\text{-}flow\text{-}2}\ c_1\ c_2\ c_2'\ s_1\ t_1\ t_2\ f\ f'\ vs_1\ vs_1'\ vs_2\ vs_2'\ ws_1\ ws_1'\ ws_2\ ws_2'\ cs \equiv \\
\quad \forall S \subseteq \{x.\ s_1 = t_1\ (\subseteq \mathit{sources\text{-}out}\ cs\ vs_1\ s_1\ f\ x)\}. \\
\quad\quad S \neq \{\} \longrightarrow \\
\quad\quad f = f'\ (\subseteq vs_1,\ vs_1',\ \bigcup\ \{\mathit{tags\text{-}out}\ cs\ vs_1\ s_1\ f\ x \mid x.\ x \in S\}) \longrightarrow \\
\quad\quad\quad (c_1,\ t_1,\ f',\ vs_1',\ ws_1') \rightarrow\!*\ (c_2',\ t_2,\ f',\ vs_2',\ ws_2') \wedge \\
\quad\quad\quad (c_2 = \mathit{SKIP}) = (c_2' = \mathit{SKIP}) \wedge \\
\quad\quad\quad \mathit{map}\ \mathit{fst}\ [p \leftarrow \mathit{drop}\ (\mathit{length}\ vs_1)\ vs_2.\ \mathit{fst}\ p \in S] = \\
\quad\quad\quad\quad \mathit{map}\ \mathit{fst}\ [p \leftarrow \mathit{drop}\ (\mathit{length}\ vs_1')\ vs_2'.\ \mathit{fst}\ p \in S] \wedge \\
\quad\quad\quad [p \leftarrow \mathit{drop}\ (\mathit{length}\ ws_1)\ ws_2.\ \mathit{fst}\ p \in S] = \\
\quad\quad\quad\quad [p \leftarrow \mathit{drop}\ (\mathit{length}\ ws_1')\ ws_2'.\ \mathit{fst}\ p \in S]
\end{array}$$

abbreviation *ok-flow* **where**

$$\begin{array}{l}
\mathit{ok\text{-}flow}\ c_1\ c_2\ s_1\ s_2\ f\ vs_1\ vs_2\ ws_1\ ws_2\ cs \equiv \\
\quad \forall t_1\ f'\ vs_1'\ ws_1'.\ \exists c_2'\ t_2\ vs_2'\ ws_2'. \\
\quad\quad \mathit{ok\text{-}flow\text{-}1}\ c_1\ c_2\ c_2'\ s_1\ s_2\ t_1\ t_2\ f\ f'\ vs_1\ vs_1'\ vs_2\ vs_2'\ ws_1'\ ws_2'\ cs \wedge \\
\quad\quad \mathit{ok\text{-}flow\text{-}2}\ c_1\ c_2\ c_2'\ s_1\ t_1\ t_2\ f\ f'\ vs_1\ vs_1'\ vs_2\ vs_2'\ ws_1\ ws_1'\ ws_2\ ws_2'\ cs
\end{array}$$

definition $\mathit{correct} :: \mathit{com} \Rightarrow \mathit{state}\ \mathit{set} \Rightarrow \mathit{vname}\ \mathit{set} \Rightarrow \mathit{bool}$ **where**

$$\begin{array}{l}
\mathit{correct}\ c\ A\ X \equiv \\
\quad \forall s \in \mathit{Univ}\ A\ (\subseteq \mathit{state} \cap X).\ \forall c_1\ c_2\ s_1\ s_2\ f\ vs\ vs_1\ vs_2\ ws\ ws_1\ ws_2\ \mathit{cfs}. \\
\quad\quad (c,\ s,\ f,\ vs,\ ws) \rightarrow\!*\ (c_1,\ s_1,\ f,\ vs_1,\ ws_1) \wedge \\
\quad\quad (c_1,\ s_1,\ f,\ vs_1,\ ws_1) \rightarrow\!*\{\mathit{cfs}\}\ (c_2,\ s_2,\ f,\ vs_2,\ ws_2) \longrightarrow \\
\quad\quad\quad \mathit{ok\text{-}flow}\ c_1\ c_2\ s_1\ s_2\ f\ vs_1\ vs_2\ ws_1\ ws_2\ (\mathit{flow}\ \mathit{cfs})
\end{array}$$

abbreviation $\mathit{noninterf\text{-}set} :: \mathit{state}\ \mathit{set} \Rightarrow \mathit{vname}\ \mathit{set} \Rightarrow \mathit{vname}\ \mathit{set} \Rightarrow \mathit{bool}$

(‹(-: - ⇝| -)› *[51, 51, 51] 50*) **where**
$A\colon X \rightsquigarrow|\ Y \equiv \forall y \in Y.\ \exists s \in A.\ \exists x \in X.\ \neg\ s\colon \textit{dom}\ x \rightsquigarrow \textit{dom}\ y$

abbreviation *ok-flow-aux-1* **where**
$\textit{ok-flow-aux-1}\ c_1\ c_2\ c_2'\ s_1\ t_1\ t_2\ f\ f'\ vs_1\ vs_1'\ vs_2\ vs_2'\ ws_1'\ ws_2'\ cs \equiv$
$\quad \forall S \subseteq \{x.\ s_1 = t_1\ (\subseteq \textit{sources-aux}\ cs\ vs_1\ s_1\ f\ x)\}.$
$\qquad S \neq \{\} \longrightarrow$
$\qquad f = f'\ (\subseteq vs_1,\ vs_1',\ \bigcup\ \{\textit{tags-aux}\ cs\ vs_1\ s_1\ f\ x \mid x.\ x \in S\}) \longrightarrow$
$\qquad\quad (c_1,\ t_1,\ f',\ vs_1',\ ws_1') \rightarrow\!*\ (c_2',\ t_2,\ f',\ vs_2',\ ws_2') \wedge$
$\qquad\quad (c_2 = \textit{SKIP}) = (c_2' = \textit{SKIP}) \wedge$
$\qquad\quad \textit{map fst}\ [p \leftarrow \textit{drop}\ (\textit{length}\ vs_1)\ vs_2.\ \textit{fst}\ p \in S] =$
$\qquad\qquad \textit{map fst}\ [p \leftarrow \textit{drop}\ (\textit{length}\ vs_1')\ vs_2'.\ \textit{fst}\ p \in S]$

abbreviation *ok-flow-aux-2* **where**
$\textit{ok-flow-aux-2}\ s_1\ s_2\ t_1\ t_2\ f\ f'\ vs_1\ vs_1'\ cs \equiv$
$\quad \forall S \subseteq \{x.\ s_1 = t_1\ (\subseteq \textit{sources}\ cs\ vs_1\ s_1\ f\ x)\}.$
$\qquad S \neq \{\} \longrightarrow$
$\qquad f = f'\ (\subseteq vs_1,\ vs_1',\ \bigcup\ \{\textit{tags}\ cs\ vs_1\ s_1\ f\ x \mid x.\ x \in S\}) \longrightarrow$
$\qquad\quad s_2 = t_2\ (\subseteq S)$

abbreviation *ok-flow-aux-3* **where**
$\textit{ok-flow-aux-3}\ s_1\ t_1\ f\ f'\ vs_1\ vs_1'\ ws_1\ ws_1'\ ws_2\ ws_2'\ cs \equiv$
$\quad \forall S \subseteq \{x.\ s_1 = t_1\ (\subseteq \textit{sources-out}\ cs\ vs_1\ s_1\ f\ x)\}.$
$\qquad S \neq \{\} \longrightarrow$
$\qquad f = f'\ (\subseteq vs_1,\ vs_1',\ \bigcup\ \{\textit{tags-out}\ cs\ vs_1\ s_1\ f\ x \mid x.\ x \in S\}) \longrightarrow$
$\qquad\quad [p \leftarrow \textit{drop}\ (\textit{length}\ ws_1)\ ws_2.\ \textit{fst}\ p \in S] =$
$\qquad\qquad [p \leftarrow \textit{drop}\ (\textit{length}\ ws_1')\ ws_2'.\ \textit{fst}\ p \in S]$

abbreviation *ok-flow-aux* :: $\textit{config set} \Rightarrow \textit{com} \Rightarrow \textit{com} \Rightarrow \textit{state} \Rightarrow \textit{state} \Rightarrow$
$\quad \textit{stream} \Rightarrow \textit{inputs} \Rightarrow \textit{inputs} \Rightarrow \textit{outputs} \Rightarrow \textit{outputs} \Rightarrow \textit{flow} \Rightarrow \textit{bool}$
where
$\textit{ok-flow-aux}\ U\ c_1\ c_2\ s_1\ s_2\ f\ vs_1\ vs_2\ ws_1\ ws_2\ cs \equiv$
$\quad (\forall t_1\ f'\ vs_1'\ ws_1'.\ \exists c_2'\ t_2\ vs_2'\ ws_2'.$
$\qquad \textit{ok-flow-aux-1}\ c_1\ c_2\ c_2'\ s_1\ t_1\ t_2\ f\ f'\ vs_1\ vs_1'\ vs_2\ vs_2'\ ws_1'\ ws_2'\ cs \wedge$
$\qquad \textit{ok-flow-aux-2}\ s_1\ s_2\ t_1\ t_2\ f\ f'\ vs_1\ vs_1'\ cs \wedge$
$\qquad \textit{ok-flow-aux-3}\ s_1\ t_1\ f\ f'\ vs_1\ vs_1'\ ws_1\ ws_1'\ ws_2\ ws_2'\ cs) \wedge$
$\quad (\forall Y.\ (\exists (A,\ X) \in U.\ A\colon X \rightsquigarrow|\ Y) \longrightarrow \textit{no-upd}\ cs\ Y)$

In addition to the equations handling the further constructs of the considered extension of language IMP, the auxiliary recursive function *ctyping1-aux* used to define the idempotent type system *ctyping1* differs from its counterpart used in my previous paper [10] also in that it records any update of a state variable using a pair of type *vname* × *val option*, where the first component is the state variable being updated, and the latter one matches *Some i* or *None* depending on whether its new value can be evaluated to an integer *i* at compile time or not.

Apart from the aforesaid type change, the equations for the constructs included in the original language IMP are the same as in my previous paper [10], whereas the equations for the additional constructs of the considered language extension are as follows.

- The equation for an input instruction *IN* x, like the one handling assignments, records the update of variable x just in case it is a state variable (as otherwise its update cannot change the applying interference relation). If so, its update is recorded with (x, *None*), since input values cannot be evaluated at compile time.
- The equation for an output instruction *OUT* x does not record any update, since output instructions leave the program state unchanged.
- The equation for a nondeterministic choice c_1 *OR* c_2 sets the returned value to $\vdash c_1 \sqcup \vdash c_2$, in the same way as the equation for a conditional statement *IF* b *THEN* c_1 *ELSE* c_2 whose boolean condition b cannot be evaluated at compile time.

As in my previous paper [10], the *state set* returned by *ctyping1* is defined so that any *indeterminate* state variable (namely, any state variable x with a latest recorded update (x, *None*)) may take an arbitrary value. Of course, a real-world implementation of this type system would not need to actually return a distinct state for any such value, but rather just to mark any indeterminate state variable in each returned state with some special value standing for *arbitrary*.

primrec *btyping1* :: *bexp* ⇒ *bool option* (‹(⊢ -)› [*51*] *55*) **where**

⊢ *Bc v* = *Some v* |

⊢ *Not b* = (*case* ⊢ *b of*
  Some v ⇒ *Some* (¬ *v*) | - ⇒ *None*) |

⊢ *And* b_1 b_2 = (*case* (⊢ b_1, ⊢ b_2) *of*
  (*Some* v_1, *Some* v_2) ⇒ *Some* ($v_1 \wedge v_2$) | - ⇒ *None*) |

⊢ *Less* a_1 a_2 = (*if avars* a_1 ∪ *avars* a_2 = {}
  then Some (*aval* a_1 (λ*x*. *0*) < *aval* a_2 (λ*x*. *0*)) *else None*)

inductive-set *ctyping1-merge-aux* :: *state-upd list set* ⇒
  state-upd list set ⇒ (*state-upd list* × *bool*) *list set*
  (**infix** ‹⨆› *55*) **for** *A* **and** *B* **where**

xs ∈ *A* ⟹ [(*xs*, *True*)] ∈ *A* ⨆ *B* |

ys ∈ *B* ⟹ [(*ys*, *False*)] ∈ *A* ⨆ *B* |

$[\![ws \in A \bigsqcup B;\ \neg\ snd\ (hd\ ws);\ xs \in A;\ (xs,\ True) \notin set\ ws]\!] \Longrightarrow$
$(xs,\ True)\ \#\ ws \in A \bigsqcup B$ |

$[\![ws \in A \bigsqcup B;\ snd\ (hd\ ws);\ ys \in B;\ (ys,\ False) \notin set\ ws]\!] \Longrightarrow$
$(ys,\ False)\ \#\ ws \in A \bigsqcup B$

declare *ctyping1-merge-aux.intros* [*intro*]

definition *ctyping1-append* ::
$state\text{-}upd\ list\ set \Rightarrow state\text{-}upd\ list\ set \Rightarrow state\text{-}upd\ list\ set$
(**infixl** ‹@› *55*) **where**
$A\ @\ B \equiv \{xs\ @\ ys \mid xs\ ys.\ xs \in A \wedge ys \in B\}$

definition *ctyping1-merge* ::
$state\text{-}upd\ list\ set \Rightarrow state\text{-}upd\ list\ set \Rightarrow state\text{-}upd\ list\ set$
(**infixl** ‹⊔› *55*) **where**
$A \sqcup B \equiv \{concat\ (map\ fst\ ws) \mid ws.\ ws \in A \bigsqcup B\}$

definition *ctyping1-merge-append* ::
$state\text{-}upd\ list\ set \Rightarrow state\text{-}upd\ list\ set \Rightarrow state\text{-}upd\ list\ set$
(**infixl** ‹$\sqcup_@$› *55*) **where**
$A \sqcup_@ B \equiv (if\ card\ B = Suc\ 0\ then\ A\ else\ A \sqcup B)\ @\ B$

primrec *ctyping1-aux* :: $com \Rightarrow state\text{-}upd\ list\ set$
(‹($\vdash$ -)› [*51*] *60*) **where**

$\vdash SKIP = \{[]\}$ |

$\vdash x ::= a = (if\ x \in state$
$then\ \{[(x,\ if\ avars\ a = \{\}\ then\ Some\ (aval\ a\ (\lambda x.\ 0))\ else\ None)]\}$
$else\ \{[]\})$ |

$\vdash IN\ x = (if\ x \in state\ then\ \{[(x,\ None)]\}\ else\ \{[]\})$ |

$\vdash OUT\ x = \{[]\}$ |

$\vdash c_1;;\ c_2 = \vdash c_1 \sqcup_@ \vdash c_2$ |

$\vdash c_1\ OR\ c_2 = \vdash c_1 \sqcup \vdash c_2$ |

$\vdash IF\ b\ THEN\ c_1\ ELSE\ c_2 = (let\ f = \vdash b\ in$
$(if\ f \in \{Some\ True,\ None\}\ then \vdash c_1\ else\ \{\}) \sqcup$
$(if\ f \in \{Some\ False,\ None\}\ then \vdash c_2\ else\ \{\}))$ |

$\vdash WHILE\ b\ DO\ c = (let\ f = \vdash b\ in$
$(if\ f \in \{Some\ False,\ None\}\ then\ \{[]\}\ else\ \{\}) \cup$
$(if\ f \in \{Some\ True,\ None\}\ then \vdash c\ else\ \{\}))$

```
definition ctyping1 :: com ⇒ state set ⇒ vname set ⇒ config
  (‹(⊢ - '(⊆ -, -'))› [51] 55) where
⊢ c (⊆ A, X) ≡ let F = {λx. [y←ys. fst y = x] | ys. ys ∈ ⊢ c} in
  ({λx. if f x = []
     then s x else case snd (last (f x)) of None ⇒ t x | Some i ⇒ i |
       f s t. f ∈ F ∧ s ∈ A},
   Univ?? A {x. ∀f ∈ F. if f x = []
     then x ∈ X else snd (last (f x)) ≠ None})
```

Finally, in the recursive definition of the main type system *ctyping2*, the equations dealing with the constructs included in the original language IMP are the same as in my previous paper [10], whereas the equations for the additional constructs of the considered language extension are as follows.

- The equation for an input instruction *IN* x sets the returned value to a *pass* verdict *Some* (B, Y) just in case each set of variables in the current scope is allowed to affect variable x in the associated set of program states. If so, then the sets B and Y are computed in the same way as with an assignment whose right-hand expression cannot be evaluated at compile time, since input values cannot be evaluated at compile time, too.
- The equation for an output instruction *OUT* x sets the returned value to a *pass* verdict *Some* (B, Y) just in case each set of variables in the current scope is allowed to affect variable x in the associated set of program states. If so, then the sets B and Y are computed in the same way as with a *SKIP* command, as output instructions leave the program state unchanged, too.
- The equation for a nondeterministic choice c_1 *OR* c_2 sets the returned value to a *pass* verdict *Some* (B, Y) just in case *pass* verdicts are returned for both branches. If so, then the sets B and Y are computed in the same way as with a conditional statement *IF* b *THEN* c_1 *ELSE* c_2 whose boolean condition b cannot be evaluated at compile time.

```
primrec btyping2-aux :: bexp ⇒ state set ⇒ vname set ⇒ state set option
  (‹(⊫ - '(⊆ -, -'))› [51] 55) where

⊫ Bc v (⊆ A, -) = Some (if v then A else {}) |

⊫ Not b (⊆ A, X) = (case ⊫ b (⊆ A, X) of
  Some B ⇒ Some (A − B) | - ⇒ None) |

⊫ And b₁ b₂ (⊆ A, X) = (case (⊫ b₁ (⊆ A, X), ⊫ b₂ (⊆ A, X)) of
```

```
 (Some B₁, Some B₂) ⇒ Some (B₁ ∩ B₂) | - ⇒ None) |

⊫ Less a₁ a₂ (⊆ A, X) = (if avars a₁ ∪ avars a₂ ⊆ state ∩ X
 then Some {s. s ∈ A ∧ aval a₁ s < aval a₂ s} else None)

definition btyping2 ::
 bexp ⇒ state set ⇒ vname set ⇒ state set × state set
 (‹(⊨ - '(⊆ -, -'))› [51] 55) where
⊨ b (⊆ A, X) ≡ case ⊫ b (⊆ A, X) of
 Some A' ⇒ (A', A − A') | - ⇒ (A, A)


abbreviation interf-set :: state set ⇒ vname set ⇒ vname set ⇒ bool
 (‹(-: - ⇝ -)› [51, 51, 51] 50) where
A: X ⇝ Y ≡ ∀ s ∈ A. ∀ x ∈ X. ∀ y ∈ Y. s: dom x ⇝ dom y

abbreviation atyping :: bool ⇒ aexp ⇒ vname set ⇒ bool
 (‹(- ⊨ - '(⊆ -'))› [51, 51] 50) where
v ⊨ a (⊆ X) ≡ avars a = {} ∨ avars a ⊆ state ∩ X ∧ v

definition univ-states-if :: state set ⇒ vname set ⇒ state set
 (‹(Univ? - -)› [51, 75] 75) where
Univ? A X ≡ if state ⊆ X then A else Univ A (⊆ {})


fun ctyping2 :: scope ⇒ com ⇒ state set ⇒ vname set ⇒ config option
 (‹(- ⊨ - '(⊆ -, -'))› [51, 51] 55) where

- ⊨ SKIP (⊆ A, X) = Some (A, Univ?? A X) |

(U, v) ⊨ x ::= a (⊆ A, X) =
 (if ∀ (B, Y) ∈ insert (Univ? A X, avars a) U. B: Y ⇝ {x}
  then Some (if x ∈ state ∧ A ≠ {}
   then if v ⊨ a (⊆ X)
    then ({s(x := aval a s) | s. s ∈ A}, insert x X) else (A, X − {x})
   else (A, Univ?? A X))
  else None) |

(U, v) ⊨ IN x (⊆ A, X) =
 (if ∀ (B, Y) ∈ U. B: Y ⇝ {x}
  then Some (if x ∈ state ∧ A ≠ {}
   then (A, X − {x}) else (A, Univ?? A X))
  else None) |

(U, v) ⊨ OUT x (⊆ A, X) =
 (if ∀ (B, Y) ∈ U. B: Y ⇝ {x}
  then Some (A, Univ?? A X)
  else None) |
```

$(U,\ v) \models c_1;;\ c_2\ (\subseteq A,\ X) =$
$\ (\mathit{case}\ (U,\ v) \models c_1\ (\subseteq A,\ X)\ \mathit{of}$
$\quad \mathit{Some}\ (B,\ Y) \Rightarrow (U,\ v) \models c_2\ (\subseteq B,\ Y)\ |\ \text{-} \Rightarrow \mathit{None})\ |$

$(U,\ v) \models c_1\ \mathit{OR}\ c_2\ (\subseteq A,\ X) =$
$\ (\mathit{case}\ ((U,\ v) \models c_1\ (\subseteq A,\ X),\ (U,\ v) \models c_2\ (\subseteq A,\ X))\ \mathit{of}$
$\quad (\mathit{Some}\ (C_1,\ Y_1),\ \mathit{Some}\ (C_2,\ Y_2)) \Rightarrow \mathit{Some}\ (C_1 \cup C_2,\ Y_1 \cap Y_2)\ |$
$\quad \text{-} \Rightarrow \mathit{None})\ |$

$(U,\ v) \models \mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2\ (\subseteq A,\ X) =$
$\ (\mathit{case}\ (\mathit{insert}\ (\mathit{Univ?}\ A\ X,\ \mathit{bvars}\ b)\ U,\ \models b\ (\subseteq A,\ X))\ \mathit{of}\ (U',\ B_1,\ B_2) \Rightarrow$
$\quad \mathit{case}\ ((U',\ v) \models c_1\ (\subseteq B_1,\ X),\ (U',\ v) \models c_2\ (\subseteq B_2,\ X))\ \mathit{of}$
$\quad\ \ (\mathit{Some}\ (C_1,\ Y_1),\ \mathit{Some}\ (C_2,\ Y_2)) \Rightarrow \mathit{Some}\ (C_1 \cup C_2,\ Y_1 \cap Y_2)\ |$
$\quad\ \ \text{-} \Rightarrow \mathit{None})\ |$

$(U,\ v) \models \mathit{WHILE}\ b\ \mathit{DO}\ c\ (\subseteq A,\ X) = (\mathit{case} \models b\ (\subseteq A,\ X)\ \mathit{of}\ (B_1,\ B_2) \Rightarrow$
$\ \ \mathit{case} \vdash c\ (\subseteq B_1,\ X)\ \mathit{of}\ (C,\ Y) \Rightarrow \mathit{case} \models b\ (\subseteq C,\ Y)\ \mathit{of}\ (B_1',\ B_2') \Rightarrow$
$\quad \mathit{if}\ \forall (B,\ W) \in \mathit{insert}\ (\mathit{Univ?}\ A\ X \cup \mathit{Univ?}\ C\ Y,\ \mathit{bvars}\ b)\ U.\ B\colon W \leadsto \mathit{UNIV}$
$\quad \mathit{then}\ \mathit{case}\ ((\{\},\ \mathit{False}) \models c\ (\subseteq B_1,\ X),\ (\{\},\ \mathit{False}) \models c\ (\subseteq B_1',\ Y))\ \mathit{of}$
$\quad\ \ (\mathit{Some}\ \text{-},\ \mathit{Some}\ \text{-}) \Rightarrow \mathit{Some}\ (B_2 \cup B_2',\ \mathit{Univ??}\ B_2\ X \cap Y)\ |$
$\quad\ \ \text{-} \Rightarrow \mathit{None}$
$\quad \mathit{else}\ \mathit{None})$

end

end

3 Idempotence of the auxiliary type system meant for loop bodies

theory *Idempotence*
imports *Definitions*
begin

As in my previous paper [10], the purpose of this section is to prove that the auxiliary type system *ctyping1* used to simulate the execution of loop bodies is idempotent, namely that if its output for a given input is the pair formed by *state set B* and *vname set Y*, then the output is the same if *B* and *Y* are fed back into the type system (lemma *ctyping1-idem*).

3.1 Local context proofs

context *noninterf*
begin

abbreviation *ctyping1-idem-lhs* **where**

ctyping1-idem-lhs s t t′ ys ys′ x ≡
if [*y*←*ys*′. *fst y* = *x*] = []
then if [*y*←*ys*. *fst y* = *x*] = []
then s x
else case snd (*last* [*y*←*ys*. *fst y* = *x*]) *of None* ⇒ *t x* | *Some i* ⇒ *i*
else case snd (*last* [*y*←*ys*′. *fst y* = *x*]) *of None* ⇒ *t*′ *x* | *Some i* ⇒ *i*

abbreviation *ctyping1-idem-rhs* **where**
ctyping1-idem-rhs f s t x ≡
if f x = []
then s x
else case snd (*last* (*f x*)) *of None* ⇒ *t x* | *Some i* ⇒ *i*

abbreviation *ctyping1-idem-pred* **where**
ctyping1-idem-pred s t t′ ys ys′ A (*S* :: *state-upd list set*) ≡ ∃ *f s*′.
(∃ *t*″. *ctyping1-idem-lhs s t t′ ys ys′* = *ctyping1-idem-rhs f s′ t*″) ∧
(∀ *x*. (*f x* = [] ⟷ [*y*←*ys* @ *ys*′. *fst y* = *x*] = []) ∧
(*f x* ≠ [] ⟶ *last* (*f x*) = *last* [*y*←*ys* @ *ys*′. *fst y* = *x*])) ∧
(∃ *ys*″. *f* = (λ*x*. [*y*←*ys*″. *fst y* = *x*]) ∧ *ys*″ ∈ *S*) ∧ *s*′ ∈ *A*

lemma *ctyping1-merge-aux-no-nil*:
ws ∈ *A* ⨆ *B* ⟹ *ws* ≠ []
by (*erule ctyping1-merge-aux.cases*, *simp-all*)

lemma *ctyping1-merge-aux-empty-lhs*:
{} ⨆ *B* = {[(*ys*, *False*)] | *ys*. *ys* ∈ *B*}
by (*rule equalityI*, *clarify*, *erule ctyping1-merge-aux.induct*, *auto*)

lemma *ctyping1-merge-aux-empty-rhs*:
A ⨆ {} = {[(*xs*, *True*)] | *xs*. *xs* ∈ *A*}
by (*rule equalityI*, *clarify*, *erule ctyping1-merge-aux.induct*, *auto*)

lemma *ctyping1-merge-empty-lhs*:
{} ⊔ *B* = *B*
by (*force simp*: *ctyping1-merge-def ctyping1-merge-aux-empty-lhs*)

lemma *ctyping1-merge-empty-rhs*:
A ⊔ {} = *A*
by (*force simp*: *ctyping1-merge-def ctyping1-merge-aux-empty-rhs*)

lemma *ctyping1-aux-nonempty*:
⊢ *c* ≠ {}
by (*induction c*, *auto simp*: *Let-def ctyping1-merge-def*
ctyping1-merge-append-def ctyping1-append-def, *fastforce*)

lemma *ctyping1-merge-idem-fst*:
assumes

```
    A: ⋀ys ys'. ys ∈ ⊢ c_1 ⟹ ys' ∈ ⊢ c_1 ⟹
      ctyping1-idem-pred s t t' ys ys' A (⊢ c_1) and
    B: ⋀ys ys'. ys ∈ ⊢ c_2 ⟹ ys' ∈ ⊢ c_2 ⟹
      ctyping1-idem-pred s t t' ys ys' A (⊢ c_2) and
    C: s ∈ A and
    D: ys ∈ ⊢ c_1 ⊔ ⊢ c_2 and
    E: ys' ∈ ⊢ c_1 ⊔ ⊢ c_2
  shows ctyping1-idem-pred s t t' ys ys' A (⊢ c_1 ⊔ ⊢ c_2)
proof −
  obtain ws where ws ∈ ⊢ c_1 ⨆ ⊢ c_2 and ys = concat (map fst ws)
    using D by (auto simp: ctyping1-merge-def)
  thus ?thesis
  proof (induction ws arbitrary: ys rule: list.induct,
   blast dest: ctyping1-merge-aux-no-nil)
    fix w ws ys
    assume
      F: ⋀xs. ws ∈ ⊢ c_1 ⨆ ⊢ c_2 ⟹ xs = concat (map fst ws) ⟹
        ctyping1-idem-pred s t t' xs ys' A (⊢ c_1 ⊔ ⊢ c_2) and
      G: w # ws ∈ ⊢ c_1 ⨆ ⊢ c_2
    assume ys = concat (map fst (w # ws))
    hence H: ys = fst w @ concat (map fst ws)
      (is ys = ?x @ ?xs)
      by simp
    have ctyping1-idem-pred s t t' ?xs ys' A (⊢ c_1 ⊔ ⊢ c_2)
    proof (cases ws)
      case Nil
      show ?thesis
        apply (rule exI [of - λx. [y←ys'. fst y = x]])
        apply (rule exI [of - s])
        apply (rule conjI)
         apply (rule exI [of - t'])
        by (auto simp: C E Nil)
    next
      case Cons
      have ws ∈ ⊢ c_1 ⨆ ⊢ c_2
        using G by (rule ctyping1-merge-aux.cases, simp-all add: Cons)
      thus ?thesis
        using F by simp
    qed
    then obtain f and s' and t'' and ys'' where
      I: ctyping1-idem-lhs s t t' ?xs ys' =
        ctyping1-idem-rhs f s' t'' and
      J: ∀x. (f x = [] ⟷ [y←?xs @ ys'. fst y = x] = []) ∧
        (f x ≠ [] ⟶ last (f x) = last [y←?xs @ ys'. fst y = x]) and
      K: f = (λx. [y←ys''. fst y = x]) and
      L: ys'' ∈ ⊢ c_1 ⊔ ⊢ c_2 and
      M: s' ∈ A
      by auto
    obtain ws'' where
```

```
  N: ws'' ∈ ⊢ c₁ ⨆ ⊢ c₂ and
  O: ys'' = concat (map fst ws'')
  using L by (auto simp: ctyping1-merge-def)
show ctyping1-idem-pred s t t' ys ys' A (⊢ c₁ ⊔ ⊢ c₂)
proof (cases w ∈ set ws'')
  assume P: w ∈ set ws''
  show ?thesis
    apply (rule exI [of - f])
    apply (rule exI [of - s'])
    apply (rule conjI)
     apply (rule exI [of - t''])
     apply (rule ext)
    subgoal for x
    proof (cases [y←ys'. fst y = x], cases [y←ys. fst y = x] = [])
      case Cons
      thus ctyping1-idem-lhs s t t' ys ys' x =
        ctyping1-idem-rhs f s' t'' x
        by (insert fun-cong [OF I, of x], simp)
    next
      case Nil
      moreover case True
      ultimately show ctyping1-idem-lhs s t t' ys ys' x =
        ctyping1-idem-rhs f s' t'' x
        using H by (insert fun-cong [OF I, of x], simp)
    next
      case Nil
      case False
      hence [y←?x. fst y = x] ≠ [] ∨ [y←?xs. fst y = x] ≠ []
        using H by simp
      moreover {
        assume [y←?x. fst y = x] ≠ []
        hence [y←ys''. fst y = x] ≠ []
          using O and P by (auto simp: filter-concat)
        hence [y←?xs. fst y = x] ≠ []
          using J and K and Nil by simp
      }
      ultimately have Q: [y←?xs. fst y = x] ≠ [] ..
      hence (case snd (last [y←?xs. fst y = x]) of
        None ⇒ t x | Some i ⇒ i) = ctyping1-idem-rhs f s' t'' x
        using Nil by (insert fun-cong [OF I, of x], simp)
      moreover have last [y←?xs. fst y = x] = last [y←ys. fst y = x]
        using H and Q by simp
      ultimately show ctyping1-idem-lhs s t t' ys ys' x =
        ctyping1-idem-rhs f s' t'' x
        using Nil and False by simp
    qed
    apply (rule conjI)
    subgoal
    proof -
```

```
      show ∀ x. (f x = [] ⟷ [y←ys @ ys'. fst y = x] = []) ∧
        (f x ≠ [] ⟶ last (f x) = last [y←ys @ ys'. fst y = x])
        (is ∀ x. ?P x ∧ ?Q x)
      proof
        fix x
        have ?P x
        proof
          assume Q: f x = []
          hence [y←?xs @ ys'. fst y = x] = []
            using J by simp
          moreover have [y←?x. fst y = x] = []
            using K and O and P and Q by (simp add: filter-concat)
          ultimately show [y←ys @ ys'. fst y = x] = []
            using H by simp
        qed (insert H J, simp)
        moreover have ?Q x
          using J and H by simp
        ultimately show ?P x ∧ ?Q x ..
      qed
    qed
    by (insert K L M, blast)
next
  assume P: w ∉ set ws''
  let ?y = fst (hd ws'')
  show ?thesis
  proof (cases snd w = snd (hd ws''))
    assume Q: snd w = snd (hd ws'')
    hence snd w ∧ snd (hd ws'') ∨ ¬ snd w ∧ ¬ snd (hd ws'')
      (is ?P ∨ ?Q)
      by simp
    moreover {
      assume ?P
      have ?x ∈ ⊢ c₁
        using G by (rule ctyping1-merge-aux.cases, insert ‹?P›, simp-all)
      moreover have ?y ∈ ⊢ c₁
        using N by (rule ctyping1-merge-aux.cases, insert ‹?P›, simp-all)
      ultimately have ctyping1-idem-pred s t t' ?x ?y A (⊢ c₁)
        using A by simp
    }
    moreover {
      assume ?Q
      have ?x ∈ ⊢ c₂
        using G by (rule ctyping1-merge-aux.cases, insert ‹?Q›, simp-all)
      moreover have ?y ∈ ⊢ c₂
        using N by (rule ctyping1-merge-aux.cases, insert ‹?Q›, simp-all)
      ultimately have ctyping1-idem-pred s t t' ?x ?y A (⊢ c₂)
        using B by simp
    }
    ultimately obtain f₀ and s₀ and t₀ and ys₀ where
```

```
  R: ctyping1-idem-lhs s t t' ?x ?y =
    ctyping1-idem-rhs f_0 s_0 t_0 and
  S: ∀ x. (f_0 x = [] ⟷ [y←?x @ ?y. fst y = x] = []) ∧
    (f_0 x ≠ [] ⟶ last (f_0 x) = last [y←?x @ ?y. fst y = x]) and
  T: f_0 = (λx. [y←ys_0. fst y = x]) and
  U: ys_0 ∈ ⊢ c_1 ∧ snd w ∨ ys_0 ∈ ⊢ c_2 ∧ ¬ snd w
  by auto
from U obtain w_0 where
  V: [w_0] ∈ ⊢ c_1 ⨆ ⊢ c_2 and
  W: ys_0 = fst w_0 and
  X: snd w_0 = snd w
  by fastforce
show ?thesis
proof (cases w_0 ∈ set ws'')
  assume Y: w_0 ∈ set ws''
  show ?thesis
    apply (rule exI [of - f])
    apply (rule exI [of - s'])
    apply (rule conjI)
     apply (rule exI [of - t''])
     apply (rule ext)
    subgoal for x
    proof (cases [y←ys'. fst y = x], cases [y←ys. fst y = x] = [])
      case Cons
      thus ctyping1-idem-lhs s t t' ys ys' x =
        ctyping1-idem-rhs f s' t'' x
        by (insert fun-cong [OF I, of x], simp)
    next
      case Nil
      moreover case True
      ultimately show ctyping1-idem-lhs s t t' ys ys' x =
        ctyping1-idem-rhs f s' t'' x
        using H by (insert fun-cong [OF I, of x], simp)
    next
      case Nil
      case False
      hence [y←?x. fst y = x] ≠ [] ∨ [y←?xs. fst y = x] ≠ []
        using H by simp
      moreover {
        assume [y←?x. fst y = x] ≠ []
        hence [y←ys_0. fst y = x] ≠ []
          using S and T by simp
        hence [y←ys''. fst y = x] ≠ []
        using O and W and Y by (auto simp: filter-concat)
        hence [y←?xs. fst y = x] ≠ []
          using J and K and Nil by simp
      }
      ultimately have Z: [y←?xs. fst y = x] ≠ [] ..
      hence (case snd (last [y←?xs. fst y = x]) of
```

```
        None ⇒ t x | Some i ⇒ i) = ctyping1-idem-rhs f s' t'' x
        using Nil by (insert fun-cong [OF I, of x], simp)
      moreover have last [y←?xs. fst y = x] = last [y←ys. fst y = x]
        using H and Z by simp
      ultimately show ctyping1-idem-lhs s t t' ys ys' x =
        ctyping1-idem-rhs f s' t'' x
        using Nil and False by simp
    qed
    apply (rule conjI)
    subgoal
    proof −
      show ∀x. (f x = [] ⟷ [y←ys @ ys'. fst y = x] = []) ∧
        (f x ≠ [] ⟶ last (f x) = last [y←ys @ ys'. fst y = x])
        (is ∀x. ?P x ∧ ?Q x)
      proof
        fix x
        have ?P x
        proof
          assume Z: f x = []
          hence [y←?xs @ ys'. fst y = x] = []
            using J by simp
          moreover have [y←ys''. fst y = x] = []
            using K and Z by simp
          hence [y←ys0. fst y = x] = []
            using O and W and Y by (simp add: filter-concat)
          hence [y←?x. fst y = x] = []
            using S and T by simp
          ultimately show [y←ys @ ys'. fst y = x] = []
            using H by simp
        qed (insert H J, simp)
        moreover have ?Q x
          using J and H by simp
        ultimately show ?P x ∧ ?Q x ..
      qed
    qed
    by (insert K L M, blast)
next
  assume Y: w0 ∉ set ws''
  let ?ws = w0 # tl ws''
  {
    assume Z: tl ws'' ≠ []
    have tl ws'' ∈ ⊢ c1 ⨆ ⊢ c2
      using N by (rule ctyping1-merge-aux.cases, insert Z, simp-all)
    moreover have snd (hd (tl ws'')) = (¬ snd w)
      using N by (rule ctyping1-merge-aux.cases, insert Q Z, simp-all)
    moreover have w0 ∉ set (tl ws'')
      using Y by (cases ws'', simp-all)
    ultimately have ?ws ∈ ⊢ c1 ⨆ ⊢ c2
      by (cases w0, insert U W X, auto)
```

```
}
hence Z: ?ws ∈ ⊢ c1 ⨆ ⊢ c2
  by (cases tl ws'', insert V, simp-all)
let ?ys = concat (map fst (tl ws''))
let ?f = λx. [y←concat (map fst ?ws). fst y = x]
let ?t = λx. if f x = [] then t0 x else t'' x
have AA: ws'' = hd ws'' # tl ws''
  by (insert ctyping1-merge-aux-no-nil [OF N], simp)
have AB: ys'' = ?y @ ?ys
  using O by (subst (asm) AA, simp)
have AC: ∀x. [y←?ys. fst y = x] ≠ [] ⟶
  last (?f x) = last (f x)
  using K and O by (subst (asm) AA, simp)
have AD: ∀x. [y←?ys. fst y = x] = [] ∧ [y←?y. fst y = x] ≠ [] ⟶
  last (?f x) = last (f x)
  (is ∀x. ?P x ∧ ?Q x ⟶ -)
proof clarify
  fix x
  assume ?P x and ?Q x
  moreover from this and S and T have
   last [y←ys0. fst y = x] = last [y←?x @ ?y. fst y = x]
    by simp
  ultimately show last (?f x) = last (f x)
    using K and W and AB by simp
qed
show ?thesis
  apply (rule exI [of - ?f])
  apply (rule exI [of - s'])
  apply (rule conjI)
   apply (rule exI [of - ?t])
   apply (rule ext)
  subgoal for x
  proof (cases [y←ys'. fst y = x], cases [y←?xs. fst y = x] = [])
    case Cons
    hence AE:
     (case snd (last [y←ys'. fst y = x]) of
        None ⇒ t' x | Some i ⇒ i) =
      (case snd (last (f x)) of None ⇒ ?t x | Some i ⇒ i)
      using J by (insert fun-cong [OF I, of x], simp)
    show ctyping1-idem-lhs s t t' ys ys' x =
      ctyping1-idem-rhs ?f s' ?t x
    proof (cases [y←?ys. fst y = x] ≠ [])
      case True
      thus ?thesis
        using AC and AE and Cons by simp
    next
      case False
      moreover have [y←ys''. fst y = x] ≠ []
        using J and K and Cons by simp
```

```
    ultimately have [y←?y. fst y = x] ≠ []
      using AB by simp
    moreover from this have ?f x ≠ []
      using S and T and W by simp
    ultimately show ?thesis
      using AD and AE and Cons and False by simp
  qed
next
  case Nil
  moreover case False
  ultimately have
   (case snd (last [y←ys. fst y = x]) of
      None ⇒ t x | Some i ⇒ i) =
    (case snd (last (f x)) of None ⇒ ?t x | Some i ⇒ i)
    using J and H by (insert fun-cong [OF I, of x], simp)
  moreover have
    AE: [y←?y. fst y = x] ≠ [] ∨ [y←?ys. fst y = x] ≠ []
    (is - ∨ ?P)
    using J and K and AB and False by auto
  hence ?f x ≠ []
    using S and T and W by (cases ?P, simp-all)
  moreover have last (?f x) = last (f x)
    using AC and AD and AE by blast
  ultimately show ctyping1-idem-lhs s t t' ys ys' x =
    ctyping1-idem-rhs ?f s' ?t x
    using H and Nil and False by auto
next
  case Nil
  moreover case True
  ultimately have AE: f x = []
    using J by simp
  hence AF: [y←?y @ ?ys. fst y = x] = []
    using K and AB by simp
  show ctyping1-idem-lhs s t t' ys ys' x =
    ctyping1-idem-rhs ?f s' ?t x
  proof (cases [y←?x. fst y = x] = [])
    assume AG: [y←?x. fst y = x] = []
    moreover from J and AE have s x = s' x
      by (insert fun-cong [OF I, of x], simp)
    moreover have [y←ys_0. fst y = x] = []
      using S and T and AF and AG by simp
    hence ?f x = []
      using W and AF by simp
    ultimately show ?thesis
      using H and Nil and True by simp
  next
    assume AG: [y←?x. fst y = x] ≠ []
    moreover from this and S and AE and AF have
     (case snd (last [y←?x. fst y = x]) of
```

```
        None ⇒ t x | Some i ⇒ i) =
      (case snd (last (f_0 x)) of None ⇒ ?t x | Some i ⇒ i)
      by (insert fun-cong [OF R, of x], simp)
    moreover have [y←ys_0. fst y = x] ≠ []
      using S and T and AG by simp
    hence ?f x ≠ []
      using W by simp
    moreover have last (?f x) = last (f_0 x)
      using T and W and AF by simp
    ultimately show ?thesis
      using H and Nil and True by auto
  qed
qed
apply (rule conjI)
subgoal
proof −
  show ∀ x. (?f x = [] ⟷ [y←ys @ ys′. fst y = x] = []) ∧
    (?f x ≠ [] ⟶ last (?f x) = last [y←ys @ ys′. fst y = x])
    (is ∀ x. ?P x ∧ ?Q x)
  proof
    fix x
    have AE: ?P x
    proof
      assume AF: ?f x = []
      hence [y←?x @ ?y. fst y = x] = []
        using S and T and W by simp
      moreover from this and J and K and AB and AF have
       [y←?xs @ ys′. fst y = x] = []
        by auto
      ultimately show [y←ys @ ys′. fst y = x] = []
        using H by simp
    next
      assume [y←ys @ ys′. fst y = x] = []
      hence [y←?x @ ?y @ ?ys. fst y = x] = []
        using H and J and K and AB by simp
      moreover from this have [y←ys_0. fst y = x] = []
        using S and T by simp
      ultimately show ?f x = []
        using W by simp
    qed
    moreover have ?Q x
    proof (clarify, cases [y←?y @ ?ys. fst y = x])
      case Nil
      hence last (?f x) = last (f_0 x)
        using T and W by simp
      moreover assume ?f x ≠ []
      hence [y←ys @ ys′. fst y = x] ≠ []
        using AE by blast
      hence [y←?x @ ?y @ ?ys. fst y = x] ≠ []
```

```
              using H and J and K and AB by simp
            ultimately have last (?f x) = last [y←?x. fst y = x]
              using S and Nil by simp
            moreover have [y←?xs @ ys'. fst y = x] = []
              using J and K and AB and Nil by simp
            ultimately show
             last (?f x) = last [y←ys @ ys'. fst y = x]
              using H by simp
          next
            case Cons
            hence [y←?y. fst y = x] ≠ [] ∨
             [y←?ys. fst y = x] ≠ []
              by auto
            hence last (?f x) = last (f x)
              using AC and AD by blast
            moreover have f x ≠ []
              using K and AB and Cons by simp
            ultimately show
             last (?f x) = last [y←ys @ ys'. fst y = x]
              using H and J by simp
          qed
          ultimately show ?P x ∧ ?Q x ..
        qed
      qed
      by (rule conjI, rule exI [of - concat (map fst ?ws)],
       insert M Z, auto simp only: ctyping1-merge-def)
  qed
next
  assume snd w ≠ snd (hd ws'')
  hence snd w ∧ ¬ snd (hd ws'') ∨ ¬ snd w ∧ snd (hd ws'')
    (is ?P ∨ ?Q)
    by simp
  moreover {
    assume ?P
    moreover have ?x ∈ ⊢ c₁
      using G by (rule ctyping1-merge-aux.cases, insert ‹?P›, simp-all)
    moreover have (?x, True) ∉ set ws''
      using P and ‹?P› by (cases w, simp)
    ultimately have w # ws'' ∈ ⊢ c₁ ⨆ ⊢ c₂
      using N by (cases w, auto)
  }
  moreover {
    assume ?Q
    moreover have ?x ∈ ⊢ c₂
      using G by (rule ctyping1-merge-aux.cases, insert ‹?Q›, simp-all)
    moreover have (?x, False) ∉ set ws''
      using P and ‹?Q› by (cases w, simp)
    ultimately have w # ws'' ∈ ⊢ c₁ ⨆ ⊢ c₂
      using N by (cases w, auto)
```

```
}
ultimately have Q: w # ws'' ∈ ⊢ c1 ⨆ ⊢ c2
 (is ?ws ∈ -) ..
let ?f = λx. [y←concat (map fst ?ws). fst y = x]
let ?t = λx. if f x = [] then t x else t'' x
show ?thesis
  apply (rule exI [of - ?f])
  apply (rule exI [of - s'])
  apply (rule conjI)
   apply (rule exI [of - ?t])
   apply (rule ext)
  subgoal for x
  proof (cases [y←ys'. fst y = x], cases [y←?xs. fst y = x] = [])
    case Cons
    moreover from this have
     (case snd (last [y←ys'. fst y = x]) of
       None ⇒ t' x | Some i ⇒ i) =
      (case snd (last (f x)) of None ⇒ ?t x | Some i ⇒ i)
      using J by (insert fun-cong [OF I, of x], simp)
    moreover have ?f x ≠ []
      using J and K and O and Cons by simp
    moreover have f x ≠ []
      using J and Cons by simp
    hence last (?f x) = last (f x)
      using K and O by simp
    ultimately show ctyping1-idem-lhs s t t' ys ys' x =
      ctyping1-idem-rhs ?f s' ?t x
      by auto
  next
    case Nil
    moreover case False
    ultimately have
     (case snd (last [y←ys. fst y = x]) of
       None ⇒ t x | Some i ⇒ i) =
      (case snd (last (f x)) of None ⇒ ?t x | Some i ⇒ i)
      using J and H by (insert fun-cong [OF I, of x], simp)
    moreover have ?f x ≠ []
      using J and K and O and False by simp
    moreover have f x ≠ []
      using J and False by simp
    hence last (?f x) = last (f x)
      using K and O by simp
    ultimately show ctyping1-idem-lhs s t t' ys ys' x =
      ctyping1-idem-rhs ?f s' ?t x
      using H and Nil and False by auto
  next
    case Nil
    moreover case True
    ultimately have R: f x = []
```

```
    using J by simp
  show ctyping1-idem-lhs s t t' ys ys' x =
    ctyping1-idem-rhs ?f s' ?t x
  proof (cases [y←?x. fst y = x] = [])
    assume [y←?x. fst y = x] = []
    moreover have [y←ys''. fst y = x] = []
      using K and R by simp
    ultimately have ?f x = []
      using O by simp
    moreover from J and R have s x = s' x
      by (insert fun-cong [OF I, of x], simp)
    ultimately show ?thesis
      using H and Nil and True by simp
  next
    assume [y←?x. fst y = x] ≠ []
    moreover have last [y←ys. fst y = x] = last [y←?x. fst y = x]
      using H and True by simp
    moreover have last (?f x) = last [y←?x. fst y = x]
      using K and O and R by simp
    ultimately show ?thesis
      using H and R and Nil by simp
  qed
qed
apply (rule conjI)
subgoal
proof -
  show ∀x. (?f x = [] ⟷ [y←ys @ ys'. fst y = x] = []) ∧
    (?f x ≠ [] ⟶ last (?f x) = last [y←ys @ ys'. fst y = x])
    (is ∀x. ?P x ∧ ?Q x)
  proof
    fix x
    have ?P x
    proof
      assume ?f x = []
      hence [y←?x @ ys''. fst y = x] = []
        using O by simp
      moreover from this have [y←?xs @ ys'. fst y = x] = []
        using J and K by simp
      ultimately show [y←ys @ ys'. fst y = x] = []
        using H by simp
    next
      assume [y←ys @ ys'. fst y = x] = []
      hence [y←?x @ ?xs @ ys'. fst y = x] = []
        using H by simp
      moreover from this have [y←ys''. fst y = x] = []
        using J and K by simp
      ultimately show ?f x = []
        using O by simp
    qed
```

```
              moreover have ?Q x
              proof (clarify, cases [y←ys''. fst y = x])
                case Nil
                hence last (?f x) = last [y←?x. fst y = x]
                  using O by simp
                moreover have [y←?xs @ ys'. fst y = x] = []
                  using J and K and Nil by simp
                hence
                 last [y←ys @ ys'. fst y = x] = last [y←?x. fst y = x]
                  using H by simp
                ultimately show
                 last (?f x) = last [y←ys @ ys'. fst y = x]
                  by simp
              next
                case Cons
                hence last (?f x) = last (f x)
                  using K and O by simp
                moreover have R: f x ≠ []
                  using K and Cons by simp
                hence last [y←?xs @ ys'. fst y = x] = last (f x)
                  using J by simp
                moreover have [y←?xs @ ys'. fst y = x] ≠ []
                  using J and R by simp
                ultimately show
                 last (?f x) = last [y←ys @ ys'. fst y = x]
                  using H by simp
              qed
              ultimately show ?P x ∧ ?Q x ..
            qed
          qed
          by (rule conjI, rule exI [of - concat (map fst ?ws)],
           insert M Q, auto simp only: ctyping1-merge-def)
      qed
    qed
  qed
qed

lemma ctyping1-merge-append-idem-fst:
  assumes
    A: ⋀ys ys'. ys ∈ ⊢ c1 ⟹ ys' ∈ ⊢ c1 ⟹
      ctyping1-idem-pred s t t' ys ys' A (⊢ c1) and
    B: ⋀ys ys'. ys ∈ ⊢ c2 ⟹ ys' ∈ ⊢ c2 ⟹
      ctyping1-idem-pred s t t' ys ys' A (⊢ c2) and
    C: s ∈ A and
    D: ys ∈ ⊢ c1 ⊔@ ⊢ c2 and
    E: ys' ∈ ⊢ c1 ⊔@ ⊢ c2
  shows ctyping1-idem-pred s t t' ys ys' A (⊢ c1 ⊔@ ⊢ c2)
  apply (subst ctyping1-merge-append-def)
  apply (split if-split)
```

```
apply (rule conjI)
subgoal
proof
  assume F: card (⊢ c2) = Suc 0
  with D obtain ys1 and ys2 where
    G: ys = ys1 @ ys2 and
    H: ys1 ∈ ⊢ c1 and
    I: ys2 ∈ ⊢ c2
    by (auto simp: ctyping1-merge-append-def ctyping1-append-def)
  from E and F obtain ys1' and ys2' where
    J: ys' = ys1' @ ys2' and
    K: ys1' ∈ ⊢ c1 and
    L: ys2' ∈ ⊢ c2
    by (auto simp: ctyping1-merge-append-def ctyping1-append-def)
  have M: ys2' = ys2
    using F and I and L by (fastforce simp: card-1-singleton-iff)
  obtain f and s' and t'' and ys1'' where
    N: ctyping1-idem-lhs s t t' ys1 ys1' =
      ctyping1-idem-rhs f s' t'' and
    O: ∀ x. (f x = [] ⟷ [y←ys1 @ ys1'. fst y = x] = []) ∧
      (f x ≠ [] ⟶ last (f x) = last [y←ys1 @ ys1'. fst y = x]) and
    P: f = (λx. [y←ys1''. fst y = x]) and
    Q: ys1'' ∈ ⊢ c1 and
    R: s' ∈ A
    using A [OF H K] by auto
  let ?f = λx. [y←ys1'' @ ys2. fst y = x]
  let ?t = λx. if [y←ys2. fst y = x] = [] then t'' x else t' x
  show ctyping1-idem-pred s t t' ys ys' A (⊢ c1 @ ⊢ c2)
    apply (rule exI [of - ?f])
    apply (rule exI [of - s'])
    apply (rule conjI)
     apply (rule exI [of - ?t])
     apply (rule ext)
    subgoal for x
    proof (cases [y←ys2. fst y = x], cases f x = [])
      case Nil
      moreover case True
      ultimately have s x = s' x
        using O by (insert fun-cong [OF N, of x], simp)
      moreover have [y←ys'. fst y = x] = []
        using J and M and O and Nil and True by simp
      moreover have [y←ys. fst y = x] = []
        using G and O and Nil and True by simp
      moreover have ?f x = []
        using P and Nil and True by simp
      ultimately show ctyping1-idem-lhs s t t' ys ys' x =
        ctyping1-idem-rhs ?f s' ?t x
        by simp
    next
```

```
  case Nil
  moreover from this have
   [y←ys'. fst y = x] = [y←ys1'. fst y = x]
    using J and M by simp
  moreover have [y←ys. fst y = x] = [y←ys1. fst y = x]
    using G and Nil by simp
  moreover case False
  moreover from this have ?f x ≠ []
    using P by simp
  moreover have last (?f x) = last (f x)
    using P and Nil by simp
  ultimately show ctyping1-idem-lhs s t t' ys ys' x =
    ctyping1-idem-rhs ?f s' ?t x
    by (insert fun-cong [OF N, of x], auto)
next
  case Cons
  moreover from this have [y←ys'. fst y = x] ≠ []
    using J and M by simp
  moreover have
   last [y←ys'. fst y = x] = last [y←ys2. fst y = x]
    using J and M and Cons by simp
  ultimately show ctyping1-idem-lhs s t t' ys ys' x =
    ctyping1-idem-rhs ?f s' ?t x
    by simp
qed
apply (rule conjI)
subgoal
proof −
  show ∀ x. (?f x = [] ⟷ [y←ys @ ys'. fst y = x] = []) ∧
    (?f x ≠ [] ⟶ last (?f x) = last [y←ys @ ys'. fst y = x])
    (is ∀ x. ?P x ∧ ?Q x)
  proof
    fix x
    have ?P x
      using G and J and M and O and P by auto
    moreover have ?Q x
    proof (clarify, cases [y←ys2. fst y = x])
      case Nil
      moreover assume ?f x ≠ []
      ultimately have
       last (?f x) = last [y←ys1 @ ys1'. fst y = x]
        using O and P by simp
      thus last (?f x) = last [y←ys @ ys'. fst y = x]
        using G and J and M and Nil by simp
    next
      case Cons
      thus last (?f x) = last [y←ys @ ys'. fst y = x]
        using J and M by simp
    qed
```

```
        ultimately show ?P x ∧ ?Q x ..
      qed
    qed
    by (rule conjI, rule exI [of - ys_1'' @ ys_2],
     insert I Q R, auto simp: ctyping1-append-def)
qed
subgoal
proof
  assume F: card (⊢ c_2) ≠ Suc 0
  with D obtain ws and xs where
    G: ys = ws @ xs and
    H: ws ∈ ⊢ c_1 ⊔ ⊢ c_2 and
    I: xs ∈ ⊢ c_2
    by (auto simp: ctyping1-merge-append-def ctyping1-append-def)
  from E and F obtain ws' and xs' where
    J: ys' = ws' @ xs' and
    K: ws' ∈ ⊢ c_1 ⊔ ⊢ c_2 and
    L: xs' ∈ ⊢ c_2
    by (auto simp: ctyping1-merge-append-def ctyping1-append-def)
  from I have [(xs, False)] ∈ ⊢ c_1 ⨆ ⊢ c_2 ..
  hence M: xs ∈ ⊢ c_1 ⊔ ⊢ c_2
    by (force simp: ctyping1-merge-def)
  obtain f and s' and r and zs where
    N: ctyping1-idem-lhs s t t' ws xs =
      ctyping1-idem-rhs f s' r and
    O: ∀x. (f x = [] ⟷ [y←ws @ xs. fst y = x] = []) ∧
      (f x ≠ [] ⟶ last (f x) = last [y←ws @ xs. fst y = x]) and
    P: f = (λx. [y←zs. fst y = x]) and
    Q: zs ∈ ⊢ c_1 ⊔ ⊢ c_2 and
    R: s' ∈ A
    using ctyping1-merge-idem-fst [OF A B C H M] by auto
  obtain f' and s'' and r' and zs' where
    S: ctyping1-idem-lhs s t t' zs ws' =
      ctyping1-idem-rhs f' s'' r' and
    T: ∀x. (f' x = [] ⟷ [y←zs @ ws'. fst y = x] = []) ∧
      (f' x ≠ [] ⟶ last (f' x) = last [y←zs @ ws'. fst y = x]) and
    U: f' = (λx. [y←zs'. fst y = x]) and
    V: zs' ∈ ⊢ c_1 ⊔ ⊢ c_2 and
    W: s'' ∈ A
    using ctyping1-merge-idem-fst [OF A B C Q K] by auto
  let ?f = λx. [y←zs' @ xs'. fst y = x]
  let ?t = λx. if [y←xs'. fst y = x] = [] then r' x else t' x
  show ctyping1-idem-pred s t t' ys ys' A (⊢ c_1 ⊔ ⊢ c_2 @ ⊢ c_2)
    apply (rule exI [of - ?f])
    apply (rule exI [of - s''])
    apply (rule conjI)
     apply (rule exI [of - ?t])
     apply (rule ext)
    subgoal for x
```

```
proof (cases [y←xs'. fst y = x], cases f' x = [])
  case Nil
  moreover case True
  hence s x = s'' x
    using T by (insert fun-cong [OF S, of x], simp)
  moreover have [y←ys'. fst y = x] = []
    using J and T and Nil and True by simp
  moreover have [y←zs. fst y = x] = []
    using T and True by simp
  hence [y←ys. fst y = x] = []
    using G and O and P by simp
  moreover have ?f x = []
    using U and Nil and True by simp
  ultimately show ctyping1-idem-lhs s t t' ys ys' x =
    ctyping1-idem-rhs ?f s'' ?t x
    by simp
next
  case Nil
  moreover from this have
    X: [y←ys'. fst y = x] = [y←ws'. fst y = x]
    using J by simp
  moreover case False
  moreover have
   [y←zs. fst y = x] ≠ [] ∧ [y←ys. fst y = x] ≠ [] ∧
      last [y←ys. fst y = x] = last [y←zs. fst y = x]
    (is ?P ∧ ?Q ∧ ?R) if
    a: [y←ys'. fst y = x] = []
  proof -
    have ?P
      using T and X and False and a by simp
    moreover from this have ?Q
      using G and O and P by simp
    moreover have ?R
      using G and O and P and ‹?P› by simp
    ultimately show ?thesis
      by simp
  qed
  moreover have ?f x ≠ []
    using U and False by simp
  moreover have last (?f x) = last (f' x)
    using U and Nil by simp
  ultimately show ctyping1-idem-lhs s t t' ys ys' x =
    ctyping1-idem-rhs ?f s'' ?t x
    by (insert fun-cong [OF S, of x], auto)
next
  case Cons
  moreover from this have [y←ys'. fst y = x] ≠ []
    using J by simp
  moreover have
```

```
   last [y←ys′. fst y = x] = last [y←xs′. fst y = x]
    using J and Cons by simp
  ultimately show ctyping1-idem-lhs s t t′ ys ys′ x =
    ctyping1-idem-rhs ?f s′′ ?t x
    by simp
qed
apply (rule conjI)
subgoal
proof −
  show ∀ x. (?f x = [] ⟷ [y←ys @ ys′. fst y = x] = []) ∧
    (?f x ≠ [] ⟶ last (?f x) = last [y←ys @ ys′. fst y = x])
    (is ∀ x. ?P x ∧ ?Q x)
  proof
    fix x
    have ?P x
    proof
      assume [y←zs′ @ xs′. fst y = x] = []
      moreover from this have [y←zs @ ws′. fst y = x] = []
        using T and U by simp
      moreover from this have [y←ws @ xs. fst y = x] = []
        using O and P by simp
      ultimately show [y←ys @ ys′. fst y = x] = []
        using G and J by simp
    next
      assume [y←ys @ ys′. fst y = x] = []
      hence [y←ws @ xs @ ws′ @ xs′. fst y = x] = []
        using G and J by simp
      moreover from this have [y←zs. fst y = x] = []
        using O and P by simp
      ultimately show [y←zs′ @ xs′. fst y = x] = []
        using T and U by simp
    qed
    moreover have ?Q x
    proof (clarify, cases [y←xs′. fst y = x])
      case Nil
      moreover assume ?f x ≠ []
      ultimately have X: f′ x ≠ []
        using U by simp
      hence Y: last (?f x) = last [y←zs @ ws′. fst y = x]
        using T and U and Nil by simp
      show last (?f x) = last [y←ys @ ys′. fst y = x]
      proof (cases [y←ws′. fst y = x] = [])
        case True
        moreover from this have f x ≠ []
          using P and T and X by simp
        ultimately have
         last (?f x) = last [y←ws @ xs. fst y = x]
          using O and P and Y by simp
        thus ?thesis
```

```
                using G and J and Nil and True by simp
            next
              case False
              thus ?thesis
                using J and Y and Nil by simp
            qed
          qed (simp add: J)
          ultimately show ?P x ∧ ?Q x ..
        qed
      qed
      by (rule conjI, rule exI [of - zs' @ xs'],
       insert L V W, auto simp: ctyping1-append-def)
  qed
  done


lemma ctyping1-aux-idem-fst:
⟦s ∈ A; ys ∈ ⊢ c; ys' ∈ ⊢ c⟧ ⟹
    ctyping1-idem-pred s t t' ys ys' A (⊢ c)
proof (induction c arbitrary: ys ys')
  fix c₁ c₂ ys ys'
  show
   ⟦⋀ys ys'. s ∈ A ⟹ ys ∈ ⊢ c₁ ⟹ ys' ∈ ⊢ c₁ ⟹
      ctyping1-idem-pred s t t' ys ys' A (⊢ c₁);
    ⋀ys ys'. s ∈ A ⟹ ys ∈ ⊢ c₂ ⟹ ys' ∈ ⊢ c₂ ⟹
      ctyping1-idem-pred s t t' ys ys' A (⊢ c₂);
    s ∈ A; ys ∈ ⊢ c₁;; c₂; ys' ∈ ⊢ c₁;; c₂⟧ ⟹
      ctyping1-idem-pred s t t' ys ys' A (⊢ c₁;; c₂)
    by (simp, rule ctyping1-merge-append-idem-fst [simplified])
next
  fix c₁ c₂ ys ys'
  show
   ⟦⋀ys ys'. s ∈ A ⟹ ys ∈ ⊢ c₁ ⟹ ys' ∈ ⊢ c₁ ⟹
      ctyping1-idem-pred s t t' ys ys' A (⊢ c₁);
    ⋀ys ys'. s ∈ A ⟹ ys ∈ ⊢ c₂ ⟹ ys' ∈ ⊢ c₂ ⟹
      ctyping1-idem-pred s t t' ys ys' A (⊢ c₂);
    s ∈ A; ys ∈ ⊢ c₁ OR c₂; ys' ∈ ⊢ c₁ OR c₂⟧ ⟹
      ctyping1-idem-pred s t t' ys ys' A (⊢ c₁ OR c₂)
    by (simp, rule ctyping1-merge-idem-fst [simplified])
next
  fix b c₁ c₂ ys ys'
  assume
    A: ⋀ys ys'. s ∈ A ⟹ ys ∈ ⊢ c₁ ⟹ ys' ∈ ⊢ c₁ ⟹
      ctyping1-idem-pred s t t' ys ys' A (⊢ c₁) and
    B: ⋀ys ys'. s ∈ A ⟹ ys ∈ ⊢ c₂ ⟹ ys' ∈ ⊢ c₂ ⟹
      ctyping1-idem-pred s t t' ys ys' A (⊢ c₂) and
    C: s ∈ A and
    D: ys ∈ ⊢ IF b THEN c₁ ELSE c₂ and
    E: ys' ∈ ⊢ IF b THEN c₁ ELSE c₂
```

```
  show ctyping1-idem-pred s t t' ys ys' A (⊢ IF b THEN c1 ELSE c2)
  proof (cases ⊢ b)
    case None
    show ?thesis
      by (insert A B C D E None, simp,
       rule ctyping1-merge-idem-fst [simplified])
  next
    case (Some v)
    show ?thesis
    proof (cases v)
      case True
      thus ?thesis
        by (insert A C D E Some, simp add: ctyping1-merge-empty-rhs)
    next
      case False
      thus ?thesis
        by (insert B C D E Some, simp add: ctyping1-merge-empty-lhs)
    qed
  qed
next
  fix b c ys ys'
  assume
    A: ⋀ys ys'. s ∈ A ⟹ ys ∈ ⊢ c ⟹ ys' ∈ ⊢ c ⟹
      ctyping1-idem-pred s t t' ys ys' A (⊢ c) and
    B: s ∈ A and
    C: ys ∈ ⊢ WHILE b DO c and
    D: ys' ∈ ⊢ WHILE b DO c
  have E: ctyping1-idem-pred s t t' ys ys' A (⊢ WHILE b DO c) if
    a: ys ∈ ⊢ c and
    b: ys' ∈ ⊢ c and
    c: ⊢ b ∈ {Some True, None}
  proof -
    have ctyping1-idem-pred s t t' ys ys' A (⊢ c)
      using A and B and a and b by simp
    then obtain f and s' and t'' and ys'' where
      E: ctyping1-idem-lhs s t t' ys ys' =
        ctyping1-idem-rhs f s' t'' and
      F: ∀ x. (f x = [] ⟷ [y←ys @ ys'. fst y = x] = []) ∧
        (f x ≠ [] ⟶ last (f x) = last [y←ys @ ys'. fst y = x]) and
      G: f = (λx. [y←ys''. fst y = x]) and
      H: ys'' ∈ ⊢ c and
      I: s' ∈ A
      by auto
    show ?thesis
      by (rule exI [of - f], insert E F G H I c, force)
  qed
  show ctyping1-idem-pred s t t' ys ys' A (⊢ WHILE b DO c)
  proof (cases ⊢ b)
    case None
```

```
  show ?thesis
  proof (cases ys')
    case Nil
    show ?thesis
    proof (cases ys = [])
      case True
      thus ?thesis
        by (insert B None Nil, force)
    next
      case False
      thus ?thesis
        by (insert B C None Nil, force)
    qed
  next
    case Cons
    show ?thesis
    proof (cases ys = [])
      case True
      show ?thesis
        apply (insert B D None Cons True)
        apply (rule exI [of - λx. [y←ys'. fst y = x]])
        apply (rule exI [of - s])
        apply (rule conjI)
         apply fastforce
        apply (rule conjI)
         apply fastforce
        apply (rule conjI)
         apply (rule exI [of - ys'])
        by simp-all
    next
      case False
      hence ys ∈ ⊢ c ∧ ys' ∈ ⊢ c
        using C and D and None and Cons by simp
      thus ?thesis
        using None by (blast intro: E)
    qed
  qed
next
  case (Some v)
  show ?thesis
  proof (cases v)
    case True
    hence ys ∈ ⊢ c ∧ ys' ∈ ⊢ c
      using C and D and Some by simp
    thus ?thesis
      using Some and True by (fastforce intro: E)
  next
    case False
    hence ys = [] ∧ ys' = []
```

```
    using C and D and Some by simp
   thus ?thesis
    by (insert B Some False, simp)
  qed
 qed
qed fastforce+


lemma ctyping1-idem-fst-1:
⟦s ∈ A; ys ∈ ⊢ c; ys′ ∈ ⊢ c⟧ ⟹ ∃f s′.
   (∃t″. ctyping1-idem-lhs s t t′ ys ys′ = ctyping1-idem-rhs f s′ t″) ∧
   (∃ys″. f = (λx. [y←ys″. fst y = x]) ∧ ys″ ∈ ⊢ c) ∧ s′ ∈ A
 apply (drule ctyping1-aux-idem-fst [where ys′ = ys], assumption+)
 apply clarify
 apply (rule exI, (rule conjI)?)+
  apply assumption
 by blast

lemma ctyping1-idem-fst-2:
⟦s ∈ A; ys ∈ ⊢ c⟧ ⟹ ∃f s′.
   (∃t′.
     (λx. if [y←ys. fst y = x] = []
       then s x
       else case snd (last [y←ys. fst y = x]) of None ⇒ t x | Some i ⇒ i) =
     (λx. if f x = []
       then s′ x
       else case snd (last (f x)) of None ⇒ t′ x | Some i ⇒ i)) ∧
   (∃ys′. f = (λx. [y←ys′. fst y = x]) ∧ ys′ ∈ ⊢ c) ∧
   (∃f′ s″.
     (∃t″. s′ = (λx. if f′ x = []
       then s″ x
       else case snd (last (f′ x)) of None ⇒ t″ x | Some i ⇒ i)) ∧
     (∃ys″. f′ = (λx. [y←ys″. fst y = x]) ∧ ys″ ∈ ⊢ c) ∧ s″ ∈ A)
 (is ⟦-; -⟧ ⟹ ∃- -. (∃-. ?f = -) ∧ -)
by (rule exI, rule exI [of - ?f], fastforce)

lemma ctyping1-idem-fst:
⊢ c (⊆ A, X) = (B, Y) ⟹ case ⊢ c (⊆ B, Y) of (B′, Y′) ⇒ B′ = B
by (auto intro: ctyping1-idem-fst-1 ctyping1-idem-fst-2 simp: ctyping1-def)


lemma ctyping1-idem-snd-1:
 assumes
   A: A ≠ {} and
   B: ∀r f s.
     (∀t. r ≠ (λx. if f x = [] then s x else case snd (last (f x)) of
       None ⇒ t x | Some i ⇒ i)) ∨
     (∀ys. f = (λx. [y←ys. fst y = x]) ⟶ ys ∉ ⊢ c) ∨ s ∉ A
     (is ∀r f s. (∀t. r ≠ ?r f s t) ∨ -)
```

```
  shows UNIV = S
proof −
  obtain s where C: s ∈ A
    using A by blast
  obtain ys where D: ys ∈ ⊢ c
    by (insert ctyping1-aux-nonempty, blast)
  let ?f = λx. [y←ys. fst y = x]
  show ?thesis
    using B [rule-format, of ?r ?f s (λx. 0) ?f s] and C and D by auto
qed

lemma ctyping1-idem-snd-2:
 {x. ∀f.
    (f x = [] ⟶ (∃ys. f = (λx. [y←ys. fst y = x]) ∧ ys ∈ ⊢ c) ⟶
      (∀f.
        (f x = [] ⟶ (∃ys. f = (λx. [y←ys. fst y = x]) ∧ ys ∈ ⊢ c) ⟶
          x ∈ X) ∧
        (f x ≠ [] ⟶ (∃ys. f = (λx. [y←ys. fst y = x]) ∧ ys ∈ ⊢ c) ⟶
          (∃i. snd (last (f x)) = Some i)))) ∧
    (f x ≠ [] ⟶ (∃ys. f = (λx. [y←ys. fst y = x]) ∧ ys ∈ ⊢ c) ⟶
      (∃i. snd (last (f x)) = Some i))} =
  {x. ∀f.
    (f x = [] ⟶ (∃ys. f = (λx. [y←ys. fst y = x]) ∧ ys ∈ ⊢ c) ⟶
      x ∈ X) ∧
    (f x ≠ [] ⟶ (∃ys. f = (λx. [y←ys. fst y = x]) ∧ ys ∈ ⊢ c) ⟶
      (∃i. snd (last (f x)) = Some i))}
by (rule equalityI, force+)

lemma ctyping1-idem-snd:
⊢ c (⊆ A, X) = (B, Y) ⟹ case ⊢ c (⊆ B, Y) of (B', Y') ⇒ Y' = Y
by (clarsimp simp: ctyping1-def ctyping1-idem-snd-1 ctyping1-idem-snd-2)


lemma ctyping1-idem:
⊢ c (⊆ A, X) = (B, Y) ⟹ ⊢ c (⊆ B, Y) = (B, Y)
by (frule ctyping1-idem-fst, drule ctyping1-idem-snd, auto)

end

end
```

4 Overapproximation of program semantics by the main type system

```
theory Overapproximation
  imports Idempotence
begin
```

As in my previous paper [10], the purpose of this section is to prove that type system *ctyping2* overapproximates program semantics, namely that if (a) $(c, s, p) \Rightarrow (t, q)$, (b) the type system outputs a *state set* B and a *vname set* Y when it is input program c, *state set* A, and *vname set* X, and (c) state s agrees with some state in A on the value of each state variable in X, then t must agree with some state in B on the value of each state variable in Y (lemma *ctyping2-approx*).

This proof makes use of the lemma *ctyping1-idem* proven in the previous section.

4.1 Global context proofs

lemma *avars-aval*:
$s = t\ (\subseteq \mathit{avars}\ a) \Longrightarrow \mathit{aval}\ a\ s = \mathit{aval}\ a\ t$
by (*induction a*, *simp-all*)

4.2 Local context proofs

context *noninterf*
begin

lemma *interf-set-mono*:
$[\![A' \subseteq A;\ X \subseteq X';\ \forall (B', Y') \in U'.\ \exists (B, Y) \in U.\ B' \subseteq B \wedge Y' \subseteq Y;$
$\quad \forall (B, Y) \in \mathit{insert}\ (\mathit{Univ?}\ A\ X,\ Z)\ U.\ B{:}\ Y \leadsto W]\!] \Longrightarrow$
$\ \forall (B, Y) \in \mathit{insert}\ (\mathit{Univ?}\ A'\ X',\ Z)\ U'.\ B{:}\ Y \leadsto W$
by (*subgoal-tac* $\mathit{Univ?}\ A'\ X' \subseteq \mathit{Univ?}\ A\ X$, *fastforce*,
auto simp: *univ-states-if-def*)

lemma *btyping1-btyping2-aux-1* [*elim*]:
assumes
A: $\mathit{avars}\ a_1 = \{\}$ **and**
B: $\mathit{avars}\ a_2 = \{\}$ **and**
C: $\mathit{aval}\ a_1\ (\lambda x.\ 0) < \mathit{aval}\ a_2\ (\lambda x.\ 0)$
shows $\mathit{aval}\ a_1\ s < \mathit{aval}\ a_2\ s$
proof –
have $\mathit{aval}\ a_1\ s = \mathit{aval}\ a_1\ (\lambda x.\ 0) \wedge \mathit{aval}\ a_2\ s = \mathit{aval}\ a_2\ (\lambda x.\ 0)$
using A **and** B **by** (*blast intro*: *avars-aval*)
thus *?thesis*
using C **by** *simp*
qed

lemma *btyping1-btyping2-aux-2* [*elim*]:
assumes
A: $\mathit{avars}\ a_1 = \{\}$ **and**
B: $\mathit{avars}\ a_2 = \{\}$ **and**
C: $\neg\ \mathit{aval}\ a_1\ (\lambda x.\ 0) < \mathit{aval}\ a_2\ (\lambda x.\ 0)$ **and**

D: $\mathit{aval}\ a_1\ s < \mathit{aval}\ a_2\ s$
shows *False*
proof −
have $\mathit{aval}\ a_1\ s = \mathit{aval}\ a_1\ (\lambda x.\ 0) \wedge \mathit{aval}\ a_2\ s = \mathit{aval}\ a_2\ (\lambda x.\ 0)$
using A **and** B **by** (*blast intro*: *avars-aval*)
thus *?thesis*
using C **and** D **by** *simp*
qed

lemma *btyping1-btyping2-aux*:
$\vdash b = \mathit{Some}\ v \Longrightarrow \models b\ (\subseteq A,\ X) = \mathit{Some}\ (\mathit{if}\ v\ \mathit{then}\ A\ \mathit{else}\ \{\})$
by (*induction b arbitrary*: *v*, *auto split*: *if-split-asm option.split-asm*)

lemma *btyping1-btyping2*:
$\vdash b = \mathit{Some}\ v \Longrightarrow \models b\ (\subseteq A,\ X) = (\mathit{if}\ v\ \mathit{then}\ (A,\ \{\})\ \mathit{else}\ (\{\},\ A))$
by (*simp add*: *btyping2-def btyping1-btyping2-aux*)

lemma *btyping2-aux-subset*:
$\models b\ (\subseteq A,\ X) = \mathit{Some}\ A' \Longrightarrow A' = \{s.\ s \in A \wedge \mathit{bval}\ b\ s\}$
by (*induction b arbitrary*: A', *auto split*: *if-split-asm option.split-asm*)

lemma *btyping2-aux-diff*:
$[\![\models b\ (\subseteq A,\ X) = \mathit{Some}\ B;\ \models b\ (\subseteq A',\ X') = \mathit{Some}\ B';\ A' \subseteq A;\ B' \subseteq B]\!] \Longrightarrow$
$A' - B' \subseteq A - B$
by (*blast dest*: *btyping2-aux-subset*)

lemma *btyping2-aux-mono*:
$[\![\models b\ (\subseteq A,\ X) = \mathit{Some}\ B;\ A' \subseteq A;\ X \subseteq X']\!] \Longrightarrow$
$\exists B'.\ \models b\ (\subseteq A',\ X') = \mathit{Some}\ B' \wedge B' \subseteq B$
by (*induction b arbitrary*: *B*, *auto dest*: *btyping2-aux-diff split*:
if-split-asm option.split-asm)

lemma *btyping2-mono*:
$[\![\models b\ (\subseteq A,\ X) = (B_1,\ B_2);\ \models b\ (\subseteq A',\ X') = (B_1{}',\ B_2{}');\ A' \subseteq A;\ X \subseteq X']\!] \Longrightarrow$
$B_1{}' \subseteq B_1 \wedge B_2{}' \subseteq B_2$
by (*simp add*: *btyping2-def split*: *option.split-asm*,
frule-tac [3−4] *btyping2-aux-mono*, *auto dest*: *btyping2-aux-subset*)

lemma *btyping2-un-eq*:
$\models b\ (\subseteq A,\ X) = (B_1,\ B_2) \Longrightarrow B_1 \cup B_2 = A$
by (*auto simp*: *btyping2-def dest*: *btyping2-aux-subset split*: *option.split-asm*)

lemma *btyping2-aux-eq*:
$[\![\models b\ (\subseteq A,\ X) = \mathit{Some}\ A';\ s = t\ (\subseteq \mathit{state} \cap X)]\!] \Longrightarrow \mathit{bval}\ b\ s = \mathit{bval}\ b\ t$
proof (*induction b arbitrary*: A')
fix $A'\ v$
show
$[\![\models \mathit{Bc}\ v\ (\subseteq A,\ X) = \mathit{Some}\ A';\ s = t\ (\subseteq \mathit{state} \cap X)]\!] \Longrightarrow$
$\mathit{bval}\ (\mathit{Bc}\ v)\ s = \mathit{bval}\ (\mathit{Bc}\ v)\ t$

```
    by simp
next
  fix A′ b
  show
   ⟦⋀A′. ⊫ b (⊆ A, X) = Some A′ ⟹ s = t (⊆ state ∩ X) ⟹
      bval b s = bval b t;
    ⊫ Not b (⊆ A, X) = Some A′; s = t (⊆ state ∩ X)⟧ ⟹
      bval (Not b) s = bval (Not b) t
    by (simp split: option.split-asm)
next
  fix A′ b₁ b₂
  show
   ⟦⋀A′. ⊫ b₁ (⊆ A, X) = Some A′ ⟹ s = t (⊆ state ∩ X) ⟹
      bval b₁ s = bval b₁ t;
    ⋀A′. ⊫ b₂ (⊆ A, X) = Some A′ ⟹ s = t (⊆ state ∩ X) ⟹
      bval b₂ s = bval b₂ t;
    ⊫ And b₁ b₂ (⊆ A, X) = Some A′; s = t (⊆ state ∩ X)⟧ ⟹
      bval (And b₁ b₂) s = bval (And b₁ b₂) t
    by (simp split: option.split-asm)
next
  fix A′ a₁ a₂
  show
   ⟦⊫ Less a₁ a₂ (⊆ A, X) = Some A′; s = t (⊆ state ∩ X)⟧ ⟹
      bval (Less a₁ a₂) s = bval (Less a₁ a₂) t
    by (subgoal-tac aval a₁ s = aval a₁ t,
     subgoal-tac aval a₂ s = aval a₂ t,
     auto intro!: avars-aval split: if-split-asm)
qed


lemma ctyping1-mono-fst:
 ⟦⊢ c (⊆ A, X) = (B, Y); ⊢ c (⊆ A′, X′) = (B′, Y′); A′ ⊆ A⟧ ⟹
    B′ ⊆ B
by (fastforce simp: ctyping1-def)

lemma ctyping1-mono:
  assumes
    A: ⊢ c (⊆ A, X) = (B, Y) and
    B: ⊢ c (⊆ A′, X′) = (B′, Y′) and
    C: A′ ⊆ A and
    D: X ⊆ X′
  shows B′ ⊆ B ∧ Y ⊆ Y′
proof (rule conjI, rule ctyping1-mono-fst [OF A B C])
  {
    fix x
    assume x ∉ Univ?? A′ {x. ∀f ∈ {λx. [y←ys. fst y = x] | ys. ys ∈ ⊢ c}.
      if f x = [] then x ∈ X′ else snd (last (f x)) ≠ None}
    moreover from this have A′ ≠ {}
      by (simp split: if-split-asm)
```

ultimately have $\neg\ (\forall f.$
$(\exists ys.\ f = (\lambda x.\ [y \leftarrow ys.\ fst\ y = x]) \wedge ys \in \vdash c) \longrightarrow$
$(if\ f\ x = []\ then\ x \in X'\ else\ snd\ (last\ (f\ x)) \neq None))$
(**is** $\neg$ *?P X'*)
by (*auto split*: *if-split-asm*)
moreover assume *?P X*
hence *?P X'*
using *D* **by** *fastforce*
ultimately have *False*
by *contradiction*
}
with *A* **and** *B* **and** *C* **show** $Y \subseteq Y'$
by (*cases* $A = \{\}$, *auto simp*: *ctyping1-def*)
qed

lemma *ctyping2-mono-skip* [*elim*!]:
$[\![(U, False) \models SKIP\ (\subseteq A, X) = Some\ (C, Z);\ A' \subseteq A;\ X \subseteq X']\!] \Longrightarrow$
$\exists C'\ Z'.\ (U', False) \models SKIP\ (\subseteq A', X') = Some\ (C', Z') \wedge$
$C' \subseteq C \wedge Z \subseteq Z'$
by (*clarsimp*, *subgoal-tac Univ?? C X* $= X$, *force*+)

lemma *ctyping2-mono-assign* [*elim*!]:
$[\![(U, False) \models x ::= a\ (\subseteq A, X) = Some\ (C, Z);\ A' \subseteq A;\ X \subseteq X';$
$\forall (B', Y') \in U'.\ \exists (B, Y) \in U.\ B' \subseteq B \wedge Y' \subseteq Y]\!] \Longrightarrow$
$\exists C'\ Z'.\ (U', False) \models x ::= a\ (\subseteq A', X') = Some\ (C', Z') \wedge$
$C' \subseteq C \wedge Z \subseteq Z'$
by (*frule interf-set-mono* [**where** $W = \{x\}$], *auto split*: *if-split-asm*)

lemma *ctyping2-mono-input* [*elim*!]:
$[\![(U, False) \models IN\ x\ (\subseteq A, X) = Some\ (C, Z);\ A' \subseteq A;\ X \subseteq X';$
$\forall (B', Y') \in U'.\ \exists (B, Y) \in U.\ B' \subseteq B \wedge Y' \subseteq Y]\!] \Longrightarrow$
$\exists C'\ Z'.\ (U', False) \models IN\ x\ (\subseteq A', X') = Some\ (C', Z') \wedge$
$C' \subseteq C \wedge Z \subseteq Z'$
by (*frule interf-set-mono* [**where** $W = \{x\}$], *auto split*: *if-split-asm*)

lemma *ctyping2-mono-output* [*elim*!]:
$[\![(U, False) \models OUT\ x\ (\subseteq A, X) = Some\ (C, Z);\ A' \subseteq A;\ X \subseteq X';$
$\forall (B', Y') \in U'.\ \exists (B, Y) \in U.\ B' \subseteq B \wedge Y' \subseteq Y]\!] \Longrightarrow$
$\exists C'\ Z'.\ (U', False) \models OUT\ x\ (\subseteq A', X') = Some\ (C', Z') \wedge$
$C' \subseteq C \wedge Z \subseteq Z'$
by (*frule interf-set-mono* [**where** $W = \{x\}$], *auto split*: *if-split-asm*)

lemma *ctyping2-mono-seq*:
assumes
A: $\bigwedge A'\ B\ X'\ Y\ U'.$
$(U, False) \models c_1\ (\subseteq A, X) = Some\ (B, Y) \Longrightarrow A' \subseteq A \Longrightarrow X \subseteq X' \Longrightarrow$
$\forall (B', Y') \in U'.\ \exists (B, Y) \in U.\ B' \subseteq B \wedge Y' \subseteq Y \Longrightarrow$
$\exists B'\ Y'.\ (U', False) \models c_1\ (\subseteq A', X') = Some\ (B', Y') \wedge$

$B' \subseteq B \wedge Y \subseteq Y'$ **and**
B: $\bigwedge p\ B\ Y\ B'\ C\ Y'\ Z\ U'.$
$(U,\ \mathit{False}) \models c_1\ (\subseteq A,\ X) = \mathit{Some}\ p \Longrightarrow (B,\ Y) = p \Longrightarrow$
$(U,\ \mathit{False}) \models c_2\ (\subseteq B,\ Y) = \mathit{Some}\ (C,\ Z) \Longrightarrow B' \subseteq B \Longrightarrow Y \subseteq Y' \Longrightarrow$
$\forall (B',\ Y') \in U'.\ \exists (B,\ Y) \in U.\ B' \subseteq B \wedge Y' \subseteq Y \Longrightarrow$
$\exists\, C'\ Z'.\ (U',\ \mathit{False}) \models c_2\ (\subseteq B',\ Y') = \mathit{Some}\ (C',\ Z') \wedge$
$C' \subseteq C \wedge Z \subseteq Z'$ **and**
C: $(U,\ \mathit{False}) \models c_1;;\ c_2\ (\subseteq A,\ X) = \mathit{Some}\ (C,\ Z)$ **and**
D: $A' \subseteq A$ **and**
E: $X \subseteq X'$ **and**
F: $\forall (B',\ Y') \in U'.\ \exists (B,\ Y) \in U.\ B' \subseteq B \wedge Y' \subseteq Y$
shows $\exists\, C'\ Z'.\ (U',\ \mathit{False}) \models c_1;;\ c_2\ (\subseteq A',\ X') = \mathit{Some}\ (C',\ Z') \wedge$
$C' \subseteq C \wedge Z \subseteq Z'$
proof –
obtain $B\ Y$ **where** $(U,\ \mathit{False}) \models c_1\ (\subseteq A,\ X) = \mathit{Some}\ (B,\ Y) \wedge$
$(U,\ \mathit{False}) \models c_2\ (\subseteq B,\ Y) = \mathit{Some}\ (C,\ Z)$
using C **by** (*auto split*: *option.split-asm*)
moreover from *this* **obtain** $B'\ Y'$ **where**
G: $(U',\ \mathit{False}) \models c_1\ (\subseteq A',\ X') = \mathit{Some}\ (B',\ Y') \wedge B' \subseteq B \wedge Y \subseteq Y'$
using A **and** D **and** E **and** F **by** *fastforce*
ultimately obtain $C'\ Z'$ **where**
$(U',\ \mathit{False}) \models c_2\ (\subseteq B',\ Y') = \mathit{Some}\ (C',\ Z') \wedge C' \subseteq C \wedge Z \subseteq Z'$
using B **and** F **by** *fastforce*
thus *?thesis*
using G **by** *simp*
qed

lemma *ctyping2-mono-or*:
assumes
A: $\bigwedge A'\ C_1\ X'\ Y_1\ U'.$
$(U,\ \mathit{False}) \models c_1\ (\subseteq A,\ X) = \mathit{Some}\ (C_1,\ Y_1) \Longrightarrow A' \subseteq A \Longrightarrow X \subseteq X' \Longrightarrow$
$\forall (B',\ Y') \in U'.\ \exists (B,\ Y) \in U.\ B' \subseteq B \wedge Y' \subseteq Y \Longrightarrow$
$\exists\, C_1'\ Y_1'.\ (U',\ \mathit{False}) \models c_1\ (\subseteq A',\ X') = \mathit{Some}\ (C_1',\ Y_1') \wedge$
$C_1' \subseteq C_1 \wedge Y_1 \subseteq Y_1'$ **and**
B: $\bigwedge A'\ C_2\ X'\ Y_2\ U'.$
$(U,\ \mathit{False}) \models c_2\ (\subseteq A,\ X) = \mathit{Some}\ (C_2,\ Y_2) \Longrightarrow A' \subseteq A \Longrightarrow X \subseteq X' \Longrightarrow$
$\forall (B',\ Y') \in U'.\ \exists (B,\ Y) \in U.\ B' \subseteq B \wedge Y' \subseteq Y \Longrightarrow$
$\exists\, C_2'\ Y_2'.\ (U',\ \mathit{False}) \models c_2\ (\subseteq A',\ X') = \mathit{Some}\ (C_2',\ Y_2') \wedge$
$C_2' \subseteq C_2 \wedge Y_2 \subseteq Y_2'$ **and**
C: $(U,\ \mathit{False}) \models c_1\ \mathit{OR}\ c_2\ (\subseteq A,\ X) = \mathit{Some}\ (C,\ Y)$ **and**
D: $A' \subseteq A$ **and**
E: $X \subseteq X'$ **and**
F: $\forall (B',\ Y') \in U'.\ \exists (B,\ Y) \in U.\ B' \subseteq B \wedge Y' \subseteq Y$
shows $\exists\, C'\ Y'.\ (U',\ \mathit{False}) \models c_1\ \mathit{OR}\ c_2\ (\subseteq A',\ X') = \mathit{Some}\ (C',\ Y') \wedge$
$C' \subseteq C \wedge Y \subseteq Y'$
proof –
obtain $C_1\ C_2\ Y_1\ Y_2$ **where**
G: $(C,\ Y) = (C_1 \cup C_2,\ Y_1 \cap Y_2) \wedge$
$\mathit{Some}\ (C_1,\ Y_1) = (U,\ \mathit{False}) \models c_1\ (\subseteq A,\ X) \wedge$

$\mathit{Some}\ (C_2,\ Y_2) = (U,\ \mathit{False}) \models c_2\ (\subseteq A,\ X)$
using C **by** $(\mathit{simp\ split}\colon \mathit{option.split\text{-}asm\ prod.split\text{-}asm})$
moreover have $H\colon \forall (B',\ Y') \in U'.\ \exists (B,\ Y) \in U.\ B' \subseteq B \wedge Y' \subseteq Y$
using F **by** simp
ultimately have $\exists\, C_1{}'\ Y_1{}'.$
$(U',\ \mathit{False}) \models c_1\ (\subseteq A',\ X') = \mathit{Some}\ (C_1{}',\ Y_1{}') \wedge C_1{}' \subseteq C_1 \wedge Y_1 \subseteq Y_1{}'$
using A **and** D **and** E **by** simp
moreover have $\exists\, C_2{}'\ Y_2{}'.$
$(U',\ \mathit{False}) \models c_2\ (\subseteq A',\ X') = \mathit{Some}\ (C_2{}',\ Y_2{}') \wedge C_2{}' \subseteq C_2 \wedge Y_2 \subseteq Y_2{}'$
using B **and** D **and** E **and** G **and** H **by** simp
ultimately show $\mathit{?thesis}$
using G **by** auto
qed

lemma $\mathit{ctyping2\text{-}mono\text{-}if}$:
assumes
$A\colon \bigwedge W\ p\ B_1\ B_2\ B_1{}'\ C_1\ X'\ Y_1\ W'.\ (W,\ p) =$
$(\mathit{insert}\ (\mathit{Univ?}\ A\ X,\ \mathit{bvars}\ b)\ U,\ \models b\ (\subseteq A,\ X)) \Longrightarrow (B_1,\ B_2) = p \Longrightarrow$
$(W,\ \mathit{False}) \models c_1\ (\subseteq B_1,\ X) = \mathit{Some}\ (C_1,\ Y_1) \Longrightarrow B_1{}' \subseteq B_1 \Longrightarrow$
$X \subseteq X' \Longrightarrow \forall (B',\ Y') \in W'.\ \exists (B,\ Y) \in W.\ B' \subseteq B \wedge Y' \subseteq Y \Longrightarrow$
$\exists\, C_1{}'\ Y_1{}'.\ (W',\ \mathit{False}) \models c_1\ (\subseteq B_1{}',\ X') = \mathit{Some}\ (C_1{}',\ Y_1{}') \wedge$
$C_1{}' \subseteq C_1 \wedge Y_1 \subseteq Y_1{}'$ **and**
$B\colon \bigwedge W\ p\ B_1\ B_2\ B_2{}'\ C_2\ X'\ Y_2\ W'.\ (W,\ p) =$
$(\mathit{insert}\ (\mathit{Univ?}\ A\ X,\ \mathit{bvars}\ b)\ U,\ \models b\ (\subseteq A,\ X)) \Longrightarrow (B_1,\ B_2) = p \Longrightarrow$
$(W,\ \mathit{False}) \models c_2\ (\subseteq B_2,\ X) = \mathit{Some}\ (C_2,\ Y_2) \Longrightarrow B_2{}' \subseteq B_2 \Longrightarrow$
$X \subseteq X' \Longrightarrow \forall (B',\ Y') \in W'.\ \exists (B,\ Y) \in W.\ B' \subseteq B \wedge Y' \subseteq Y \Longrightarrow$
$\exists\, C_2{}'\ Y_2{}'.\ (W',\ \mathit{False}) \models c_2\ (\subseteq B_2{}',\ X') = \mathit{Some}\ (C_2{}',\ Y_2{}') \wedge$
$C_2{}' \subseteq C_2 \wedge Y_2 \subseteq Y_2{}'$ **and**
$C\colon (U,\ \mathit{False}) \models \mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2\ (\subseteq A,\ X) = \mathit{Some}\ (C,\ Y)$ **and**
$D\colon A' \subseteq A$ **and**
$E\colon X \subseteq X'$ **and**
$F\colon \forall (B',\ Y') \in U'.\ \exists (B,\ Y) \in U.\ B' \subseteq B \wedge Y' \subseteq Y$
shows $\exists\, C'\ Y'.\ (U',\ \mathit{False}) \models \mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2\ (\subseteq A',\ X') =$
$\mathit{Some}\ (C',\ Y') \wedge C' \subseteq C \wedge Y \subseteq Y'$
proof $-$
let $\mathit{?W} = \mathit{insert}\ (\mathit{Univ?}\ A\ X,\ \mathit{bvars}\ b)\ U$
let $\mathit{?W'} = \mathit{insert}\ (\mathit{Univ?}\ A'\ X',\ \mathit{bvars}\ b)\ U'$
obtain $B_1\ B_2\ C_1\ C_2\ Y_1\ Y_2$ **where**
$G\colon (C,\ Y) = (C_1 \cup C_2,\ Y_1 \cap Y_2) \wedge (B_1,\ B_2) = \models b\ (\subseteq A,\ X) \wedge$
$\mathit{Some}\ (C_1,\ Y_1) = (\mathit{?W},\ \mathit{False}) \models c_1\ (\subseteq B_1,\ X) \wedge$
$\mathit{Some}\ (C_2,\ Y_2) = (\mathit{?W},\ \mathit{False}) \models c_2\ (\subseteq B_2,\ X)$
using C **by** $(\mathit{simp\ split}\colon \mathit{option.split\text{-}asm\ prod.split\text{-}asm})$
moreover obtain $B_1{}'\ B_2{}'$ **where** $H\colon (B_1{}',\ B_2{}') = \models b\ (\subseteq A',\ X')$
by $(\mathit{cases} \models b\ (\subseteq A',\ X'),\ \mathit{simp})$
ultimately have $I\colon B_1{}' \subseteq B_1 \wedge B_2{}' \subseteq B_2$
by $(\mathit{metis\ btyping2\text{-}mono}\ D\ E)$
moreover have $J\colon \forall (B',\ Y') \in \mathit{?W'}.\ \exists (B,\ Y) \in \mathit{?W}.\ B' \subseteq B \wedge Y' \subseteq Y$
using D **and** E **and** F **by** $(\mathit{auto\ simp}\colon \mathit{univ\text{-}states\text{-}if\text{-}def})$
ultimately have $\exists\, C_1{}'\ Y_1{}'.$

$(\mathit{?W'}, \mathit{False}) \models c_1\ (\subseteq B_1', X') = \mathit{Some}\ (C_1', Y_1') \wedge C_1' \subseteq C_1 \wedge Y_1 \subseteq Y_1'$
using A **and** E **and** G **by** *force*
moreover have $\exists\, C_2'\ Y_2'.$
$(\mathit{?W'}, \mathit{False}) \models c_2\ (\subseteq B_2', X') = \mathit{Some}\ (C_2', Y_2') \wedge C_2' \subseteq C_2 \wedge Y_2 \subseteq Y_2'$
using B **and** E **and** G **and** I **and** J **by** *force*
ultimately show *?thesis*
using G **and** H **by** (*auto split*: *prod.split*)
qed

lemma *ctyping2-mono-while*:
assumes
A: $\bigwedge B_1\ B_2\ C\ Y\ B_1'\ B_2'\ D_1\ E\ X'\ V\ U'.\ (B_1, B_2) = \models b\ (\subseteq A, X) \Longrightarrow$
$(C, Y) = \vdash c\ (\subseteq B_1, X) \Longrightarrow (B_1', B_2') = \models b\ (\subseteq C, Y) \Longrightarrow$
$\forall (B, W) \in \mathit{insert}\ (\mathit{Univ?}\ A\ X \cup \mathit{Univ?}\ C\ Y, \mathit{bvars}\ b)\ U.$
$B\colon W \leadsto \mathit{UNIV} \Longrightarrow$
$(\{\}, \mathit{False}) \models c\ (\subseteq B_1, X) = \mathit{Some}\ (E, V) \Longrightarrow D_1 \subseteq B_1 \Longrightarrow$
$X \subseteq X' \Longrightarrow \forall (B', Y') \in U'.\ \exists (B, Y) \in \{\}.\ B' \subseteq B \wedge Y' \subseteq Y \Longrightarrow$
$\exists E'\ V'.\ (U', \mathit{False}) \models c\ (\subseteq D_1, X') = \mathit{Some}\ (E', V') \wedge$
$E' \subseteq E \wedge V \subseteq V'$ **and**
B: $\bigwedge B_1\ B_2\ C\ Y\ B_1'\ B_2'\ D_1'\ F\ Y'\ W\ U'.\ (B_1, B_2) = \models b\ (\subseteq A, X) \Longrightarrow$
$(C, Y) = \vdash c\ (\subseteq B_1, X) \Longrightarrow (B_1', B_2') = \models b\ (\subseteq C, Y) \Longrightarrow$
$\forall (B, W) \in \mathit{insert}\ (\mathit{Univ?}\ A\ X \cup \mathit{Univ?}\ C\ Y, \mathit{bvars}\ b)\ U.$
$B\colon W \leadsto \mathit{UNIV} \Longrightarrow$
$(\{\}, \mathit{False}) \models c\ (\subseteq B_1', Y) = \mathit{Some}\ (F, W) \Longrightarrow D_1' \subseteq B_1' \Longrightarrow$
$Y \subseteq Y' \Longrightarrow \forall (B', Y') \in U'.\ \exists (B, Y) \in \{\}.\ B' \subseteq B \wedge Y' \subseteq Y \Longrightarrow$
$\exists F'\ W'.\ (U', \mathit{False}) \models c\ (\subseteq D_1', Y') = \mathit{Some}\ (F', W') \wedge$
$F' \subseteq F \wedge W \subseteq W'$ **and**
C: $(U, \mathit{False}) \models \mathit{WHILE}\ b\ \mathit{DO}\ c\ (\subseteq A, X) = \mathit{Some}\ (B, Z)$ **and**
D: $A' \subseteq A$ **and**
E: $X \subseteq X'$ **and**
F: $\forall (B', Y') \in U'.\ \exists (B, Y) \in U.\ B' \subseteq B \wedge Y' \subseteq Y$
shows $\exists B'\ Z'.\ (U', \mathit{False}) \models \mathit{WHILE}\ b\ \mathit{DO}\ c\ (\subseteq A', X') = \mathit{Some}\ (B', Z') \wedge$
$B' \subseteq B \wedge Z \subseteq Z'$
proof –
obtain $B_1\ B_1'\ B_2\ B_2'\ C\ E\ F\ V\ W\ Y$ **where** G: $(B_1, B_2) = \models b\ (\subseteq A, X) \wedge$
$(C, Y) = \vdash c\ (\subseteq B_1, X) \wedge (B_1', B_2') = \models b\ (\subseteq C, Y) \wedge$
$(\forall (B, W) \in \mathit{insert}\ (\mathit{Univ?}\ A\ X \cup \mathit{Univ?}\ C\ Y, \mathit{bvars}\ b)\ U.\ B\colon W \leadsto \mathit{UNIV}) \wedge$
$\mathit{Some}\ (E, V) = (\{\}, \mathit{False}) \models c\ (\subseteq B_1, X) \wedge$
$\mathit{Some}\ (F, W) = (\{\}, \mathit{False}) \models c\ (\subseteq B_1', Y) \wedge$
$(B, Z) = (B_2 \cup B_2', \mathit{Univ??}\ B_2\ X \cap Y)$
using C **by** (*force split*: *if-split-asm option.split-asm prod.split-asm*)
moreover obtain $D_1\ D_2$ **where** H: $\models b\ (\subseteq A', X') = (D_1, D_2)$
by (*cases* $\models b\ (\subseteq A', X')$, *simp*)
ultimately have I: $D_1 \subseteq B_1 \wedge D_2 \subseteq B_2$
by (*smt* (*verit*) *btyping2-mono D E*)
moreover obtain $C'\ Y'$ **where** J: $\vdash c\ (\subseteq D_1, X') = (C', Y')$
by (*cases* $\vdash c\ (\subseteq D_1, X')$, *simp*)
ultimately have K: $C' \subseteq C \wedge Y \subseteq Y'$
by (*smt* (*verit*) *ctyping1-mono E G*)

```
  moreover obtain D1' D2' where L: ⊨ b (⊆ C', Y') = (D1', D2')
    by (cases ⊨ b (⊆ C', Y'), simp)
  ultimately have M: D1' ⊆ B1' ∧ D2' ⊆ B2'
    by (smt (verit) btyping2-mono G)
  then obtain F' W' where
   ({}, False) ⊨ c (⊆ D1', Y') = Some (F', W') ∧ F' ⊆ F ∧ W ⊆ W'
    using B and F and G and K by force
  moreover obtain E' V' where
   ({}, False) ⊨ c (⊆ D1, X') = Some (E', V') ∧ E' ⊆ E ∧ V ⊆ V'
    using A and E and F and G and I by force
  moreover have Univ? A' X' ⊆ Univ? A X
    using D and E by (auto simp: univ-states-if-def)
  moreover have Univ? C' Y' ⊆ Univ? C Y
    using K by (auto simp: univ-states-if-def)
  ultimately have (U', False) ⊨ WHILE b DO c (⊆ A', X') =
    Some (D2 ∪ D2', Univ?? D2 X' ∩ Y')
    using F and G and H and J and L by force
  moreover have D2 ∪ D2' ⊆ B
    using G and I and M by auto
  moreover have Z ⊆ Univ?? D2 X' ∩ Y'
    using E and G and I and K by auto
  ultimately show ?thesis
    by simp
qed

lemma ctyping2-mono:
 ⟦(U, False) ⊨ c (⊆ A, X) = Some (C, Z); A' ⊆ A; X ⊆ X';
   ∀(B', Y') ∈ U'. ∃(B, Y) ∈ U. B' ⊆ B ∧ Y' ⊆ Y⟧ ⟹
  ∃C' Z'. (U', False) ⊨ c (⊆ A', X') = Some (C', Z') ∧ C' ⊆ C ∧ Z ⊆ Z'
  apply (induction (U, False) c A X arbitrary: A' C X' Z U U'
   rule: ctyping2.induct)
         apply fastforce
        apply fastforce
       apply fastforce
      apply fastforce
     apply (erule ctyping2-mono-seq, assumption+)
    apply (erule ctyping2-mono-or, assumption+)
   apply (erule ctyping2-mono-if, assumption+)
  apply (erule ctyping2-mono-while, assumption+)
  done


lemma ctyping1-ctyping2-fst-assign [elim!]:
  assumes
    A: ⊢ x ::= a (⊆ A, X) = (C, Z) and
    B: (U, False) ⊨ x ::= a (⊆ A, X) = Some (C', Z')
  shows C' ⊆ C
proof -
  let ?F = λx' w. if x = x'
```

```
    then (x, w) # [y←[]. fst y = x']
    else [y←[]. fst y = x']
  {
    fix s'
    assume s' ∈ C'
    moreover assume x ∈ state and C: avars a = {}
    ultimately obtain s where D: s ∈ A and E: s' = s(x := aval a s)
      using B by (auto split: if-split-asm)
    have ∃ s.
      (∃ t. s' = (λx'. if ?F x' (Some (aval a (λx. 0))) = []
        then s x'
        else case snd (last (?F x' (Some (aval a (λx. 0))))) of
          None ⇒ t x' | Some i ⇒ i)) ∧
      s ∈ A
      apply (insert C E)
      apply (rule exI [of - s])
      apply (rule conjI [OF - D])
      apply (rule exI [of - λx. 0])
      by (fastforce intro: avars-aval)
  }
  moreover {
    fix s'
    assume s' ∈ C'
    moreover assume x ∈ state and avars a ≠ {}
    ultimately obtain s where C: s ∈ A and D: s' = s
      using B by (simp split: if-split-asm)
    have ∃ s.
      (∃ t. s' = (λx'. if ?F x' None = []
        then s x'
        else case snd (last (?F x' None)) of
          None ⇒ t x' | Some i ⇒ i)) ∧
      s ∈ A
      apply (insert D)
      apply (rule exI [of - s])
      apply (rule conjI [OF - C])
      apply (rule exI [of - s])
      by auto
  }
  moreover {
    fix s'
    assume s' ∈ C' and x ∉ state
    hence s' ∈ A
      using B by (simp split: if-split-asm)
  }
  ultimately show ?thesis
    using A by (fastforce simp: ctyping1-def)
qed

lemma ctyping1-ctyping2-fst-input [elim!]:
```

```
  assumes
    A: ⊢ IN x (⊆ A, X) = (C, Z) and
    B: (U, False) ⊨ IN x (⊆ A, X) = Some (C', Z')
  shows C' ⊆ C
proof -
  let ?F = λx'. if x = x'
    then (x, None) # [y←[]. fst y = x']
    else [y←[]. fst y = x']
  {
    fix s'
    assume s' ∈ C'
    moreover assume x ∈ state
    ultimately obtain s where C: s ∈ A and D: s' = s
      using B by (simp split: if-split-asm)
    have ∃ s.
      (∃ t. s' = (λx'. if ?F x' = []
        then s x'
        else case snd (last (?F x')) of
          None ⇒ t x' | Some i ⇒ i)) ∧
      s ∈ A
      apply (insert D)
      apply (rule exI [of - s])
      apply (rule conjI [OF - C])
      apply (rule exI [of - s])
      by auto
  }
  moreover {
    fix s'
    assume s' ∈ C' and x ∉ state
    hence s' ∈ A
      using B by (simp split: if-split-asm)
  }
  ultimately show ?thesis
    using A by (fastforce simp: ctyping1-def)
qed

lemma ctyping1-ctyping2-fst-output [elim!]:
⟦⊢ OUT x (⊆ A, X) = (C, Z);
    (U, False) ⊨ OUT x (⊆ A, X) = Some (C', Z')⟧ ⟹
  C' ⊆ C
by (simp add: ctyping1-def split: if-split-asm)

lemma ctyping1-ctyping2-fst-seq:
  assumes
    A: ⊢ c₁;; c₂ (⊆ A, X) = (C, Z) and
    B: (U, False) ⊨ c₁;; c₂ (⊆ A, X) = Some (C', Z') and
    C: ⋀B B' Y Y'. ⊢ c₁ (⊆ A, X) = (B, Y) ⟹
      (U, False) ⊨ c₁ (⊆ A, X) = Some (B', Y') ⟹ B' ⊆ B and
    D: ⋀p B' Y' D' C' W' Z'.
```

```
    (U, False) ⊨ c1 (⊆ A, X) = Some p ⟹ (B', Y') = p ⟹
      ⊢ c2 (⊆ B', Y') = (D', W') ⟹
        (U, False) ⊨ c2 (⊆ B', Y') = Some (C', Z') ⟹ C' ⊆ D'
  shows C' ⊆ C
proof -
  obtain B' Y' where E: (U, False) ⊨ c1 (⊆ A, X) = Some (B', Y') and
   (U, False) ⊨ c2 (⊆ B', Y') = Some (C', Z')
    using B by (auto split: option.split-asm)
  moreover obtain D' W' where F: ⊢ c2 (⊆ B', Y') = (D', W')
    by (cases ⊢ c2 (⊆ B', Y'), simp)
  ultimately have G: C' ⊆ D'
    using D by simp
  obtain B Y where H: ⊢ c1 (⊆ A, X) = (B, Y)
    by (cases ⊢ c1 (⊆ A, X), simp)
  hence B' ⊆ B
    using C and E by simp
  moreover obtain D W where I: ⊢ c2 (⊆ B, Y) = (D, W)
    by (cases ⊢ c2 (⊆ B, Y), simp)
  ultimately have D' ⊆ D
    using F by (blast dest: ctyping1-mono-fst)
  moreover {
    fix ys ys' s t and t' :: state
    assume K: s ∈ A
    assume ys ∈ ⊢ c1 and ys' ∈ ⊢ c2
    hence L: ys @ ys' ∈ ⊢ c1 ⊔@ ⊢ c2
      by (force simp: ctyping1-merge-append-def
       ctyping1-append-def ctyping1-merge-def)
    let ?f = λx. [y←ys @ ys'. fst y = x]
    let ?t = λx. if [y←ys'. fst y = x] = [] then t x else t' x
    have ∃f s'.
      (∃t''.
        (λx. if [y←ys'. fst y = x] = []
          then if [y←ys. fst y = x] = []
            then s x
            else case snd (last [y←ys. fst y = x]) of
              None ⇒ t x | Some i ⇒ i
          else case snd (last [y←ys'. fst y = x]) of
            None ⇒ t' x | Some i ⇒ i) =
        (λx. if f x = []
          then s' x
          else case snd (last (f x)) of None ⇒ t'' x | Some i ⇒ i)) ∧
      (∃ys''. f = (λx. [y←ys''. fst y = x]) ∧ ys'' ∈ ⊢ c1 ⊔@ ⊢ c2) ∧ s' ∈ A
      apply (insert K L)
      apply (rule exI [of - ?f])
      apply (rule exI [of - s])
      apply (rule conjI)
       apply (rule exI [of - ?t])
       apply fastforce
      apply (rule conjI)
```

```
      apply (rule exI [of - ys @ ys'])
      by simp-all
  }
  hence D ⊆ C
    using A and H and I by (auto simp: ctyping1-def)
  ultimately show ?thesis
    using G by simp
qed

lemma ctyping1-ctyping2-fst-or:
  assumes
    A: ⊢ c1 OR c2 (⊆ A, X) = (C, Y) and
    B: (U, False) ⊨ c1 OR c2 (⊆ A, X) = Some (C', Y') and
    C: ⋀C C' Y Y'. ⊢ c1 (⊆ A, X) = (C, Y) ⟹
      (U, False) ⊨ c1 (⊆ A, X) = Some (C', Y') ⟹ C' ⊆ C and
    D: ⋀C C' Y Y'. ⊢ c2 (⊆ A, X) = (C, Y) ⟹
      (U, False) ⊨ c2 (⊆ A, X) = Some (C', Y') ⟹ C' ⊆ C
  shows C' ⊆ C
proof –
  obtain C1' C2' Y1' Y2' where
    E: (C', Y') = (C1' ∪ C2', Y1' ∩ Y2') and
    F: (U, False) ⊨ c1 (⊆ A, X) = Some (C1', Y1') and
    G: (U, False) ⊨ c2 (⊆ A, X) = Some (C2', Y2')
    using B by (auto split: option.split-asm prod.split-asm)
  obtain C1 Y1 where H: ⊢ c1 (⊆ A, X) = (C1, Y1)
    by (cases ⊢ c1 (⊆ A, X), simp)
  hence C1' ⊆ C1
    using C and F by simp
  moreover obtain C2 Y2 where I: ⊢ c2 (⊆ A, X) = (C2, Y2)
    by (cases ⊢ c2 (⊆ A, X), simp)
  hence C2' ⊆ C2
    using D and G by simp
  ultimately have C' ⊆ C1 ∪ C2
    using E by blast
  moreover {
    fix ys s t
    assume s ∈ A
    moreover assume ys ∈ ⊢ c1
    hence ys ∈ ⊢ c1 ⊔ ⊢ c2
      by (force simp: ctyping1-merge-def)
    ultimately have ∃f s'.
      (∃t'.
        (λx. if [y←ys. fst y = x] = []
          then s x
          else case snd (last [y←ys. fst y = x]) of
            None ⇒ t x | Some i ⇒ i) =
        (λx. if f x = []
          then s' x
          else case snd (last (f x)) of None ⇒ t' x | Some i ⇒ i)) ∧
```

```
      (∃ ys'. f = (λx. [y←ys'. fst y = x]) ∧ ys' ∈ ⊢ c_1 ⊔ ⊢ c_2) ∧ s' ∈ A
      by fastforce
  }
  hence C_1 ⊆ C
    using A and H by (auto simp: ctyping1-def)
  moreover {
    fix ys s t
    assume s ∈ A
    moreover assume ys ∈ ⊢ c_2
    hence ys ∈ ⊢ c_1 ⊔ ⊢ c_2
      by (force simp: ctyping1-merge-def)
    ultimately have ∃ f s'.
      (∃ t'.
        (λx. if [y←ys. fst y = x] = []
          then s x
          else case snd (last [y←ys. fst y = x]) of
            None ⇒ t x | Some i ⇒ i) =
        (λx. if f x = []
          then s' x
          else case snd (last (f x)) of None ⇒ t' x | Some i ⇒ i)) ∧
      (∃ ys'. f = (λx. [y←ys'. fst y = x]) ∧ ys' ∈ ⊢ c_1 ⊔ ⊢ c_2) ∧ s' ∈ A
      by fastforce
  }
  hence C_2 ⊆ C
    using A and I by (auto simp: ctyping1-def)
  ultimately show ?thesis
    by blast
qed

lemma ctyping1-ctyping2-fst-if:
  assumes
    A: ⊢ IF b THEN c_1 ELSE c_2 (⊆ A, X) = (C, Y) and
    B: (U, False) ⊨ IF b THEN c_1 ELSE c_2 (⊆ A, X) = Some (C', Y') and
    C: ⋀U' p B_1 B_2 C C' Y Y'.
      (U', p) = (insert (Univ? A X, bvars b) U, ⊨ b (⊆ A, X)) ⟹
        (B_1, B_2) = p ⟹ ⊢ c_1 (⊆ B_1, X) = (C, Y) ⟹
          (U', False) ⊨ c_1 (⊆ B_1, X) = Some (C', Y') ⟹ C' ⊆ C and
    D: ⋀U' p B_1 B_2 C C' Y Y'.
      (U', p) = (insert (Univ? A X, bvars b) U, ⊨ b (⊆ A, X)) ⟹
        (B_1, B_2) = p ⟹ ⊢ c_2 (⊆ B_2, X) = (C, Y) ⟹
          (U', False) ⊨ c_2 (⊆ B_2, X) = Some (C', Y') ⟹ C' ⊆ C
  shows C' ⊆ C
proof −
  let ?U' = insert (Univ? A X, bvars b) U
  obtain B_1 B_2 C_1' C_2' Y_1' Y_2' where
    E: (C', Y') = (C_1' ∪ C_2', Y_1' ∩ Y_2') and
    F: ⊨ b (⊆ A, X) = (B_1, B_2) and
    G: (?U', False) ⊨ c_1 (⊆ B_1, X) = Some (C_1', Y_1') and
    H: (?U', False) ⊨ c_2 (⊆ B_2, X) = Some (C_2', Y_2')
```

```
  using B by (auto split: option.split-asm prod.split-asm)
obtain C1 Y1 where I: ⊢ c1 (⊆ B1, X) = (C1, Y1)
  by (cases ⊢ c1 (⊆ B1, X), simp)
hence C1' ⊆ C1
  using C and F and G by simp
moreover obtain C2 Y2 where J: ⊢ c2 (⊆ B2, X) = (C2, Y2)
  by (cases ⊢ c2 (⊆ B2, X), simp)
hence C2' ⊆ C2
  using D and F and H by simp
ultimately have K: C' ⊆ C1 ∪ C2
  using E by blast
{
  fix ys s t
  assume s ∈ B1
  hence s ∈ A
    using F by (blast dest: btyping2-un-eq)
  moreover assume ys ∈ ⊢ c1
  hence ys ∈ ⊢ c1 ⊔ ⊢ c2
    by (force simp: ctyping1-merge-def)
  ultimately have ∃f s'.
    (∃t'.
      (λx. if [y←ys. fst y = x] = []
        then s x
        else case snd (last [y←ys. fst y = x]) of
          None ⇒ t x | Some i ⇒ i) =
      (λx. if f x = []
        then s' x
        else case snd (last (f x)) of None ⇒ t' x | Some i ⇒ i)) ∧
    (∃ys'. f = (λx. [y←ys'. fst y = x]) ∧ ys' ∈ ⊢ c1 ⊔ ⊢ c2) ∧ s' ∈ A
    by fastforce
}
moreover {
  fix ys s t
  assume s ∈ B1
  moreover assume ys ∈ ⊢ c1
  hence ys ∈ ⊢ c1 ⊔ {}
    by (force simp: ctyping1-merge-def)
  ultimately have ∃f s'.
    (∃t'.
      (λx. if [y←ys. fst y = x] = []
        then s x
        else case snd (last [y←ys. fst y = x]) of
          None ⇒ t x | Some i ⇒ i) =
      (λx. if f x = []
        then s' x
        else case snd (last (f x)) of None ⇒ t' x | Some i ⇒ i)) ∧
    (∃ys'. f = (λx. [y←ys'. fst y = x]) ∧ ys' ∈ ⊢ c1 ⊔ {}) ∧ s' ∈ B1
    by fastforce
}
```

```
  ultimately have L: C₁ ⊆ C
    using A and F and I by (cases ⊢ b, auto
     dest!: btyping1-btyping2 [of - - A X] simp: ctyping1-def)
  {
    fix ys s t
    assume s ∈ B₂
    hence s ∈ A
      using F by (blast dest: btyping2-un-eq)
    moreover assume ys ∈ ⊢ c₂
    hence ys ∈ ⊢ c₁ ⊔ ⊢ c₂
      by (force simp: ctyping1-merge-def)
    ultimately have ∃f s'.
      (∃t'.
        (λx. if [y←ys. fst y = x] = []
          then s x
          else case snd (last [y←ys. fst y = x]) of
            None ⇒ t x | Some i ⇒ i) =
        (λx. if f x = []
          then s' x
          else case snd (last (f x)) of None ⇒ t' x | Some i ⇒ i)) ∧
      (∃ys'. f = (λx. [y←ys'. fst y = x]) ∧ ys' ∈ ⊢ c₁ ⊔ ⊢ c₂) ∧ s' ∈ A
      by fastforce
  }
  moreover {
    fix ys s t
    assume s ∈ B₂
    moreover assume ys ∈ ⊢ c₂
    hence ys ∈ {} ⊔ ⊢ c₂
      by (force simp: ctyping1-merge-def)
    ultimately have ∃f s'.
      (∃t'.
        (λx. if [y←ys. fst y = x] = []
          then s x
          else case snd (last [y←ys. fst y = x]) of
            None ⇒ t x | Some i ⇒ i) =
        (λx. if f x = []
          then s' x
          else case snd (last (f x)) of None ⇒ t' x | Some i ⇒ i)) ∧
      (∃ys'. f = (λx. [y←ys'. fst y = x]) ∧ ys' ∈ {} ⊔ ⊢ c₂) ∧ s' ∈ B₂
      by fastforce
  }
  ultimately have C₂ ⊆ C
    using A and F and J by (cases ⊢ b, auto
     dest!: btyping1-btyping2 [of - - A X] simp: ctyping1-def)
  with K and L show ?thesis
    by blast
qed

lemma ctyping1-ctyping2-fst-while:
```

```
  assumes
    A: ⊢ WHILE b DO c (⊆ A, X) = (B, Z) and
    B: (U, False) ⊨ WHILE b DO c (⊆ A, X) = Some (B′, Z′)
  shows B′ ⊆ B
proof −
  obtain B₁ B₁′ B₂ B₂′ C Y where
    C: ⊨ b (⊆ A, X) = (B₁, B₂) and
    D: ⊢ c (⊆ B₁, X) = (C, Y) and
    E: ⊨ b (⊆ C, Y) = (B₁′, B₂′) and
    F: (B′, Z′) = (B₂ ∪ B₂′, Univ?? B₂ X ∩ Y)
    using B by (force split: if-split-asm option.split-asm prod.split-asm)
  {
    fix s
    assume s ∈ B₂
    hence s ∈ A
      using C by (blast dest: btyping2-un-eq)
    hence ∃ f s′.
      (∃ t. s = (λx. if f x = []
        then s′ x
        else case snd (last (f x)) of None ⇒ t x | Some i ⇒ i)) ∧
      (∃ ys. f = (λx. [y←ys. fst y = x]) ∧ (ys = [] ∨ ys ∈ ⊢ c)) ∧ s′ ∈ A
      by force
  }
  with A and C have G: B₂ ⊆ B
    by (cases ⊢ b, auto dest!: btyping1-btyping2 [of - - A X]
     simp: ctyping1-def)
  {
    fix s
    assume s ∈ B₂′
    hence s ∈ C
      using E by (blast dest: btyping2-un-eq)
    then obtain f s′ where H:
     (∃ t. s = (λx. if f x = []
        then s′ x
        else case snd (last (f x)) of None ⇒ t x | Some i ⇒ i)) ∧
      (∃ ys. f = (λx. [y←ys. fst y = x]) ∧ ys ∈ ⊢ c) ∧ s′ ∈ B₁
      using D by (fastforce simp: ctyping1-def)
    hence I: s′ ∈ A
      using C by (blast dest: btyping2-un-eq)
    have ∃ f s′.
      (∃ t. s = (λx. if f x = []
        then s′ x
        else case snd (last (f x)) of None ⇒ t x | Some i ⇒ i)) ∧
      (∃ ys. f = (λx. [y←ys. fst y = x]) ∧ (ys = [] ∨ ys ∈ ⊢ c)) ∧ s′ ∈ A
      by (rule exI [of - f], insert H I, auto)
  }
  moreover {
    fix s
    assume s ∈ B₂′
```

```
    moreover assume ⊢ b = Some True
    ultimately have ∃f s'.
      (∃t. s = (λx. if f x = []
        then s' x
        else case snd (last (f x)) of None ⇒ t x | Some i ⇒ i)) ∧
      (∃ys. f = (λx. [y←ys. fst y = x]) ∧ ys ∈ ⊢ c) ∧ s' ∈ A
      using E by (auto dest: btyping1-btyping2 [of - - C Y])
  }
  moreover {
    fix s
    assume s ∈ B2'
    hence C ≠ {}
      using E by (blast dest: btyping2-un-eq)
    hence B1 ≠ {}
      using D by (auto simp: ctyping1-def)
    moreover assume ⊢ b = Some False
    ultimately have s ∈ A
      using C by (auto dest: btyping1-btyping2 [of - - A X])
  }
  ultimately have B2' ⊆ B
    using A by (cases ⊢ b, auto simp: ctyping1-def)
  with F and G show ?thesis
    by simp
qed

lemma ctyping1-ctyping2-fst:
 ⟦⊢ c (⊆ A, X) = (C, Z); (U, False) ⊨ c (⊆ A, X) = Some (C', Z')⟧ ⟹
    C' ⊆ C
  apply (induction (U, False) c A X arbitrary: C C' Z Z' U
   rule: ctyping2.induct)
         apply (fastforce simp: ctyping1-def)
        apply fastforce
       apply fastforce
      apply fastforce
     apply (erule ctyping1-ctyping2-fst-seq, assumption+)
    apply (erule ctyping1-ctyping2-fst-or, assumption+)
   apply (erule ctyping1-ctyping2-fst-if, assumption+)
  apply (erule ctyping1-ctyping2-fst-while, assumption+)
  done


lemma ctyping1-ctyping2-snd-skip [elim!]:
 ⟦⊢ SKIP (⊆ A, X) = (C, Z);
    (U, False) ⊨ SKIP (⊆ A, X) = Some (C', Z')⟧ ⟹
  Z ⊆ Z'
by (simp add: ctyping1-def split: if-split-asm)

lemma ctyping1-ctyping2-snd-assign [elim!]:
 ⟦⊢ x ::= a (⊆ A, X) = (C, Z);
```

$(U,\ \mathit{False}) \models x ::= a\ (\subseteq A,\ X) = \mathit{Some}\ (C',\ Z')\rrbracket \Longrightarrow$
$Z \subseteq Z'$
by (*auto simp*: *ctyping1-def split*: *if-split-asm*)

lemma *ctyping1-ctyping2-snd-input* [*elim*!]:
$\llbracket\vdash \mathit{IN}\ x\ (\subseteq A,\ X) = (C,\ Z);$
$(U,\ \mathit{False}) \models \mathit{IN}\ x\ (\subseteq A,\ X) = \mathit{Some}\ (C',\ Z')\rrbracket \Longrightarrow$
$Z \subseteq Z'$
by (*auto simp*: *ctyping1-def split*: *if-split-asm*)

lemma *ctyping1-ctyping2-snd-output* [*elim*!]:
$\llbracket\vdash \mathit{OUT}\ x\ (\subseteq A,\ X) = (C,\ Z);$
$(U,\ \mathit{False}) \models \mathit{OUT}\ x\ (\subseteq A,\ X) = \mathit{Some}\ (C',\ Z')\rrbracket \Longrightarrow$
$Z \subseteq Z'$
by (*simp add*: *ctyping1-def split*: *if-split-asm*)

lemma *ctyping1-ctyping2-snd-seq*:
assumes
A: $\vdash c_1;;\ c_2\ (\subseteq A,\ X) = (C,\ Z)$ **and**
B: $(U,\ \mathit{False}) \models c_1;;\ c_2\ (\subseteq A,\ X) = \mathit{Some}\ (C',\ Z')$ **and**
C: $\bigwedge B\ B'\ Y\ Y'.\ \vdash c_1\ (\subseteq A,\ X) = (B,\ Y) \Longrightarrow$
$(U,\ \mathit{False}) \models c_1\ (\subseteq A,\ X) = \mathit{Some}\ (B',\ Y') \Longrightarrow Y \subseteq Y'$ **and**
D: $\bigwedge p\ B'\ Y'\ D'\ C'\ W'\ Z'.$
$(U,\ \mathit{False}) \models c_1\ (\subseteq A,\ X) = \mathit{Some}\ p \Longrightarrow (B',\ Y') = p \Longrightarrow$
$\vdash c_2\ (\subseteq B',\ Y') = (D',\ W') \Longrightarrow$
$(U,\ \mathit{False}) \models c_2\ (\subseteq B',\ Y') = \mathit{Some}\ (C',\ Z') \Longrightarrow W' \subseteq Z'$
shows $Z \subseteq Z'$
proof −
obtain $B'\ Y'$ **where** E: $(U,\ \mathit{False}) \models c_1\ (\subseteq A,\ X) = \mathit{Some}\ (B',\ Y')$ **and**
$(U,\ \mathit{False}) \models c_2\ (\subseteq B',\ Y') = \mathit{Some}\ (C',\ Z')$
using B **by** (*auto split*: *option.split-asm*)
moreover obtain $D'\ W'$ **where** F: $\vdash c_2\ (\subseteq B',\ Y') = (D',\ W')$
by (*cases* $\vdash c_2\ (\subseteq B',\ Y')$, *simp*)
ultimately have G: $W' \subseteq Z'$
using D **by** *simp*
obtain $B\ Y$ **where** H: $\vdash c_1\ (\subseteq A,\ X) = (B,\ Y)$
by (*cases* $\vdash c_1\ (\subseteq A,\ X)$, *simp*)
hence $Y \subseteq Y'$
using C **and** E **by** *simp*
moreover have $B' \subseteq B$
using H **and** E **by** (*rule ctyping1-ctyping2-fst*)
moreover obtain $D\ W$ **where** I: $\vdash c_2\ (\subseteq B,\ Y) = (D,\ W)$
by (*cases* $\vdash c_2\ (\subseteq B,\ Y)$, *simp*)
ultimately have $W \subseteq W'$
using F **by** (*blast dest*: *ctyping1-mono*)
moreover {
fix x
assume J: $\forall f.\ (\exists ys.\ f = (\lambda x.\ [y\leftarrow ys.\ \mathit{fst}\ y = x]) \wedge ys \in\ \vdash c_1 \sqcup_{@} \vdash c_2) \longrightarrow$
$(\mathit{if}\ f\ x = []\ \mathit{then}\ x \in X\ \mathit{else}\ \mathit{snd}\ (\mathit{last}\ (f\ x)) \neq \mathit{None})$

```
{
  fix ys' ys
  assume ys ∈ ⊢ c1 and ys' ∈ ⊢ c2
  hence ys @ ys' ∈ ⊢ c1 ⊔@ ⊢ c2
    by (force simp: ctyping1-merge-append-def
      ctyping1-append-def ctyping1-merge-def)
  moreover assume [y←ys. fst y = x] = [] and [y←ys'. fst y = x] = []
  ultimately have x ∈ X
    using J by auto
}
moreover {
  fix ys ys'
  assume ys ∈ ⊢ c1 and ys' ∈ ⊢ c2
  hence ys @ ys' ∈ ⊢ c1 ⊔@ ⊢ c2
    by (force simp: ctyping1-merge-append-def
      ctyping1-append-def ctyping1-merge-def)
  moreover assume [y←ys. fst y = x] ≠ [] and [y←ys'. fst y = x] = []
  ultimately have ∃ i. snd (last [y←ys. fst y = x]) = Some i
    using J by auto
}
moreover {
  fix ys'
  assume ys' ∈ ⊢ c2
  moreover obtain ys where ys ∈ ⊢ c1
    by (insert ctyping1-aux-nonempty, blast)
  ultimately have ys @ ys' ∈ ⊢ c1 ⊔@ ⊢ c2
    by (force simp: ctyping1-merge-append-def
      ctyping1-append-def ctyping1-merge-def)
  moreover assume [y←ys'. fst y = x] ≠ []
  ultimately have ∃ i. snd (last [y←ys'. fst y = x]) = Some i
    using J by auto
}
ultimately have x ∈ {x. ∀ f ∈ {λx. [y←ys. fst y = x] | ys. ys ∈ ⊢ c2}.
  if f x = []
  then x ∈ {x. ∀ f. (∃ ys. f = (λx. [y←ys. fst y = x]) ∧ ys ∈ ⊢ c1) ⟶
    (if f x = [] then x ∈ X else snd (last (f x)) ≠ None)}
  else snd (last (f x)) ≠ None}
  (is - ∈ ?X)
  by auto
moreover assume x ∉ (if ∀ x f s.
  (∀ t. x ≠ (λx. if f x = [] then s x else case snd (last (f x)) of
    None ⇒ t x | Some i ⇒ i)) ∨
  (∀ ys. f = (λx. [y←ys. fst y = x]) ⟶ ys ∉ ⊢ c1) ∨ s ∉ A
    then UNIV else ?X)
hence x ∉ ?X
  by (auto split: if-split-asm)
ultimately have False
  by contradiction
}
```

hence $Z \subseteq W$
using A **and** H **and** I **by** (*cases* $A = \{\}$, *auto simp*: *ctyping1-def*)
ultimately show *?thesis*
using G **by** *simp*
qed

lemma *ctyping1-ctyping2-snd-or*:
assumes
A: $\vdash c_1$ *OR* c_2 $(\subseteq A,\ X) = (C,\ Y)$ **and**
B: $(U,\ \mathit{False}) \models c_1$ *OR* c_2 $(\subseteq A,\ X) = \mathit{Some}\ (C',\ Y')$ **and**
C: $\bigwedge C\ C'\ Y\ Y'.\ \vdash c_1\ (\subseteq A,\ X) = (C,\ Y) \Longrightarrow$
$(U,\ \mathit{False}) \models c_1\ (\subseteq A,\ X) = \mathit{Some}\ (C',\ Y') \Longrightarrow Y \subseteq Y'$ **and**
D: $\bigwedge C\ C'\ Y\ Y'.\ \vdash c_2\ (\subseteq A,\ X) = (C,\ Y) \Longrightarrow$
$(U,\ \mathit{False}) \models c_2\ (\subseteq A,\ X) = \mathit{Some}\ (C',\ Y') \Longrightarrow Y \subseteq Y'$
shows $Y \subseteq Y'$
proof $-$
obtain $C_1'\ C_2'\ Y_1'\ Y_2'$ **where**
E: $(C',\ Y') = (C_1' \cup C_2',\ Y_1' \cap Y_2')$ **and**
F: $(U,\ \mathit{False}) \models c_1\ (\subseteq A,\ X) = \mathit{Some}\ (C_1',\ Y_1')$ **and**
G: $(U,\ \mathit{False}) \models c_2\ (\subseteq A,\ X) = \mathit{Some}\ (C_2',\ Y_2')$
using B **by** (*auto split*: *option.split-asm prod.split-asm*)
obtain $C_1\ Y_1$ **where** H: $\vdash c_1\ (\subseteq A,\ X) = (C_1,\ Y_1)$
by (*cases* $\vdash c_1\ (\subseteq A,\ X)$, *simp*)
hence $Y_1 \subseteq Y_1'$
using C **and** F **by** *simp*
moreover obtain $C_2\ Y_2$ **where** I: $\vdash c_2\ (\subseteq A,\ X) = (C_2,\ Y_2)$
by (*cases* $\vdash c_2\ (\subseteq A,\ X)$, *simp*)
hence $Y_2 \subseteq Y_2'$
using D **and** G **by** *simp*
ultimately have $Y_1 \cap Y_2 \subseteq Y'$
using E **by** *blast*
moreover {
fix $x\ ys$
assume $\forall f.\ (\exists ys.\ f = (\lambda x.\ [y \leftarrow ys.\ \mathit{fst}\ y = x]) \wedge ys \in \vdash c_1 \sqcup \vdash c_2) \longrightarrow$
$(\mathit{if}\ f\ x = []\ \mathit{then}\ x \in X\ \mathit{else}\ \mathit{snd}\ (\mathit{last}\ (f\ x)) \neq \mathit{None})$
moreover assume $ys \in \vdash c_1$
hence $ys \in \vdash c_1 \sqcup \vdash c_2$
by (*force simp*: *ctyping1-merge-def*)
ultimately have $\mathit{if}\ [y \leftarrow ys.\ \mathit{fst}\ y = x] = []$
$\mathit{then}\ x \in X\ \mathit{else}\ \mathit{snd}\ (\mathit{last}\ [y \leftarrow ys.\ \mathit{fst}\ y = x]) \neq \mathit{None}$
(**is** *?P*)
by *blast*
moreover assume $\neg$ *?P*
ultimately have *False*
by *contradiction*
}
hence $Y \subseteq Y_1$
using A **and** H **by** (*cases* $A = \{\}$, *auto simp*: *ctyping1-def*)
moreover {

```
    fix x ys
    assume ∀f. (∃ys. f = (λx. [y←ys. fst y = x]) ∧ ys ∈ ⊢ c₁ ⊔ ⊢ c₂) ⟶
      (if f x = [] then x ∈ X else snd (last (f x)) ≠ None)
    moreover assume ys ∈ ⊢ c₂
    hence ys ∈ ⊢ c₁ ⊔ ⊢ c₂
      by (force simp: ctyping1-merge-def)
    ultimately have if [y←ys. fst y = x] = []
      then x ∈ X else snd (last [y←ys. fst y = x]) ≠ None
      (is ?P)
      by blast
    moreover assume ¬ ?P
    ultimately have False
      by contradiction
  }
  hence Y ⊆ Y₂
    using A and I by (cases A = {}, auto simp: ctyping1-def)
  ultimately show ?thesis
    by blast
qed

lemma ctyping1-ctyping2-snd-if:
  assumes
    A: ⊢ IF b THEN c₁ ELSE c₂ (⊆ A, X) = (C, Y) and
    B: (U, False) ⊨ IF b THEN c₁ ELSE c₂ (⊆ A, X) = Some (C', Y') and
    C: ⋀U' p B₁ B₂ C C' Y Y'.
      (U', p) = (insert (Univ? A X, bvars b) U, ⊨ b (⊆ A, X)) ⟹
        (B₁, B₂) = p ⟹ ⊢ c₁ (⊆ B₁, X) = (C, Y) ⟹
          (U', False) ⊨ c₁ (⊆ B₁, X) = Some (C', Y') ⟹ Y ⊆ Y' and
    D: ⋀U' p B₁ B₂ C C' Y Y'.
      (U', p) = (insert (Univ? A X, bvars b) U, ⊨ b (⊆ A, X)) ⟹
        (B₁, B₂) = p ⟹ ⊢ c₂ (⊆ B₂, X) = (C, Y) ⟹
          (U', False) ⊨ c₂ (⊆ B₂, X) = Some (C', Y') ⟹ Y ⊆ Y'
  shows Y ⊆ Y'
proof –
  let ?U' = insert (Univ? A X, bvars b) U
  obtain B₁ B₂ C₁' C₂' Y₁' Y₂' where
    E: (C', Y') = (C₁' ∪ C₂', Y₁' ∩ Y₂') and
    F: ⊨ b (⊆ A, X) = (B₁, B₂) and
    G: (?U', False) ⊨ c₁ (⊆ B₁, X) = Some (C₁', Y₁') and
    H: (?U', False) ⊨ c₂ (⊆ B₂, X) = Some (C₂', Y₂')
    using B by (auto split: option.split-asm prod.split-asm)
  obtain C₁ Y₁ where I: ⊢ c₁ (⊆ B₁, X) = (C₁, Y₁)
    by (cases ⊢ c₁ (⊆ B₁, X), simp)
  hence Y₁ ⊆ Y₁'
    using C and F and G by simp
  moreover obtain C₂ Y₂ where J: ⊢ c₂ (⊆ B₂, X) = (C₂, Y₂)
    by (cases ⊢ c₂ (⊆ B₂, X), simp)
  hence Y₂ ⊆ Y₂'
    using D and F and H by simp
```

ultimately have $Y_1 \cap Y_2 \subseteq Y'$
using E **by** *blast*
moreover have K: $B_1 \cup B_2 = A$
using F **by** (*rule btyping2-un-eq*)
{
fix $x\ x'\ ys$
assume $x \in (\mathit{if}\ B_1 = \{\} \wedge B_2 = \{\}\ \mathit{then}\ \mathit{UNIV}\ \mathit{else}$
$\{x.\ \forall f \in \{\lambda x.\ [y \leftarrow ys.\ \mathit{fst}\ y = x] \mid ys.\ ys \in \vdash c_1 \sqcup \vdash c_2\}.$
$\mathit{if}\ f\ x = []\ \mathit{then}\ x \in X\ \mathit{else}\ \mathit{snd}\ (\mathit{last}\ (f\ x)) \neq \mathit{None}\})$ **and**
$x' \in B_1$
hence $\forall f.\ (\exists ys.\ f = (\lambda x.\ [y \leftarrow ys.\ \mathit{fst}\ y = x]) \wedge ys \in \vdash c_1 \sqcup \vdash c_2) \longrightarrow$
$(\mathit{if}\ f\ x = []\ \mathit{then}\ x \in X\ \mathit{else}\ \mathit{snd}\ (\mathit{last}\ (f\ x)) \neq \mathit{None})$
by (*auto split*: *if-split-asm*)
moreover assume $ys \in \vdash c_1$
hence $ys \in \vdash c_1 \sqcup \vdash c_2$
by (*force simp*: *ctyping1-merge-def*)
ultimately have $\mathit{if}\ [y \leftarrow ys.\ \mathit{fst}\ y = x] = []$
$\mathit{then}\ x \in X\ \mathit{else}\ \mathit{snd}\ (\mathit{last}\ [y \leftarrow ys.\ \mathit{fst}\ y = x]) \neq \mathit{None}$
(**is** *?P*)
by *blast*
moreover assume $\neg$ *?P*
ultimately have *False*
by *contradiction*
}
note $L = \mathit{this}$
{
fix $x\ x'\ ys\ v$
assume $x \in (\mathit{if}\ B_1 = \{\} \wedge B_2 = \{\}\ \mathit{then}\ \mathit{UNIV}\ \mathit{else}$
$\{x.\ \forall f \in \{\lambda x.\ [y \leftarrow ys.\ \mathit{fst}\ y = x] \mid ys.$
$ys \in (\mathit{if}\ v\ \mathit{then} \vdash c_1\ \mathit{else}\ \{\}) \sqcup (\mathit{if}\ \neg\ v\ \mathit{then} \vdash c_2\ \mathit{else}\ \{\})\}.$
$\mathit{if}\ f\ x = []\ \mathit{then}\ x \in X\ \mathit{else}\ \mathit{snd}\ (\mathit{last}\ (f\ x)) \neq \mathit{None}\})$
moreover assume M: $x' \in B_1$ **and**
$(\mathit{if}\ v\ \mathit{then}\ (B_1 \cup B_2,\ \{\})\ \mathit{else}\ (\{\},\ B_1 \cup B_2)) = (B_1,\ B_2)$
hence v
by (*simp split*: *if-split-asm*)
ultimately have
$\forall f.\ (\exists ys.\ f = (\lambda x.\ [y \leftarrow ys.\ \mathit{fst}\ y = x]) \wedge ys \in \vdash c_1 \sqcup \{\}) \longrightarrow$
$(\mathit{if}\ f\ x = []\ \mathit{then}\ x \in X\ \mathit{else}\ \mathit{snd}\ (\mathit{last}\ (f\ x)) \neq \mathit{None})$
using M **by** (*auto split*: *if-split-asm*)
moreover assume $ys \in \vdash c_1$
hence $ys \in \vdash c_1 \sqcup \{\}$
by (*force simp*: *ctyping1-merge-def*)
ultimately have $\mathit{if}\ [y \leftarrow ys.\ \mathit{fst}\ y = x] = []$
$\mathit{then}\ x \in X\ \mathit{else}\ \mathit{snd}\ (\mathit{last}\ [y \leftarrow ys.\ \mathit{fst}\ y = x]) \neq \mathit{None}$
(**is** *?P*)
by *blast*
moreover assume $\neg$ *?P*
ultimately have *False*
by *contradiction*

```
}
note M = this
from A and F and I and K have Y ⊆ Y1
  apply (cases B1 = {})
   apply (fastforce simp: ctyping1-def)
  apply (cases ⊢ b)
  by (auto dest!: btyping1-btyping2 [of - - A X] L M simp: ctyping1-def)
moreover {
  fix x x' ys
  assume x ∈ (if B1 = {} ∧ B2 = {} then UNIV else
    {x. ∀f ∈ {λx. [y←ys. fst y = x] | ys. ys ∈ ⊢ c1 ⊔ ⊢ c2}.
      if f x = [] then x ∈ X else snd (last (f x)) ≠ None}) and
   x' ∈ B2
  hence ∀f. (∃ys. f = (λx. [y←ys. fst y = x]) ∧ ys ∈ ⊢ c1 ⊔ ⊢ c2) ⟶
    (if f x = [] then x ∈ X else snd (last (f x)) ≠ None)
    by (auto split: if-split-asm)
  moreover assume ys ∈ ⊢ c2
  hence ys ∈ ⊢ c1 ⊔ ⊢ c2
    by (force simp: ctyping1-merge-def)
  ultimately have if [y←ys. fst y = x] = []
    then x ∈ X else snd (last [y←ys. fst y = x]) ≠ None
    (is ?P)
    by blast
  moreover assume ¬ ?P
  ultimately have False
    by contradiction
}
note N = this
{
  fix x x' ys v
  assume x ∈ (if B1 = {} ∧ B2 = {} then UNIV else
    {x. ∀f ∈ {λx. [y←ys. fst y = x] | ys.
      ys ∈ (if v then ⊢ c1 else {}) ⊔ (if ¬ v then ⊢ c2 else {})}.
        if f x = [] then x ∈ X else snd (last (f x)) ≠ None})
  moreover assume O: x' ∈ B2 and
   (if v then (B1 ∪ B2, {}) else ({}, B1 ∪ B2)) = (B1, B2)
  hence ¬ v
    by (simp split: if-split-asm)
  ultimately have
   ∀f. (∃ys. f = (λx. [y←ys. fst y = x]) ∧ ys ∈ {} ⊔ ⊢ c2) ⟶
     (if f x = [] then x ∈ X else snd (last (f x)) ≠ None)
    using O by (auto split: if-split-asm)
  moreover assume ys ∈ ⊢ c2
  hence ys ∈ {} ⊔ ⊢ c2
    by (force simp: ctyping1-merge-def)
  ultimately have if [y←ys. fst y = x] = []
    then x ∈ X else snd (last [y←ys. fst y = x]) ≠ None
    (is ?P)
    by blast
```

```
    moreover assume ¬ ?P
    ultimately have False
      by contradiction
  }
  note O = this
  from A and F and J and K have Y ⊆ Y2
    apply (cases B2 = {})
     apply (fastforce simp: ctyping1-def)
    apply (cases ⊢ b)
    by (auto dest!: btyping1-btyping2 [of - - A X] N O simp: ctyping1-def)
  ultimately show ?thesis
    by blast
qed

lemma ctyping1-ctyping2-snd-while:
  assumes
    A: ⊢ WHILE b DO c (⊆ A, X) = (B, Z) and
    B: (U, False) ⊨ WHILE b DO c (⊆ A, X) = Some (B', Z')
  shows Z ⊆ Z'
proof −
  obtain B1 B1' B2 B2' C Y where
    C: ⊨ b (⊆ A, X) = (B1, B2) and
    D: ⊢ c (⊆ B1, X) = (C, Y) and
    E: ⊨ b (⊆ C, Y) = (B1', B2') and
    F: (B', Z') = (B2 ∪ B2', Univ?? B2 X ∩ Y)
    using B by (force split: if-split-asm option.split-asm prod.split-asm)
  have G: B1 ∪ B2 = A
    using C by (rule btyping2-un-eq)
  {
    fix x x'
    assume x ∈ (if B1 = {} ∧ B2 = {} then UNIV else
      {x. ∀f ∈ {λx. [y←ys. fst y = x] | ys. ys = [] ∨ ys ∈ ⊢ c}.
        if f x = [] then x ∈ X else snd (last (f x)) ≠ None}) and
     x' ∈ B2
    hence ∀f ∈ {λx. [y←ys. fst y = x] | ys. ys = [] ∨ ys ∈ ⊢ c}.
      (if f x = [] then x ∈ X else snd (last (f x)) ≠ None)
      by (auto split: if-split-asm)
    hence x ∈ X
      by fastforce
    moreover assume x ∉ X
    ultimately have False
      by contradiction
  }
  note H = this
  {
    fix x x' v
    assume x ∈ (if B1 = {} ∧ B2 = {} then UNIV else
      {x. ∀f ∈ {λx. [y←ys. fst y = x] | ys.
        ys ∈ (if ¬ v then {[]} else {}) ∨ ys ∈ (if v then ⊢ c else {})}.
```

```
      if f x = [] then x ∈ X else snd (last (f x)) ≠ None})
  moreover assume H: x' ∈ B₂ and
   (if v then (B₁ ∪ B₂, {}) else ({}, B₁ ∪ B₂)) = (B₁, B₂)
  hence ¬ v
    by (simp split: if-split-asm)
  ultimately have x ∈ X
    using H by (auto split: if-split-asm)
  moreover assume x ∉ X
  ultimately have False
    by contradiction
}
note I = this
from A and C and G have Z ⊆ Univ?? B₂ X
  apply (cases B₂ = {})
   apply fastforce
  apply (cases ⊢ b)
  by (auto dest!: btyping1-btyping2 [of - - A X] H I simp: ctyping1-def)
moreover {
  fix x
  assume x ∉ Univ?? B₁ {x. ∀f ∈ {λx. [y←ys. fst y = x] | ys. ys ∈ ⊢ c}.
    if f x = [] then x ∈ X else snd (last (f x)) ≠ None}
  moreover from this have B₁ ≠ {}
    by (simp split: if-split-asm)
  ultimately have ¬ (∀f.
    (∃ys. f = (λx. [y←ys. fst y = x]) ∧ (ys = [] ∨ ys ∈ ⊢ c)) ⟶
      (if f x = [] then x ∈ X else snd (last (f x)) ≠ None))
    (is ¬ ?P)
    by (auto split: if-split-asm)
  moreover assume ?P
  ultimately have False
    by contradiction
}
note J = this
{
  fix x v
  assume x ∉ Univ?? B₁ {x. ∀f ∈ {λx. [y←ys. fst y = x] | ys. ys ∈ ⊢ c}.
    if f x = [] then x ∈ X else snd (last (f x)) ≠ None}
  moreover from this have K: B₁ ≠ {}
    by (simp split: if-split-asm)
  ultimately have L: ¬ (∀f.
    (∃ys. f = (λx. [y←ys. fst y = x]) ∧ ys ∈ ⊢ c) ⟶
      (if f x = [] then x ∈ X else snd (last (f x)) ≠ None))
    (is ¬ ?P)
    by (auto split: if-split-asm)
  assume ⊢ b = Some v
  with C and K have v
    by (auto dest: btyping1-btyping2 [of - - A X])
  moreover assume ∀f. (∃ys. f = (λx. [y←ys. fst y = x]) ∧
    (ys ∈ (if ¬ v then {[]} else {}) ∨ ys ∈ (if v then ⊢ c else {}))) ⟶
```

$(\mathit{if}\ f\ x = []\ \mathit{then}\ x \in X\ \mathit{else}\ \mathit{snd}\ (\mathit{last}\ (f\ x)) \neq \mathit{None})$
ultimately have *?P*
by *simp*
with L **have** *False*
by *contradiction*
}
note $K = \mathit{this}$
from A **and** D **and** G **have** $Z \subseteq Y$
apply $(\mathit{cases}\ A = \{\})$
apply (*fastforce simp*: *ctyping1-def*)
apply $(\mathit{cases} \vdash b)$
by (*auto dest*: $J\ K$ *simp*: *ctyping1-def*)
ultimately show *?thesis*
using F **by** *simp*
qed

lemma *ctyping1-ctyping2-snd*:
$[\![\vdash c\ (\subseteq A,\ X) = (C,\ Z);\ (U,\ \mathit{False}) \models c\ (\subseteq A,\ X) = \mathit{Some}\ (C',\ Z')]\!] \Longrightarrow$
$Z \subseteq Z'$
apply (*induction* $(U,\ \mathit{False})\ c\ A\ X$ *arbitrary*: $C\ C'\ Z\ Z'\ U$
rule: *ctyping2.induct*)
apply *fastforce*
apply *fastforce*
apply *fastforce*
apply *fastforce*
apply (*erule ctyping1-ctyping2-snd-seq*, *assumption+*)
apply (*erule ctyping1-ctyping2-snd-or*, *assumption+*)
apply (*erule ctyping1-ctyping2-snd-if*, *assumption+*)
apply (*erule ctyping1-ctyping2-snd-while*, *assumption+*)
done

lemma *ctyping1-ctyping2*:
$[\![\vdash c\ (\subseteq A,\ X) = (C,\ Z);\ (U,\ \mathit{False}) \models c\ (\subseteq A,\ X) = \mathit{Some}\ (C',\ Z')]\!] \Longrightarrow$
$C' \subseteq C \wedge Z \subseteq Z'$
by (*blast dest*: *ctyping1-ctyping2-fst ctyping1-ctyping2-snd*)

lemma *btyping2-aux-approx-1* [*elim*]:
assumes
A: $\models b_1\ (\subseteq A,\ X) = \mathit{Some}\ B_1$ **and**
B: $\models b_2\ (\subseteq A,\ X) = \mathit{Some}\ B_2$ **and**
C: $\mathit{bval}\ b_1\ s$ **and**
D: $\mathit{bval}\ b_2\ s$ **and**
E: $r \in A$ **and**
F: $s = r\ (\subseteq \mathit{state} \cap X)$
shows $\exists\, r' \in B_1 \cap B_2.\ r = r'\ (\subseteq \mathit{state} \cap X)$
proof −
from A **and** C **and** E **and** F **have** $r \in B_1$

by (*frule-tac btyping2-aux-subset*, *drule-tac btyping2-aux-eq*, *auto*)
moreover from B **and** D **and** E **and** F **have** $r \in B_2$
by (*frule-tac btyping2-aux-subset*, *drule-tac btyping2-aux-eq*, *auto*)
ultimately show *?thesis*
by *blast*
qed

lemma *btyping2-aux-approx-2* [*elim*]:
assumes
A: *avars* $a_1 \subseteq$ *state* **and**
B: *avars* $a_2 \subseteq$ *state* **and**
C: *avars* $a_1 \subseteq X$ **and**
D: *avars* $a_2 \subseteq X$ **and**
E: *aval* a_1 $s <$ *aval* a_2 s **and**
F: $r \in A$ **and**
G: $s = r$ ($\subseteq$ *state* $\cap$ X)
shows $\exists\, r'.\ r' \in A \wedge$ *aval* a_1 $r' <$ *aval* a_2 $r' \wedge r = r'$ ($\subseteq$ *state* $\cap$ X)
proof $-$
have *aval* a_1 $s =$ *aval* a_1 $r \wedge$ *aval* a_2 $s =$ *aval* a_2 r
using A **and** B **and** C **and** D **and** G **by** (*blast intro*: *avars-aval*)
thus *?thesis*
using E **and** F **by** *auto*
qed

lemma *btyping2-aux-approx-3* [*elim*]:
assumes
A: *avars* $a_1 \subseteq$ *state* **and**
B: *avars* $a_2 \subseteq$ *state* **and**
C: *avars* $a_1 \subseteq X$ **and**
D: *avars* $a_2 \subseteq X$ **and**
E: $\neg$ *aval* a_1 $s <$ *aval* a_2 s **and**
F: $r \in A$ **and**
G: $s = r$ ($\subseteq$ *state* $\cap$ X)
shows $\exists\, r' \in A - \{s \in A.$ *aval* a_1 $s <$ *aval* a_2 $s\}.\ r = r'$ ($\subseteq$ *state* $\cap$ X)
proof $-$
have *aval* a_1 $s =$ *aval* a_1 $r \wedge$ *aval* a_2 $s =$ *aval* a_2 r
using A **and** B **and** C **and** D **and** G **by** (*blast intro*: *avars-aval*)
thus *?thesis*
using E **and** F **by** *auto*
qed

lemma *btyping2-aux-approx*:
$[\![\models b\ (\subseteq A,\ X) =$ *Some* A'; $s \in$ *Univ* A ($\subseteq$ *state* $\cap$ X)$]\!] \Longrightarrow$
$s \in$ *Univ* (*if bval b s then* A' *else* $A - A'$) ($\subseteq$ *state* $\cap$ X)
by (*induction b arbitrary*: A', *auto dest*: *btyping2-aux-subset*
split: *if-split-asm option.split-asm*)

lemma *btyping2-approx*:
$[\![\models b\ (\subseteq A,\ X) = (B_1,\ B_2)$; $s \in$ *Univ* A ($\subseteq$ *state* $\cap$ X)$]\!] \Longrightarrow$

$s \in \mathit{Univ}\ (\mathit{if}\ \mathit{bval}\ b\ s\ \mathit{then}\ B_1\ \mathit{else}\ B_2)\ (\subseteq \mathit{state} \cap X)$
by (*drule sym*, *simp add*: *btyping2-def split*: *option.split-asm*,
drule btyping2-aux-approx, *auto*)

lemma *ctyping2-approx-assign* [*elim*!]:
$[\![\forall t'.\ \mathit{aval}\ a\ s = t'\ x \longrightarrow (\forall s.\ t' = s(x := \mathit{aval}\ a\ s) \longrightarrow s \notin A) \vee$
$(\exists y \in \mathit{state} \cap X.\ y \neq x \wedge t\ y \neq t'\ y);$
$v \models a\ (\subseteq X);\ t \in A;\ s = t\ (\subseteq \mathit{state} \cap X)]\!] \Longrightarrow \mathit{False}$
by (*drule spec* [*of* - $t(x := \mathit{aval}\ a\ t)$], *cases a*,
(*fastforce simp del*: *aval.simps*(*3*) *intro*: *avars-aval*)+)

lemma *ctyping2-approx-if-1*:
$[\![\mathit{bval}\ b\ s;\ \models b\ (\subseteq A,\ X) = (B_1,\ B_2);\ r \in A;\ s = r\ (\subseteq \mathit{state} \cap X);$
$(\mathit{insert}\ (\mathit{Univ?}\ A\ X,\ \mathit{bvars}\ b)\ U,\ v) \models c_1\ (\subseteq B_1,\ X) = \mathit{Some}\ (C_1,\ Y_1);$
$\bigwedge A\ B\ X\ Y\ U\ v.\ (U,\ v) \models c_1\ (\subseteq A,\ X) = \mathit{Some}\ (B,\ Y) \Longrightarrow$
$\exists r \in A.\ s = r\ (\subseteq \mathit{state} \cap X) \Longrightarrow \exists r' \in B.\ t = r'\ (\subseteq \mathit{state} \cap Y)]\!] \Longrightarrow$
$\exists r' \in C_1 \cup C_2.\ t = r'\ (\subseteq \mathit{state} \cap (Y_1 \cap Y_2))$
by (*drule btyping2-approx*, *blast*, *fastforce*)

lemma *ctyping2-approx-if-2*:
$[\![\neg\ \mathit{bval}\ b\ s;\ \models b\ (\subseteq A,\ X) = (B_1,\ B_2);\ r \in A;\ s = r\ (\subseteq \mathit{state} \cap X);$
$(\mathit{insert}\ (\mathit{Univ?}\ A\ X,\ \mathit{bvars}\ b)\ U,\ v) \models c_2\ (\subseteq B_2,\ X) = \mathit{Some}\ (C_2,\ Y_2);$
$\bigwedge A\ B\ X\ Y\ U\ v.\ (U,\ v) \models c_2\ (\subseteq A,\ X) = \mathit{Some}\ (B,\ Y) \Longrightarrow$
$\exists r \in A.\ s = r\ (\subseteq \mathit{state} \cap X) \Longrightarrow \exists r' \in B.\ t = r'\ (\subseteq \mathit{state} \cap Y)]\!] \Longrightarrow$
$\exists r' \in C_1 \cup C_2.\ t = r'\ (\subseteq \mathit{state} \cap (Y_1 \cap Y_2))$
by (*drule btyping2-approx*, *blast*, *fastforce*)

lemma *ctyping2-approx-while-1* [*elim*]:
$[\![\neg\ \mathit{bval}\ b\ s;\ r \in A;\ s = r\ (\subseteq \mathit{state} \cap X);\ \models b\ (\subseteq A,\ X) = (B,\ \{\})]\!] \Longrightarrow$
$\exists t \in C.\ s = t\ (\subseteq \mathit{state} \cap Y)$
by (*drule btyping2-approx*, *blast*, *simp*)

lemma *ctyping2-approx-while-2* [*elim*]:
$[\![\forall t \in B_2 \cup B_2'.\ \exists x \in \mathit{state} \cap (X \cap Y).\ r\ x \neq t\ x;\ \neg\ \mathit{bval}\ b\ s;$
$r \in A;\ s = r\ (\subseteq \mathit{state} \cap X);\ \models b\ (\subseteq A,\ X) = (B_1,\ B_2)]\!] \Longrightarrow \mathit{False}$
by (*drule btyping2-approx*, *blast*, *auto*)

lemma *ctyping2-approx-while-aux*:
assumes
A: $\models b\ (\subseteq A,\ X) = (B_1,\ B_2)$ **and**
B: $\vdash c\ (\subseteq B_1,\ X) = (C,\ Y)$ **and**
C: $\models b\ (\subseteq C,\ Y) = (B_1',\ B_2')$ **and**
D: $(\{\},\ \mathit{False}) \models c\ (\subseteq B_1,\ X) = \mathit{Some}\ (D,\ Z)$ **and**
E: $(\{\},\ \mathit{False}) \models c\ (\subseteq B_1',\ Y) = \mathit{Some}\ (D',\ Z')$ **and**
F: $r_1 \in A$ **and**
G: $s_1 = r_1\ (\subseteq \mathit{state} \cap X)$ **and**
H: $\mathit{bval}\ b\ s_1$ **and**
I: $\bigwedge C\ B\ Y\ W\ U.\ (\mathit{case} \models b\ (\subseteq C,\ Y)\ \mathit{of}\ (B_1',\ B_2') \Rightarrow$

$case \vdash c\ (\subseteq B_1{}',\ Y)\ of\ (C',\ Y') \Rightarrow$
$case \models b\ (\subseteq C',\ Y')\ of\ (B_1{}'',\ B_2{}'') \Rightarrow if$
$(\forall s \in Univ?\ C\ Y \cup Univ?\ C'\ Y'.\ \forall x \in bvars\ b.\ \forall y.\ s{:}\ dom\ x \rightsquigarrow dom\ y) \wedge$
$(\forall p \in U.\ case\ p\ of\ (B,\ W) \Rightarrow \forall s \in B.\ \forall x \in W.\ \forall y.\ s{:}\ dom\ x \rightsquigarrow dom\ y)$
$then\ case\ (\{\},\ False) \models c\ (\subseteq B_1{}',\ Y)\ of$
$None \Rightarrow None\ |\ Some\ \text{-} \Rightarrow case\ (\{\},\ False) \models c\ (\subseteq B_1{}'',\ Y')\ of$
$None \Rightarrow None\ |\ Some\ \text{-} \Rightarrow Some\ (B_2{}' \cup B_2{}'',\ Univ??\ B_2{}'\ Y \cap Y')$
$else\ None) = Some\ (B,\ W) \Longrightarrow$
$\exists r \in C.\ s_2 = r\ (\subseteq state \cap Y) \Longrightarrow \exists r \in B.\ s_3 = r\ (\subseteq state \cap W)$
(**is** $\bigwedge C\ B\ Y\ W\ U.\ ?P\ C\ B\ Y\ W\ U \Longrightarrow \text{-} \Longrightarrow \text{-}$) **and**
$J{:}\ \bigwedge A\ B\ X\ Y\ U\ v.\ (U,\ v) \models c\ (\subseteq A,\ X) = Some\ (B,\ Y) \Longrightarrow$
$\exists r \in A.\ s_1 = r\ (\subseteq state \cap X) \Longrightarrow \exists r \in B.\ s_2 = r\ (\subseteq state \cap Y)$ **and**
$K{:}\ \forall s \in Univ?\ A\ X \cup Univ?\ C\ Y.\ \forall x \in bvars\ b.\ \forall y.\ s{:}\ dom\ x \rightsquigarrow dom\ y$ **and**
$L{:}\ \forall p \in U.\ \forall B\ W.\ p = (B,\ W) \longrightarrow$
$(\forall s \in B.\ \forall x \in W.\ \forall y.\ s{:}\ dom\ x \rightsquigarrow dom\ y)$
shows $\exists r \in B_2 \cup B_2{}'.\ s_3 = r\ (\subseteq state \cap Univ??\ B_2\ X \cap Y)$
proof –
obtain $C'\ Y'$ **where** $M{:}\ \vdash c\ (\subseteq B_1{}',\ Y) = (C',\ Y')$
by (*cases* $\vdash c\ (\subseteq B_1{}',\ Y)$, *simp*)
obtain $B_1{}''\ B_2{}''$ **where** $N{:}\ (B_1{}'',\ B_2{}'') = \models b\ (\subseteq C',\ Y')$
by (*cases* $\models b\ (\subseteq C',\ Y')$, *simp*)
let $?B = B_2{}' \cup B_2{}''$
let $?W = Univ??\ B_2{}'\ Y \cap Y'$
have $\vdash c\ (\subseteq C,\ Y) = (C,\ Y)$
using *ctyping1-idem* **and** B **by** *auto*
moreover have $B_1{}' \subseteq C$
using C **by** (*blast dest*: *btyping2-un-eq*)
ultimately have $O{:}\ C' \subseteq C \wedge Y \subseteq Y'$
by (*rule ctyping1-mono* [*OF* - *M*], *simp*)
hence $Univ?\ C'\ Y' \subseteq Univ?\ C\ Y$
by (*auto simp*: *univ-states-if-def*)
moreover from I **have** $?P\ C\ ?B\ Y\ ?W\ U \Longrightarrow$
$\exists r \in C.\ s_2 = r\ (\subseteq state \cap Y) \Longrightarrow \exists r \in ?B.\ s_3 = r\ (\subseteq state \cap ?W)$ **.**
ultimately have $(case\ (\{\},\ False) \models c\ (\subseteq B_1{}'',\ Y')\ of$
$None \Rightarrow None\ |\ Some\ \text{-} \Rightarrow Some\ (?B,\ ?W)) = Some\ (?B,\ ?W) \Longrightarrow$
$\exists r \in C.\ s_2 = r\ (\subseteq state \cap Y) \Longrightarrow \exists r \in ?B.\ s_3 = r\ (\subseteq state \cap ?W)$
using C **and** E **and** K **and** L **and** M **and** N
by (*fastforce split*: *if-split-asm prod.split-asm*)
moreover have $P{:}\ B_1{}'' \subseteq B_1{}' \wedge B_2{}'' \subseteq B_2{}'$
by (*metis btyping2-mono C N O*)
hence $\exists D''\ Z''.\ (\{\},\ False) \models c\ (\subseteq B_1{}'',\ Y') =$
$Some\ (D'',\ Z'') \wedge D'' \subseteq D' \wedge Z' \subseteq Z''$
using E **and** O **by** (*auto intro*: *ctyping2-mono*)
ultimately have
$\exists r \in C.\ s_2 = r\ (\subseteq state \cap Y) \Longrightarrow \exists r \in ?B.\ s_3 = r\ (\subseteq state \cap ?W)$
by *fastforce*
moreover from A **and** D **and** F **and** G **and** H **and** J **obtain** r_2 **where**
$r_2 \in D$ **and** $s_2 = r_2\ (\subseteq state \cap Z)$
by (*drule-tac btyping2-approx*, *blast*, *force*)

moreover have $D \subseteq C \wedge Y \subseteq Z$
using B **and** D **by** (*rule ctyping1-ctyping2*)
ultimately obtain r_3 **where** Q: $r_3 \in ?B$ **and** R: $s_3 = r_3\ (\subseteq \mathit{state} \cap ?W)$
by *blast*
show *?thesis*
proof (*rule bexI* [*of* - r_3])
show $s_3 = r_3\ (\subseteq \mathit{state} \cap \mathit{Univ??}\ B_2\ X \cap Y)$
using O **and** R **by** *auto*
next
show $r_3 \in B_2 \cup B_2{}'$
using P **and** Q **by** *blast*
qed
qed

lemmas *ctyping2-approx-while-3* =
ctyping2-approx-while-aux [**where** $B_2 = \{\}$, *simplified*]

lemma *ctyping2-approx-while-4*:
$[\![\models b\ (\subseteq A,\ X) = (B_1,\ B_2);$
$\vdash c\ (\subseteq B_1,\ X) = (C,\ Y);$
$\models b\ (\subseteq C,\ Y) = (B_1{}',\ B_2{}');$
$(\{\},\ \mathit{False}) \models c\ (\subseteq B_1,\ X) = \mathit{Some}\ (D,\ Z);$
$(\{\},\ \mathit{False}) \models c\ (\subseteq B_1{}',\ Y) = \mathit{Some}\ (D',\ Z');$
$r_1 \in A;\ s_1 = r_1\ (\subseteq \mathit{state} \cap X);\ \mathit{bval}\ b\ s_1;$
$\bigwedge C\ B\ Y\ W\ U.\ (\mathit{case} \models b\ (\subseteq C,\ Y)\ \mathit{of}\ (B_1{}',\ B_2{}') \Rightarrow$
$\mathit{case} \vdash c\ (\subseteq B_1{}',\ Y)\ \mathit{of}\ (C',\ Y') \Rightarrow$
$\mathit{case} \models b\ (\subseteq C',\ Y')\ \mathit{of}\ (B_1{}'',\ B_2{}'') \Rightarrow$
$\mathit{if}\ (\forall s \in \mathit{Univ?}\ C\ Y \cup \mathit{Univ?}\ C'\ Y'.\ \forall x \in \mathit{bvars}\ b.\ \forall y.\ s\text{: } \mathit{dom}\ x \leadsto \mathit{dom}\ y) \wedge$
$(\forall p \in U.\ \mathit{case}\ p\ \mathit{of}\ (B,\ W) \Rightarrow \forall s \in B.\ \forall x \in W.\ \forall y.\ s\text{: } \mathit{dom}\ x \leadsto \mathit{dom}\ y)$
$\mathit{then}\ \mathit{case}\ (\{\},\ \mathit{False}) \models c\ (\subseteq B_1{}',\ Y)\ \mathit{of}$
$\mathit{None} \Rightarrow \mathit{None} \mid \mathit{Some}\ \text{-} \Rightarrow \mathit{case}\ (\{\},\ \mathit{False}) \models c\ (\subseteq B_1{}'',\ Y')\ \mathit{of}$
$\mathit{None} \Rightarrow \mathit{None} \mid \mathit{Some}\ \text{-} \Rightarrow \mathit{Some}\ (B_2{}' \cup B_2{}'',\ \mathit{Univ??}\ B_2{}'\ Y \cap Y')$
$\mathit{else}\ \mathit{None}) = \mathit{Some}\ (B,\ W) \Longrightarrow$
$\exists r \in C.\ s_2 = r\ (\subseteq \mathit{state} \cap Y) \Longrightarrow \exists r \in B.\ s_3 = r\ (\subseteq \mathit{state} \cap W);$
$\bigwedge A\ B\ X\ Y\ U\ v.\ (U,\ v) \models c\ (\subseteq A,\ X) = \mathit{Some}\ (B,\ Y) \Longrightarrow$
$\exists r \in A.\ s_1 = r\ (\subseteq \mathit{state} \cap X) \Longrightarrow \exists r \in B.\ s_2 = r\ (\subseteq \mathit{state} \cap Y);$
$\forall s \in \mathit{Univ?}\ A\ X \cup \mathit{Univ?}\ C\ Y.\ \forall x \in \mathit{bvars}\ b.\ \forall y.\ s\text{: } \mathit{dom}\ x \leadsto \mathit{dom}\ y;$
$\forall p \in U.\ \forall B\ W.\ p = (B,\ W) \longrightarrow (\forall s \in B.\ \forall x \in W.\ \forall y.\ s\text{: } \mathit{dom}\ x \leadsto \mathit{dom}\ y);$
$\forall r \in B_2 \cup B_2{}'.\ \exists x \in \mathit{state} \cap (X \cap Y).\ s_3\ x \neq r\ x]\!] \Longrightarrow$
False
by (*drule ctyping2-approx-while-aux*, *assumption*+, *auto*)

lemma *ctyping2-approx*:
$[\![(c,\ s,\ p) \Rightarrow (t,\ q);\ (U,\ v) \models c\ (\subseteq A,\ X) = \mathit{Some}\ (B,\ Y);$
$s \in \mathit{Univ}\ A\ (\subseteq \mathit{state} \cap X)]\!] \Longrightarrow t \in \mathit{Univ}\ B\ (\subseteq \mathit{state} \cap Y)$
proof (*induction* $(c,\ s,\ p)\ (t,\ q)$ *arbitrary*: $A\ B\ X\ Y\ U\ v\ c\ s\ p\ t\ q$
rule: *big-step.induct*)
fix $A\ C\ X\ Z\ U\ v\ c_1\ c_2\ s\ p\ t\ q$ **and** p' :: *stage*
show

$\llbracket \bigwedge r\ q\ A\ B\ X\ Y\ U\ v.\ p' = (r,\ q) \Longrightarrow$
$(U,\ v) \models c_1\ (\subseteq A,\ X) = \mathit{Some}\ (B,\ Y) \Longrightarrow$
$s \in \mathit{Univ}\ A\ (\subseteq \mathit{state} \cap X) \Longrightarrow r \in \mathit{Univ}\ B\ (\subseteq \mathit{state} \cap Y);$
$\bigwedge r\ q\ B\ C\ Y\ Z\ U\ v.\ p' = (r,\ q) \Longrightarrow$
$(U,\ v) \models c_2\ (\subseteq B,\ Y) = \mathit{Some}\ (C,\ Z) \Longrightarrow$
$r \in \mathit{Univ}\ B\ (\subseteq \mathit{state} \cap Y) \Longrightarrow t \in \mathit{Univ}\ C\ (\subseteq \mathit{state} \cap Z);$
$(U,\ v) \models c_1;;\ c_2\ (\subseteq A,\ X) = \mathit{Some}\ (C,\ Z);$
$s \in \mathit{Univ}\ A\ (\subseteq \mathit{state} \cap X)\rrbracket \Longrightarrow$
$t \in \mathit{Univ}\ C\ (\subseteq \mathit{state} \cap Z)$
by (*cases* p', *auto split*: *option.split-asm prod.split-asm*)
next
fix $A\ C\ X\ Y\ U\ v\ c_1\ c_2\ s\ p\ t\ q$
show
$\llbracket \bigwedge A\ C\ X\ Y\ U\ v.\ (U,\ v) \models c_1\ (\subseteq A,\ X) = \mathit{Some}\ (C,\ Y) \Longrightarrow$
$s \in \mathit{Univ}\ A\ (\subseteq \mathit{state} \cap X) \Longrightarrow t \in \mathit{Univ}\ C\ (\subseteq \mathit{state} \cap Y);$
$(U,\ v) \models c_1\ \mathit{OR}\ c_2\ (\subseteq A,\ X) = \mathit{Some}\ (C,\ Y);$
$s \in \mathit{Univ}\ A\ (\subseteq \mathit{state} \cap X)\rrbracket \Longrightarrow$
$t \in \mathit{Univ}\ C\ (\subseteq \mathit{state} \cap Y)$
by (*fastforce split*: *option.split-asm*)
next
fix $A\ C\ X\ Y\ U\ v\ c_1\ c_2\ s\ p\ t\ q$
show
$\llbracket \bigwedge A\ C\ X\ Y\ U\ v.\ (U,\ v) \models c_2\ (\subseteq A,\ X) = \mathit{Some}\ (C,\ Y) \Longrightarrow$
$s \in \mathit{Univ}\ A\ (\subseteq \mathit{state} \cap X) \Longrightarrow t \in \mathit{Univ}\ C\ (\subseteq \mathit{state} \cap Y);$
$(U,\ v) \models c_1\ \mathit{OR}\ c_2\ (\subseteq A,\ X) = \mathit{Some}\ (C,\ Y);$
$s \in \mathit{Univ}\ A\ (\subseteq \mathit{state} \cap X)\rrbracket \Longrightarrow$
$t \in \mathit{Univ}\ C\ (\subseteq \mathit{state} \cap Y)$
by (*fastforce split*: *option.split-asm*)
next
fix $A\ B\ X\ Y\ U\ v\ b\ c_1\ c_2\ s\ p\ t\ q$
show
$\llbracket \mathit{bval}\ b\ s;\ (c_1,\ s,\ p) \Rightarrow (t,\ q);$
$\bigwedge A\ C\ X\ Y\ U\ v.\ (U,\ v) \models c_1\ (\subseteq A,\ X) = \mathit{Some}\ (C,\ Y) \Longrightarrow$
$s \in \mathit{Univ}\ A\ (\subseteq \mathit{state} \cap X) \Longrightarrow t \in \mathit{Univ}\ C\ (\subseteq \mathit{state} \cap Y);$
$(U,\ v) \models \mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2\ (\subseteq A,\ X) = \mathit{Some}\ (B,\ Y);$
$s \in \mathit{Univ}\ A\ (\subseteq \mathit{state} \cap X)\rrbracket \Longrightarrow$
$t \in \mathit{Univ}\ B\ (\subseteq \mathit{state} \cap Y)$
by (*auto split*: *option.split-asm prod.split-asm*,
rule ctyping2-approx-if-1)
next
fix $A\ B\ X\ Y\ U\ v\ b\ c_1\ c_2\ s\ p\ t\ q$
show
$\llbracket \neg\ \mathit{bval}\ b\ s;\ (c_2,\ s,\ p) \Rightarrow (t,\ q);$
$\bigwedge A\ C\ X\ Y\ U\ v.\ (U,\ v) \models c_2\ (\subseteq A,\ X) = \mathit{Some}\ (C,\ Y) \Longrightarrow$
$s \in \mathit{Univ}\ A\ (\subseteq \mathit{state} \cap X) \Longrightarrow t \in \mathit{Univ}\ C\ (\subseteq \mathit{state} \cap Y);$
$(U,\ v) \models \mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2\ (\subseteq A,\ X) = \mathit{Some}\ (B,\ Y);$
$s \in \mathit{Univ}\ A\ (\subseteq \mathit{state} \cap X)\rrbracket \Longrightarrow$
$t \in \mathit{Univ}\ B\ (\subseteq \mathit{state} \cap Y)$
by (*auto split*: *option.split-asm prod.split-asm*,

rule ctyping2-approx-if-2)
next
fix $A\ B\ X\ Y\ U\ v\ b\ c\ s_1\ p_1\ s_2\ p_2\ s_3\ p_3$
show
$[\![\mathit{bval}\ b\ s_1;\ (c,\ s_1,\ p_1) \Rightarrow (s_2,\ p_2);$
$\bigwedge A\ B\ X\ Y\ U\ v.\ (U,\ v) \models c\ (\subseteq A,\ X) = \mathit{Some}\ (B,\ Y) \Longrightarrow$
$s_1 \in \mathit{Univ}\ A\ (\subseteq \mathit{state} \cap X) \Longrightarrow s_2 \in \mathit{Univ}\ B\ (\subseteq \mathit{state} \cap Y);$
$(\mathit{WHILE}\ b\ \mathit{DO}\ c,\ s_2,\ p_2) \Rightarrow (s_3,\ p_3);$
$\bigwedge A\ B\ X\ Y\ U\ v.\ (U,\ v) \models \mathit{WHILE}\ b\ \mathit{DO}\ c\ (\subseteq A,\ X) = \mathit{Some}\ (B,\ Y) \Longrightarrow$
$s_2 \in \mathit{Univ}\ A\ (\subseteq \mathit{state} \cap X) \Longrightarrow s_3 \in \mathit{Univ}\ B\ (\subseteq \mathit{state} \cap Y);$
$(U,\ v) \models \mathit{WHILE}\ b\ \mathit{DO}\ c\ (\subseteq A,\ X) = \mathit{Some}\ (B,\ Y);$
$s_1 \in \mathit{Univ}\ A\ (\subseteq \mathit{state} \cap X)]\!] \Longrightarrow$
$s_3 \in \mathit{Univ}\ B\ (\subseteq \mathit{state} \cap Y)$
by (*auto split*: *if-split-asm option.split-asm prod.split-asm*,
erule-tac [*2*] *ctyping2-approx-while-4*,
erule ctyping2-approx-while-3)
qed (*auto split*: *if-split-asm option.split-asm prod.split-asm*)

end

end

5 Sufficiency of well-typedness for information flow correctness: propaedeutic lemmas

theory *Correctness-Lemmas*
imports *Overapproximation*
begin

The purpose of this section is to prove some further lemmas used in the proof of the main theorem, which is the subject of the next section.

The proof of one of these lemmas uses the lemmas *ctyping1-idem* and *ctyping2-approx* proven in the previous sections.

5.1 Global context proofs

lemma *bvars-bval*:
$s = t\ (\subseteq \mathit{bvars}\ b) \Longrightarrow \mathit{bval}\ b\ s = \mathit{bval}\ b\ t$
by (*induction b*, *simp-all*, *rule arg-cong2*, *auto intro*: *avars-aval*)

lemma *eq-streams-subset*:
$[\![f = f'\ (\subseteq vs,\ vs',\ T);\ T' \subseteq T]\!] \Longrightarrow f = f'\ (\subseteq vs,\ vs',\ T')$
by (*auto simp*: *eq-streams-def*)

lemma *flow-append-1*:
assumes A: $\bigwedge \mathit{cfs}' :: (\mathit{com} \times \mathit{stage})\ \mathit{list}.$

c # map fst (cfs :: (com × stage) list) = map fst cfs′ ⟹
flow-aux (map fst cfs′ @ map fst cfs″) =
flow-aux (map fst cfs′) @ flow-aux (map fst cfs″)
shows *flow-aux (c # map fst cfs @ map fst cfs″) =*
flow-aux (c # map fst cfs) @ flow-aux (map fst cfs″)
using *A [of (c, λx. 0, λx n. 0, [], []) # cfs]* **by** *simp*

lemma *flow-append*:
flow (cfs @ cfs′) = flow cfs @ flow cfs′
by (*simp add*: *flow-def*, *induction map fst cfs arbitrary*: *cfs*
rule: *flow-aux.induct*, *auto*, *rule flow-append-1*)

lemma *flow-cons*:
flow (cf # cfs) = flow-aux (fst cf # []) @ flow cfs
by (*subgoal-tac cf # cfs = [cf] @ cfs*, *simp only*: *flow-append*,
simp-all add: *flow-def*)

lemma *in-flow-length*:
length [p←in-flow cs vs f. fst p = x] = length [c←cs. c = IN x]
by (*induction cs vs f rule*: *in-flow.induct*, *simp-all*)

lemma *in-flow-append*:
in-flow (cs @ cs′) vs f =
in-flow cs vs f @ in-flow cs′ (vs @ in-flow cs vs f) f
by (*induction cs′ vs f rule*: *in-flow.induct*,
(*simp only*: *append-assoc [symmetric] in-flow.simps*,
simp add: *in-flow-length ac-simps*)+)

lemma *in-flow-one*:
in-flow [c] vs f = (case c of
IN x ⇒ [(x, f x (length [p←vs. fst p = x]))] | - ⇒ [])
by (*subst append-Nil [symmetric]*, *cases c*, *simp-all only*: *in-flow.simps*,
simp-all)

lemma *run-flow-append*:
run-flow (cs @ cs′) vs s f =
run-flow cs′ (vs @ in-flow cs vs f) (run-flow cs vs s f) f
by (*induction cs′ vs s f rule*: *run-flow.induct*,
(*simp only*: *append-assoc [symmetric] run-flow.simps*,
simp add: *in-flow-length ac-simps*)+)

lemma *run-flow-one*:
run-flow [c] vs s f = (case c of
x ::= a ⇒ s(x := aval a s) |
IN x ⇒ s(x := f x (length [p←vs. fst p = x])) |
- ⇒ s)
by (*subst append-Nil [symmetric]*, *cases c*, *simp-all only*: *run-flow.simps*,

simp-all)

lemma run-flow-observe:
run-flow (⟨X⟩ # cs) vs s f = run-flow cs vs s f
apply (rule subst [of ([] @ [⟨X⟩]) @ cs -
λcs′. run-flow cs′ vs s f = run-flow cs vs s f])
apply fastforce
by (subst run-flow-append, simp only: in-flow.simps run-flow.simps, simp)

lemma out-flow-append:
out-flow (cs @ cs′) vs s f =
out-flow cs vs s f @
out-flow cs′ (vs @ in-flow cs vs f) (run-flow cs vs s f) f
by (induction cs′ vs s f rule: out-flow.induct,
(simp only: append-assoc [symmetric] out-flow.simps,
simp add: run-flow-append)+)

lemma out-flow-one:
out-flow [c] vs s f = (case c of
OUT x ⇒ [(x, s x)] | - ⇒ [])
by (subst append-Nil [symmetric], cases c, simp-all only: out-flow.simps,
simp-all)

lemma no-upd-empty:
no-upd cs {}
by (induction cs {} :: vname set rule: no-upd.induct, simp-all)

lemma no-upd-append:
no-upd (cs @ cs′) X = (no-upd cs X ∧ no-upd cs′ X)
by (induction cs X rule: no-upd.induct, simp-all)

lemma no-upd-in-flow:
no-upd cs X ⟹ [p←in-flow cs vs f. fst p ∈ X] = []
by (induction cs vs f rule: in-flow.induct, simp-all add: no-upd-append)

lemma no-upd-run-flow:
no-upd cs X ⟹ run-flow cs vs s f = s (⊆ X)
by (induction cs vs s f rule: run-flow.induct, auto simp: Let-def no-upd-append)

lemma no-upd-out-flow:
no-upd cs X ⟹ [p←out-flow cs vs s f. fst p ∈ X] = []
by (induction cs vs s f rule: out-flow.induct, simp-all add: no-upd-append)

lemma small-stepsl-append:
⟦cf →*{cfs} cf′; cf′ →*{cfs′} cf″⟧ ⟹ cf →*{cfs @ cfs′} cf″
by (induction cf′ cfs′ cf″ rule: small-stepsl.induct, simp,

simp only: *append-assoc* [*symmetric*] *small-stepsl.simps*)

lemma *small-step-stream*:
(*c*, *s*, *f*, *vs*, *ws*) → (*c′*, *p*) ⟹ ∃ *s′ vs′ ws′*. *p* = (*s′*, *f*, *vs′*, *ws′*)
by (*induction* (*c*, *s*, *f*, *vs*, *ws*) (*c′*, *p*) *arbitrary*: *c s f vs ws c′ p*
rule: *small-step.induct*, *simp-all*)

lemma *small-stepsl-stream*:
(*c*, *s*, *f*, *vs*, *ws*) →∗{*cfs*} (*c′*, *p*) ⟹ ∃ *s′ vs′ ws′*. *p* = (*s′*, *f*, *vs′*, *ws′*)
by (*induction* (*c*, *s*, *f*, *vs*, *ws*) *cfs* (*c′*, *p*) *arbitrary*: *c s f vs ws c′ p*
rule: *small-stepsl.induct*, *auto dest*: *small-step-stream*)

lemma *small-steps-stepsl-1*:
∃ *cfs*. *cf* →∗{*cfs*} *cf*
by (*rule exI* [*of* - []], *simp*)

lemma *small-steps-stepsl-2*:
⟦*cf* → *cf′*; *cf′* →∗{*cfs*} *cf″*⟧ ⟹ ∃ *cfs′*. *cf* →∗{*cfs′*} *cf″*
by (*rule exI* [*of* - [*cf*] @ *cfs*], *rule small-stepsl-append*,
subst append-Nil [*symmetric*], *simp only*: *small-stepsl.simps*)

lemma *small-steps-stepsl*:
cf →∗ *cf′* ⟹ ∃ *cfs*. *cf* →∗{*cfs*} *cf′*
by (*induction cf cf′ rule*: *star.induct*, *rule small-steps-stepsl-1*,
blast intro: *small-steps-stepsl-2*)

lemma *small-stepsl-steps*:
cf →∗{*cfs*} *cf′* ⟹ *cf* →∗ *cf′*
by (*induction cf cfs cf′ rule*: *small-stepsl.induct*, *auto intro*: *star-trans*)

lemma *small-steps-stream*:
(*c*, *s*, *f*, *vs*, *ws*) →∗ (*c′*, *p*) ⟹ ∃ *s′ vs′ ws′*. *p* = (*s′*, *f*, *vs′*, *ws′*)
by (*blast dest*: *small-steps-stepsl intro*: *small-stepsl-stream*)

lemma *small-stepsl-cons-1*:
cf →∗{[*cf′*]} *cf″* ⟹ *cf′* = *cf* ∧ (∃ *cf′*. *cf* → *cf′* ∧ *cf′* →∗{[]} *cf″*)
by (*subst* (*asm*) *append-Nil* [*symmetric*], *simp only*: *small-stepsl.simps*,
cases cf″, *simp*)

lemma *small-stepsl-cons-2*:
⟦*cf* →∗{*cf′* # *cfs*} *cf″* ⟹
    cf′ = *cf* ∧ (∃ *cf′*. *cf* → *cf′* ∧ *cf′* →∗{*cfs*} *cf″*);
  cf →∗{*cf′* # *cfs* @ [*cf″*]} *cf‴*⟧ ⟹
    cf′ = *cf* ∧ (∃ *cf′*. *cf* → *cf′* ∧ *cf′* →∗{*cfs* @ [*cf″*]} *cf‴*)
by (*simp only*: *append-Cons* [*symmetric*], *simp only*: *small-stepsl.simps*, *simp*)

lemma *small-stepsl-cons*:

$cf \rightarrow\!*\{cf' \# \mathit{cfs}\}\ cf'' \Longrightarrow$
$\quad cf' = cf\ \wedge$
$\quad (\exists\, cf'.\ cf \rightarrow cf' \wedge cf' \rightarrow\!*\{\mathit{cfs}\}\ cf'')$
by (*induction cf cfs cf'' rule*: *small-stepsl.induct*,
erule small-stepsl-cons-1, *rule small-stepsl-cons-2*)

lemma *small-stepsl-skip*:
$(\mathit{SKIP},\ p) \rightarrow\!*\{\mathit{cfs}\}\ cf \Longrightarrow cf = (\mathit{SKIP},\ p) \wedge \mathit{flow}\ \mathit{cfs} = []$
by (*induction* (*SKIP*, *p*) *cfs cf rule*: *small-stepsl.induct*,
auto simp: *flow-def*)

lemma *small-stepsl-assign*:
$(x ::= a,\ s,\ p) \rightarrow\!*\{\mathit{cfs}\}\ cf \Longrightarrow$
$\quad cf = (x ::= a,\ s,\ p)\ \wedge$
$\quad\ \ \mathit{flow}\ \mathit{cfs} = []\ \vee$
$\quad cf = (\mathit{SKIP},\ s(x := \mathit{aval}\ a\ s),\ p)\ \wedge$
$\quad\ \ \mathit{flow}\ \mathit{cfs} = [x ::= a]$
by (*induction* (*x* ::= *a* :: *com*, *s*, *p*) *cfs cf rule*: *small-stepsl.induct*,
force simp: *flow-def*, *auto simp*: *flow-append*, *simp-all add*: *flow-def*)

lemma *small-stepsl-input*:
$(\mathit{IN}\ x,\ s,\ f,\ \mathit{vs},\ \mathit{ws}) \rightarrow\!*\{\mathit{cfs}\}\ cf \Longrightarrow$
$\quad cf = (\mathit{IN}\ x,\ s,\ f,\ \mathit{vs},\ \mathit{ws})\ \wedge$
$\quad\ \ \mathit{flow}\ \mathit{cfs} = []\ \vee$
$\quad (\mathit{let}\ n = \mathit{length}\ [p\leftarrow \mathit{vs}.\ \mathit{fst}\ p = x]$
$\quad\ \ \mathit{in}\ cf = (\mathit{SKIP},\ s(x := f\ x\ n),\ f,\ \mathit{vs}\ @\ [(x,\ f\ x\ n)],\ \mathit{ws})\ \wedge$
$\quad\ \ \mathit{flow}\ \mathit{cfs} = [\mathit{IN}\ x])$
by (*induction* (*IN x* :: *com*, *s*, *f*, *vs*, *ws*) *cfs cf rule*:
small-stepsl.induct, *force simp*: *flow-def*, *auto simp*: *Let-def flow-append*,
simp-all add: *flow-def*)

lemma *small-stepsl-output*:
$(\mathit{OUT}\ x,\ s,\ f,\ \mathit{vs},\ \mathit{ws}) \rightarrow\!*\{\mathit{cfs}\}\ cf \Longrightarrow$
$\quad cf = (\mathit{OUT}\ x,\ s,\ f,\ \mathit{vs},\ \mathit{ws})\ \wedge$
$\quad\ \ \mathit{flow}\ \mathit{cfs} = []\ \vee$
$\quad cf = (\mathit{SKIP},\ s,\ f,\ \mathit{vs},\ \mathit{ws}\ @\ [(x,\ s\ x)])\ \wedge$
$\quad\ \ \mathit{flow}\ \mathit{cfs} = [\mathit{OUT}\ x]$
by (*induction* (*OUT x* :: *com*, *s*, *f*, *vs*, *ws*) *cfs cf rule*:
small-stepsl.induct, *force simp*: *flow-def*, *auto simp*: *flow-append*,
simp-all add: *flow-def*)

lemma *small-stepsl-seq-1*:
$(c_1;;\ c_2,\ p) \rightarrow\!*\{[]\}\ (c,\ q) \Longrightarrow$
$\quad (\exists\, c'\ \mathit{cfs}'.\ c = c';;\ c_2\ \wedge$
$\quad\ \ (c_1,\ p) \rightarrow\!*\{\mathit{cfs}'\}\ (c',\ q)\ \wedge$
$\quad\ \ \mathit{flow}\ [] = \mathit{flow}\ \mathit{cfs}')\ \vee$
$\quad (\exists\, p'\ \mathit{cfs}'\ \mathit{cfs}''.\ \mathit{length}\ \mathit{cfs}'' < \mathit{length}\ []\ \wedge$

```
      (c₁, p) →*{cfs'} (SKIP, p') ∧
      (c₂, p') →*{cfs''} (c, q) ∧
      flow [] = flow cfs' @ flow cfs'')
by force

lemma small-stepsl-seq-2:
  assumes A: ⋀c' q'. cf = (c', q') ⟹
    (c₁;; c₂, p) →*{cfs} (c', q') ⟹
      (∃ c'' cfs'. c' = c'';; c₂ ∧
        (c₁, p) →*{cfs'} (c'', q') ∧
        flow cfs = flow cfs') ∨
      (∃ p' cfs' cfs''. length cfs'' < length cfs ∧
        (c₁, p) →*{cfs'} (SKIP, p') ∧
        (c₂, p') →*{cfs''} (c', q') ∧
        flow cfs = flow cfs' @ flow cfs'')
    (is ⋀c' q'. - ⟹ - ⟹
      (∃ c'' cfs'. ?P c' q' c'' cfs') ∨
      (∃ p' cfs' cfs''. ?Q c' q' p' cfs' cfs''))
  assumes B: (c₁;; c₂, p) →*{cfs @ [cf]} (c, q)
  shows
   (∃ c' cfs'. c = c';; c₂ ∧
      (c₁, p) →*{cfs'} (c', q) ∧
      flow (cfs @ [cf]) = flow cfs') ∨
    (∃ p' cfs' cfs''. length cfs'' < length (cfs @ [cf]) ∧
      (c₁, p) →*{cfs'} (SKIP, p') ∧
      (c₂, p') →*{cfs''} (c, q) ∧
      flow (cfs @ [cf]) = flow cfs' @ flow cfs'')
    (is ?T ∨ ?U)
proof (cases cf)
  fix c' q'
  assume C: cf = (c', q')
  moreover {
    assume D: (c', q') → (c, q)
    assume
     (∃ c'' cfs'. ?P c' q' c'' cfs') ∨
      (∃ p' cfs' cfs''. ?Q c' q' p' cfs' cfs'')
    hence ?thesis
    proof
      assume ∃ c'' cfs'. ?P c' q' c'' cfs'
      then obtain c'' and cfs' where
        E: c' = c'';; c₂ and
        F: (c₁, p) →*{cfs'} (c'', q') and
        G: flow cfs = flow cfs'
        by blast
      hence (c'';; c₂, q') → (c, q)
        using D by simp
      moreover {
        assume
          H: c'' = SKIP and
```

```
      I: (c, q) = (c₂, q')
    have ?U
    proof (rule exI [of - q'], rule exI [of - cfs'],
     rule exI [of - []])
      from C and E and F and G and H and I show
       length [] < length (cfs @ [cf]) ∧
        (c₁, p) →*{cfs'} (SKIP, q') ∧
        (c₂, q') →*{[]} (c, q) ∧
        flow (cfs @ [cf]) = flow cfs' @ flow []
        by (simp add: flow-append, simp add: flow-def)
    qed
  }
  moreover {
    fix d q''
    assume
      H: (c'', q') → (d, q'') and
      I: (c, q) = (d;; c₂, q'')
    have ?T
    proof (rule exI [of - d],
     rule exI [of - cfs' @ [(c'', q')]])
      from C and E and F and G and H and I show
       c = d;; c₂ ∧
        (c₁, p) →*{cfs' @ [(c'', q')]} (d, q) ∧
        flow (cfs @ [cf]) = flow (cfs' @ [(c'', q')])
        by (simp add: flow-append, simp add: flow-def)
    qed
  }
  ultimately show ?thesis
    by blast
next
  assume ∃ p' cfs' cfs''. ?Q c' q' p' cfs' cfs''
  then obtain p' and cfs' and cfs'' where
    E: length cfs'' < length cfs and
    F: (c₁, p) →*{cfs'} (SKIP, p') and
    G: (c₂, p') →*{cfs''} (c', q') and
    H: flow cfs = flow cfs' @ flow cfs''
    by blast
  show ?thesis
  proof (rule disjI2, rule exI [of - p'], rule exI [of - cfs'],
   rule exI [of - cfs'' @ [(c', q')]])
    from C and D and E and F and G and H show
     length (cfs'' @ [(c', q')]) < length (cfs @ [cf]) ∧
      (c₁, p) →*{cfs'} (SKIP, p') ∧
      (c₂, p') →*{cfs'' @ [(c', q')]} (c, q) ∧
      flow (cfs @ [cf]) = flow cfs' @ flow (cfs'' @ [(c', q')])
      by (simp add: flow-append)
  qed
 qed
}
```

```
  ultimately show ?thesis
    using A and B by simp
qed

lemma small-stepsl-seq:
 (c1;; c2, p) →∗{cfs} (c, q) ⟹
    (∃ c′ cfs′. c = c′;; c2 ∧
      (c1, p) →∗{cfs′} (c′, q) ∧
      flow cfs = flow cfs′) ∨
    (∃ p′ cfs′ cfs′′. length cfs′′ < length cfs ∧
      (c1, p) →∗{cfs′} (SKIP, p′) ∧ (c2, p′) →∗{cfs′′} (c, q) ∧
      flow cfs = flow cfs′ @ flow cfs′′)
by (induction (c1;; c2, p) cfs (c, q) arbitrary: c1 c2 p c q
 rule: small-stepsl.induct, erule small-stepsl-seq-1,
 rule small-stepsl-seq-2)


lemma small-stepsl-or-1:
  assumes A: (c1 OR c2, p) →∗{cfs} cf ⟹
    cf = (c1 OR c2, p) ∧
      flow cfs = [] ∨
    (c1, p) →∗{tl cfs} cf ∧
      flow cfs = flow (tl cfs) ∨
    (c2, p) →∗{tl cfs} cf ∧
      flow cfs = flow (tl cfs)
    (is - ⟹ ?P ∨ ?Q ∨ ?R)
  assumes B: (c1 OR c2, p) →∗{cfs @ [cf]} cf′
  shows
   cf′ = (c1 OR c2, p) ∧
      flow (cfs @ [cf]) = [] ∨
    (c1, p) →∗{tl (cfs @ [cf])} cf′ ∧
      flow (cfs @ [cf]) = flow (tl (cfs @ [cf])) ∨
    (c2, p) →∗{tl (cfs @ [cf])} cf′ ∧
      flow (cfs @ [cf]) = flow (tl (cfs @ [cf]))
    (is - ∨ ?T)
proof −
  {
    assume
      C: (c1 OR c2, p) →∗{cfs} cf and
      D: cf → cf′
    assume ?P ∨ ?Q ∨ ?R
    hence ?T
    proof (rule disjE, erule-tac [2] disjE)
      assume ?P
      moreover from this have (c1 OR c2, p) → cf′
        using D by simp
      ultimately show ?thesis
        using C by (auto dest: small-stepsl-cons
         simp: tl-append flow-cons split: list.split)
```

```
    next
      assume ?Q
      with C and D show ?thesis
        by (auto simp: tl-append flow-cons split: list.split)
    next
      assume ?R
      with C and D show ?thesis
        by (auto simp: tl-append flow-cons split: list.split)
    qed
  }
  with A and B show ?thesis
    by simp
qed

lemma small-stepsl-or:
 (c₁ OR c₂, p) →∗{cfs} cf ⟹
    cf = (c₁ OR c₂, p) ∧
      flow cfs = [] ∨
    (c₁, p) →∗{tl cfs} cf ∧
      flow cfs = flow (tl cfs) ∨
    (c₂, p) →∗{tl cfs} cf ∧
      flow cfs = flow (tl cfs)
by (induction (c₁ OR c₂, p) cfs cf rule: small-stepsl.induct,
 force simp: flow-def, rule small-stepsl-or-1)


lemma small-stepsl-if-1:
  assumes A: (IF b THEN c₁ ELSE c₂, s, p) →∗{cfs} cf ⟹
    cf = (IF b THEN c₁ ELSE c₂, s, p) ∧
      flow cfs = [] ∨
    bval b s ∧ (c₁, s, p) →∗{tl cfs} cf ∧
      flow cfs = ⟨bvars b⟩ # flow (tl cfs) ∨
    ¬ bval b s ∧ (c₂, s, p) →∗{tl cfs} cf ∧
      flow cfs = ⟨bvars b⟩ # flow (tl cfs)
    (is - ⟹ ?P ∨ ?Q ∨ ?R)
  assumes B: (IF b THEN c₁ ELSE c₂, s, p) →∗{cfs @ [cf]} cf′
  shows
   cf′ = (IF b THEN c₁ ELSE c₂, s, p) ∧
      flow (cfs @ [cf]) = [] ∨
    bval b s ∧ (c₁, s, p) →∗{tl (cfs @ [cf])} cf′ ∧
      flow (cfs @ [cf]) = ⟨bvars b⟩ # flow (tl (cfs @ [cf])) ∨
    ¬ bval b s ∧ (c₂, s, p) →∗{tl (cfs @ [cf])} cf′ ∧
      flow (cfs @ [cf]) = ⟨bvars b⟩ # flow (tl (cfs @ [cf]))
    (is - ∨ ?T)
proof –
  {
    assume
      C: (IF b THEN c₁ ELSE c₂, s, p) →∗{cfs} cf and
      D: cf → cf′
```

```
    assume ?P ∨ ?Q ∨ ?R
    hence ?T
    proof (rule disjE, erule-tac [2] disjE)
      assume ?P
      moreover from this have (IF b THEN c1 ELSE c2, s, p) → cf'
        using D by simp
      ultimately show ?thesis
        using C by (auto dest: small-stepsl-cons
         simp: tl-append flow-cons split: list.split)
    next
      assume ?Q
      with D show ?thesis
        by (auto simp: tl-append flow-cons split: list.split)
    next
      assume ?R
      with D show ?thesis
        by (auto simp: tl-append flow-cons split: list.split)
    qed
  }
  with A and B show ?thesis
    by simp
qed

lemma small-stepsl-if:
 (IF b THEN c1 ELSE c2, s, p) →∗{cfs} cf ⟹
    cf = (IF b THEN c1 ELSE c2, s, p) ∧
      flow cfs = [] ∨
    bval b s ∧ (c1, s, p) →∗{tl cfs} cf ∧
      flow cfs = ⟨bvars b⟩ # flow (tl cfs) ∨
    ¬ bval b s ∧ (c2, s, p) →∗{tl cfs} cf ∧
      flow cfs = ⟨bvars b⟩ # flow (tl cfs)
by (induction (IF b THEN c1 ELSE c2, s, p) cfs cf rule:
 small-stepsl.induct, force simp: flow-def, rule small-stepsl-if-1)


lemma small-stepsl-while-1:
  assumes A: (WHILE b DO c, s, p) →∗{cfs} cf ⟹
    cf = (WHILE b DO c, s, p) ∧
      flow cfs = [] ∨
    bval b s ∧ (c;; WHILE b DO c, s, p) →∗{tl cfs} cf ∧
      flow cfs = ⟨bvars b⟩ # flow (tl cfs) ∨
    ¬ bval b s ∧ cf = (SKIP, s, p) ∧
      flow cfs = [⟨bvars b⟩]
    (is - ⟹ ?P ∨ ?Q ∨ ?R)
  assumes B: (WHILE b DO c, s, p) →∗{cfs @ [cf]} cf'
  shows
   cf' = (WHILE b DO c, s, p) ∧
      flow (cfs @ [cf]) = [] ∨
    bval b s ∧ (c;; WHILE b DO c, s, p) →∗{tl (cfs @ [cf])} cf' ∧
```

```
    flow (cfs @ [cf]) = ⟨bvars b⟩ # flow (tl (cfs @ [cf])) ∨
  ¬ bval b s ∧ cf' = (SKIP, s, p) ∧
    flow (cfs @ [cf]) = [⟨bvars b⟩]
  (is - ∨ ?T)
proof –
 {
  assume
    C: (WHILE b DO c, s, p) →*{cfs} cf and
    D: cf → cf'
  assume ?P ∨ ?Q ∨ ?R
  hence ?T
  proof (rule disjE, erule-tac [2] disjE)
    assume ?P
    moreover from this have (WHILE b DO c, s, p) → cf'
      using D by simp
    ultimately show ?thesis
      using C by (auto dest: small-stepsl-cons
       simp: tl-append flow-cons split: list.split)
  next
    assume ?Q
    with D show ?thesis
      by (auto simp: tl-append flow-cons split: list.split)
  next
    assume ?R
    with D show ?thesis
      by blast
  qed
 }
 with A and B show ?thesis
  by simp
qed

lemma small-stepsl-while:
 (WHILE b DO c, s, p) →*{cfs} cf ⟹
  cf = (WHILE b DO c, s, p) ∧
    flow cfs = [] ∨
  bval b s ∧ (c;; WHILE b DO c, s, p) →*{tl cfs} cf ∧
    flow cfs = ⟨bvars b⟩ # flow (tl cfs) ∨
  ¬ bval b s ∧ cf = (SKIP, s, p) ∧
    flow cfs = [⟨bvars b⟩]
by (induction (WHILE b DO c, s, p) cfs cf rule: small-stepsl.induct,
 force simp: flow-def, rule small-stepsl-while-1)


lemma small-steps-in-flow-1:
 ⟦(c, s, f, vs, ws) → (c', s', f', vs', ws');
  vs'' = vs' @ drop (length vs') vs''⟧ ⟹
 vs'' = vs @ drop (length vs) vs''
by (induction (c, s, f, vs, ws) (c', s', f', vs', ws')
```

arbitrary: *c c′ s s′ f f′ vs vs′ ws ws′ rule*: *small-step.induct*,
auto elim: *ssubst*)

lemma *small-steps-in-flow*:
(*c*, *s*, *f*, *vs*, *ws*) →∗ (*c′*, *s′*, *f′*, *vs′*, *ws′*) ⟹
vs′ = *vs* @ *drop* (*length vs*) *vs′*
by (*induction* (*c*, *s*, *f*, *vs*, *ws*) (*c′*, *s′*, *f′*, *vs′*, *ws′*)
arbitrary: *c c′ s s′ f f′ vs vs′ ws ws′ rule*: *star.induct*,
auto intro: *small-steps-in-flow-1*)

lemma *small-steps-out-flow-1*:
⟦(*c*, *s*, *f*, *vs*, *ws*) → (*c′*, *s′*, *f′*, *vs′*, *ws′*);
ws″ = *ws′* @ *drop* (*length ws′*) *ws″*⟧ ⟹
ws″ = *ws* @ *drop* (*length ws*) *ws″*
by (*induction* (*c*, *s*, *f*, *vs*, *ws*) (*c′*, *s′*, *f′*, *vs′*, *ws′*)
arbitrary: *c c′ s s′ f f′ vs vs′ ws ws′ rule*: *small-step.induct*,
auto elim: *ssubst*)

lemma *small-steps-out-flow*:
(*c*, *s*, *f*, *vs*, *ws*) →∗ (*c′*, *s′*, *f′*, *vs′*, *ws′*) ⟹
ws′ = *ws* @ *drop* (*length ws*) *ws′*
by (*induction* (*c*, *s*, *f*, *vs*, *ws*) (*c′*, *s′*, *f′*, *vs′*, *ws′*)
arbitrary: *c c′ s s′ f f′ vs vs′ ws ws′ rule*: *star.induct*,
auto intro: *small-steps-out-flow-1*)

lemma *small-stepsl-in-flow-1*:
assumes
A: (*c*, *s*, *f*, *vs*, *ws*) →∗{*cfs*} (*c′*, *s′*, *f′*, *vs* @ *vs′*, *ws′*) **and**
B: (*c′*, *s′*, *f′*, *vs* @ *vs′*, *ws′*) → (*c″*, *s″*, *f″*, *vs″*, *ws″*)
shows *vs″* = *vs* @ *vs′* @
in-flow (*flow* [(*c′*, *s′*, *f′*, *vs* @ *vs′*, *ws′*)]) (*vs* @ *vs′*) *f*
using *small-stepsl-stream* [*OF A*] **and** *B*
by (*induction* [*c′*] *arbitrary*: *c′ c″ rule*: *flow-aux.induct*,
auto simp: *flow-def in-flow-one*)

lemma *small-stepsl-in-flow*:
(*c*, *s*, *f*, *vs*, *ws*) →∗{*cfs*} (*c′*, *s′*, *f′*, *vs′*, *ws′*) ⟹
vs′ = *vs* @ *in-flow* (*flow cfs*) *vs f*
by (*induction* (*c*, *s*, *f*, *vs*, *ws*) *cfs* (*c′*, *s′*, *f′*, *vs′*, *ws′*)
arbitrary: *c′ s′ f′ vs′ ws′ rule*: *small-stepsl.induct*, *simp add*: *flow-def*,
auto intro: *small-stepsl-in-flow-1 simp*: *flow-append in-flow-append*)

lemma *small-stepsl-run-flow-1*:
assumes
A: (*c*, *s*, *f*, *vs*, *ws*) →∗{*cfs*}
(*c′*, *run-flow* (*flow cfs*) *vs s f*, *f′*, *vs′*, *ws′*) **and**

B: (c', *run-flow* (*flow cfs*) *vs s f*, f', vs', ws') →
(c'', s'', f'', vs'', ws'')
shows s'' = *run-flow* (*flow* [(c', *run-flow* (*flow cfs*) *vs s f*, f', vs', ws')])
(*vs* @ *in-flow* (*flow cfs*) *vs f*) (*run-flow* (*flow cfs*) *vs s f*) *f*
using *small-stepsl-stream* [*OF A*] **and** *small-stepsl-in-flow* [*OF A*] **and** *B*
by (*induction* [c'] *arbitrary*: c' c'' *rule*: *flow-aux.induct*,
auto simp: *flow-def run-flow-one*)

lemma *small-stepsl-run-flow*:
(*c*, *s*, *f*, *vs*, *ws*) →∗{*cfs*} (c', s', f', vs', ws') ⟹
s' = *run-flow* (*flow cfs*) *vs s f*
by (*induction* (*c*, *s*, *f*, *vs*, *ws*) *cfs* (c', s', f', vs', ws')
arbitrary: c' s' f' vs' ws' *rule*: *small-stepsl.induct*, *simp add*: *flow-def*,
auto intro: *small-stepsl-run-flow-1 simp*: *flow-append run-flow-append*)

lemma *small-stepsl-out-flow-1*:
assumes
A: (*c*, *s*, *f*, *vs*, *ws*) →∗{*cfs*} (c', s', f', vs', *ws* @ ws') **and**
B: (c', s', f', vs', *ws* @ ws') → (c'', s'', f'', vs'', ws'')
shows ws'' = *ws* @ ws' @
out-flow (*flow* [(c', s', f', vs', *ws* @ ws')]) (*vs* @ *in-flow* (*flow cfs*) *vs f*)
(*run-flow* (*flow cfs*) *vs s f*) *f*
using *small-stepsl-run-flow* [*OF A*] **and** *B*
by (*induction* [c'] *arbitrary*: c' c'' *rule*: *flow-aux.induct*,
auto simp: *flow-def out-flow-one*)

lemma *small-stepsl-out-flow*:
(*c*, *s*, *f*, *vs*, *ws*) →∗{*cfs*} (c', s', f', vs', ws') ⟹
ws' = *ws* @ *out-flow* (*flow cfs*) *vs s f*
by (*induction* (*c*, *s*, *f*, *vs*, *ws*) *cfs* (c', s', f', vs', ws')
arbitrary: c' s' f' vs' ws' *rule*: *small-stepsl.induct*, *simp add*: *flow-def*,
auto intro: *small-stepsl-out-flow-1 simp*: *flow-append out-flow-append*)

lemma *small-steps-inputs*:
assumes
A: (*c*, *s*, *f*, vs_0, ws_0) →∗{cfs_1} (c_0, s_1, *f*, vs_1, ws_1) **and**
B: (c_1, s_1, *f*, vs_1, ws_1) →∗{cfs_2} (c_2, s_2, *f*, vs_2, ws_2) **and**
C: (*c*, s', f', vs_0', ws_0') →∗ (c_0', s_1', f', vs_1', ws_1') **and**
D: (c_1', s_1', f', vs_1', ws_1') →∗ (c_2', s_2', f', vs_2', ws_2') **and**
E: *map fst* [*p*←*drop* (*length* vs_0) vs_1. *P p*] =
map fst [*p*←*drop* (*length* vs_0') vs_1'. *P p*] **and**
F: *map fst* [*p*←*drop* (*length* vs_1) vs_2. *P p*] =
map fst [*p*←*drop* (*length* vs_1') vs_2'. *P p*]
shows *map fst* [*p*←*drop* (*length* vs_0) vs_2. *P p*] =
map fst [*p*←*drop* (*length* vs_0') vs_2'. *P p*]
proof −
have *G*: vs_1 = vs_0 @ *drop* (*length* vs_0) vs_1

using *small-stepsl-steps* [*OF A*] **by** (*rule small-steps-in-flow*)
have $vs_2 = vs_1$ @ *drop* (*length* vs_1) vs_2
using *small-stepsl-steps* [*OF B*] **by** (*rule small-steps-in-flow*)
hence *H*: $vs_2 = vs_0$ @ *drop* (*length* vs_0) vs_1 @ *drop* (*length* vs_1) vs_2
by (*subst* (*asm*) *G*, *simp*)
have *I*: $vs_1' = vs_0'$ @ *drop* (*length* vs_0') vs_1'
using *C* **by** (*rule small-steps-in-flow*)
have $vs_2' = vs_1'$ @ *drop* (*length* vs_1') vs_2'
using *D* **by** (*rule small-steps-in-flow*)
hence *J*: $vs_2' = vs_0'$ @ *drop* (*length* vs_0') vs_1' @ *drop* (*length* vs_1') vs_2'
by (*subst* (*asm*) *I*, *simp*)
from *E* **and** *F* **show** *?thesis*
by (*subst H*, *subst J*, *simp*)
qed

lemma *small-steps-outputs*:
assumes
A: $(c, s, f, vs_0, ws_0) \to*\{cfs_1\} (c_0, s_1, f, vs_1, ws_1)$ **and**
B: $(c_1, s_1, f, vs_1, ws_1) \to*\{cfs_2\} (c_2, s_2, f, vs_2, ws_2)$ **and**
C: $(c, s', f', vs_0', ws_0') \to* (c_0', s_1', f', vs_1', ws_1')$ **and**
D: $(c_1', s_1', f', vs_1', ws_1') \to* (c_2', s_2', f', vs_2', ws_2')$ **and**
E: [*p*←*drop* (*length* ws_0) ws_1. *P p*] =
[*p*←*drop* (*length* ws_0') ws_1'. *P p*] **and**
F: [*p*←*drop* (*length* ws_1) ws_2. *P p*] =
[*p*←*drop* (*length* ws_1') ws_2'. *P p*]
shows [*p*←*drop* (*length* ws_0) ws_2. *P p*] =
[*p*←*drop* (*length* ws_0') ws_2'. *P p*]
proof −
have *G*: $ws_1 = ws_0$ @ *drop* (*length* ws_0) ws_1
using *small-stepsl-steps* [*OF A*] **by** (*rule small-steps-out-flow*)
have $ws_2 = ws_1$ @ *drop* (*length* ws_1) ws_2
using *small-stepsl-steps* [*OF B*] **by** (*rule small-steps-out-flow*)
hence *H*: $ws_2 = ws_0$ @ *drop* (*length* ws_0) ws_1 @ *drop* (*length* ws_1) ws_2
by (*subst* (*asm*) *G*, *simp*)
have *I*: $ws_1' = ws_0'$ @ *drop* (*length* ws_0') ws_1'
using *C* **by** (*rule small-steps-out-flow*)
have $ws_2' = ws_1'$ @ *drop* (*length* ws_1') ws_2'
using *D* **by** (*rule small-steps-out-flow*)
hence *J*: $ws_2' = ws_0'$ @ *drop* (*length* ws_0') ws_1' @ *drop* (*length* ws_1') ws_2'
by (*subst* (*asm*) *I*, *simp*)
from *E* **and** *F* **show** *?thesis*
by (*subst H*, *subst J*, *simp*)
qed

5.2 Local context proofs

context *noninterf*
begin

lemma *no-upd-sources*:
no-upd cs X ⟹ ∀ *x* ∈ *X*. *x* ∈ *sources cs vs s f x*
by (*induction cs rule*: *rev-induct*, *auto simp*: *no-upd-append*
split: *com-flow.split*)

lemma *sources-aux-append*:
sources-aux cs vs s f x ⊆ *sources-aux* (*cs* @ *cs′*) *vs s f x*
by (*induction cs′ rule*: *rev-induct*, *simp*, *subst append-assoc* [*symmetric*],
auto simp del: *append-assoc split*: *com-flow.split*)

lemma *sources-out-append*:
sources-out cs vs s f x ⊆ *sources-out* (*cs* @ *cs′*) *vs s f x*
by (*induction cs′ rule*: *rev-induct*, *simp*, *subst append-assoc* [*symmetric*],
auto simp del: *append-assoc split*: *com-flow.split*)

lemma *sources-aux-sources*:
sources-aux cs vs s f x ⊆ *sources cs vs s f x*
by (*induction cs rule*: *rev-induct*, *auto split*: *com-flow.split*)

lemma *sources-aux-sources-out*:
sources-aux cs vs s f x ⊆ *sources-out cs vs s f x*
by (*induction cs rule*: *rev-induct*, *auto split*: *com-flow.split*)

lemma *sources-aux-observe-hd-1*:
∀ *y* ∈ *X*. *s*: *dom y* ⇝ *dom x* ⟹ *X* ⊆ *sources-aux* [⟨*X*⟩] *vs s f x*
by (*subst append-Nil* [*symmetric*], *subst sources-aux.simps*, *auto*)

lemma *sources-aux-observe-hd-2*:
⟦∀ *y* ∈ *X*. *s*: *dom y* ⇝ *dom x* ⟹ *X* ⊆ *sources-aux* (⟨*X*⟩ # *xs*) *vs s f x*;
∀ *y* ∈ *X*. *s*: *dom y* ⇝ *dom x*⟧ ⟹
X ⊆ *sources-aux* (⟨*X*⟩ # *xs* @ [*x′*]) *vs s f x*
by (*subst append-Cons* [*symmetric*], *subst sources-aux.simps*,
auto split: *com-flow.split*)

lemma *sources-aux-observe-hd*:
∀ *y* ∈ *X*. *s*: *dom y* ⇝ *dom x* ⟹ *X* ⊆ *sources-aux* (⟨*X*⟩ # *cs*) *vs s f x*
by (*induction cs rule*: *rev-induct*,
erule sources-aux-observe-hd-1, *rule sources-aux-observe-hd-2*)

lemma *sources-aux-bval*:
assumes
A: *S* ⊆ {*x*. *s* = *t* (⊆ *sources-aux* (⟨*bvars b*⟩ # *cs*) *vs s f x*)} **and**
B: *s* ∈ *Univ A* (⊆ *state* ∩ *X*) **and**
C: *bval b s* ≠ *bval b t*
shows *Univ? A X*: *bvars b* ⇝| *S*
proof −
have ¬ *s* = *t* (⊆ *bvars b*)
using *A* **and** *C* **by** (*erule-tac contrapos-nn*, *auto dest*: *bvars-bval*)

hence $\forall x \in S.\ \neg\ bvars\ b \subseteq sources\text{-}aux\ ((\langle bvars\ b\rangle\ \#\ cs)\ vs\ s\ f\ x$
using *A* **by** *blast*
hence *D*: $\{s\}$: *bvars b* $\rightsquigarrow|\ S$
by (*fastforce dest*: *sources-aux-observe-hd*)
{
fix *r y*
assume $r \in A$ **and** $y \in S$
moreover assume $s = r\ (\subseteq state \cap X)$ **and** $state \subseteq X$
hence *interf s* = *interf r*
by (*blast intro*: *interf-state*)
ultimately have *A*: *bvars b* $\rightsquigarrow|\ \{y\}$
using *D* **by** *fastforce*
}
with *B* **and** *D* **show** *?thesis*
by (*fastforce simp*: *univ-states-if-def*)
qed

lemma *ok-flow-aux-degen*:
assumes *A*: $\nexists S.\ S \neq \{\} \wedge S \subseteq \{x.\ s_1 = t_1\ (\subseteq sources\text{-}aux\ cs\ vs_1\ s_1\ f\ x)\}$
shows $\forall c_2'\ t_2\ vs_2'\ ws_2'.$
$ok\text{-}flow\text{-}aux\text{-}1\ c_1\ c_2\ c_2'\ s_1\ t_1\ t_2\ f\ f'\ vs_1\ vs_1'\ vs_2\ vs_2'\ ws_1'\ ws_2'\ cs\ \wedge$
$ok\text{-}flow\text{-}aux\text{-}2\ s_1\ s_2\ t_1\ t_2\ f\ f'\ vs_1\ vs_1'\ cs\ \wedge$
$ok\text{-}flow\text{-}aux\text{-}3\ s_1\ t_1\ f\ f'\ vs_1\ vs_1'\ ws_1\ ws_1'\ ws_2\ ws_2'\ cs$
(**is** $\forall c_2'\ t_2\ vs_2'\ ws_2'.\ ?P1\ c_2'\ t_2\ vs_2'\ ws_2' \wedge ?P2\ t_2 \wedge ?P3\ ws_2'$)
proof *clarify*
fix $c_2'\ t_2\ vs_2'\ ws_2'$
{
fix *S*
assume $S \neq \{\}$ **and** $S \subseteq \{x.\ s_1 = t_1\ (\subseteq sources\text{-}aux\ cs\ vs_1\ s_1\ f\ x)\}$
hence $?P1\ c_2'\ t_2\ vs_2'\ ws_2'$
using *A* **by** *blast*
}
moreover {
fix *S*
assume $S \subseteq \{x.\ s_1 = t_1\ (\subseteq sources\ cs\ vs_1\ s_1\ f\ x)\}$
moreover have $\forall x.\ sources\text{-}aux\ cs\ vs_1\ s_1\ f\ x \subseteq sources\ cs\ vs_1\ s_1\ f\ x$
by (*blast intro*: *subsetD* [*OF sources-aux-sources*])
ultimately have $S \subseteq \{x.\ s_1 = t_1\ (\subseteq sources\text{-}aux\ cs\ vs_1\ s_1\ f\ x)\}$
by *blast*
moreover assume $S \neq \{\}$
ultimately have $?P2\ t_2$
using *A* **by** *blast*
}
moreover {
fix *S*
assume $S \subseteq \{x.\ s_1 = t_1\ (\subseteq sources\text{-}out\ cs\ vs_1\ s_1\ f\ x)\}$
moreover have $\forall x.\ sources\text{-}aux\ cs\ vs_1\ s_1\ f\ x \subseteq sources\text{-}out\ cs\ vs_1\ s_1\ f\ x$
by (*blast intro*: *subsetD* [*OF sources-aux-sources-out*])
ultimately have $S \subseteq \{x.\ s_1 = t_1\ (\subseteq sources\text{-}aux\ cs\ vs_1\ s_1\ f\ x)\}$

by *blast*
moreover assume $S \neq \{\}$
ultimately have *?P3* $ws_2{}'$
using *A* **by** *blast*
}
ultimately show *?P1* $c_2{}'$ t_2 $vs_2{}'$ $ws_2{}'$ $\wedge$ *?P2* t_2 $\wedge$ *?P3* $ws_2{}'$
by *auto*
qed

lemma *tags-aux-append*:
tags-aux cs vs s f x $\subseteq$ *tags-aux* (*cs* @ *cs'*) *vs s f x*
by (*induction cs' rule*: *rev-induct*, *simp*, *subst append-assoc* [*symmetric*],
auto simp del: *append-assoc split*: *com-flow.split*)

lemma *tags-out-append*:
tags-out cs vs s f x $\subseteq$ *tags-out* (*cs* @ *cs'*) *vs s f x*
by (*induction cs' rule*: *rev-induct*, *simp*, *subst append-assoc* [*symmetric*],
auto simp del: *append-assoc split*: *com-flow.split*)

lemma *tags-aux-tags*:
tags-aux cs vs s f x $\subseteq$ *tags cs vs s f x*
by (*induction cs rule*: *rev-induct*, *auto split*: *com-flow.split*)

lemma *tags-aux-tags-out*:
tags-aux cs vs s f x $\subseteq$ *tags-out cs vs s f x*
by (*induction cs rule*: *rev-induct*, *auto split*: *com-flow.split*)

lemma *tags-ubound-1*:
assumes
A: (*y*, *Suc* (*length* [*c*←*cs*. *c* = *IN y*] + *n*)) $\in$ *tags-aux cs vs s f x* **and**
B: $\bigwedge$*z n*. *y* = *z* $\Longrightarrow$
(*z*, *length* [*c*←*cs*. *c* = *IN z*] + *n*) $\notin$ *tags-aux cs vs s f x*
shows *False*
proof −
have (*y*, *length* [*c*←*cs*. *c* = *IN y*] + *Suc n*) $\notin$ *tags-aux cs vs s f x*
using *B* **by** *blast*
thus *?thesis*
using *A* **by** *simp*
qed

lemma *tags-ubound-2*:
assumes
A: (*y*, *Suc* (*length* [*c*←*cs*. *c* = *IN y*] + *n*)) $\in$ *tags cs vs s f x* **and**
B: $\bigwedge$*z n*. *y* = *z* $\Longrightarrow$ *z* $\neq$ *x* $\Longrightarrow$
(*z*, *length* [*c*←*cs*. *c* = *IN z*] + *n*) $\notin$ *tags cs vs s f x* **and**
C: *y* $\neq$ *x*
shows *False*

proof −
  have (*y*, *length* [*c*←*cs*. *c* = *IN y*] + *Suc n*) ∉ *tags cs vs s f x*
    using *B* **and** *C* **by** *blast*
  thus *?thesis*
    using *A* **by** *simp*
qed

lemma *tags-ubound*:
 (*y*, *length* [*c*←*cs*. *c* = *IN y*] + *n*) ∉ *tags cs vs s f x*
and *tags-aux-ubound*:
 (*y*, *length* [*c*←*cs*. *c* = *IN y*] + *n*) ∉ *tags-aux cs vs s f x*
by (*induction cs vs s f x* **and** *cs vs s f x arbitrary*: *n* **and** *n*
 rule: *tags-induct*, *auto intro*: *tags-ubound-1 tags-ubound-2*
 split: *if-split-asm com-flow.split-asm*)

lemma *tags-out-ubound-1*:
  assumes
    A: (*y*, *Suc* (*length* [*c*←*cs*. *c* = *IN y*] + *n*)) ∈ *tags-out cs vs s f x* **and**
    B: ⋀*z n*. *y* = *z* ⟹
      (*z*, *length* [*c*←*cs*. *c* = *IN z*] + *n*) ∉ *tags-out cs vs s f x*
  shows *False*
proof −
  have (*y*, *length* [*c*←*cs*. *c* = *IN y*] + *Suc n*) ∉ *tags-out cs vs s f x*
    using *B* **by** *blast*
  thus *?thesis*
    using *A* **by** *simp*
qed

lemma *tags-out-ubound*:
 (*y*, *length* [*c*←*cs*. *c* = *IN y*] + *n*) ∉ *tags-out cs vs s f x*
by (*induction cs vs s f x arbitrary*: *n rule*: *tags-out.induct*, *auto*
 intro: *notE* [*OF tags-ubound*] *tags-out-ubound-1*
 split: *if-split-asm com-flow.split-asm*)

lemma *tags-less*:
 (*y*, *n*) ∈ *tags cs vs s f x* ⟹ *n* < *length* [*c*←*cs*. *c* = *IN y*]
  apply (*rule ccontr*)
  apply (*drule add-diff-inverse-nat*)
  apply (*drule ssubst*, *assumption*)
by (*simp add*: *tags-ubound*)

lemma *tags-aux-less*:
 (*y*, *n*) ∈ *tags-aux cs vs s f x* ⟹ *n* < *length* [*c*←*cs*. *c* = *IN y*]
  apply (*rule ccontr*)
  apply (*drule add-diff-inverse-nat*)
  apply (*drule ssubst*, *assumption*)
by (*simp add*: *tags-aux-ubound*)

```
lemma tags-out-less:
 (y, n) ∈ tags-out cs vs s f x ⟹ n < length [c←cs. c = IN y]
  apply (rule ccontr)
  apply (drule add-diff-inverse-nat)
  apply (drule ssubst, assumption)
by (simp add: tags-out-ubound)


lemma sources-observe-tl-1:
  assumes
    A: ⋀z a. c = (z ::= a :: com-flow) ⟹ z = x ⟹
      sources-aux cs vs s f x ⊆ sources-aux (⟨X⟩ # cs) vs s f x and
    B: ⋀z a b w. c = (z ::= a :: com-flow) ⟹ z = x ⟹
      sources cs vs s f w ⊆ sources (⟨X⟩ # cs) vs s f w and
    C: ⋀z a. c = (z ::= a :: com-flow) ⟹ z ≠ x ⟹
      sources cs vs s f x ⊆ sources (⟨X⟩ # cs) vs s f x and
    D: ⋀z. c = (IN z :: com-flow) ⟹ z = x ⟹
      sources-aux cs vs s f x ⊆ sources-aux (⟨X⟩ # cs) vs s f x and
    E: ⋀z. c = (IN z :: com-flow) ⟹ z ≠ x ⟹
      sources cs vs s f x ⊆ sources (⟨X⟩ # cs) vs s f x and
    F: ⋀z. c = (OUT z :: com-flow) ⟹
      sources cs vs s f x ⊆ sources (⟨X⟩ # cs) vs s f x and
    G: ⋀Y b w. c = ⟨Y⟩ ⟹
      sources cs vs s f w ⊆ sources (⟨X⟩ # cs) vs s f w
  shows sources (cs @ [c]) vs s f x ⊆ sources (⟨X⟩ # cs @ [c]) vs s f x
    (is - ⊆ ?F c)
  apply (subst sources.simps)
  apply (split com-flow.split)
  apply (rule conjI)
  subgoal
  proof –
    show ∀ z a. c = z ::= a ⟶ (if z = x
      then sources-aux cs vs s f x ∪ ⋃ {sources cs vs s f y | y.
        run-flow cs vs s f: dom y ⇝ dom x ∧ y ∈ avars a}
      else sources cs vs s f x) ⊆ ?F c
      (is ∀ - a. - ⟶ (if - then ?A ∪ ?G a else ?B) ⊆ -)
    proof (clarify, split if-split-asm)
      fix y z a
      assume H: c = z ::= a and I: z = x
      hence ?F (z ::= a) = sources-aux (⟨X⟩ # cs) vs s f x ∪
        ⋃ {sources (⟨X⟩ # cs) vs s f y | y.
          run-flow cs vs s f: dom y ⇝ dom x ∧ y ∈ avars a}
        (is - = ?A' ∪ ?G' a)
        by (simp only: append-Cons [symmetric] sources.simps,
         simp add: run-flow-observe)
      moreover assume y ∈ ?A ∪ ?G a
      moreover {
        assume y ∈ ?A
```

```
      hence y ∈ ?A′
        using A and H and I by blast
    }
    moreover {
      assume y ∈ ?G a
      hence y ∈ ?G′ a
        using B and H and I by blast
    }
    ultimately show y ∈ ?F (z ::= a)
      by blast
  next
    fix y z a
    assume c = z ::= a and z ≠ x
    moreover from this have ?F (z ::= a) = sources (⟨X⟩ # cs) vs s f x
      by (simp only: append-Cons [symmetric] sources.simps, simp)
    moreover assume y ∈ ?B
    ultimately show y ∈ ?F (z ::= a)
      using C by blast
  qed
qed
apply (rule conjI)
subgoal
proof −
  show ∀ z. c = IN z ⟶ (if z = x
    then sources-aux cs vs s f x else sources cs vs s f x) ⊆ ?F c
    (is ∀ -. - ⟶ (if - then ?A else ?B) ⊆ -)
  proof (clarify, split if-split-asm)
    fix y z
    assume c = IN z and z = x
    moreover from this have ?F (IN z) = sources-aux (⟨X⟩ # cs) vs s f x
      by (simp only: append-Cons [symmetric] sources.simps, simp)
    moreover assume y ∈ ?A
    ultimately show y ∈ ?F (IN z)
      using D by blast
  next
    fix y z
    assume c = IN z and z ≠ x
    moreover from this have ?F (IN z) = sources (⟨X⟩ # cs) vs s f x
      by (simp only: append-Cons [symmetric] sources.simps, simp)
    moreover assume y ∈ ?B
    ultimately show y ∈ ?F (IN z)
      using E by blast
  qed
qed
apply (rule conjI)
subgoal by (simp only: append-Cons [symmetric] sources.simps, simp add: F)
subgoal
proof −
  show ∀ Y. c = ⟨Y⟩ ⟶ sources cs vs s f x ∪
```

```
      ⋃ {sources cs vs s f y | y.
        run-flow cs vs s f: dom y ⇝ dom x ∧ y ∈ Y} ⊆ ?F c
      (is ∀ Y. - ⟶ ?A ∪ ?G Y ⊆ -)
    proof clarify
      fix y Y
      assume H: c = ⟨Y⟩
      hence ?F (⟨Y⟩) = sources (⟨X⟩ # cs) vs s f x ∪
        ⋃ {sources (⟨X⟩ # cs) vs s f y | y.
          run-flow cs vs s f: dom y ⇝ dom x ∧ y ∈ Y}
        (is - = ?A' ∪ ?G' Y)
        by (simp only: append-Cons [symmetric] sources.simps,
         simp add: run-flow-observe)
      moreover assume y ∈ ?A ∪ ?G Y
      moreover {
        assume y ∈ ?A
        hence y ∈ ?A'
          using G and H by blast
      }
      moreover {
        assume y ∈ ?G Y
        hence y ∈ ?G' Y
          using G and H by blast
      }
      ultimately show y ∈ ?F (⟨Y⟩)
        by blast
    qed
  qed
  done

lemma sources-observe-tl-2:
  assumes
    A: ⋀z a. c = (z ::= a :: com-flow) ⟹
      sources-aux cs vs s f x ⊆ sources-aux (⟨X⟩ # cs) vs s f x and
    B: ⋀z. c = (IN z :: com-flow) ⟹
      sources-aux cs vs s f x ⊆ sources-aux (⟨X⟩ # cs) vs s f x and
    C: ⋀z. c = (OUT z :: com-flow) ⟹
      sources-aux cs vs s f x ⊆ sources-aux (⟨X⟩ # cs) vs s f x and
    D: ⋀Y. c = ⟨Y⟩ ⟹
      sources-aux cs vs s f x ⊆ sources-aux (⟨X⟩ # cs) vs s f x and
    E: ⋀Y b w. c = ⟨Y⟩ ⟹
      sources cs vs s f w ⊆ sources (⟨X⟩ # cs) vs s f w
  shows sources-aux (cs @ [c]) vs s f x ⊆
    sources-aux (⟨X⟩ # cs @ [c]) vs s f x
    (is - ⊆ ?F c)
  apply (subst sources-aux.simps)
  apply (split com-flow.split)
  apply (rule conjI)
   defer
   apply (rule conjI)
```

```
    defer
    apply (rule conjI)
     defer
  subgoal
  proof -
    show ∀ Y. c = ⟨Y⟩ ⟶ sources-aux cs vs s f x ∪
      ⋃ {sources cs vs s f y | y.
        run-flow cs vs s f: dom y ⇝ dom x ∧ y ∈ Y} ⊆ ?F c
      (is ∀ Y. - ⟶ ?A ∪ ?G Y ⊆ -)
    proof clarify
      fix y Y
      assume F: c = ⟨Y⟩
      hence ?F (⟨Y⟩) = sources-aux (⟨X⟩ # cs) vs s f x ∪
        ⋃ {sources (⟨X⟩ # cs) vs s f y | y.
          run-flow cs vs s f: dom y ⇝ dom x ∧ y ∈ Y}
        (is - = ?A' ∪ ?G' Y)
        by (simp only: append-Cons [symmetric] sources-aux.simps,
         simp add: run-flow-observe)
      moreover assume y ∈ ?A ∪ ?G Y
      moreover {
        assume y ∈ ?A
        hence y ∈ ?A'
          using D and F by blast
      }
      moreover {
        assume y ∈ ?G Y
        hence y ∈ ?G' Y
          using E and F by blast
      }
      ultimately show y ∈ ?F (⟨Y⟩)
        by blast
    qed
  qed
by (simp only: append-Cons [symmetric] sources-aux.simps, simp add: A B C)+

lemma sources-observe-tl:
 sources cs vs s f x ⊆ sources (⟨X⟩ # cs) vs s f x
and sources-aux-observe-tl:
 sources-aux cs vs s f x ⊆ sources-aux (⟨X⟩ # cs) vs s f x
   apply (induction cs vs s f x and cs vs s f x rule: sources-induct)
  subgoal by (erule sources-observe-tl-1, assumption+)
  subgoal by (simp, subst append-Nil [symmetric], subst sources.simps, simp)
  subgoal by (erule sources-observe-tl-2, assumption+)
  by simp


lemma sources-out-observe-tl-1:
  assumes
    A: ⋀z a. c = (z ::= a :: com-flow) ⟹
```

```
    sources-out cs vs s f x ⊆ sources-out (⟨X⟩ # cs) vs s f x and
  B: ⋀z. c = (IN z :: com-flow) ⟹
    sources-out cs vs s f x ⊆ sources-out (⟨X⟩ # cs) vs s f x and
  C: ⋀z. c = (OUT z :: com-flow) ⟹
    sources-out cs vs s f x ⊆ sources-out (⟨X⟩ # cs) vs s f x and
  D: ⋀Y. c = ⟨Y⟩ ⟹
    sources-out cs vs s f x ⊆ sources-out (⟨X⟩ # cs) vs s f x
shows sources-out (cs @ [c]) vs s f x ⊆
  sources-out (⟨X⟩ # cs @ [c]) vs s f x
  (is - ⊆ ?F c)
apply (subst sources-out.simps)
apply (split com-flow.split)
apply (rule conjI)
 defer
 apply (rule conjI)
  defer
subgoal
proof
  show ∀ z. c = OUT z ⟶ sources-out cs vs s f x ∪
    (if z = x then sources cs vs s f x else {}) ⊆ ?F c
    (is ∀ -. - ⟶ ?A ∪ (if - then ?B else -) ⊆ -)
  proof (clarify, split if-split-asm)
    fix y z
    assume E: c = OUT z and F: z = x
    assume y ∈ ?A ∪ ?B
    moreover {
      assume y ∈ ?A
      hence y ∈ sources-out (⟨X⟩ # cs) vs s f x
        using C and E by blast
    }
    moreover {
      assume y ∈ ?B
      hence y ∈ sources (⟨X⟩ # cs) vs s f x
        by (rule subsetD [OF sources-observe-tl])
    }
    ultimately show y ∈ ?F (OUT z)
      using F by (simp only: append-Cons [symmetric] sources-out.simps,
       auto)
  next
    fix y z
    assume c = OUT z and y ∈ sources-out cs vs s f x ∪ {}
    hence y ∈ sources-out (⟨X⟩ # cs) vs s f x
      using C by blast
    thus y ∈ ?F (OUT z)
      by (simp only: append-Cons [symmetric] sources-out.simps, simp)
  qed
next
  show ∀ Y. c = ⟨Y⟩ ⟶ sources-out cs vs s f x ∪
    ⋃ {sources cs vs s f y | y.
```

run-flow cs vs s f: *dom y* ⇝ *dom x* ∧ *y* ∈ *Y*} ⊆ *?F c*
(**is** ∀ *Y*. - ⟶ *?A* ∪ *?G Y* ⊆ -)
proof *clarify*
fix *y Y*
assume *E*: *c* = ⟨*Y*⟩
assume *y* ∈ *?A* ∪ *?G Y*
moreover {
assume *y* ∈ *?A*
hence *y* ∈ *sources-out* (⟨*X*⟩ # *cs*) *vs s f x*
using *D* **and** *E* **by** *blast*
}
moreover {
assume *y* ∈ *?G Y*
hence *y* ∈ ⋃ {*sources* (⟨*X*⟩ # *cs*) *vs s f y* | *y*.
run-flow (⟨*X*⟩ # *cs*) *vs s f*: *dom y* ⇝ *dom x* ∧ *y* ∈ *Y*}
by (*auto intro*: *subsetD* [*OF sources-observe-tl*]
simp: *run-flow-observe*)
}
ultimately show *y* ∈ *?F* (⟨*Y*⟩)
by (*simp only*: *append-Cons* [*symmetric*] *sources-out.simps*, *auto*)
qed
qed
by (*simp only*: *append-Cons* [*symmetric*] *sources-out.simps*, *simp add*: *A B*)+

lemma *sources-out-observe-tl*:
sources-out cs vs s f x ⊆ *sources-out* (⟨*X*⟩ # *cs*) *vs s f x*
by (*induction cs vs s f x rule*: *sources-out.induct*,
erule sources-out-observe-tl-1, *simp-all*)

lemma *tags-observe-tl-1*:
⟦⋀*z a*. *c* = *z* ::= *a* ⟹ *z* = *x* ⟹
tags-aux (⟨*X*⟩ # *cs*) *vs s f x* = *tags-aux cs vs s f x*;
⋀*z a b w*. *c* = *z* ::= *a* ⟹ *z* = *x* ⟹
tags (⟨*X*⟩ # *cs*) *vs s f w* = *tags cs vs s f w*;
⋀*z a*. *c* = *z* ::= *a* ⟹ *z* ≠ *x* ⟹
tags (⟨*X*⟩ # *cs*) *vs s f x* = *tags cs vs s f x*;
⋀*z*. *c* = *IN z* ⟹ *z* = *x* ⟹
tags-aux (⟨*X*⟩ # *cs*) *vs s f x* = *tags-aux cs vs s f x*;
⋀*z*. *c* = *IN z* ⟹ *z* ≠ *x* ⟹
tags (⟨*X*⟩ # *cs*) *vs s f x* = *tags cs vs s f x*;
⋀*z*. *c* = *OUT z* ⟹
tags (⟨*X*⟩ # *cs*) *vs s f x* = *tags cs vs s f x*;
⋀*Y b w*. *c* = ⟨*Y*⟩ ⟹
tags (⟨*X*⟩ # *cs*) *vs s f w* = *tags cs vs s f w*⟧ ⟹
tags (⟨*X*⟩ # *cs* @ [*c*]) *vs s f x* = *tags* (*cs* @ [*c*]) *vs s f x*
by (*subst tags.simps*, *split com-flow.split*, *simp-all only*: *append-Cons*
[*symmetric*] *tags.simps*, *simp-all add*: *run-flow-observe*)

lemma *tags-observe-tl-2*:
$[\![\bigwedge z\ a.\ c = z ::= a \Longrightarrow$
$\quad \textit{tags-aux}\ (\langle X\rangle\ \#\ cs)\ vs\ s\ f\ x = \textit{tags-aux}\ cs\ vs\ s\ f\ x;$
$\bigwedge z.\ c = \textit{IN}\ z \Longrightarrow$
$\quad \textit{tags-aux}\ (\langle X\rangle\ \#\ cs)\ vs\ s\ f\ x = \textit{tags-aux}\ cs\ vs\ s\ f\ x;$
$\bigwedge z.\ c = \textit{OUT}\ z \Longrightarrow$
$\quad \textit{tags-aux}\ (\langle X\rangle\ \#\ cs)\ vs\ s\ f\ x = \textit{tags-aux}\ cs\ vs\ s\ f\ x;$
$\bigwedge Y.\ c = \langle Y\rangle \Longrightarrow$
$\quad \textit{tags-aux}\ (\langle X\rangle\ \#\ cs)\ vs\ s\ f\ x = \textit{tags-aux}\ cs\ vs\ s\ f\ x;$
$\bigwedge Y\ b\ w.\ c = \langle Y\rangle \Longrightarrow$
$\quad \textit{tags}\ (\langle X\rangle\ \#\ cs)\ vs\ s\ f\ w = \textit{tags}\ cs\ vs\ s\ f\ w]\!] \Longrightarrow$
$\textit{tags-aux}\ (\langle X\rangle\ \#\ cs\ @\ [c])\ vs\ s\ f\ x = \textit{tags-aux}\ (cs\ @\ [c])\ vs\ s\ f\ x$
by (*subst tags-aux.simps*, *split com-flow.split*, *simp-all only*: *append-Cons*
[*symmetric*] *tags-aux.simps*, *simp-all add*: *run-flow-observe*)

lemma *tags-observe-tl*:
$\textit{tags}\ (\langle X\rangle\ \#\ cs)\ vs\ s\ f\ x = \textit{tags}\ cs\ vs\ s\ f\ x$
and *tags-aux-observe-tl*:
$\textit{tags-aux}\ (\langle X\rangle\ \#\ cs)\ vs\ s\ f\ x = \textit{tags-aux}\ cs\ vs\ s\ f\ x$
apply (*induction cs vs s f x* **and** *cs vs s f x rule*: *tags-induct*)
subgoal by (*erule tags-observe-tl-1*, *assumption*+)
subgoal by (*subst append-Nil* [*symmetric*], *subst tags.simps tags-aux.simps*, *simp*)
subgoal by (*erule tags-observe-tl-2*, *assumption*+)
subgoal by (*subst append-Nil* [*symmetric*], *subst tags.simps tags-aux.simps*, *simp*)
done

lemma *tags-out-observe-tl-1*:
$[\![\bigwedge z\ a.\ c = z ::= a \Longrightarrow$
$\quad \textit{tags-out}\ (\langle X\rangle\ \#\ cs)\ vs\ s\ f\ x = \textit{tags-out}\ cs\ vs\ s\ f\ x;$
$\bigwedge z.\ c = \textit{IN}\ z \Longrightarrow$
$\quad \textit{tags-out}\ (\langle X\rangle\ \#\ cs)\ vs\ s\ f\ x = \textit{tags-out}\ cs\ vs\ s\ f\ x;$
$\bigwedge z.\ c = \textit{OUT}\ z \Longrightarrow$
$\quad \textit{tags-out}\ (\langle X\rangle\ \#\ cs)\ vs\ s\ f\ x = \textit{tags-out}\ cs\ vs\ s\ f\ x;$
$\bigwedge Y.\ c = \langle Y\rangle \Longrightarrow$
$\quad \textit{tags-out}\ (\langle X\rangle\ \#\ cs)\ vs\ s\ f\ x = \textit{tags-out}\ cs\ vs\ s\ f\ x]\!] \Longrightarrow$
$\textit{tags-out}\ (\langle X\rangle\ \#\ cs\ @\ [c])\ vs\ s\ f\ x = \textit{tags-out}\ (cs\ @\ [c])\ vs\ s\ f\ x$
by (*subst tags-out.simps*, *split com-flow.split*, *simp-all only*: *append-Cons*
[*symmetric*] *tags-out.simps*, *simp-all add*: *run-flow-observe tags-observe-tl*)

lemma *tags-out-observe-tl*:
$\textit{tags-out}\ (\langle X\rangle\ \#\ cs)\ vs\ s\ f\ x = \textit{tags-out}\ cs\ vs\ s\ f\ x$
apply (*induction cs vs s f x rule*: *tags-out.induct*)
apply (*erule tags-out-observe-tl-1*, *assumption*+)
by (*subst append-Nil* [*symmetric*], *subst tags-out.simps*, *simp*)

lemma *tags-sources-1*:
assumes
A: $\bigwedge z\ a.\ c = (z ::= a :: \textit{com-flow}) \Longrightarrow z = x \Longrightarrow$

```
   (y, n) ∈ tags-aux cs vs s f x ⟹
    let m = Suc (Max {k. k ≤ length cs ∧
     length [c←take k cs. c = IN y] ≤ n})
    in y ∈ sources-aux (drop m cs) (vs @ in-flow (take m cs) vs f)
     (run-flow (take m cs) vs s f) f x
   (is ⋀- -. - ⟹ - ⟹ - ⟹ let m = Suc (Max (?F cs)) in
    - ∈ sources-aux - (?G m cs) (?H m cs) - -)
 assumes
  B: ⋀z a b w. c = (z ::= a :: com-flow) ⟹ z = x ⟹
   (y, n) ∈ tags cs vs s f w ⟹ let m = Suc (Max (?F cs)) in
    y ∈ sources (drop m cs) (?G m cs) (?H m cs) f w and
  C: ⋀z a. c = (z ::= a :: com-flow) ⟹ z ≠ x ⟹
   (y, n) ∈ tags cs vs s f x ⟹ let m = Suc (Max (?F cs)) in
    y ∈ sources (drop m cs) (?G m cs) (?H m cs) f x and
  D: ⋀z. c = (IN z :: com-flow) ⟹ z = x ⟹
   (y, n) ∈ tags-aux cs vs s f x ⟹ let m = Suc (Max (?F cs)) in
    y ∈ sources-aux (drop m cs) (?G m cs) (?H m cs) f x and
  E: ⋀z. c = (IN z :: com-flow) ⟹ z ≠ x ⟹
   (y, n) ∈ tags cs vs s f x ⟹ let m = Suc (Max (?F cs)) in
    y ∈ sources (drop m cs) (?G m cs) (?H m cs) f x and
  F: ⋀z. c = (OUT z :: com-flow) ⟹
   (y, n) ∈ tags cs vs s f x ⟹ let m = Suc (Max (?F cs)) in
    y ∈ sources (drop m cs) (?G m cs) (?H m cs) f x and
  G: ⋀X b w. c = ⟨X⟩ ⟹
   (y, n) ∈ tags cs vs s f w ⟹ let m = Suc (Max (?F cs)) in
    y ∈ sources (drop m cs) (?G m cs) (?H m cs) f w and
  H: (y, n) ∈ tags (cs @ [c]) vs s f x
 shows let m = Suc (Max (?F (cs @ [c]))) in
  y ∈ sources (drop m (cs @ [c])) (?G m (cs @ [c])) (?H m (cs @ [c])) f x
proof −
 have I: n < length [c←cs @ [c]. c = IN y]
  using H by (rule tags-less)
 hence ?F (cs @ [c]) = ?F cs
  using le-Suc-eq by auto
 moreover have c ≠ IN y ∨ n < length [c←cs. c = IN y] ⟹
  Suc (Max (?F cs)) ≤ length cs
  (is - ⟹ ?m ≤ -)
  using I by (subst Suc-le-eq, subst Max-less-iff,
   auto elim: le-neq-implies-less)
 ultimately have J: c ≠ IN y ∨ n < length [c←cs. c = IN y] ⟹
  take (Suc (Max (?F (cs @ [c])))) (cs @ [c]) = take ?m cs ∧
  drop (Suc (Max (?F (cs @ [c])))) (cs @ [c]) = drop ?m cs @ [c]
  by simp
 from H show ?thesis
 proof (subst (asm) tags.simps, split com-flow.split-asm)
  fix z a
  assume K: c = z ::= a
  show (y, n) ∈ (if z = x
   then tags-aux cs vs s f x ∪ ⋃ {tags cs vs s f y | y.
```

```
      run-flow cs vs s f: dom y ⇝ dom x ∧ y ∈ avars a}
    else tags cs vs s f x) ⟹ ?thesis
    (is - ∈ (if - then ?A ∪ ?B else ?C) ⟹ -)
  proof (split if-split-asm)
    assume L: z = x and (y, n) ∈ ?A ∪ ?B
    moreover {
      assume (y, n) ∈ ?A
      hence y ∈ sources-aux (drop ?m cs) (?G ?m cs) (?H ?m cs) f x
        using A and K and L by simp
    }
    moreover {
      assume (y, n) ∈ ?B
      hence y ∈ ⋃ {sources (drop ?m cs) (?G ?m cs) (?H ?m cs) f y | y.
        run-flow (drop ?m cs) (?G ?m cs) (?H ?m cs) f:
          dom y ⇝ dom x ∧ y ∈ avars a}
        using B and K and L by (auto simp: run-flow-append [symmetric])
    }
    ultimately show ?thesis
      using J and K by auto
  next
    assume z ≠ x and (y, n) ∈ ?C
    moreover from this have
     y ∈ sources (drop ?m cs) (?G ?m cs) (?H ?m cs) f x
      using C and K by simp
    ultimately show ?thesis
      using J and K by simp
  qed
next
  fix z
  assume K: c = IN z
  show (y, n) ∈ (if z = x
    then insert (x, length [c←cs. c = IN x]) (tags-aux cs vs s f x)
    else tags cs vs s f x) ⟹ ?thesis
    (is - ∈ (if - then insert - ?A else ?B) ⟹ -)
  proof (split if-split-asm, erule insertE)
    assume (y, n) = (x, length [c←cs. c = IN x]) and z = x
    moreover from this have Max (?F (cs @ [c])) = length cs
      using K by (subst Max-eq-iff, auto elim: le-SucE)
    ultimately show ?thesis
      by simp
  next
    assume L: (y, n) ∈ tags-aux cs vs s f x and z = x
    moreover from this have
     y ∈ sources-aux (drop ?m cs) (?G ?m cs) (?H ?m cs) f x
      using D and K by simp
    moreover have n < length [c←cs. c = IN y]
      using L by (rule tags-aux-less)
    ultimately show ?thesis
      using J and K by simp
```

```
    next
      assume L: (y, n) ∈ tags cs vs s f x and z ≠ x
      moreover from this have
       y ∈ sources (drop ?m cs) (?G ?m cs) (?H ?m cs) f x
        using E and K by simp
      moreover have n < length [c←cs. c = IN y]
        using L by (rule tags-less)
      ultimately show ?thesis
        using J and K by simp
    qed
  next
    fix z
    assume c = OUT z and (y, n) ∈ tags cs vs s f x
    moreover from this have
     y ∈ sources (drop ?m cs) (?G ?m cs) (?H ?m cs) f x
      using F by simp
    ultimately show ?thesis
      using J by simp
  next
    fix X
    assume K: c = ⟨X⟩
    assume (y, n) ∈ tags cs vs s f x ∪ ⋃ {tags cs vs s f y | y.
      run-flow cs vs s f: dom y ⇝ dom x ∧ y ∈ X}
      (is - ∈ ?A ∪ ?B)
    moreover {
      assume (y, n) ∈ ?A
      hence y ∈ sources (drop ?m cs) (?G ?m cs) (?H ?m cs) f x
        using G and K by simp
    }
    moreover {
      assume (y, n) ∈ ?B
      hence y ∈ ⋃ {sources (drop ?m cs) (?G ?m cs) (?H ?m cs) f y | y.
        run-flow (drop ?m cs) (?G ?m cs) (?H ?m cs) f:
          dom y ⇝ dom x ∧ y ∈ X}
        using G and K by (auto simp: run-flow-append [symmetric])
    }
    ultimately show ?thesis
      using J and K by auto
  qed
qed

lemma tags-sources-2:
  assumes
    A: ⋀z a. c = (z ::= a :: com-flow) ⟹
      (y, n) ∈ tags-aux cs vs s f x ⟹
        let m = Suc (Max {k. k ≤ length cs ∧
          length [c←take k cs. c = IN y] ≤ n})
        in y ∈ sources-aux (drop m cs) (vs @ in-flow (take m cs) vs f)
          (run-flow (take m cs) vs s f) f x
```

```
      (is ⋀- -. - ⟹ - ⟹ let m = Suc (Max (?F cs)) in
        - ∈ sources-aux - (?G m cs) (?H m cs) - -)
  assumes
    B: ⋀z. c = (IN z :: com-flow) ⟹
      (y, n) ∈ tags-aux cs vs s f x ⟹ let m = Suc (Max (?F cs)) in
        y ∈ sources-aux (drop m cs) (?G m cs) (?H m cs) f x and
    C: ⋀z. c = (OUT z :: com-flow) ⟹
      (y, n) ∈ tags-aux cs vs s f x ⟹ let m = Suc (Max (?F cs)) in
        y ∈ sources-aux (drop m cs) (?G m cs) (?H m cs) f x and
    D: ⋀X. c = ⟨X⟩ ⟹
      (y, n) ∈ tags-aux cs vs s f x ⟹ let m = Suc (Max (?F cs)) in
        y ∈ sources-aux (drop m cs) (?G m cs) (?H m cs) f x and
    E: ⋀X b w. c = ⟨X⟩ ⟹
      (y, n) ∈ tags cs vs s f w ⟹ let m = Suc (Max (?F cs)) in
        y ∈ sources (drop m cs) (?G m cs) (?H m cs) f w and
    F: (y, n) ∈ tags-aux (cs @ [c]) vs s f x
  shows let m = Suc (Max (?F (cs @ [c]))) in
    y ∈ sources-aux (drop m (cs @ [c])) (?G m (cs @ [c])) (?H m (cs @ [c])) f x
proof –
  have G: n < length [c←cs @ [c]. c = IN y]
    using F by (rule tags-aux-less)
  hence ?F (cs @ [c]) = ?F cs
    using le-Suc-eq by auto
  moreover have c ≠ IN y ∨ n < length [c←cs. c = IN y] ⟹
    Suc (Max (?F cs)) ≤ length cs
    (is - ⟹ ?m ≤ -)
    using G by (subst Suc-le-eq, subst Max-less-iff,
     auto elim: le-neq-implies-less)
  ultimately have H: c ≠ IN y ∨ n < length [c←cs. c = IN y] ⟹
    take (Suc (Max (?F (cs @ [c])))) (cs @ [c]) = take ?m cs ∧
    drop (Suc (Max (?F (cs @ [c])))) (cs @ [c]) = drop ?m cs @ [c]
    by simp
  from F show ?thesis
  proof (subst (asm) tags-aux.simps, split com-flow.split-asm)
    fix z a
    assume c = z ::= a and (y, n) ∈ tags-aux cs vs s f x
    moreover from this have
     y ∈ sources-aux (drop ?m cs) (?G ?m cs) (?H ?m cs) f x
      using A by simp
    ultimately show ?thesis
      using H by simp
  next
    fix z
    assume c = IN z and I: (y, n) ∈ tags-aux cs vs s f x
    moreover from this have
     y ∈ sources-aux (drop ?m cs) (?G ?m cs) (?H ?m cs) f x
      using B by simp
    moreover have n < length [c←cs. c = IN y]
      using I by (rule tags-aux-less)
```

```
    ultimately show ?thesis
      using H by simp
  next
    fix z
    assume c = OUT z and (y, n) ∈ tags-aux cs vs s f x
    moreover from this have
     y ∈ sources-aux (drop ?m cs) (?G ?m cs) (?H ?m cs) f x
      using C by simp
    ultimately show ?thesis
      using H by simp
  next
    fix X
    assume I: c = ⟨X⟩
    assume (y, n) ∈ tags-aux cs vs s f x ∪ ⋃ {tags cs vs s f y | y.
      run-flow cs vs s f: dom y ⇝ dom x ∧ y ∈ X}
      (is - ∈ ?A ∪ ?B)
    moreover {
      assume (y, n) ∈ ?A
      hence y ∈ sources-aux (drop ?m cs) (?G ?m cs) (?H ?m cs) f x
        using D and I by simp
    }
    moreover {
      assume (y, n) ∈ ?B
      hence y ∈ ⋃ {sources (drop ?m cs) (?G ?m cs) (?H ?m cs) f y | y.
        run-flow (drop ?m cs) (?G ?m cs) (?H ?m cs) f:
          dom y ⇝ dom x ∧ y ∈ X}
        using E and I by (auto simp: run-flow-append [symmetric])
    }
    ultimately show ?thesis
      using H and I by auto
  qed
qed

lemma tags-sources:
 (y, n) ∈ tags cs vs s f x ⟹
    let m = Suc (Max {k. k ≤ length cs ∧
      length [c←take k cs. c = IN y] ≤ n})
    in y ∈ sources (drop m cs) (vs @ in-flow (take m cs) vs f)
      (run-flow (take m cs) vs s f) f x
and tags-aux-sources-aux:
 (y, n) ∈ tags-aux cs vs s f x ⟹
    let m = Suc (Max {k. k ≤ length cs ∧
      length [c←take k cs. c = IN y] ≤ n})
    in y ∈ sources-aux (drop m cs) (vs @ in-flow (take m cs) vs f)
      (run-flow (take m cs) vs s f) f x
by (induction cs vs s f x and cs vs s f x rule: tags-induct,
 erule-tac [3] tags-sources-2, erule tags-sources-1, simp-all)
```

```
lemma tags-out-sources-out-1:
  assumes
    A: ⋀z a. c = (z ::= a :: com-flow) ⟹
      (y, n) ∈ tags-out cs vs s f x ⟹
        let m = Suc (Max {k. k ≤ length cs ∧
          length [c←take k cs. c = IN y] ≤ n})
        in y ∈ sources-out (drop m cs) (vs @ in-flow (take m cs) vs f)
          (run-flow (take m cs) vs s f) f x
      (is ⋀- -. - ⟹ - ⟹ let m = Suc (Max (?F cs)) in
        - ∈ sources-out - (?G m cs) (?H m cs) - -)
  assumes
    B: ⋀z. c = (IN z :: com-flow) ⟹
      (y, n) ∈ tags-out cs vs s f x ⟹ let m = Suc (Max (?F cs)) in
        y ∈ sources-out (drop m cs) (?G m cs) (?H m cs) f x and
    C: ⋀z. c = (OUT z :: com-flow) ⟹
      (y, n) ∈ tags-out cs vs s f x ⟹ let m = Suc (Max (?F cs)) in
        y ∈ sources-out (drop m cs) (?G m cs) (?H m cs) f x and
    D: ⋀X. c = ⟨X⟩ ⟹
      (y, n) ∈ tags-out cs vs s f x ⟹ let m = Suc (Max (?F cs)) in
        y ∈ sources-out (drop m cs) (?G m cs) (?H m cs) f x and
    E: (y, n) ∈ tags-out (cs @ [c]) vs s f x
  shows let m = Suc (Max (?F (cs @ [c]))) in
    y ∈ sources-out (drop m (cs @ [c])) (?G m (cs @ [c])) (?H m (cs @ [c])) f x
proof −
  have F: n < length [c←cs @ [c]. c = IN y]
    using E by (rule tags-out-less)
  hence ?F (cs @ [c]) = ?F cs
    using le-Suc-eq by auto
  moreover have c ≠ IN y ∨ n < length [c←cs. c = IN y] ⟹
    Suc (Max (?F cs)) ≤ length cs
    (is - ⟹ ?m ≤ -)
    using F by (subst Suc-le-eq, subst Max-less-iff,
     auto elim: le-neq-implies-less)
  ultimately have G: c ≠ IN y ∨ n < length [c←cs. c = IN y] ⟹
    take (Suc (Max (?F (cs @ [c])))) (cs @ [c]) = take ?m cs ∧
    drop (Suc (Max (?F (cs @ [c])))) (cs @ [c]) = drop ?m cs @ [c]
    by simp
  from E show ?thesis
  proof (subst (asm) tags-out.simps, split com-flow.split-asm)
    fix z a
    assume c = z ::= a and (y, n) ∈ tags-out cs vs s f x
    moreover from this have
     y ∈ sources-out (drop ?m cs) (?G ?m cs) (?H ?m cs) f x
      using A by simp
    ultimately show ?thesis
      using G by simp
  next
    fix z
    assume c = IN z and H: (y, n) ∈ tags-out cs vs s f x
```

```
  moreover from this have
   y ∈ sources-out (drop ?m cs) (?G ?m cs) (?H ?m cs) f x
    using B by simp
  moreover have n < length [c←cs. c = IN y]
    using H by (rule tags-out-less)
  ultimately show ?thesis
    using G by simp
next
  fix z
  assume H: c = OUT z
  show (y, n) ∈ tags-out cs vs s f x ∪
    (if z = x then tags cs vs s f x else {}) ⟹ ?thesis
    (is - ∈ ?A ∪ (if - then ?B else -) ⟹ -)
  proof (split if-split-asm)
    assume z = x and (y, n) ∈ ?A ∪ ?B
    moreover {
      assume (y, n) ∈ ?A
      hence y ∈ sources-out (drop ?m cs) (?G ?m cs) (?H ?m cs) f x
        using C and H by simp
    }
    moreover {
      assume (y, n) ∈ ?B
      hence y ∈ sources (drop ?m cs) (?G ?m cs) (?H ?m cs) f x
        by (auto dest: tags-sources)
    }
    ultimately show ?thesis
      using G and H by auto
  next
    assume (y, n) ∈ ?A ∪ {}
    moreover from this have
     y ∈ sources-out (drop ?m cs) (?G ?m cs) (?H ?m cs) f x
      using C and H by simp
    ultimately show ?thesis
      using G and H by simp
  qed
next
  fix X
  assume H: c = ⟨X⟩
  assume (y, n) ∈ tags-out cs vs s f x ∪ ⋃ {tags cs vs s f y | y.
    run-flow cs vs s f: dom y ⇝ dom x ∧ y ∈ X}
    (is - ∈ ?A ∪ ?B)
  moreover {
    assume (y, n) ∈ ?A
    hence y ∈ sources-out (drop ?m cs) (?G ?m cs) (?H ?m cs) f x
      using D and H by simp
  }
  moreover {
    assume (y, n) ∈ ?B
    hence y ∈ ⋃ {sources (drop ?m cs) (?G ?m cs) (?H ?m cs) f y | y.
```

```
        run-flow (drop ?m cs) (?G ?m cs) (?H ?m cs) f:
          dom y ⇝ dom x ∧ y ∈ X}
        by (fastforce dest: tags-sources simp: run-flow-append [symmetric])
    }
    ultimately show ?thesis
      using G and H by auto
  qed
qed

lemma tags-out-sources-out:
 (y, n) ∈ tags-out cs vs s f x ⟹
    let m = Suc (Max {k. k ≤ length cs ∧
      length [c←take k cs. c = IN y] ≤ n})
    in y ∈ sources-out (drop m cs) (vs @ in-flow (take m cs) vs f)
      (run-flow (take m cs) vs s f) f x
by (induction cs vs s f x rule: tags-out.induct,
 erule tags-out-sources-out-1, simp-all)


lemma sources-member-1:
  assumes
    A: ⋀z a. c = (z ::= a :: com-flow) ⟹ z = x ⟹
      y ∈ sources-aux cs' (vs @ in-flow cs vs f) (run-flow cs vs s f) f x ⟹
        sources cs vs s f y ⊆ sources-aux (cs @ cs') vs s f x
      (is ⋀- -. - ⟹ - ⟹ - ∈ sources-aux - ?vs' ?s' - - ⟹
        - ⊆ sources-aux ?cs - - - -)
  assumes
    B: ⋀z a b w. c = (z ::= a :: com-flow) ⟹ z = x ⟹
      y ∈ sources cs' ?vs' ?s' f w ⟹
        sources cs vs s f y ⊆ sources ?cs vs s f w and
    C: ⋀z a. c = (z ::= a :: com-flow) ⟹ z ≠ x ⟹
      y ∈ sources cs' ?vs' ?s' f x ⟹
        sources cs vs s f y ⊆ sources ?cs vs s f x and
    D: ⋀z. c = (IN z :: com-flow) ⟹ z = x ⟹
      y ∈ sources-aux cs' ?vs' ?s' f x ⟹
        sources cs vs s f y ⊆ sources-aux ?cs vs s f x and
    E: ⋀z. c = (IN z :: com-flow) ⟹ z ≠ x ⟹
      y ∈ sources cs' ?vs' ?s' f x ⟹
        sources cs vs s f y ⊆ sources ?cs vs s f x and
    F: ⋀z. c = (OUT z :: com-flow) ⟹
      y ∈ sources cs' ?vs' ?s' f x ⟹
        sources cs vs s f y ⊆ sources ?cs vs s f x and
    G: ⋀X b w. c = ⟨X⟩ ⟹
      y ∈ sources cs' ?vs' ?s' f w ⟹
        sources cs vs s f y ⊆ sources ?cs vs s f w
  shows y ∈ sources (cs' @ [c]) ?vs' ?s' f x ⟹
    sources cs vs s f y ⊆ sources (cs @ cs' @ [c]) vs s f x
proof (subst (asm) sources.simps, split com-flow.split-asm)
  fix z a
```

```
  assume H: c = z ::= a
  show y ∈ (if z = x
    then sources-aux cs' ?vs' ?s' f x ∪ ⋃ {sources cs' ?vs' ?s' f w | w.
      run-flow cs' ?vs' ?s' f: dom w ⇝ dom x ∧ w ∈ avars a}
    else sources cs' ?vs' ?s' f x) ⟹ ?thesis
    (is - ∈ (if - then ?A ∪ ?B else ?C) ⟹ -)
  proof (split if-split-asm)
    assume I: z = x and y ∈ ?A ∪ ?B
    moreover {
      assume y ∈ ?A
      hence sources cs vs s f y ⊆ sources-aux ?cs vs s f x
        using A and H and I by simp
    }
    moreover {
      assume y ∈ ?B
      hence sources cs vs s f y ⊆ ⋃ {sources ?cs vs s f w | w.
        run-flow ?cs vs s f: dom w ⇝ dom x ∧ w ∈ avars a}
        using B and H and I by (fastforce simp: run-flow-append)
    }
    ultimately show ?thesis
      using H by (simp only: append-assoc [symmetric] sources.simps, auto)
  next
    assume z ≠ x and y ∈ ?C
    moreover from this have sources cs vs s f y ⊆ sources ?cs vs s f x
      using C and H by simp
    ultimately show ?thesis
      using H by (simp only: append-assoc [symmetric] sources.simps, auto)
  qed
next
  fix z
  assume H: c = IN z
  show y ∈ (if z = x
    then sources-aux cs' ?vs' ?s' f x
    else sources cs' ?vs' ?s' f x) ⟹ ?thesis
    (is - ∈ (if - then ?A else ?B) ⟹ -)
  proof (split if-split-asm)
    assume z = x and y ∈ ?A
    moreover from this have sources cs vs s f y ⊆ sources-aux ?cs vs s f x
      using D and H by simp
    ultimately show ?thesis
      using H by (simp only: append-assoc [symmetric] sources.simps, auto)
  next
    assume z ≠ x and y ∈ ?B
    moreover from this have sources cs vs s f y ⊆ sources ?cs vs s f x
      using E and H by simp
    ultimately show ?thesis
      using H by (simp only: append-assoc [symmetric] sources.simps, auto)
  qed
next
```

fix *z*
assume *c* = *OUT z* **and** *y* ∈ *sources cs′ ?vs′ ?s′ f x*
moreover from *this* **have** *sources cs vs s f y* ⊆ *sources ?cs vs s f x*
using *F* **by** *simp*
ultimately show *?thesis*
by (*simp only*: *append-assoc* [*symmetric*] *sources.simps*, *auto*)
next
fix *X*
assume *H*: *c* = ⟨*X*⟩
assume *y* ∈ *sources cs′ ?vs′ ?s′ f x* ∪ ⋃ {*sources cs′ ?vs′ ?s′ f w* | *w*.
run-flow cs′ ?vs′ ?s′ f: *dom w* ⇝ *dom x* ∧ *w* ∈ *X*}
(**is** - ∈ *?A* ∪ *?B*)
moreover {
assume *y* ∈ *?A*
hence *sources cs vs s f y* ⊆ *sources ?cs vs s f x*
using *G* **and** *H* **by** *simp*
}
moreover {
assume *y* ∈ *?B*
hence *sources cs vs s f y* ⊆ ⋃ {*sources ?cs vs s f w* | *w*.
run-flow ?cs vs s f: *dom w* ⇝ *dom x* ∧ *w* ∈ *X*}
using *G* **and** *H* **by** (*auto simp*: *run-flow-append*)
}
ultimately show *?thesis*
using *H* **by** (*simp only*: *append-assoc* [*symmetric*] *sources.simps*, *auto*)
qed

lemma *sources-member-2*:
assumes
A: ⋀*z a*. *c* = (*z* ::= *a* :: *com-flow*) ⟹
y ∈ *sources-aux cs′* (*vs* @ *in-flow cs vs f*) (*run-flow cs vs s f*) *f x* ⟹
sources cs vs s f y ⊆ *sources-aux* (*cs* @ *cs′*) *vs s f x*
(**is** ⋀- -. - ⟹ - ∈ *sources-aux* - *?vs′ ?s′* - - ⟹
- ⊆ *sources-aux ?cs* - - - -)
assumes
B: ⋀*z*. *c* = (*IN z* :: *com-flow*) ⟹
y ∈ *sources-aux cs′ ?vs′ ?s′ f x* ⟹
sources cs vs s f y ⊆ *sources-aux ?cs vs s f x* **and**
C: ⋀*z*. *c* = (*OUT z* :: *com-flow*) ⟹
y ∈ *sources-aux cs′ ?vs′ ?s′ f x* ⟹
sources cs vs s f y ⊆ *sources-aux ?cs vs s f x* **and**
D: ⋀*X*. *c* = ⟨*X*⟩ ⟹
y ∈ *sources-aux cs′ ?vs′ ?s′ f x* ⟹
sources cs vs s f y ⊆ *sources-aux ?cs vs s f x* **and**
E: ⋀*X b w*. *c* = ⟨*X*⟩ ⟹
y ∈ *sources cs′ ?vs′ ?s′ f w* ⟹
sources cs vs s f y ⊆ *sources ?cs vs s f w*
shows *y* ∈ *sources-aux* (*cs′* @ [*c*]) *?vs′ ?s′ f x* ⟹
sources cs vs s f y ⊆ *sources-aux* (*cs* @ *cs′* @ [*c*]) *vs s f x*

```
proof (subst (asm) sources-aux.simps, split com-flow.split-asm)
  fix z a
  assume c = z ::= a and y ∈ sources-aux cs' ?vs' ?s' f x
  moreover from this have sources cs vs s f y ⊆ sources-aux ?cs vs s f x
    using A by simp
  ultimately show ?thesis
    by (simp only: append-assoc [symmetric] sources-aux.simps, auto)
next
  fix z
  assume c = IN z and y ∈ sources-aux cs' ?vs' ?s' f x
  moreover from this have sources cs vs s f y ⊆ sources-aux ?cs vs s f x
    using B by simp
  ultimately show ?thesis
    by (simp only: append-assoc [symmetric] sources-aux.simps, auto)
next
  fix z
  assume c = OUT z and y ∈ sources-aux cs' ?vs' ?s' f x
  moreover from this have sources cs vs s f y ⊆ sources-aux ?cs vs s f x
    using C by simp
  ultimately show ?thesis
    by (simp only: append-assoc [symmetric] sources-aux.simps, auto)
next
  fix X
  assume F: c = ⟨X⟩
  assume y ∈ sources-aux cs' ?vs' ?s' f x ∪ ⋃ {sources cs' ?vs' ?s' f w | w.
    run-flow cs' ?vs' ?s' f: dom w ⇝ dom x ∧ w ∈ X}
    (is - ∈ ?A ∪ ?B)
  moreover {
    assume y ∈ ?A
    hence sources cs vs s f y ⊆ sources-aux ?cs vs s f x
      using D and F by simp
  }
  moreover {
    assume y ∈ ?B
    hence sources cs vs s f y ⊆ ⋃ {sources ?cs vs s f w | w.
      run-flow ?cs vs s f: dom w ⇝ dom x ∧ w ∈ X}
      using E and F by (auto simp: run-flow-append)
  }
  ultimately show ?thesis
    using F by (simp only: append-assoc [symmetric] sources-aux.simps, auto)
qed

lemma sources-member:
 y ∈ sources cs' (vs @ in-flow cs vs f) (run-flow cs vs s f) f x ⟹
   sources cs vs s f y ⊆ sources (cs @ cs') vs s f x
and sources-aux-member:
 y ∈ sources-aux cs' (vs @ in-flow cs vs f) (run-flow cs vs s f) f x ⟹
   sources cs vs s f y ⊆ sources-aux (cs @ cs') vs s f x
by (induction cs' vs s f x and cs' vs s f x rule: sources-induct,
```

```
  erule-tac [3] sources-member-2, erule sources-member-1, simp-all)


lemma sources-out-member-1:
  assumes
    A: ⋀z a. c = (z ::= a :: com-flow) ⟹
      y ∈ sources-out cs′ (vs @ in-flow cs vs f) (run-flow cs vs s f) f x ⟹
        sources cs vs s f y ⊆ sources-out (cs @ cs′) vs s f x
      (is ⋀- -. - ⟹ - ∈ sources-out - ?vs′ ?s′ - - ⟹
        - ⊆ sources-out ?cs - - - -)
  assumes
    B: ⋀z. c = (IN z :: com-flow) ⟹
      y ∈ sources-out cs′ ?vs′ ?s′ f x ⟹
        sources cs vs s f y ⊆ sources-out ?cs vs s f x and
    C: ⋀z. c = (OUT z :: com-flow) ⟹
      y ∈ sources-out cs′ ?vs′ ?s′ f x ⟹
        sources cs vs s f y ⊆ sources-out ?cs vs s f x and
    D: ⋀X. c = ⟨X⟩ ⟹
      y ∈ sources-out cs′ ?vs′ ?s′ f x ⟹
        sources cs vs s f y ⊆ sources-out ?cs vs s f x
  shows y ∈ sources-out (cs′ @ [c]) ?vs′ ?s′ f x ⟹
    sources cs vs s f y ⊆ sources-out (cs @ cs′ @ [c]) vs s f x
proof (subst (asm) sources-out.simps, split com-flow.split-asm)
  fix z a
  assume c = z ::= a and y ∈ sources-out cs′ ?vs′ ?s′ f x
  moreover from this have sources cs vs s f y ⊆ sources-out ?cs vs s f x
    using A by simp
  ultimately show ?thesis
    by (simp only: append-assoc [symmetric] sources-out.simps, auto)
next
  fix z
  assume c = IN z and y ∈ sources-out cs′ ?vs′ ?s′ f x
  moreover from this have sources cs vs s f y ⊆ sources-out ?cs vs s f x
    using B by simp
  ultimately show ?thesis
    by (simp only: append-assoc [symmetric] sources-out.simps, auto)
next
  fix z
  assume E: c = OUT z
  show y ∈ sources-out cs′ ?vs′ ?s′ f x ∪
    (if z = x then sources cs′ ?vs′ ?s′ f x else {}) ⟹ ?thesis
    (is - ∈ ?A ∪ (if - then ?B else -) ⟹ -)
  proof (split if-split-asm)
    assume z = x and y ∈ ?A ∪ ?B
    moreover {
      assume y ∈ ?A
      hence sources cs vs s f y ⊆ sources-out ?cs vs s f x
        using C and E by simp
    }
```

```
    moreover {
      assume y ∈ ?B
      hence sources cs vs s f y ⊆ sources ?cs vs s f x
        by (rule sources-member)
    }
    ultimately show ?thesis
      using E by (simp only: append-assoc [symmetric] sources-out.simps, auto)
  next
    assume y ∈ ?A ∪ {}
    moreover from this have sources cs vs s f y ⊆ sources-out ?cs vs s f x
      using C and E by simp
    ultimately show ?thesis
      using E by (simp only: append-assoc [symmetric] sources-out.simps, auto)
  qed
next
  fix X
  assume E: c = ⟨X⟩
  assume y ∈ sources-out cs' ?vs' ?s' f x ∪ ⋃ {sources cs' ?vs' ?s' f w | w.
    run-flow cs' ?vs' ?s' f: dom w ⇝ dom x ∧ w ∈ X}
    (is - ∈ ?A ∪ ?B)
  moreover {
    assume y ∈ ?A
    hence sources cs vs s f y ⊆ sources-out ?cs vs s f x
      using D and E by simp
  }
  moreover {
    assume y ∈ ?B
    hence sources cs vs s f y ⊆ ⋃ {sources ?cs vs s f w | w.
      run-flow ?cs vs s f: dom w ⇝ dom x ∧ w ∈ X}
      by (auto dest: sources-member simp: run-flow-append)
  }
  ultimately show ?thesis
    using E by (simp only: append-assoc [symmetric] sources-out.simps, auto)
qed

lemma sources-out-member:
 y ∈ sources-out cs' (vs @ in-flow cs vs f) (run-flow cs vs s f) f x ⟹
   sources cs vs s f y ⊆ sources-out (cs @ cs') vs s f x
by (induction cs' vs s f x rule: sources-out.induct,
 erule sources-out-member-1, simp-all)


lemma tags-member-1:
  assumes
    A: ⋀z a. c = (z ::= a :: com-flow) ⟹ z = x ⟹
      y ∈ sources-aux cs' (vs @ in-flow cs vs f) (run-flow cs vs s f) f x ⟹
        tags cs vs s f y ⊆ tags-aux (cs @ cs') vs s f x
    (is ⋀- -. - ⟹ - ⟹ - ∈ sources-aux - ?vs' ?s' - - ⟹
      - ⊆ tags-aux ?cs - - - -)
```

```
  assumes
    B: ⋀z a b w. c = (z ::= a :: com-flow) ⟹ z = x ⟹
      y ∈ sources cs' ?vs' ?s' f w ⟹
        tags cs vs s f y ⊆ tags ?cs vs s f w and
    C: ⋀z a. c = (z ::= a :: com-flow) ⟹ z ≠ x ⟹
      y ∈ sources cs' ?vs' ?s' f x ⟹
        tags cs vs s f y ⊆ tags ?cs vs s f x and
    D: ⋀z. c = (IN z :: com-flow) ⟹ z = x ⟹
      y ∈ sources-aux cs' ?vs' ?s' f x ⟹
        tags cs vs s f y ⊆ tags-aux ?cs vs s f x and
    E: ⋀z. c = (IN z :: com-flow) ⟹ z ≠ x ⟹
      y ∈ sources cs' ?vs' ?s' f x ⟹
        tags cs vs s f y ⊆ tags ?cs vs s f x and
    F: ⋀z. c = (OUT z :: com-flow) ⟹
      y ∈ sources cs' ?vs' ?s' f x ⟹
        tags cs vs s f y ⊆ tags ?cs vs s f x and
    G: ⋀X b w. c = ⟨X⟩ ⟹
      y ∈ sources cs' ?vs' ?s' f w ⟹
        tags cs vs s f y ⊆ tags ?cs vs s f w
  shows y ∈ sources (cs' @ [c]) ?vs' ?s' f x ⟹
    tags cs vs s f y ⊆ tags (cs @ cs' @ [c]) vs s f x
proof (subst (asm) sources.simps, split com-flow.split-asm)
  fix z a
  assume H: c = z ::= a
  show y ∈ (if z = x
    then sources-aux cs' ?vs' ?s' f x ∪ ⋃ {sources cs' ?vs' ?s' f w | w.
      run-flow cs' ?vs' ?s' f: dom w ⇝ dom x ∧ w ∈ avars a}
    else sources cs' ?vs' ?s' f x) ⟹ ?thesis
    (is - ∈ (if - then ?A ∪ ?B else ?C) ⟹ -)
  proof (split if-split-asm)
    assume I: z = x and y ∈ ?A ∪ ?B
    moreover {
      assume y ∈ ?A
      hence tags cs vs s f y ⊆ tags-aux ?cs vs s f x
        using A and H and I by simp
    }
    moreover {
      assume y ∈ ?B
      hence tags cs vs s f y ⊆ ⋃ {tags ?cs vs s f w | w.
        run-flow ?cs vs s f: dom w ⇝ dom x ∧ w ∈ avars a}
        using B and H and I by (fastforce simp: run-flow-append)
    }
    ultimately show ?thesis
      using H by (simp only: append-assoc [symmetric] tags.simps, auto)
  next
    assume z ≠ x and y ∈ ?C
    moreover from this have tags cs vs s f y ⊆ tags ?cs vs s f x
      using C and H by simp
    ultimately show ?thesis
```

```
      using H by (simp only: append-assoc [symmetric] tags.simps, auto)
  qed
next
  fix z
  assume H: c = IN z
  show y ∈ (if z = x
    then sources-aux cs′ ?vs′ ?s′ f x
    else sources cs′ ?vs′ ?s′ f x) ⟹ ?thesis
    (is - ∈ (if - then ?A else ?B) ⟹ -)
  proof (split if-split-asm)
    assume z = x and y ∈ ?A
    moreover from this have tags cs vs s f y ⊆ tags-aux ?cs vs s f x
      using D and H by simp
    ultimately show ?thesis
      using H by (simp only: append-assoc [symmetric] tags.simps, auto)
  next
    assume z ≠ x and y ∈ ?B
    moreover from this have tags cs vs s f y ⊆ tags ?cs vs s f x
      using E and H by simp
    ultimately show ?thesis
      using H by (simp only: append-assoc [symmetric] tags.simps, auto)
  qed
next
  fix z
  assume c = OUT z and y ∈ sources cs′ ?vs′ ?s′ f x
  moreover from this have tags cs vs s f y ⊆ tags ?cs vs s f x
    using F by simp
  ultimately show ?thesis
    by (simp only: append-assoc [symmetric] tags.simps, auto)
next
  fix X
  assume H: c = ⟨X⟩
  assume y ∈ sources cs′ ?vs′ ?s′ f x ∪ ⋃ {sources cs′ ?vs′ ?s′ f w | w.
    run-flow cs′ ?vs′ ?s′ f: dom w ⇝ dom x ∧ w ∈ X}
    (is - ∈ ?A ∪ ?B)
  moreover {
    assume y ∈ ?A
    hence tags cs vs s f y ⊆ tags ?cs vs s f x
      using G and H by simp
  }
  moreover {
    assume y ∈ ?B
    hence tags cs vs s f y ⊆ ⋃ {tags ?cs vs s f w | w.
      run-flow ?cs vs s f: dom w ⇝ dom x ∧ w ∈ X}
      using G and H by (auto simp: run-flow-append)
  }
  ultimately show ?thesis
    using H by (simp only: append-assoc [symmetric] tags.simps, auto)
qed
```

```
lemma tags-member-2:
  assumes
    A: ⋀z a. c = (z ::= a :: com-flow) ⟹
      y ∈ sources-aux cs' (vs @ in-flow cs vs f) (run-flow cs vs s f) f x ⟹
        tags cs vs s f y ⊆ tags-aux (cs @ cs') vs s f x
      (is ⋀- -. - ⟹ - ∈ sources-aux - ?vs' ?s' - - ⟹
        - ⊆ tags-aux ?cs - - - -)
  assumes
    B: ⋀z. c = (IN z :: com-flow) ⟹
      y ∈ sources-aux cs' ?vs' ?s' f x ⟹
        tags cs vs s f y ⊆ tags-aux ?cs vs s f x and
    C: ⋀z. c = (OUT z :: com-flow) ⟹
      y ∈ sources-aux cs' ?vs' ?s' f x ⟹
        tags cs vs s f y ⊆ tags-aux ?cs vs s f x and
    D: ⋀X. c = ⟨X⟩ ⟹
      y ∈ sources-aux cs' ?vs' ?s' f x ⟹
        tags cs vs s f y ⊆ tags-aux ?cs vs s f x and
    E: ⋀X b w. c = ⟨X⟩ ⟹
      y ∈ sources cs' ?vs' ?s' f w ⟹
        tags cs vs s f y ⊆ tags ?cs vs s f w
  shows y ∈ sources-aux (cs' @ [c]) ?vs' ?s' f x ⟹
    tags cs vs s f y ⊆ tags-aux (cs @ cs' @ [c]) vs s f x
proof (subst (asm) sources-aux.simps, split com-flow.split-asm)
  fix z a
  assume c = z ::= a and y ∈ sources-aux cs' ?vs' ?s' f x
  moreover from this have tags cs vs s f y ⊆ tags-aux ?cs vs s f x
    using A by simp
  ultimately show ?thesis
    by (simp only: append-assoc [symmetric] tags-aux.simps, auto)
next
  fix z
  assume c = IN z and y ∈ sources-aux cs' ?vs' ?s' f x
  moreover from this have tags cs vs s f y ⊆ tags-aux ?cs vs s f x
    using B by simp
  ultimately show ?thesis
    by (simp only: append-assoc [symmetric] tags-aux.simps, auto)
next
  fix z
  assume c = OUT z and y ∈ sources-aux cs' ?vs' ?s' f x
  moreover from this have tags cs vs s f y ⊆ tags-aux ?cs vs s f x
    using C by simp
  ultimately show ?thesis
    by (simp only: append-assoc [symmetric] tags-aux.simps, auto)
next
  fix X
  assume F: c = ⟨X⟩
  assume y ∈ sources-aux cs' ?vs' ?s' f x ∪ ⋃ {sources cs' ?vs' ?s' f w | w.
    run-flow cs' ?vs' ?s' f: dom w ⇝ dom x ∧ w ∈ X}
```

```
    (is - ∈ ?A ∪ ?B)
  moreover {
    assume y ∈ ?A
    hence tags cs vs s f y ⊆ tags-aux ?cs vs s f x
      using D and F by simp
  }
  moreover {
    assume y ∈ ?B
    hence tags cs vs s f y ⊆ ⋃ {tags ?cs vs s f w | w.
      run-flow ?cs vs s f: dom w ⇝ dom x ∧ w ∈ X}
      using E and F by (auto simp: run-flow-append)
  }
  ultimately show ?thesis
    using F by (simp only: append-assoc [symmetric] tags-aux.simps, auto)
qed

lemma tags-member:
 y ∈ sources cs' (vs @ in-flow cs vs f) (run-flow cs vs s f) f x ⟹
    tags cs vs s f y ⊆ tags (cs @ cs') vs s f x
and tags-aux-member:
 y ∈ sources-aux cs' (vs @ in-flow cs vs f) (run-flow cs vs s f) f x ⟹
    tags cs vs s f y ⊆ tags-aux (cs @ cs') vs s f x
by (induction cs' vs s f x and cs' vs s f x rule: tags-induct,
 erule-tac [3] tags-member-2, erule tags-member-1, simp-all)


lemma tags-out-member-1:
  assumes
    A: ⋀z a. c = (z ::= a :: com-flow) ⟹
      y ∈ sources-out cs' (vs @ in-flow cs vs f) (run-flow cs vs s f) f x ⟹
        tags cs vs s f y ⊆ tags-out (cs @ cs') vs s f x
      (is ⋀- -. - ⟹ - ∈ sources-out - ?vs' ?s' - - ⟹
        - ⊆ tags-out ?cs - - - -)
  assumes
    B: ⋀z. c = (IN z :: com-flow) ⟹
      y ∈ sources-out cs' ?vs' ?s' f x ⟹
        tags cs vs s f y ⊆ tags-out ?cs vs s f x and
    C: ⋀z. c = (OUT z :: com-flow) ⟹
      y ∈ sources-out cs' ?vs' ?s' f x ⟹
        tags cs vs s f y ⊆ tags-out ?cs vs s f x and
    D: ⋀X. c = ⟨X⟩ ⟹
      y ∈ sources-out cs' ?vs' ?s' f x ⟹
        tags cs vs s f y ⊆ tags-out ?cs vs s f x
  shows y ∈ sources-out (cs' @ [c]) ?vs' ?s' f x ⟹
    tags cs vs s f y ⊆ tags-out (cs @ cs' @ [c]) vs s f x
proof (subst (asm) sources-out.simps, split com-flow.split-asm)
  fix z a
  assume c = z ::= a and y ∈ sources-out cs' ?vs' ?s' f x
  moreover from this have tags cs vs s f y ⊆ tags-out ?cs vs s f x
```

```
    using A by simp
  ultimately show ?thesis
    by (simp only: append-assoc [symmetric] tags-out.simps, auto)
next
  fix z
  assume c = IN z and y ∈ sources-out cs' ?vs' ?s' f x
  moreover from this have tags cs vs s f y ⊆ tags-out ?cs vs s f x
    using B by simp
  ultimately show ?thesis
    by (simp only: append-assoc [symmetric] tags-out.simps, auto)
next
  fix z
  assume E: c = OUT z
  show y ∈ sources-out cs' ?vs' ?s' f x ∪
    (if z = x then sources cs' ?vs' ?s' f x else {}) ⟹ ?thesis
    (is - ∈ ?A ∪ (if - then ?B else -) ⟹ -)
  proof (split if-split-asm)
    assume z = x and y ∈ ?A ∪ ?B
    moreover {
      assume y ∈ ?A
      hence tags cs vs s f y ⊆ tags-out ?cs vs s f x
        using C and E by simp
    }
    moreover {
      assume y ∈ ?B
      hence tags cs vs s f y ⊆ tags ?cs vs s f x
        by (rule tags-member)
    }
    ultimately show ?thesis
      using E by (simp only: append-assoc [symmetric] tags-out.simps, auto)
  next
    assume y ∈ ?A ∪ {}
    moreover from this have tags cs vs s f y ⊆ tags-out ?cs vs s f x
      using C and E by simp
    ultimately show ?thesis
      using E by (simp only: append-assoc [symmetric] tags-out.simps, auto)
  qed
next
  fix X
  assume E: c = ⟨X⟩
  assume y ∈ sources-out cs' ?vs' ?s' f x ∪ ⋃ {sources cs' ?vs' ?s' f w | w.
    run-flow cs' ?vs' ?s' f: dom w ⇝ dom x ∧ w ∈ X}
    (is - ∈ ?A ∪ ?B)
  moreover {
    assume y ∈ ?A
    hence tags cs vs s f y ⊆ tags-out ?cs vs s f x
      using D and E by simp
  }
  moreover {
```

```
    assume y ∈ ?B
    hence tags cs vs s f y ⊆ ⋃ {tags ?cs vs s f w | w.
      run-flow ?cs vs s f: dom w ⇝ dom x ∧ w ∈ X}
      by (auto dest: tags-member simp: run-flow-append)
  }
  ultimately show ?thesis
    using E by (simp only: append-assoc [symmetric] tags-out.simps, auto)
qed

lemma tags-out-member:
 y ∈ sources-out cs' (vs @ in-flow cs vs f) (run-flow cs vs s f) f x ⟹
   tags cs vs s f y ⊆ tags-out (cs @ cs') vs s f x
by (induction cs' vs s f x rule: tags-out.induct,
 erule tags-out-member-1, simp-all)


lemma tags-suffix-1:
  assumes
    A: ⋀z a. c = (z ::= a :: com-flow) ⟹ z = x ⟹
      tags-aux cs' (vs @ in-flow cs vs f) (run-flow cs vs s f) f x =
      {p. case p of (w, n) ⇒ (w, length [c←cs. c = IN w] + n)
        ∈ tags-aux (cs @ cs') vs s f x}
      (is ⋀- -. - ⟹ - ⟹ tags-aux - ?vs' ?s' - - = -)
  assumes
    B: ⋀z a b y. c = (z ::= a :: com-flow) ⟹ z = x ⟹
      tags cs' ?vs' ?s' f y =
      {p. case p of (w, n) ⇒ (w, length [c←cs. c = IN w] + n)
        ∈ tags (cs @ cs') vs s f y} and
    C: ⋀z a. c = (z ::= a :: com-flow) ⟹ z ≠ x ⟹
      tags cs' ?vs' ?s' f x =
      {p. case p of (w, n) ⇒ (w, length [c←cs. c = IN w] + n)
        ∈ tags (cs @ cs') vs s f x} and
    D: ⋀z. c = (IN z :: com-flow) ⟹ z = x ⟹
      tags-aux cs' ?vs' ?s' f x =
      {p. case p of (w, n) ⇒ (w, length [c←cs. c = IN w] + n)
        ∈ tags-aux (cs @ cs') vs s f x} and
    E: ⋀z. c = (IN z :: com-flow) ⟹ z ≠ x ⟹
      tags cs' ?vs' ?s' f x =
      {p. case p of (w, n) ⇒ (w, length [c←cs. c = IN w] + n)
        ∈ tags (cs @ cs') vs s f x} and
    F: ⋀z. c = (OUT z :: com-flow) ⟹
      tags cs' ?vs' ?s' f x =
      {p. case p of (w, n) ⇒ (w, length [c←cs. c = IN w] + n)
        ∈ tags (cs @ cs') vs s f x} and
    G: ⋀X b y. c = ⟨X⟩ ⟹
      tags cs' ?vs' ?s' f y =
      {p. case p of (w, n) ⇒ (w, length [c←cs. c = IN w] + n)
        ∈ tags (cs @ cs') vs s f y}
  shows tags (cs' @ [c]) ?vs' ?s' f x =
```

```
  {p. case p of (w, n) ⇒ (w, length [c←cs. c = IN w] + n)
    ∈ tags (cs @ cs' @ [c]) vs s f x}
  (is - = {p. case p of (w, n) ⇒ ?P w n c})
apply (subst tags.simps)
apply (split com-flow.split)
apply (rule conjI)
subgoal
proof −
  show ∀ z a. c = z ::= a ⟶ (if z = x
    then tags-aux cs' ?vs' ?s' f x ∪ ⋃ {tags cs' ?vs' ?s' f y | y.
      run-flow cs' ?vs' ?s' f: dom y ⇝ dom x ∧ y ∈ avars a}
    else tags cs' ?vs' ?s' f x) =
    {(w, n). ?P w n c}
    (is ∀ - a. - ⟶ (if - then ?A ∪ ?F a else ?B) = -)
    apply clarify
    apply (split if-split)
    apply (rule conjI)
    subgoal for z a
    proof
      assume H: z = x and I: c = z ::= a
      hence ?A = {(w, n). (w, length [c←cs. c = IN w] + n)
        ∈ tags-aux (cs @ cs') vs s f x}
        using A by simp
      moreover have ∀ y. tags cs' ?vs' ?s' f y = {(w, n).
        (w, length [c←cs. c = IN w] + n) ∈ tags (cs @ cs') vs s f y}
        using B and H and I by simp
      hence ?F a = {(w, n). (w, length [c←cs. c = IN w] + n)
        ∈ ⋃ {tags (cs @ cs') vs s f y | y.
          run-flow cs' ?vs' ?s' f: dom y ⇝ dom x ∧ y ∈ avars a}}
        by blast
      ultimately show ?A ∪ ?F a = {(w, n). ?P w n (z ::= a)}
        using H by (subst append-assoc [symmetric], subst tags.simps,
         auto simp: run-flow-append)
    qed
    subgoal for z a
    proof
      assume z ≠ x and c = z ::= a
      moreover from this have ?B = {(w, n).
        (w, length [c←cs. c = IN w] + n) ∈ tags (cs @ cs') vs s f x}
        using C by simp
      ultimately show ?B = {(w, n). ?P w n (z ::= a)}
        by (subst append-assoc [symmetric], subst tags.simps, simp)
    qed
    done
qed
apply (rule conjI)
subgoal
proof −
  show ∀ z. c = IN z ⟶ (if z = x
```

```
    then insert (x, length [c←cs'. c = IN x]) (tags-aux cs' ?vs' ?s' f x)
    else tags cs' ?vs' ?s' f x) =
    {(w, n). ?P w n c}
    (is ∀ -. - ⟶ (if - then insert ?p ?A else ?B) = -)
    apply clarify
    apply (split if-split)
    apply (rule conjI)
    subgoal for z
    proof
      assume z = x and c = IN z
      moreover from this have ?A = {(w, n).
        (w, length [c←cs. c = IN w] + n) ∈ tags-aux (cs @ cs') vs s f x}
        using D by simp
      ultimately show insert ?p ?A = {(w, n). ?P w n (IN z)}
        by (subst append-assoc [symmetric], subst tags.simps, auto)
    qed
    subgoal for z
    proof
      assume z ≠ x and c = IN z
      moreover from this have ?B = {(w, n).
        (w, length [c←cs. c = IN w] + n) ∈ tags (cs @ cs') vs s f x}
        using E by simp
      ultimately show ?B = {(w, n). ?P w n (IN z)}
        by (subst append-assoc [symmetric], subst tags.simps, simp)
    qed
    done
qed
apply (rule conjI)
subgoal by (subst append-assoc [symmetric], subst tags.simps, simp add: F)
subgoal
proof −
  show ∀ X. c = ⟨X⟩ ⟶
    tags cs' ?vs' ?s' f x ∪ ⋃ {tags cs' ?vs' ?s' f y | y.
      run-flow cs' ?vs' ?s' f: dom y ⇝ dom x ∧ y ∈ X} =
    {(w, n). ?P w n c}
    (is ∀ X. - ⟶ ?A ∪ ?F X = -)
  proof clarify
    fix X
    assume H: c = ⟨X⟩
    hence ?A = {(w, n). (w, length [c←cs. c = IN w] + n)
      ∈ tags (cs @ cs') vs s f x}
      using G by simp
    moreover have ∀ y. tags cs' ?vs' ?s' f y = {(w, n).
      (w, length [c←cs. c = IN w] + n) ∈ tags (cs @ cs') vs s f y}
      using G and H by simp
    hence ?F X = {(w, n). (w, length [c←cs. c = IN w] + n)
      ∈ ⋃ {tags (cs @ cs') vs s f y | y.
        run-flow cs' ?vs' ?s' f: dom y ⇝ dom x ∧ y ∈ X}}
      by blast
```

```
    ultimately show ?A ∪ ?F X = {(w, n). ?P w n (⟨X⟩)}
      by (subst append-assoc [symmetric], subst tags.simps,
       auto simp: run-flow-append)
  qed
qed
done

lemma tags-suffix-2:
  assumes
    A: ⋀z a. c = (z ::= a :: com-flow) ⟹
      tags-aux cs′ (vs @ in-flow cs vs f) (run-flow cs vs s f) f x =
      {p. case p of (w, n) ⇒ (w, length [c←cs. c = IN w] + n)
        ∈ tags-aux (cs @ cs′) vs s f x}
      (is ⋀- -. - ⟹ tags-aux - ?vs′ ?s′ - - = -)
  assumes
    B: ⋀z. c = (IN z :: com-flow) ⟹
      tags-aux cs′ ?vs′ ?s′ f x =
      {p. case p of (w, n) ⇒ (w, length [c←cs. c = IN w] + n)
        ∈ tags-aux (cs @ cs′) vs s f x} and
    C: ⋀z. c = (OUT z :: com-flow) ⟹
      tags-aux cs′ ?vs′ ?s′ f x =
      {p. case p of (w, n) ⇒ (w, length [c←cs. c = IN w] + n)
        ∈ tags-aux (cs @ cs′) vs s f x} and
    D: ⋀X. c = ⟨X⟩ ⟹
      tags-aux cs′ ?vs′ ?s′ f x =
      {p. case p of (w, n) ⇒ (w, length [c←cs. c = IN w] + n)
        ∈ tags-aux (cs @ cs′) vs s f x} and
    E: ⋀X b y. c = ⟨X⟩ ⟹
      tags cs′ ?vs′ ?s′ f y =
      {p. case p of (w, n) ⇒ (w, length [c←cs. c = IN w] + n)
        ∈ tags (cs @ cs′) vs s f y}
  shows tags-aux (cs′ @ [c]) ?vs′ ?s′ f x =
    {p. case p of (w, n) ⇒ (w, length [c←cs. c = IN w] + n)
      ∈ tags-aux (cs @ cs′ @ [c]) vs s f x}
    (is - = {p. case p of (w, n) ⇒ ?P w n c})
  apply (subst tags-aux.simps)
  apply (split com-flow.split)
  apply (rule conjI)
   defer
   apply (rule conjI)
    defer
    apply (rule conjI)
     defer
  subgoal
  proof −
    show ∀X. c = ⟨X⟩ ⟶
      tags-aux cs′ ?vs′ ?s′ f x ∪ ⋃ {tags cs′ ?vs′ ?s′ f y | y.
        run-flow cs′ ?vs′ ?s′ f: dom y ⇝ dom x ∧ y ∈ X} =
      {(w, n). ?P w n c}
```

```
      (is ∀ X. - ⟶ ?A ∪ ?F X = -)
    proof clarify
      fix X
      assume F: c = ⟨X⟩
      hence ?A = {(w, n). (w, length [c←cs. c = IN w] + n)
        ∈ tags-aux (cs @ cs') vs s f x}
        using D by simp
      moreover have ∀ y. tags cs' ?vs' ?s' f y = {(w, n).
        (w, length [c←cs. c = IN w] + n) ∈ tags (cs @ cs') vs s f y}
        using E and F by simp
      hence ?F X = {(w, n). (w, length [c←cs. c = IN w] + n)
        ∈ ⋃ {tags (cs @ cs') vs s f y | y.
          run-flow cs' ?vs' ?s' f: dom y ⇝ dom x ∧ y ∈ X}}
        by blast
      ultimately show ?A ∪ ?F X = {(w, n). ?P w n (⟨X⟩)}
        by (subst append-assoc [symmetric], subst tags-aux.simps,
         auto simp: run-flow-append)
    qed
  qed
by (subst append-assoc [symmetric], subst tags-aux.simps, simp add: A B C)+

lemma tags-suffix:
 tags cs' (vs @ in-flow cs vs f) (run-flow cs vs s f) f x = {(w, n).
    (w, length [c←cs. c = IN w] + n) ∈ tags (cs @ cs') vs s f x}
and tags-aux-suffix:
 tags-aux cs' (vs @ in-flow cs vs f) (run-flow cs vs s f) f x = {(w, n).
    (w, length [c←cs. c = IN w] + n) ∈ tags-aux (cs @ cs') vs s f x}
by (induction cs' vs s f x and cs' vs s f x rule: tags-induct,
 erule-tac [3] tags-suffix-2, erule tags-suffix-1, simp-all
 add: tags-ubound tags-aux-ubound)


lemma tags-out-suffix-1:
  assumes
    A: ⋀z a. c = (z ::= a :: com-flow) ⟹
      tags-out cs' (vs @ in-flow cs vs f) (run-flow cs vs s f) f x =
      {p. case p of (w, n) ⇒ (w, length [c←cs. c = IN w] + n)
        ∈ tags-out (cs @ cs') vs s f x}
      (is ⋀- -. - ⟹ tags-out - ?vs' ?s' - - = -)
  assumes
    B: ⋀z. c = (IN z :: com-flow) ⟹
      tags-out cs' ?vs' ?s' f x =
      {p. case p of (w, n) ⇒ (w, length [c←cs. c = IN w] + n)
        ∈ tags-out (cs @ cs') vs s f x} and
    C: ⋀z. c = (OUT z :: com-flow) ⟹
      tags-out cs' ?vs' ?s' f x =
      {p. case p of (w, n) ⇒ (w, length [c←cs. c = IN w] + n)
        ∈ tags-out (cs @ cs') vs s f x} and
    D: ⋀X. c = ⟨X⟩ ⟹
```

```
    tags-out cs′ ?vs′ ?s′ f x =
    {p. case p of (w, n) ⇒ (w, length [c←cs. c = IN w] + n)
      ∈ tags-out (cs @ cs′) vs s f x}
shows tags-out (cs′ @ [c]) ?vs′ ?s′ f x =
  {p. case p of (w, n) ⇒ (w, length [c←cs. c = IN w] + n)
    ∈ tags-out (cs @ cs′ @ [c]) vs s f x}
  (is - = {p. case p of (w, n) ⇒ ?P w n c})
apply (subst tags-out.simps)
apply (split com-flow.split)
apply (rule conjI)
 defer
 apply (rule conjI)
  defer
subgoal
proof
  show ∀ z. c = OUT z ⟶
    tags-out cs′ ?vs′ ?s′ f x ∪
      (if z = x then tags cs′ ?vs′ ?s′ f x else {}) =
    {(w, n). ?P w n c}
    (is ∀ -. - ⟶ ?A ∪ (if - then ?B else -) = -)
    apply clarify
    apply (split if-split)
    apply (rule conjI)
    subgoal for z
    proof
      assume c = OUT z and z = x
      moreover from this have ?A = {p. case p of (w, n) ⇒
        (w, length [c←cs. c = IN w] + n) ∈ tags-out (cs @ cs′) vs s f x}
        using C by simp
      moreover have ?B = {(w, n).
        (w, length [c←cs. c = IN w] + n) ∈ tags (cs @ cs′) vs s f x}
        by (rule tags-suffix)
      ultimately show ?A ∪ ?B = {(w, n). ?P w n (OUT z)}
        by (subst append-assoc [symmetric], subst tags-out.simps, auto)
    qed
    subgoal for z
    proof
      assume c = OUT z and z ≠ x
      moreover from this have ?A = {p. case p of (w, n) ⇒
        (w, length [c←cs. c = IN w] + n) ∈ tags-out (cs @ cs′) vs s f x}
        using C by simp
      ultimately show ?A ∪ {} = {(w, n). ?P w n (OUT z)}
        by (subst append-assoc [symmetric], subst tags-out.simps, simp)
    qed
    done
next
  show ∀ X. c = ⟨X⟩ ⟶
    tags-out cs′ ?vs′ ?s′ f x ∪ ⋃ {tags cs′ ?vs′ ?s′ f y | y.
      run-flow cs′ ?vs′ ?s′ f: dom y ⇝ dom x ∧ y ∈ X} =
```

```
      {(w, n). ?P w n c}
      (is ∀X. - ⟶ ?A ∪ ?F X = -)
    proof clarify
      fix X
      assume c = ⟨X⟩
      hence ?A = {(w, n). (w, length [c←cs. c = IN w] + n)
        ∈ tags-out (cs @ cs') vs s f x}
        using D by simp
      moreover have ∀y. tags cs' ?vs' ?s' f y = {(w, n).
        (w, length [c←cs. c = IN w] + n) ∈ tags (cs @ cs') vs s f y}
        by (blast intro!: tags-suffix)
      hence ?F X = {(w, n). (w, length [c←cs. c = IN w] + n)
        ∈ ⋃ {tags (cs @ cs') vs s f y | y.
          run-flow cs' ?vs' ?s' f: dom y ⇝ dom x ∧ y ∈ X}}
        by blast
      ultimately show ?A ∪ ?F X = {(w, n). ?P w n (⟨X⟩)}
        by (subst append-assoc [symmetric], subst tags-out.simps,
         auto simp: run-flow-append)
    qed
  qed
by (subst append-assoc [symmetric], subst tags-out.simps, simp add: A B)+

lemma tags-out-suffix:
 tags-out cs' (vs @ in-flow cs vs f) (run-flow cs vs s f) f x = {(w, n).
   (w, length [c←cs. c = IN w] + n) ∈ tags-out (cs @ cs') vs s f x}
by (induction cs' vs s f x rule: tags-out.induct,
 erule tags-out-suffix-1, simp-all add: tags-out-ubound)


lemma sources-aux-rhs:
  assumes
    A: S ⊆ {x. s₁ = t₁ (⊆ sources-aux (flow cfs @ cs') vs₁ s₁ f x)}
      (is - ⊆ {-. - = - (⊆ sources-aux (?cs @ -) - - - -)})
  assumes
    B: f = f' (⊆ vs₁, vs₁',
      ⋃ {tags-aux (?cs @ cs') vs₁ s₁ f x | x. x ∈ S}) and
    C: (c₁, s₁, f, vs₁, ws₁) →*{cfs} (c₂, s₂, f, vs₂, ws₂) and
    D: ok-flow-aux-2 s₁ s₂ t₁ t₂ f f' vs₁ vs₁' ?cs
  shows S ⊆ {x. s₂ = t₂ (⊆ sources-aux cs' vs₂ s₂ f x)}
proof clarify
  fix x y
  assume E: y ∈ sources-aux cs' vs₂ s₂ f x
  moreover have F: s₂ = run-flow ?cs vs₁ s₁ f
    using C by (rule small-stepsl-run-flow)
  moreover have G: vs₂ = vs₁ @ in-flow ?cs vs₁ f
    using C by (rule small-stepsl-in-flow)
  ultimately have sources ?cs vs₁ s₁ f y ⊆ sources-aux (?cs @ cs') vs₁ s₁ f x
    by (blast dest: sources-aux-member)
  moreover assume H: x ∈ S
```

ultimately have $s_1 = t_1\ (\subseteq \mathit{sources}\ ?cs\ vs_1\ s_1\ f\ y)$
using *A* **by** *blast*
moreover have $\mathit{tags}\ ?cs\ vs_1\ s_1\ f\ y \subseteq \mathit{tags\text{-}aux}\ (?cs\ @\ cs')\ vs_1\ s_1\ f\ x$
using *E* **and** *F* **and** *G* **by** (*blast dest*: *tags-aux-member*)
hence $\mathit{tags}\ ?cs\ vs_1\ s_1\ f\ y \subseteq \bigcup\ \{\mathit{tags\text{-}aux}\ (?cs\ @\ cs')\ vs_1\ s_1\ f\ x \mid x.\ x \in S\}$
using *H* **by** *blast*
with *B* **have** $f = f'\ (\subseteq vs_1,\ vs_1',\ \mathit{tags}\ ?cs\ vs_1\ s_1\ f\ y)$
by (*rule eq-streams-subset*)
ultimately show $s_2\ y = t_2\ y$
using *D* [*rule-format*, *of* $\{y\}$] **by** *simp*
qed

lemma *sources-rhs*:
assumes
A: $S \subseteq \{x.\ s_1 = t_1\ (\subseteq \mathit{sources}\ (\mathit{flow}\ cfs\ @\ cs')\ vs_1\ s_1\ f\ x)\}$
(**is** - $\subseteq$ {-. - = - ($\subseteq$ *sources* (*?cs* @ -) - - - -)})
assumes
B: $f = f'\ (\subseteq vs_1,\ vs_1',$
$\bigcup\ \{\mathit{tags}\ (?cs\ @\ cs')\ vs_1\ s_1\ f\ x \mid x.\ x \in S\})$ **and**
C: $(c_1,\ s_1,\ f,\ vs_1,\ ws_1) \rightarrow\!*\{cfs\}\ (c_2,\ s_2,\ f,\ vs_2,\ ws_2)$ **and**
D: $\mathit{ok\text{-}flow\text{-}aux\text{-}2}\ s_1\ s_2\ t_1\ t_2\ f\ f'\ vs_1\ vs_1'\ ?cs$
shows $S \subseteq \{x.\ s_2 = t_2\ (\subseteq \mathit{sources}\ cs'\ vs_2\ s_2\ f\ x)\}$
proof *clarify*
fix *x y*
assume *E*: $y \in \mathit{sources}\ cs'\ vs_2\ s_2\ f\ x$
moreover have *F*: $s_2 = \mathit{run\text{-}flow}\ ?cs\ vs_1\ s_1\ f$
using *C* **by** (*rule small-stepsl-run-flow*)
moreover have *G*: $vs_2 = vs_1\ @\ \mathit{in\text{-}flow}\ ?cs\ vs_1\ f$
using *C* **by** (*rule small-stepsl-in-flow*)
ultimately have $\mathit{sources}\ ?cs\ vs_1\ s_1\ f\ y \subseteq \mathit{sources}\ (?cs\ @\ cs')\ vs_1\ s_1\ f\ x$
by (*blast dest*: *sources-member*)
moreover assume *H*: $x \in S$
ultimately have $s_1 = t_1\ (\subseteq \mathit{sources}\ ?cs\ vs_1\ s_1\ f\ y)$
using *A* **by** *blast*
moreover have $\mathit{tags}\ ?cs\ vs_1\ s_1\ f\ y \subseteq \mathit{tags}\ (?cs\ @\ cs')\ vs_1\ s_1\ f\ x$
using *E* **and** *F* **and** *G* **by** (*blast dest*: *tags-member*)
hence $\mathit{tags}\ ?cs\ vs_1\ s_1\ f\ y \subseteq \bigcup\ \{\mathit{tags}\ (?cs\ @\ cs')\ vs_1\ s_1\ f\ x \mid x.\ x \in S\}$
using *H* **by** *blast*
with *B* **have** $f = f'\ (\subseteq vs_1,\ vs_1',\ \mathit{tags}\ ?cs\ vs_1\ s_1\ f\ y)$
by (*rule eq-streams-subset*)
ultimately show $s_2\ y = t_2\ y$
using *D* [*rule-format*, *of* $\{y\}$] **by** *simp*
qed

lemma *sources-out-rhs*:
assumes
A: $S \subseteq \{x.\ s_1 = t_1\ (\subseteq \mathit{sources\text{-}out}\ (\mathit{flow}\ cfs\ @\ cs')\ vs_1\ s_1\ f\ x)\}$
(**is** - $\subseteq$ {-. - = - ($\subseteq$ *sources-out* (*?cs* @ -) - - - -)})
assumes

B: $f = f'$ (⊆ vs_1, vs_1',
⋃ {*tags-out* (*?cs* @ *cs'*) vs_1 s_1 *f x* | *x*. *x* ∈ *S*}) **and**
C: (c_1, s_1, *f*, vs_1, ws_1) →∗{*cfs*} (c_2, s_2, *f*, vs_2, ws_2) **and**
D: *ok-flow-aux-2* s_1 s_2 t_1 t_2 *f f'* vs_1 vs_1' *?cs*
shows *S* ⊆ {*x*. $s_2 = t_2$ (⊆ *sources-out cs'* vs_2 s_2 *f x*)}
proof *clarify*
fix *x y*
assume *E*: *y* ∈ *sources-out cs'* vs_2 s_2 *f x*
moreover have *F*: s_2 = *run-flow ?cs* vs_1 s_1 *f*
using *C* **by** (*rule small-stepsl-run-flow*)
moreover have *G*: vs_2 = vs_1 @ *in-flow ?cs* vs_1 *f*
using *C* **by** (*rule small-stepsl-in-flow*)
ultimately have *sources ?cs* vs_1 s_1 *f y* ⊆ *sources-out* (*?cs* @ *cs'*) vs_1 s_1 *f x*
by (*blast dest*: *sources-out-member*)
moreover assume *H*: *x* ∈ *S*
ultimately have $s_1 = t_1$ (⊆ *sources ?cs* vs_1 s_1 *f y*)
using *A* **by** *blast*
moreover have *tags ?cs* vs_1 s_1 *f y* ⊆ *tags-out* (*?cs* @ *cs'*) vs_1 s_1 *f x*
using *E* **and** *F* **and** *G* **by** (*blast dest*: *tags-out-member*)
hence *tags ?cs* vs_1 s_1 *f y* ⊆ ⋃ {*tags-out* (*?cs* @ *cs'*) vs_1 s_1 *f x* | *x*. *x* ∈ *S*}
using *H* **by** *blast*
with *B* **have** $f = f'$ (⊆ vs_1, vs_1', *tags ?cs* vs_1 s_1 *f y*)
by (*rule eq-streams-subset*)
ultimately show s_2 *y* = t_2 *y*
using *D* [*rule-format*, *of* {*y*}] **by** *simp*
qed

lemma *tags-aux-rhs*:
assumes
A: *S* ⊆ {*x*. $s_1 = t_1$ (⊆ *sources-aux* (*flow cfs* @ *cs'*) vs_1 s_1 *f x*)}
(**is** - ⊆ {-. - = - (⊆ *sources-aux* (*?cs* @ -) - - - -)})
assumes
B: $f = f'$ (⊆ vs_1, vs_1',
⋃ {*tags-aux* (*?cs* @ *cs'*) vs_1 s_1 *f x* | *x*. *x* ∈ *S*}) **and**
C: (c_1, s_1, *f*, vs_1, ws_1) →∗{*cfs*} (c_2, s_2, *f*, vs_2, ws_2) **and**
D: (c_1', t_1, *f'*, vs_1', ws_1') →∗ (c_2', t_2, *f'*, vs_2', ws_2') **and**
E: *ok-flow-aux-1* c_1 c_2 c_2' s_1 t_1 t_2 *f f'* vs_1 vs_1' vs_2 vs_2' ws_1' ws_2' *?cs*
shows $f = f'$ (⊆ vs_2, vs_2', ⋃ {*tags-aux cs'* vs_2 s_2 *f x* | *x*. *x* ∈ *S*})
proof (*subst eq-streams-def*, *clarify*)
fix *x y n*
have *F*: vs_2 = vs_1 @ *drop* (*length* vs_1) vs_2
using *small-stepsl-steps* [*OF C*] **by** (*rule small-steps-in-flow*)
have *G*: vs_2' = vs_1' @ *drop* (*length* vs_1') vs_2'
using *D* **by** (*rule small-steps-in-flow*)
assume (*y*, *n*) ∈ *tags-aux cs'* vs_2 s_2 *f x*
moreover have s_2 = *run-flow ?cs* vs_1 s_1 *f*
using *C* **by** (*rule small-stepsl-run-flow*)
moreover have *H*: vs_2 = vs_1 @ *in-flow ?cs* vs_1 *f*

using *C* **by** (*rule small-stepsl-in-flow*)
ultimately have *I*: (*y*, *length* [*c*←*?cs*. *c* = *IN y*] + *n*)
∈ *tags-aux* (*?cs* @ *cs′*) vs_1 s_1 *f x*
(**is** (-, *?k* + -) ∈ -)
by (*simp add*: *tags-aux-suffix*)
let *?m* = *Suc* (*Max* {*k*. *k* ≤ *length* (*?cs* @ *cs′*) ∧
length [*c*←*take k* (*?cs* @ *cs′*). *c* = *IN y*] ≤ *?k* + *n*})
have *J*: *y* ∈ *sources-aux* (*drop ?m* (*?cs* @ *cs′*))
(vs_1 @ *in-flow* (*take ?m* (*?cs* @ *cs′*)) vs_1 *f*)
(*run-flow* (*take ?m* (*?cs* @ *cs′*)) vs_1 s_1 *f*) *f x*
using *I* **by** (*auto dest*: *tags-aux-sources-aux*)
hence *sources* (*take ?m* (*?cs* @ *cs′*)) vs_1 s_1 *f y* ⊆
sources-aux (*take ?m* (*?cs* @ *cs′*) @ *drop ?m* (*?cs* @ *cs′*)) vs_1 s_1 *f x*
by (*rule sources-aux-member*)
moreover have *K*: *length ?cs* ≤ *?m*
by (*rule le-SucI*, *rule Max-ge*, *simp-all*)
ultimately have
sources (*?cs* @ *take* (*?m* − *length ?cs*) *cs′*) vs_1 s_1 *f y* ⊆
sources-aux (*?cs* @ *cs′*) vs_1 s_1 *f x*
by *simp*
moreover have
sources-aux (*?cs* @ *take* (*?m* − *length ?cs*) *cs′*) vs_1 s_1 *f y* ⊆
sources (*?cs* @ *take* (*?m* − *length ?cs*) *cs′*) vs_1 s_1 *f y*
by (*rule sources-aux-sources*)
moreover have *sources-aux ?cs* vs_1 s_1 *f y* ⊆
sources-aux (*?cs* @ *take* (*?m* − *length ?cs*) *cs′*) vs_1 s_1 *f y*
by (*rule sources-aux-append*)
moreover assume *L*: *x* ∈ *S*
hence s_1 = t_1 (⊆ *sources-aux* (*?cs* @ *cs′*) vs_1 s_1 *f x*)
using *A* **by** *blast*
ultimately have *M*: s_1 = t_1 (⊆ *sources-aux ?cs* vs_1 s_1 *f y*)
by *blast*
have *tags* (*take ?m* (*?cs* @ *cs′*)) vs_1 s_1 *f y* ⊆
tags-aux (*take ?m* (*?cs* @ *cs′*) @ *drop ?m* (*?cs* @ *cs′*)) vs_1 s_1 *f x*
using *J* **by** (*rule tags-aux-member*)
hence *tags* (*?cs* @ *take* (*?m* − *length ?cs*) *cs′*) vs_1 s_1 *f y* ⊆
tags-aux (*?cs* @ *cs′*) vs_1 s_1 *f x*
using *K* **by** *simp*
moreover have
tags-aux (*?cs* @ *take* (*?m* − *length ?cs*) *cs′*) vs_1 s_1 *f y* ⊆
tags (*?cs* @ *take* (*?m* − *length ?cs*) *cs′*) vs_1 s_1 *f y*
by (*rule tags-aux-tags*)
moreover have *tags-aux ?cs* vs_1 s_1 *f y* ⊆
tags-aux (*?cs* @ *take* (*?m* − *length ?cs*) *cs′*) vs_1 s_1 *f y*
by (*rule tags-aux-append*)
ultimately have *tags-aux ?cs* vs_1 s_1 *f y* ⊆
⋃ {*tags-aux* (*?cs* @ *cs′*) vs_1 s_1 *f x* | *x*. *x* ∈ *S*}
using *L* **by** *blast*
with *B* **have** *f* = *f′* (⊆ vs_1, $vs_1{}'$, *tags-aux ?cs* vs_1 s_1 *f y*)

```
    by (rule eq-streams-subset)
  hence map fst [p←drop (length vs1) vs2. fst p = y] =
    map fst [p←drop (length vs1') vs2'. fst p = y]
    using E [rule-format, of {y}] and M by simp
  hence length [p←drop (length vs1) vs2. fst p = y] =
    length [p←drop (length vs1') vs2'. fst p = y]
    by (drule-tac arg-cong [where f = length],
     subst (asm) (1 2) length-map)
  hence length [p←drop (length vs1) vs2. fst p = y] = ?k ∧
    length [p←drop (length vs1') vs2'. fst p = y] = ?k
    using H by (simp add: in-flow-length)
  moreover have f y (length [p←vs1. fst p = y] + ?k + n) =
    f' y (length [p←vs1'. fst p = y] + ?k + n)
    using B and I and L by (fastforce simp: eq-streams-def ac-simps)
  ultimately show f y (length [p←vs2. fst p = y] + n) =
    f' y (length [p←vs2'. fst p = y] + n)
    by (subst F, subst G, simp)
qed

lemma tags-rhs:
  assumes
    A: S ⊆ {x. s1 = t1 (⊆ sources (flow cfs @ cs') vs1 s1 f x)}
      (is - ⊆ {-. - = - (⊆ sources (?cs @ -) - - - -)})
  assumes
    B: f = f' (⊆ vs1, vs1',
      ⋃ {tags (?cs @ cs') vs1 s1 f x | x. x ∈ S}) and
    C: (c1, s1, f, vs1, ws1) →*{cfs} (c2, s2, f, vs2, ws2) and
    D: (c1', t1, f', vs1', ws1') →* (c2', t2, f', vs2', ws2') and
    E: ok-flow-aux-1 c1 c2 c2' s1 t1 t2 f f' vs1 vs1' vs2 vs2' ws1' ws2' ?cs
  shows f = f' (⊆ vs2, vs2', ⋃ {tags cs' vs2 s2 f x | x. x ∈ S})
proof (subst eq-streams-def, clarify)
  fix x y n
  have F: vs2 = vs1 @ drop (length vs1) vs2
    using small-stepsl-steps [OF C] by (rule small-steps-in-flow)
  have G: vs2' = vs1' @ drop (length vs1') vs2'
    using D by (rule small-steps-in-flow)
  assume (y, n) ∈ tags cs' vs2 s2 f x
  moreover have s2 = run-flow ?cs vs1 s1 f
    using C by (rule small-stepsl-run-flow)
  moreover have H: vs2 = vs1 @ in-flow ?cs vs1 f
    using C by (rule small-stepsl-in-flow)
  ultimately have I: (y, length [c←?cs. c = IN y] + n)
    ∈ tags (?cs @ cs') vs1 s1 f x
    (is (-, ?k + -) ∈ -)
    by (simp add: tags-suffix)
  let ?m = Suc (Max {k. k ≤ length (?cs @ cs') ∧
    length [c←take k (?cs @ cs'). c = IN y] ≤ ?k + n})
  have J: y ∈ sources (drop ?m (?cs @ cs'))
    (vs1 @ in-flow (take ?m (?cs @ cs')) vs1 f)
```

(*run-flow* (*take ?m* (*?cs* @ *cs′*)) vs_1 s_1 *f*) *f x*
using *I* **by** (*auto dest*: *tags-sources*)
hence *sources* (*take ?m* (*?cs* @ *cs′*)) vs_1 s_1 *f y* $\subseteq$
sources (*take ?m* (*?cs* @ *cs′*) @ *drop ?m* (*?cs* @ *cs′*)) vs_1 s_1 *f x*
by (*rule sources-member*)
moreover have *K*: *length ?cs* $\leq$ *?m*
by (*rule le-SucI*, *rule Max-ge*, *simp-all*)
ultimately have
sources (*?cs* @ *take* (*?m* $-$ *length ?cs*) *cs′*) vs_1 s_1 *f y* $\subseteq$
sources (*?cs* @ *cs′*) vs_1 s_1 *f x*
by *simp*
moreover have
sources-aux (*?cs* @ *take* (*?m* $-$ *length ?cs*) *cs′*) vs_1 s_1 *f y* $\subseteq$
sources (*?cs* @ *take* (*?m* $-$ *length ?cs*) *cs′*) vs_1 s_1 *f y*
by (*rule sources-aux-sources*)
moreover have *sources-aux ?cs* vs_1 s_1 *f y* $\subseteq$
sources-aux (*?cs* @ *take* (*?m* $-$ *length ?cs*) *cs′*) vs_1 s_1 *f y*
by (*rule sources-aux-append*)
moreover assume *L*: $x \in S$
hence $s_1 = t_1$ ($\subseteq$ *sources* (*?cs* @ *cs′*) vs_1 s_1 *f x*)
using *A* **by** *blast*
ultimately have *M*: $s_1 = t_1$ ($\subseteq$ *sources-aux ?cs* vs_1 s_1 *f y*)
by *blast*
have *tags* (*take ?m* (*?cs* @ *cs′*)) vs_1 s_1 *f y* $\subseteq$
tags (*take ?m* (*?cs* @ *cs′*) @ *drop ?m* (*?cs* @ *cs′*)) vs_1 s_1 *f x*
using *J* **by** (*rule tags-member*)
hence *tags* (*?cs* @ *take* (*?m* $-$ *length ?cs*) *cs′*) vs_1 s_1 *f y* $\subseteq$
tags (*?cs* @ *cs′*) vs_1 s_1 *f x*
using *K* **by** *simp*
moreover have
tags-aux (*?cs* @ *take* (*?m* $-$ *length ?cs*) *cs′*) vs_1 s_1 *f y* $\subseteq$
tags (*?cs* @ *take* (*?m* $-$ *length ?cs*) *cs′*) vs_1 s_1 *f y*
by (*rule tags-aux-tags*)
moreover have *tags-aux ?cs* vs_1 s_1 *f y* $\subseteq$
tags-aux (*?cs* @ *take* (*?m* $-$ *length ?cs*) *cs′*) vs_1 s_1 *f y*
by (*rule tags-aux-append*)
ultimately have *tags-aux ?cs* vs_1 s_1 *f y* $\subseteq$
$\bigcup$ {*tags* (*?cs* @ *cs′*) vs_1 s_1 *f x* | *x*. $x \in S$}
using *L* **by** *blast*
with *B* **have** $f = f'$ ($\subseteq$ vs_1, vs_1', *tags-aux ?cs* vs_1 s_1 *f y*)
by (*rule eq-streams-subset*)
hence *map fst* [*p*←*drop* (*length* vs_1) vs_2. *fst p* = *y*] =
map fst [*p*←*drop* (*length* vs_1') vs_2'. *fst p* = *y*]
using *E* [*rule-format*, *of* {*y*}] **and** *M* **by** *simp*
hence *length* [*p*←*drop* (*length* vs_1) vs_2. *fst p* = *y*] =
length [*p*←*drop* (*length* vs_1') vs_2'. *fst p* = *y*]
by (*drule-tac arg-cong* [**where** *f* = *length*],
subst (*asm*) (*1 2*) *length-map*)
hence *length* [*p*←*drop* (*length* vs_1) vs_2. *fst p* = *y*] = *?k* $\wedge$

length [p←*drop* (*length* vs_1') vs_2'. *fst* p = y] = *?k*
using *H* **by** (*simp add*: *in-flow-length*)
moreover have *f y* (*length* [p←vs_1. *fst* p = y] + *?k* + n) =
f′ y (*length* [p←vs_1'. *fst* p = y] + *?k* + n)
using *B* **and** *I* **and** *L* **by** (*fastforce simp*: *eq-streams-def ac-simps*)
ultimately show *f y* (*length* [p←vs_2. *fst* p = y] + n) =
f′ y (*length* [p←vs_2'. *fst* p = y] + n)
by (*subst F*, *subst G*, *simp*)
qed

lemma *tags-out-rhs*:
assumes
A: *S* ⊆ {x. s_1 = t_1 (⊆ *sources-out* (*flow cfs* @ *cs′*) vs_1 s_1 *f x*)}
(**is** - ⊆ {-. - = - (⊆ *sources-out* (*?cs* @ -) - - - -)})
assumes
B: *f* = *f′* (⊆ vs_1, vs_1',
⋃ {*tags-out* (*?cs* @ *cs′*) vs_1 s_1 *f x* | x. x ∈ *S*}) **and**
C: (c_1, s_1, *f*, vs_1, ws_1) →∗{*cfs*} (c_2, s_2, *f*, vs_2, ws_2) **and**
D: (c_1', t_1, *f′*, vs_1', ws_1') →∗ (c_2', t_2, *f′*, vs_2', ws_2') **and**
E: *ok-flow-aux-1* c_1 c_2 c_2' s_1 t_1 t_2 *f f′* vs_1 vs_1' vs_2 vs_2' ws_1' ws_2' *?cs*
shows *f* = *f′* (⊆ vs_2, vs_2', ⋃ {*tags-out cs′* vs_2 s_2 *f x* | x. x ∈ *S*})
proof (*subst eq-streams-def*, *clarify*)
fix *x y n*
have *F*: vs_2 = vs_1 @ *drop* (*length* vs_1) vs_2
using *small-stepsl-steps* [*OF C*] **by** (*rule small-steps-in-flow*)
have *G*: vs_2' = vs_1' @ *drop* (*length* vs_1') vs_2'
using *D* **by** (*rule small-steps-in-flow*)
assume (y, n) ∈ *tags-out cs′* vs_2 s_2 *f x*
moreover have s_2 = *run-flow ?cs* vs_1 s_1 *f*
using *C* **by** (*rule small-stepsl-run-flow*)
moreover have *H*: vs_2 = vs_1 @ *in-flow ?cs* vs_1 *f*
using *C* **by** (*rule small-stepsl-in-flow*)
ultimately have *I*: (y, *length* [c←*?cs*. c = *IN* y] + n)
∈ *tags-out* (*?cs* @ *cs′*) vs_1 s_1 *f x*
(**is** (-, *?k* + -) ∈ -)
by (*simp add*: *tags-out-suffix*)
let *?m* = *Suc* (*Max* {k. k ≤ *length* (*?cs* @ *cs′*) ∧
length [c←*take* k (*?cs* @ *cs′*). c = *IN* y] ≤ *?k* + n})
have *J*: y ∈ *sources-out* (*drop ?m* (*?cs* @ *cs′*))
(vs_1 @ *in-flow* (*take ?m* (*?cs* @ *cs′*)) vs_1 *f*)
(*run-flow* (*take ?m* (*?cs* @ *cs′*)) vs_1 s_1 *f*) *f x*
using *I* **by** (*auto dest*: *tags-out-sources-out*)
hence *sources* (*take ?m* (*?cs* @ *cs′*)) vs_1 s_1 *f y* ⊆
sources-out (*take ?m* (*?cs* @ *cs′*) @ *drop ?m* (*?cs* @ *cs′*)) vs_1 s_1 *f x*
by (*rule sources-out-member*)
moreover have *K*: *length ?cs* ≤ *?m*
by (*rule le-SucI*, *rule Max-ge*, *simp-all*)
ultimately have
sources (*?cs* @ *take* (*?m* − *length ?cs*) *cs′*) vs_1 s_1 *f y* ⊆

```
    sources-out (?cs @ cs′) vs₁ s₁ f x
    by simp
  moreover have
   sources-aux (?cs @ take (?m − length ?cs) cs′) vs₁ s₁ f y ⊆
    sources (?cs @ take (?m − length ?cs) cs′) vs₁ s₁ f y
    by (rule sources-aux-sources)
  moreover have sources-aux ?cs vs₁ s₁ f y ⊆
    sources-aux (?cs @ take (?m − length ?cs) cs′) vs₁ s₁ f y
    by (rule sources-aux-append)
  moreover assume L: x ∈ S
  hence s₁ = t₁ (⊆ sources-out (?cs @ cs′) vs₁ s₁ f x)
    using A by blast
  ultimately have M: s₁ = t₁ (⊆ sources-aux ?cs vs₁ s₁ f y)
    by blast
  have tags (take ?m (?cs @ cs′)) vs₁ s₁ f y ⊆
    tags-out (take ?m (?cs @ cs′) @ drop ?m (?cs @ cs′)) vs₁ s₁ f x
    using J by (rule tags-out-member)
  hence tags (?cs @ take (?m − length ?cs) cs′) vs₁ s₁ f y ⊆
    tags-out (?cs @ cs′) vs₁ s₁ f x
    using K by simp
  moreover have
   tags-aux (?cs @ take (?m − length ?cs) cs′) vs₁ s₁ f y ⊆
    tags (?cs @ take (?m − length ?cs) cs′) vs₁ s₁ f y
    by (rule tags-aux-tags)
  moreover have tags-aux ?cs vs₁ s₁ f y ⊆
    tags-aux (?cs @ take (?m − length ?cs) cs′) vs₁ s₁ f y
    by (rule tags-aux-append)
  ultimately have tags-aux ?cs vs₁ s₁ f y ⊆
    ⋃ {tags-out (?cs @ cs′) vs₁ s₁ f x | x. x ∈ S}
    using L by blast
  with B have f = f′ (⊆ vs₁, vs₁′, tags-aux ?cs vs₁ s₁ f y)
    by (rule eq-streams-subset)
  hence map fst [p←drop (length vs₁) vs₂. fst p = y] =
    map fst [p←drop (length vs₁′) vs₂′. fst p = y]
    using E [rule-format, of {y}] and M by simp
  hence length [p←drop (length vs₁) vs₂. fst p = y] =
    length [p←drop (length vs₁′) vs₂′. fst p = y]
    by (drule-tac arg-cong [where f = length],
     subst (asm) (1 2) length-map)
  hence length [p←drop (length vs₁) vs₂. fst p = y] = ?k ∧
    length [p←drop (length vs₁′) vs₂′. fst p = y] = ?k
    using H by (simp add: in-flow-length)
  moreover have f y (length [p←vs₁. fst p = y] + ?k + n) =
    f′ y (length [p←vs₁′. fst p = y] + ?k + n)
    using B and I and L by (fastforce simp: eq-streams-def ac-simps)
  ultimately show f y (length [p←vs₂. fst p = y] + n) =
    f′ y (length [p←vs₂′. fst p = y] + n)
    by (subst F, subst G, simp)
qed
```

```
lemma ctyping2-term-seq:
  assumes
    A: ⋀B Y p. (U, v) ⊨ c₁ (⊆ A, X) = Some (B, Y) ⟹
      ∃(C, Z) ∈ U. ¬ C: Z ⇝ UNIV ⟹ ∃p′. (c₁, p) ⇒ p′ and
    B: ⋀q B Y B′ Y′ p. (U, v) ⊨ c₁ (⊆ A, X) = Some q ⟹ (B, Y) = q ⟹
      (U, v) ⊨ c₂ (⊆ B, Y) = Some (B′, Y′) ⟹
        ∃(C, Z) ∈ U. ¬ C: Z ⇝ UNIV ⟹ ∃p′. (c₂, p) ⇒ p′ and
    C: (U, v) ⊨ c₁;; c₂ (⊆ A, X) = Some (B′, Y′) and
    D: ∃(C, Z) ∈ U. ¬ C: Z ⇝ UNIV
  shows ∃p′. (c₁;; c₂, p) ⇒ p′
proof −
  obtain B and Y where
    E: (U, v) ⊨ c₁ (⊆ A, X) = Some (B, Y) and
    F: (U, v) ⊨ c₂ (⊆ B, Y) = Some (B′, Y′)
    using C by (auto split: option.split-asm)
  obtain p′ where (c₁, p) ⇒ p′
    using A [OF E D] by blast
  moreover obtain p′′ where (c₂, p′) ⇒ p′′
    using B [OF E - F D] by blast
  ultimately show ?thesis
    by blast
qed

lemma ctyping2-term-or:
  assumes
    A: ⋀B Y p. (U, v) ⊨ c₁ (⊆ A, X) = Some (B, Y) ⟹
      ∃(C, Z) ∈ U. ¬ C: Z ⇝ UNIV ⟹ ∃p′. (c₁, p) ⇒ p′ and
    B: (U, v) ⊨ c₁ OR c₂ (⊆ A, X) = Some (B′, Y′) and
    C: ∃(C, Z) ∈ U. ¬ C: Z ⇝ UNIV
  shows ∃p′. (c₁ OR c₂, p) ⇒ p′
proof −
  obtain B and Y where (U, v) ⊨ c₁ (⊆ A, X) = Some (B, Y)
    using B by (auto split: option.split-asm)
  thus ?thesis
    using A and C by blast
qed

lemma ctyping2-term-if:
  assumes
    A: ⋀U′ q B₁ B₂ B Y p.
      (U′, q) = (insert (Univ? A X, bvars b) U, ⊨ b (⊆ A, X)) ⟹
        (B₁, B₂) = q ⟹ (U′, v) ⊨ c₁ (⊆ B₁, X) = Some (B, Y) ⟹
          ∃(C, Z) ∈ U′. ¬ C: Z ⇝ UNIV ⟹ ∃p′. (c₁, p) ⇒ p′ and
    B: ⋀U′ q B₁ B₂ B Y p.
      (U′, q) = (insert (Univ? A X, bvars b) U, ⊨ b (⊆ A, X)) ⟹
        (B₁, B₂) = q ⟹ (U′, v) ⊨ c₂ (⊆ B₂, X) = Some (B, Y) ⟹
          ∃(C, Z) ∈ U′. ¬ C: Z ⇝ UNIV ⟹ ∃p′. (c₂, p) ⇒ p′ and
```

```
    C: (U, v) ⊨ IF b THEN c₁ ELSE c₂ (⊆ A, X) = Some (B, Y) and
    D: ∃(C, Z) ∈ U. ¬ C: Z ⇝ UNIV
  shows ∃p'. (IF b THEN c₁ ELSE c₂, p) ⇒ p'
proof –
  let ?U' = insert (Univ? A X, bvars b) U
  obtain B₁ and B₁' and Y₁ and B₂ and B₂' and Y₂ where
    E: ⊨ b (⊆ A, X) = (B₁, B₂) and
    F: (?U', v) ⊨ c₁ (⊆ B₁, X) = Some (B₁', Y₁) and
    G: (?U', v) ⊨ c₂ (⊆ B₂, X) = Some (B₂', Y₂)
    using C by (auto split: option.split-asm prod.split-asm)
  obtain s and q where p = (s, q)
    by (cases p)
  moreover {
    assume bval b s
    moreover obtain p' where (c₁, s, q) ⇒ p'
      using A [OF - - F, of - B₂ (s, q)] and D and E by auto
    ultimately have ∃p'. (IF b THEN c₁ ELSE c₂, s, q) ⇒ p'
      by blast
  }
  moreover {
    assume ¬ bval b s
    moreover obtain p' where (c₂, s, q) ⇒ p'
      using B [OF - - G, of - B₁ (s, q)] and D and E by auto
    ultimately have ∃p'. (IF b THEN c₁ ELSE c₂, s, q) ⇒ p'
      by blast
  }
  ultimately show ?thesis
    by blast
qed

lemma ctyping2-term:
⟦(U, v) ⊨ c (⊆ A, X) = Some (B, Y); ∃(C, Z) ∈ U. ¬ C: Z ⇝ UNIV⟧ ⟹
   ∃p'. (c, p) ⇒ p'
proof (induction (U, v) c A X arbitrary: B Y U v p rule: ctyping2.induct,
 blast)
  fix A X B Y U v c₁ c₂ p
  show
   ⟦⋀B Y p. (U, v) ⊨ c₁ (⊆ A, X) = Some (B, Y) ⟹
      ∃(C, Z) ∈ U. ¬ C: Z ⇝ UNIV ⟹ ∃p'. (c₁, p) ⇒ p';
    ⋀q B Y B' Y' p. (U, v) ⊨ c₁ (⊆ A, X) = Some q ⟹ (B, Y) = q ⟹
      (U, v) ⊨ c₂ (⊆ B, Y) = Some (B', Y') ⟹
        ∃(C, Z) ∈ U. ¬ C: Z ⇝ UNIV ⟹ ∃p'. (c₂, p) ⇒ p';
    (U, v) ⊨ c₁;; c₂ (⊆ A, X) = Some (B, Y);
    ∃(C, Z) ∈ U. ¬ C: Z ⇝ UNIV⟧ ⟹
      ∃p'. (c₁;; c₂, p) ⇒ p'
    by (rule ctyping2-term-seq)
next
  fix A X B Y U v c₁ c₂ p
  show
```

```
  ⟦⋀B Y p. (U, v) ⊨ c1 (⊆ A, X) = Some (B, Y) ⟹
    ∃(C, Z) ∈ U. ¬ C: Z ⇝ UNIV ⟹ ∃p'. (c1, p) ⇒ p';
   (U, v) ⊨ c1 OR c2 (⊆ A, X) = Some (B, Y);
   ∃(C, Z) ∈ U. ¬ C: Z ⇝ UNIV⟧ ⟹
    ∃p'. (c1 OR c2, p) ⇒ p'
   by (rule ctyping2-term-or)
next
  fix A X B Y U v b c1 c2 p
  show
   ⟦⋀U' q B1 B2 B Y p.
     (U', q) = (insert (Univ? A X, bvars b) U, ⊨ b (⊆ A, X)) ⟹
       (B1, B2) = q ⟹ (U', v) ⊨ c1 (⊆ B1, X) = Some (B, Y) ⟹
         ∃(C, Z) ∈ U'. ¬ C: Z ⇝ UNIV ⟹ ∃p'. (c1, p) ⇒ p';
    ⋀U' q B1 B2 B Y p.
     (U', q) = (insert (Univ? A X, bvars b) U, ⊨ b (⊆ A, X)) ⟹
       (B1, B2) = q ⟹ (U', v) ⊨ c2 (⊆ B2, X) = Some (B, Y) ⟹
         ∃(C, Z) ∈ U'. ¬ C: Z ⇝ UNIV ⟹ ∃p'. (c2, p) ⇒ p';
    (U, v) ⊨ IF b THEN c1 ELSE c2 (⊆ A, X) = Some (B, Y);
    ∃(C, Z) ∈ U. ¬ C: Z ⇝ UNIV⟧ ⟹
     ∃p'. (IF b THEN c1 ELSE c2, p) ⇒ p'
   by (rule ctyping2-term-if)
qed (fastforce split: if-split-asm prod.split-asm)+


lemma ctyping2-confine-seq:
  assumes
    A: ⋀s' f' vs' ws' A B X Y U v. p = (s', f', vs', ws') ⟹
     (U, v) ⊨ c1 (⊆ A, X) = Some (B, Y) ⟹ ∃(C, Z) ∈ U. C: Z ⇝| S ⟹
       s = s' (⊆ S) ∧
       [p←drop (length vs) vs'. fst p ∈ S] = [] ∧
       [p←drop (length ws) ws'. fst p ∈ S] = []
     (is ⋀s' - vs' ws' - - - - - -. - ⟹ - ⟹ - ⟹
       ?P s s' vs vs' ws ws')
  assumes
    B: ⋀s' f' vs' ws' A B X Y U v. p = (s', f', vs', ws') ⟹
     (U, v) ⊨ c2 (⊆ A, X) = Some (B, Y) ⟹ ∃(C, Z) ∈ U. C: Z ⇝| S ⟹
       ?P s' s'' vs' vs'' ws' ws'' and
    C: (c1, s, f, vs, ws) ⇒ p and
    D: (c2, p) ⇒ (s'', f'', vs'', ws'') and
    E: (U, v) ⊨ c1;; c2 (⊆ A, X) = Some (B', Y') and
    F: ∃(C, Z) ∈ U. C: Z ⇝| S
  shows ?P s s'' vs vs'' ws ws''
proof –
  obtain s' and f' and vs' and ws' where G: p = (s', f', vs', ws')
   by (cases p)
  have H: (c1, s, f, vs, ws) →* (SKIP, s', f', vs', ws')
   using C and G by (simp add: big-iff-small)
  have I: (c2, s', f', vs', ws') →* (SKIP, s'', f'', vs'', ws'')
   using D and G by (simp add: big-iff-small)
```

```
  have J: vs′ = vs @ drop (length vs) vs′
    using H by (rule small-steps-in-flow)
  have vs′′ = vs′ @ drop (length vs′) vs′′
    using I by (rule small-steps-in-flow)
  hence K: vs′′ = vs @ drop (length vs) vs′ @ drop (length vs′) vs′′
    by (subst (asm) J, simp)
  have L: ws′ = ws @ drop (length ws) ws′
    using H by (rule small-steps-out-flow)
  have ws′′ = ws′ @ drop (length ws′) ws′′
    using I by (rule small-steps-out-flow)
  hence M: ws′′ = ws @ drop (length ws) ws′ @ drop (length ws′) ws′′
    by (subst (asm) L, simp)
  obtain B and Y where
    N: (U, v) ⊨ c₁ (⊆ A, X) = Some (B, Y) and
    O: (U, v) ⊨ c₂ (⊆ B, Y) = Some (B′, Y′)
    using E by (auto split: option.split-asm)
  from A [OF G N F] and B [OF G O F] show ?thesis
    by (subst K, subst M, simp)
qed

lemma ctyping2-confine-or-lhs:
  assumes
    A: ⋀A B X Y U v. (U, v) ⊨ c₁ (⊆ A, X) = Some (B, Y) ⟹
      ∃(C, Z) ∈ U. C: Z ⇝| S ⟹
        s = s′ (⊆ S) ∧
        [p←drop (length vs) vs′. fst p ∈ S] = [] ∧
        [p←drop (length ws) ws′. fst p ∈ S] = []
      (is ⋀- - - - - -. - ⟹ - ⟹ ?P)
  assumes
    B: (U, v) ⊨ c₁ OR c₂ (⊆ A, X) = Some (B′, Y′) and
    C: ∃(C, Z) ∈ U. C: Z ⇝| S
  shows ?P
proof −
  obtain B and Y where (U, v) ⊨ c₁ (⊆ A, X) = Some (B, Y)
    using B by (auto split: option.split-asm)
  with A and C show ?thesis
    by simp
qed

lemma ctyping2-confine-or-rhs:
  assumes
    A: ⋀A B X Y U v. (U, v) ⊨ c₂ (⊆ A, X) = Some (B, Y) ⟹
      ∃(C, Z) ∈ U. C: Z ⇝| S ⟹
        s = s′ (⊆ S) ∧
        [p←drop (length vs) vs′. fst p ∈ S] = [] ∧
        [p←drop (length ws) ws′. fst p ∈ S] = []
      (is ⋀- - - - - -. - ⟹ - ⟹ ?P)
  assumes
    B: (U, v) ⊨ c₁ OR c₂ (⊆ A, X) = Some (B′, Y′) and
```

C: $\exists (C,\ Z) \in U.\ C$: $Z \leadsto|\ S$
shows *?P*
proof –
obtain B **and** Y **where** $(U,\ v) \models c_2\ (\subseteq A,\ X) = $ *Some* $(B,\ Y)$
using B **by** (*auto split*: *option.split-asm*)
with A **and** C **show** *?thesis*
by *simp*
qed

lemma *ctyping2-confine-if-true*:
assumes
A: $\bigwedge A\ B\ X\ Y\ U\ v.\ (U,\ v) \models c_1\ (\subseteq A,\ X) = $ *Some* $(B,\ Y) \Longrightarrow$
$\exists (C,\ Z) \in U.\ C$: $Z \leadsto|\ S \Longrightarrow$
$s = s'\ (\subseteq S) \wedge$
$[p \leftarrow$ *drop* (*length vs*) *vs*$'$. *fst* $p \in S] = [] \wedge$
$[p \leftarrow$ *drop* (*length ws*) *ws*$'$. *fst* $p \in S] = []$
(**is** $\bigwedge$- - - - - -. - $\Longrightarrow$ - $\Longrightarrow$ *?P*)
assumes
B: $(U,\ v) \models$ *IF b THEN* c_1 *ELSE* $c_2\ (\subseteq A,\ X) = $ *Some* $(B,\ Y)$ **and**
C: $\exists (C,\ Z) \in U.\ C$: $Z \leadsto|\ S$
shows *?P*
proof –
obtain B_1 **and** $B_1{}'$ **and** Y_1 **where**
(*insert* (*Univ? A X*, *bvars b*) U, v) $\models c_1\ (\subseteq B_1,\ X) = $ *Some* $(B_1{}',\ Y_1)$
using B **by** (*auto split*: *option.split-asm prod.split-asm*)
with A **and** C **show** *?thesis*
by *simp*
qed

lemma *ctyping2-confine-if-false*:
assumes
A: $\bigwedge A\ B\ X\ Y\ U\ v.\ (U,\ v) \models c_2\ (\subseteq A,\ X) = $ *Some* $(B,\ Y) \Longrightarrow$
$\exists (C,\ Z) \in U.\ C$: $Z \leadsto|\ S \Longrightarrow$
$s = s'\ (\subseteq S) \wedge$
$[p \leftarrow$ *drop* (*length vs*) *vs*$'$. *fst* $p \in S] = [] \wedge$
$[p \leftarrow$ *drop* (*length ws*) *ws*$'$. *fst* $p \in S] = []$
(**is** $\bigwedge$- - - - - -. - $\Longrightarrow$ - $\Longrightarrow$ *?P*)
assumes
B: $(U,\ v) \models$ *IF b THEN* c_1 *ELSE* $c_2\ (\subseteq A,\ X) = $ *Some* $(B,\ Y)$ **and**
C: $\exists (C,\ Z) \in U.\ C$: $Z \leadsto|\ S$
shows *?P*
proof –
obtain B_2 **and** $B_2{}'$ **and** Y_2 **where**
(*insert* (*Univ? A X*, *bvars b*) U, v) $\models c_2\ (\subseteq B_2,\ X) = $ *Some* $(B_2{}',\ Y_2)$
using B **by** (*auto split*: *option.split-asm prod.split-asm*)
with A **and** C **show** *?thesis*
by *simp*
qed

```
lemma ctyping2-confine:
 ⟦(c, s, f, vs, ws) ⇒ (s', f', vs', ws');
    (U, v) ⊨ c (⊆ A, X) = Some (B, Y); ∃(C, Z) ∈ U. C: Z ↝| S⟧ ⟹
  s = s' (⊆ S) ∧
  [p←drop (length vs) vs'. fst p ∈ S] = [] ∧
  [p←drop (length ws) ws'. fst p ∈ S] = []
  (is ⟦-; -; -⟧ ⟹ ?P s s' vs vs' ws ws')
proof (induction (c, s, f, vs, ws) (s', f', vs', ws') arbitrary:
 c s f vs ws s' f' vs' ws' A B X Y U v rule: big-step.induct)
  fix A B X Y U v c₁ c₂ p s f vs ws s' f' vs' ws'
  show
   ⟦⋀s' f' vs' ws' A B X Y U v. p = (s', f', vs', ws') ⟹
     (U, v) ⊨ c₁ (⊆ A, X) = Some (B, Y) ⟹
       ∃(C, Z) ∈ U. C: Z ↝| S ⟹ ?P s s' vs vs' ws ws';
    ⋀s f vs ws A B X Y U v. p = (s, f, vs, ws) ⟹
     (U, v) ⊨ c₂ (⊆ A, X) = Some (B, Y) ⟹
       ∃(C, Z) ∈ U. C: Z ↝| S ⟹ ?P s s' vs vs' ws ws';
    (c₁, s, f, vs, ws) ⇒ p;
    (c₂, p) ⇒ (s', f', vs', ws');
    (U, v) ⊨ c₁;; c₂ (⊆ A, X) = Some (B, Y);
    ∃(C, Z) ∈ U. C: Z ↝| S⟧ ⟹
      ?P s s' vs vs' ws ws'
    by (rule ctyping2-confine-seq)
next
  fix A B X Y U v c₁ c₂ s vs ws s' vs' ws'
  show
   ⟦⋀A B X Y U v. (U, v) ⊨ c₁ (⊆ A, X) = Some (B, Y) ⟹
      ∃(C, Z) ∈ U. C: Z ↝| S ⟹ ?P s s' vs vs' ws ws';
    (U, v) ⊨ c₁ OR c₂ (⊆ A, X) = Some (B, Y);
    ∃(C, Z) ∈ U. C: Z ↝| S⟧ ⟹
      ?P s s' vs vs' ws ws'
    by (rule ctyping2-confine-or-lhs)
next
  fix A B X Y U v c₁ c₂ s vs ws s' vs' ws'
  show
   ⟦⋀A B X Y U v. (U, v) ⊨ c₂ (⊆ A, X) = Some (B, Y) ⟹
      ∃(C, Z) ∈ U. C: Z ↝| S ⟹ ?P s s' vs vs' ws ws';
    (U, v) ⊨ c₁ OR c₂ (⊆ A, X) = Some (B, Y);
    ∃(C, Z) ∈ U. C: Z ↝| S⟧ ⟹
      ?P s s' vs vs' ws ws'
    by (rule ctyping2-confine-or-rhs)
next
  fix A B X Y U v b c₁ c₂ s vs ws s' vs' ws'
  show
   ⟦⋀A B X Y U v. (U, v) ⊨ c₁ (⊆ A, X) = Some (B, Y) ⟹
      ∃(C, Z) ∈ U. C: Z ↝| S ⟹ ?P s s' vs vs' ws ws';
    (U, v) ⊨ IF b THEN c₁ ELSE c₂ (⊆ A, X) = Some (B, Y);
    ∃(C, Z) ∈ U. C: Z ↝| S⟧ ⟹
      ?P s s' vs vs' ws ws'
```

```
    by (rule ctyping2-confine-if-true)
next
  fix A B X Y U v b c₁ c₂ s vs ws s' vs' ws'
  show
    ⟦⋀A B X Y U v. (U, v) ⊨ c₂ (⊆ A, X) = Some (B, Y) ⟹
      ∃(C, Z) ∈ U. C: Z ⇝| S ⟹ ?P s s' vs vs' ws ws';
    (U, v) ⊨ IF b THEN c₁ ELSE c₂ (⊆ A, X) = Some (B, Y);
    ∃(C, Z) ∈ U. C: Z ⇝| S⟧ ⟹
      ?P s s' vs vs' ws ws'
    by (rule ctyping2-confine-if-false)
qed (force split: if-split-asm prod.split-asm)+


lemma eq-states-assign:
  assumes
    A: S ⊆ {y. s = t (⊆ sources [x ::= a] vs s f y)} and
    B: x ∈ S and
    C: s ∈ Univ A (⊆ state ∩ X) and
    D: Univ? A X: avars a ⇝ {x}
  shows s = t (⊆ avars a)
proof –
  obtain r where E: r ∈ A and F: s = r (⊆ state ∩ X)
    using C by blast
  have avars a ⊆ {y. s: dom y ⇝ dom x}
  proof (cases state ⊆ X)
    case True
    with F have interf s = interf r
      by (blast intro: interf-state)
    with D and E show ?thesis
      by (auto simp: univ-states-if-def split: if-split-asm)
  next
    case False
    with D and E show ?thesis
      by (auto simp: univ-states-if-def split: if-split-asm)
  qed
  moreover have s = t (⊆ sources [x ::= a] vs s f x)
    using A and B by blast
  hence s = t (⊆ {y. s: dom y ⇝ dom x ∧ y ∈ avars a})
    by (subst (asm) append-Nil [symmetric],
     simp only: sources.simps, auto)
  ultimately show ?thesis
    by blast
qed

lemma eq-states-while:
  assumes
    A: S ⊆ {x. s = t (⊆ sources-aux (⟨bvars b⟩ # cs) vs s f x)} and
    B: S ≠ {} and
    C: s ∈ Univ A (⊆ state ∩ X) ∪ Univ C (⊆ state ∩ Y) and
```

```
    D: Univ? A X ∪ Univ? C Y: bvars b ⇝ UNIV
  shows s = t (⊆ bvars b)
proof –
  from C have {s}: bvars b ⇝ UNIV
  proof
    assume s ∈ Univ A (⊆ state ∩ X)
    then obtain r where E: r ∈ A and F: s = r (⊆ state ∩ X)
      by blast
    show ?thesis
    proof (cases state ⊆ X)
      case True
      with F have interf s = interf r
        by (blast intro: interf-state)
      with D and E show ?thesis
        by (auto simp: univ-states-if-def split: if-split-asm)
    qed (insert D E, auto simp: univ-states-if-def split: if-split-asm)
  next
    assume s ∈ Univ C (⊆ state ∩ Y)
    then obtain r where E: r ∈ C and F: s = r (⊆ state ∩ Y)
      by blast
    show ?thesis
    proof (cases state ⊆ Y)
      case True
      with F have interf s = interf r
        by (blast intro: interf-state)
      with D and E show ?thesis
        by (auto simp: univ-states-if-def split: if-split-asm)
    qed (insert D E, auto simp: univ-states-if-def split: if-split-asm)
  qed
  hence ∀ x. bvars b ⊆ sources-aux (⟨bvars b⟩ # cs) vs s f x
    by (blast intro!: sources-aux-observe-hd)
  thus ?thesis
    using A and B by blast
qed

lemma univ-states-while:
  assumes
    A: (c, s, p) ⇒ (s', p') and
    B: ⊨ b (⊆ A, X) = (B₁, B₂) and
    C: ⊢ c (⊆ B₁, X) = (C, Y) and
    D: ⊨ b (⊆ C, Y) = (B₁', B₂') and
    E: ({}, False) ⊨ c (⊆ B₁, X) = Some (D, Z) and
    F: ({}, False) ⊨ c (⊆ B₁', Y) = Some (D', Z') and
    G: bval b s
  shows s ∈ Univ A (⊆ state ∩ X) ∪ Univ C (⊆ state ∩ Y) ⟹
    s' ∈ Univ A (⊆ state ∩ X) ∪ Univ C (⊆ state ∩ Y)
proof (erule UnE)
  assume H: s ∈ Univ A (⊆ state ∩ X)
  have s ∈ Univ B₁ (⊆ state ∩ X)
```

```
    using G by (insert btyping2-approx [OF B H], simp)
  with A and E have s' ∈ Univ D (⊆ state ∩ Z)
    by (rule ctyping2-approx)
  moreover have D ⊆ C ∧ Y ⊆ Z
    using C and E by (rule ctyping1-ctyping2)
  ultimately show ?thesis
    by blast
next
  assume H: s ∈ Univ C (⊆ state ∩ Y)
  have s ∈ Univ B₁' (⊆ state ∩ Y)
    using G by (insert btyping2-approx [OF D H], simp)
  with A and F have s' ∈ Univ D' (⊆ state ∩ Z')
    by (rule ctyping2-approx)
  moreover obtain C' and Y' where I: ⊢ c (⊆ B₁', Y) = (C', Y')
    by (cases ⊢ c (⊆ B₁', Y), simp)
  hence D' ⊆ C' ∧ Y' ⊆ Z'
    using F by (rule ctyping1-ctyping2)
  ultimately have s' ∈ Univ C' (⊆ state ∩ Y')
    by blast
  moreover have J: ⊢ c (⊆ C, Y) = (C, Y)
    using C by (rule ctyping1-idem)
  have B₁' ⊆ C
    using D by (blast dest: btyping2-un-eq)
  with J and I have C' ⊆ C ∧ Y ⊆ Y'
    by (rule ctyping1-mono, simp)
  ultimately show ?thesis
    by blast
qed

end

end
```

6 Sufficiency of well-typedness for information flow correctness: main theorem

```
theory Correctness-Theorem
  imports Correctness-Lemmas
begin
```

The purpose of this section is to prove that type system *ctyping2* is correct in that it guarantees that well-typed programs satisfy the information flow correctness criterion expressed by predicate *correct*, namely that if the type system outputs a value other than *None* (that is, a *pass* verdict) when it is input program *c*, *state set A*, and *vname set X*, then *correct c A X* (theorem *ctyping2-correct*).

This proof makes use of the lemma *ctyping2-approx* proven in a previous section.

6.1 Local context proofs

context *noninterf*
begin

lemma *ctyping2-correct-aux-skip* [*elim*!]:
$[\![(\mathit{SKIP},\ s,\ f,\ vs_0,\ ws_0) \rightarrow*\{cfs_1\}\ (c_1,\ s_1,\ f,\ vs_1,\ ws_1);$
$(c_1,\ s_1,\ f,\ vs_1,\ ws_1) \rightarrow*\{cfs_2\}\ (c_2,\ s_2,\ f,\ vs_2,\ ws_2)]\!] \Longrightarrow$
$\mathit{ok\text{-}flow\text{-}aux}\ U\ c_1\ c_2\ s_1\ s_2\ f\ vs_1\ vs_2\ ws_1\ ws_2\ (\mathit{flow}\ cfs_2)$
by (*fastforce dest*: *small-stepsl-skip*)

lemma *ctyping2-correct-aux-assign*:
assumes
A: $(U,\ v) \models x ::= a\ (\subseteq A,\ X) = \mathit{Some}\ (C,\ Y)$ **and**
B: $s \in \mathit{Univ}\ A\ (\subseteq \mathit{state} \cap X)$ **and**
C: $(x ::= a,\ s,\ f,\ vs_0,\ ws_0) \rightarrow*\{cfs_1\}\ (c_1,\ s_1,\ f,\ vs_1,\ ws_1)$ **and**
D: $(c_1,\ s_1,\ f,\ vs_1,\ ws_1) \rightarrow*\{cfs_2\}\ (c_2,\ s_2,\ f,\ vs_2,\ ws_2)$
shows $\mathit{ok\text{-}flow\text{-}aux}\ U\ c_1\ c_2\ s_1\ s_2\ f\ vs_1\ vs_2\ ws_1\ ws_2\ (\mathit{flow}\ cfs_2)$
proof –
from A **have** E: $\forall (B,\ Y) \in \mathit{insert}\ (\mathit{Univ?}\ A\ X,\ \mathit{avars}\ a)\ U.\ B{:}\ Y \leadsto \{x\}$
by (*simp split*: *if-split-asm*)
have
$(c_1,\ s_1,\ f,\ vs_1,\ ws_1) = (x ::= a,\ s,\ f,\ vs_0,\ ws_0)\ \vee$
$(c_1,\ s_1,\ f,\ vs_1,\ ws_1) = (\mathit{SKIP},\ s(x := \mathit{aval}\ a\ s),\ f,\ vs_0,\ ws_0)$
(**is** *?P* ∨ *?Q*)
using C **by** (*blast dest*: *small-stepsl-assign*)
thus *?thesis*
proof
assume *?P*
hence $(x ::= a,\ s,\ f,\ vs_0,\ ws_0) \rightarrow*\{cfs_2\}\ (c_2,\ s_2,\ f,\ vs_2,\ ws_2)$
using D **by** *simp*
hence
$(c_2,\ s_2,\ f,\ vs_2,\ ws_2) = (x ::= a,\ s,\ f,\ vs_0,\ ws_0)\ \wedge$
$\mathit{flow}\ cfs_2 = []\ \vee$
$(c_2,\ s_2,\ f,\ vs_2,\ ws_2) = (\mathit{SKIP},\ s(x := \mathit{aval}\ a\ s),\ f,\ vs_0,\ ws_0)\ \wedge$
$\mathit{flow}\ cfs_2 = [x ::= a]$
(**is** *?P'* ∨ -)
by (*rule small-stepsl-assign*)
thus *?thesis*
proof (*rule disjE*, *erule-tac* [*2*] *conjE*)
assume *?P'*
with ‹*?P*› **show** *?thesis*
by *fastforce*
next
assume

```
      F: (c_2, s_2, f, vs_2, ws_2) = (SKIP, s(x := aval a s), f, vs_0, ws_0) and
      G: flow cfs_2 = [x ::= a]
        (is ?cs = -)
    show ?thesis
    proof (rule conjI, clarify)
      fix t_1 f' vs_1' ws_1'
      let ?t_2 = t_1(x := aval a t_1)
      show ∃ c_2' t_2 vs_2' ws_2'.
        ok-flow-aux-1 c_1 c_2 c_2' s_1 t_1 t_2 f f'
          vs_1 vs_1' vs_2 vs_2' ws_1' ws_2' ?cs ∧
        ok-flow-aux-2 s_1 s_2 t_1 t_2 f f' vs_1 vs_1' ?cs ∧
        ok-flow-aux-3 s_1 t_1 f f' vs_1 vs_1' ws_1 ws_1' ws_2 ws_2' ?cs
      proof (rule exI [of - SKIP], rule exI [of - ?t_2],
       rule exI [of - vs_1'], rule exI [of - ws_1'])
        {
          fix S
          assume S ⊆ {y. s = t_1 (⊆ sources [x ::= a] vs_0 s f y)} and
           x ∈ S
          hence s = t_1 (⊆ avars a)
            using B by (rule eq-states-assign, insert E, simp)
          hence aval a s = aval a t_1
            by (rule avars-aval)
        }
        moreover {
          fix S y
          assume S ⊆ {y. s = t_1 (⊆ sources [x ::= a] vs_0 s f y)} and
           y ∈ S
          hence s = t_1 (⊆ sources [x ::= a] vs_0 s f y)
            by blast
          moreover assume y ≠ x
          ultimately have s y = t_1 y
            by (subst (asm) append-Nil [symmetric],
             simp only: sources.simps, simp)
        }
        ultimately show
         ok-flow-aux-1 c_1 c_2 SKIP s_1 t_1 ?t_2 f f'
           vs_1 vs_1' vs_2 vs_1' ws_1' ws_1' ?cs ∧
          ok-flow-aux-2 s_1 s_2 t_1 ?t_2 f f' vs_1 vs_1' ?cs ∧
          ok-flow-aux-3 s_1 t_1 f f' vs_1 vs_1' ws_1 ws_1' ws_2 ws_1' ?cs
          using F and G and ‹?P› by auto
      qed
    qed (insert E G, fastforce)
  qed
next
  assume ?Q
  moreover from this have
   (c_2, s_2, f, vs_2, ws_2) = (SKIP, s_1, f, vs_1, ws_1) ∧ flow cfs_2 = []
    using D by (blast intro!: small-stepsl-skip)
  ultimately show ?thesis
```

```
      by fastforce
  qed
qed

lemma ctyping2-correct-aux-input:
  assumes
    A: (U, v) ⊨ IN x (⊆ A, X) = Some (C, Y) and
    B: (IN x, s, f, $vs_0$, $ws_0$) →*{$cfs_1$} ($c_1$, $s_1$, f, $vs_1$, $ws_1$) and
    C: ($c_1$, $s_1$, f, $vs_1$, $ws_1$) →*{$cfs_2$} ($c_2$, $s_2$, f, $vs_2$, $ws_2$)
  shows ok-flow-aux U $c_1$ $c_2$ $s_1$ $s_2$ f $vs_1$ $vs_2$ $ws_1$ $ws_2$ (flow $cfs_2$)
proof −
  from A have D: ∀ (B, Y) ∈ U. B: Y ⇝ {x}
    by (simp split: if-split-asm)
  let ?n = length [p←$vs_0$. fst p = x]
  have
   ($c_1$, $s_1$, f, $vs_1$, $ws_1$) = (IN x, s, f, $vs_0$, $ws_0$) ∨
    ($c_1$, $s_1$, f, $vs_1$, $ws_1$) =
      (SKIP, s(x := f x ?n), f, $vs_0$ @ [(x, f x ?n)], $ws_0$)
    (is ?P ∨ ?Q)
    using B by (auto dest: small-stepsl-input simp: Let-def)
  thus ?thesis
  proof
    assume ?P
    hence (IN x, s, f, $vs_0$, $ws_0$) →*{$cfs_2$} ($c_2$, $s_2$, f, $vs_2$, $ws_2$)
      using C by simp
    hence
     ($c_2$, $s_2$, f, $vs_2$, $ws_2$) = (IN x, s, f, $vs_0$, $ws_0$) ∧
        flow $cfs_2$ = [] ∨
      ($c_2$, $s_2$, f, $vs_2$, $ws_2$) =
        (SKIP, s(x := f x ?n), f, $vs_0$ @ [(x, f x ?n)], $ws_0$) ∧
        flow $cfs_2$ = [IN x]
      (is ?P' ∨ -)
      by (auto dest: small-stepsl-input simp: Let-def)
    thus ?thesis
    proof (rule disjE, erule-tac [2] conjE)
      assume ?P'
      with ‹?P› show ?thesis
        by fastforce
    next
      assume
        E: ($c_2$, $s_2$, f, $vs_2$, $ws_2$) =
          (SKIP, s(x := f x ?n), f, $vs_0$ @ [(x, f x ?n)], $ws_0$) and
        F: flow $cfs_2$ = [IN x]
          (is ?cs = -)
      show ?thesis
      proof (rule conjI, clarify)
        fix $t_1$ f' $vs_1$' $ws_1$'
        let ?n' = length [p←$vs_1$' :: inputs. fst p = x]
        let ?$t_2$ = $t_1$(x := f' x ?n') :: state and
```

```
        ?vs2' = vs1' @ [(x, f' x ?n')]
      show ∃ c2' t2 vs2' ws2'.
        ok-flow-aux-1 c1 c2 c2' s1 t1 t2 f f'
          vs1 vs1' vs2 vs2' ws1' ws2' ?cs ∧
        ok-flow-aux-2 s1 s2 t1 t2 f f' vs1 vs1' ?cs ∧
        ok-flow-aux-3 s1 t1 f f' vs1 vs1' ws1 ws1' ws2 ws2' ?cs
      proof (rule exI [of - SKIP], rule exI [of - ?t2],
       rule exI [of - ?vs2'], rule exI [of - ws1'])
        {
          fix S
          assume f = f' (⊆ vs0, vs1',
            ⋃ {tags [IN x] vs0 s f y | y. y ∈ S})
              (is - = - (⊆ -, -, ?T))
          moreover assume x ∈ S
          hence tags [IN x] vs0 s f x ⊆ ?T
            by blast
          ultimately have f = f' (⊆ vs0, vs1', tags [IN x] vs0 s f x)
            by (rule eq-streams-subset)
          moreover have tags [IN x] vs0 s f x = {(x, 0)}
            by (subst append-Nil [symmetric],
             simp only: tags.simps, simp)
          ultimately have f x (length [p←vs0. fst p = x]) =
            f' x (length [p←vs1'. fst p = x])
            by (simp add: eq-streams-def)
        }
        moreover
        {
          fix S y
          assume S ⊆ {y. s = t1 (⊆ sources [IN x] vs0 s f y)} and
           y ∈ S
          hence s = t1 (⊆ sources [IN x] vs0 s f y)
            by blast
          moreover assume y ≠ x
          ultimately have s y = t1 y
            by (subst (asm) append-Nil [symmetric],
             simp only: sources.simps, simp)
        }
        ultimately show
         ok-flow-aux-1 c1 c2 SKIP s1 t1 ?t2 f f'
            vs1 vs1' vs2 ?vs2' ws1' ws1' ?cs ∧
          ok-flow-aux-2 s1 s2 t1 ?t2 f f' vs1 vs1' ?cs ∧
          ok-flow-aux-3 s1 t1 f f' vs1 vs1' ws1 ws1' ws2 ws1' ?cs
          using E and F and ‹?P› by auto
      qed
    qed (insert D F, fastforce)
  qed
next
  assume ?Q
  moreover from this have
```

```
      (c2, s2, f, vs2, ws2) = (SKIP, s1, f, vs1, ws1) ∧ flow cfs2 = []
        using C by (blast intro!: small-stepsl-skip)
      ultimately show ?thesis
        by fastforce
    qed
qed

lemma ctyping2-correct-aux-output:
  assumes
    A: (U, v) |= OUT x (⊆ A, X) = Some (B, Y) and
    B: (OUT x, s, f, vs0, ws0) →*{cfs1} (c1, s1, f, vs1, ws1) and
    C: (c1, s1, f, vs1, ws1) →*{cfs2} (c2, s2, f, vs2, ws2)
  shows ok-flow-aux U c1 c2 s1 s2 f vs1 vs2 ws1 ws2 (flow cfs2)
proof –
  from A have D: ∀ (B, Y) ∈ U. B: Y ⇝ {x}
    by (simp split: if-split-asm)
  have
   (c1, s1, f, vs1, ws1) = (OUT x, s, f, vs0, ws0) ∨
    (c1, s1, f, vs1, ws1) = (SKIP, s, f, vs0, ws0 @ [(x, s x)])
    (is ?P ∨ ?Q)
    using B by (blast dest: small-stepsl-output)
  thus ?thesis
  proof
    assume ?P
    hence (OUT x, s, f, vs0, ws0) →*{cfs2} (c2, s2, f, vs2, ws2)
      using C by simp
    hence
     (c2, s2, f, vs2, ws2) = (OUT x, s, f, vs0, ws0) ∧
        flow cfs2 = [] ∨
      (c2, s2, f, vs2, ws2) = (SKIP, s, f, vs0, ws0 @ [(x, s x)]) ∧
        flow cfs2 = [OUT x]
      (is ?P' ∨ -)
      by (rule small-stepsl-output)
    thus ?thesis
    proof (rule disjE, erule-tac [2] conjE)
      assume ?P'
      with ‹?P› show ?thesis
        by fastforce
    next
      assume
        E: (c2, s2, f, vs2, ws2) = (SKIP, s, f, vs0, ws0 @ [(x, s x)]) and
        F: flow cfs2 = [OUT x]
          (is ?cs = -)
      show ?thesis
      proof (rule conjI, clarify)
        fix t1 f' vs1' ws1'
        let ?ws2' = ws1' @ [(x, t1 x)] :: outputs
        show ∃ c2' t2 vs2' ws2'.
          ok-flow-aux-1 c1 c2 c2' s1 t1 t2 f f'
```

$vs_1\ vs_1'\ vs_2\ vs_2'\ ws_1'\ ws_2'$ *?cs* $\wedge$
ok-flow-aux-2 $s_1\ s_2\ t_1\ t_2\ f\, f'\ vs_1\ vs_1'$ *?cs* $\wedge$
ok-flow-aux-3 $s_1\ t_1\ f\, f'\ vs_1\ vs_1'\ ws_1\ ws_1'\ ws_2\ ws_2'$ *?cs*
proof (*rule exI* [*of* - *SKIP*], *rule exI* [*of* - t_1],
rule exI [*of* - vs_1'], *rule exI* [*of* - *?*ws_2'])
{
fix $S\ y$
assume $S \subseteq \{y.\ s = t_1\ (\subseteq$ *sources* $[OUT\ x]\ vs_0\ s\ f\ y)\}$ **and**
$y \in S$
hence $s = t_1\ (\subseteq$ *sources* $[OUT\ x]\ vs_0\ s\ f\ y)$
by *blast*
hence $s\ y = t_1\ y$
by (*subst* (*asm*) *append-Nil* [*symmetric*],
simp only: *sources.simps*, *simp*)
}
moreover {
fix S
assume $S \subseteq \{y.\ s = t_1\ (\subseteq$ *sources-out* $[OUT\ x]\ vs_0\ s\ f\ y)\}$ **and**
$x \in S$
hence $s = t_1\ (\subseteq$ *sources-out* $[OUT\ x]\ vs_0\ s\ f\ x)$
by *blast*
hence $s\ x = t_1\ x$
by (*subst* (*asm*) *append-Nil* [*symmetric*],
simp only: *sources-out.simps*, *simp*)
}
ultimately show
ok-flow-aux-1 $c_1\ c_2$ *SKIP* $s_1\ t_1\ t_1\ f\, f'$
$vs_1\ vs_1'\ vs_2\ vs_1'\ ws_1'$ *?*ws_2' *?cs* $\wedge$
ok-flow-aux-2 $s_1\ s_2\ t_1\ t_1\ f\, f'\ vs_1\ vs_1'$ *?cs* $\wedge$
ok-flow-aux-3 $s_1\ t_1\ f\, f'\ vs_1\ vs_1'\ ws_1\ ws_1'\ ws_2$ *?*ws_2' *?cs*
using E **and** F **and** ‹*?P*› **by** *auto*
qed
qed (*insert* $D\ F$, *fastforce*)
qed
next
assume *?Q*
moreover from *this* **have**
$(c_2,\ s_2,\ f,\ vs_2,\ ws_2) = (SKIP,\ s_1,\ f,\ vs_1,\ ws_1) \wedge$ *flow* $cfs_2 = []$
using C **by** (*blast intro*!: *small-stepsl-skip*)
ultimately show *?thesis*
by *fastforce*
qed
qed

lemma *ctyping2-correct-aux-seq*:
assumes
A: $(U,\ v) \models c_1;;\ c_2\ (\subseteq A,\ X) = Some\ (C,\ Z)$ **and**
B: $\bigwedge B\ Y\ c'\ c''\ s\ s_1\ s_2\ vs_0\ vs_1\ vs_2\ ws_0\ ws_1\ ws_2\ cfs_1\ cfs_2.$
$(U,\ v) \models c_1\ (\subseteq A,\ X) = Some\ (B,\ Y) \Longrightarrow$

```
    s ∈ Univ A (⊆ state ∩ X) ⟹
    (c_1, s, f, vs_0, ws_0) →*{cfs_1} (c', s_1, f, vs_1, ws_1) ⟹
    (c', s_1, f, vs_1, ws_1) →*{cfs_2} (c'', s_2, f, vs_2, ws_2) ⟹
      ok-flow-aux U c' c'' s_1 s_2 f vs_1 vs_2 ws_1 ws_2 (flow cfs_2) and
   C: ⋀p B Y C Z c' c'' s s_1 s_2 vs_0 vs_1 vs_2 ws_0 ws_1 ws_2 cfs_1 cfs_2.
    (U, v) ⊨ c_1 (⊆ A, X) = Some p ⟹ (B, Y) = p ⟹
    (U, v) ⊨ c_2 (⊆ B, Y) = Some (C, Z) ⟹
    s ∈ Univ B (⊆ state ∩ Y) ⟹
    (c_2, s, f, vs_0, ws_0) →*{cfs_1} (c', s_1, f, vs_1, ws_1) ⟹
    (c', s_1, f, vs_1, ws_1) →*{cfs_2} (c'', s_2, f, vs_2, ws_2) ⟹
      ok-flow-aux U c' c'' s_1 s_2 f vs_1 vs_2 ws_1 ws_2 (flow cfs_2) and
   D: s ∈ Univ A (⊆ state ∩ X) and
   E: (c_1;; c_2, s, f, vs_0, ws_0) →*{cfs_1} (c', s_1, f, vs_1, ws_1) and
   F: (c', s_1, f, vs_1, ws_1) →*{cfs_2} (c'', s_2, f, vs_2, ws_2)
  shows ok-flow-aux U c' c'' s_1 s_2 f vs_1 vs_2 ws_1 ws_2 (flow cfs_2)
proof −
  from A obtain B and Y where
   G: (U, v) ⊨ c_1 (⊆ A, X) = Some (B, Y) and
   H: (U, v) ⊨ c_2 (⊆ B, Y) = Some (C, Z)
   by (auto split: option.split-asm)
  have
   (∃ c cfs. c' = c;; c_2 ∧
     (c_1, s, f, vs_0, ws_0) →*{cfs} (c, s_1, f, vs_1, ws_1)) ∨
   (∃ s' p cfs cfs'.
     (c_1, s, f, vs_0, ws_0) →*{cfs} (SKIP, s', p) ∧
     (c_2, s', p) →*{cfs'} (c', s_1, f, vs_1, ws_1))
   using E by (fastforce dest: small-stepsl-seq)
  thus ?thesis
  proof (rule disjE, (erule-tac exE)+, (erule-tac [2] exE)+,
   erule-tac [!] conjE)
   fix c_1' cfs
   assume
     I: c' = c_1';; c_2 and
     J: (c_1, s, f, vs_0, ws_0) →*{cfs} (c_1', s_1, f, vs_1, ws_1)
   hence (c_1';; c_2, s_1, f, vs_1, ws_1) →*{cfs_2} (c'', s_2, f, vs_2, ws_2)
     using F by simp
   hence
    (∃ d cfs'. c'' = d;; c_2 ∧
      (c_1', s_1, f, vs_1, ws_1) →*{cfs'} (d, s_2, f, vs_2, ws_2) ∧
      flow cfs_2 = flow cfs') ∨
    (∃ p cfs' cfs''.
      (c_1', s_1, f, vs_1, ws_1) →*{cfs'} (SKIP, p) ∧
      (c_2, p) →*{cfs''} (c'', s_2, f, vs_2, ws_2) ∧
      flow cfs_2 = flow cfs' @ flow cfs'')
     by (blast dest: small-stepsl-seq)
   thus ?thesis
   proof (rule disjE, (erule-tac exE)+, (erule-tac [2] exE)+,
    (erule-tac [!] conjE)+)
     fix c_1'' cfs'
```

```
  assume
    K: c'' = c₁'';; c₂ and
    L: (c₁', s₁, f, vs₁, ws₁) →*{cfs'} (c₁'', s₂, f, vs₂, ws₂) and
    M: flow cfs₂ = flow cfs'
      (is ?cs = ?cs')
  have N: ok-flow-aux U c₁' c₁'' s₁ s₂ f vs₁ vs₂ ws₁ ws₂ ?cs'
    using B [OF G D J L] .
  show ?thesis
  proof (rule conjI, clarify)
    fix t₁ f' vs₁' ws₁'
    obtain c₂' and t₂ and vs₂' and ws₂' where
     ok-flow-aux-1 c₁' c₁'' c₂' s₁ t₁ t₂ f f'
       vs₁ vs₁' vs₂ vs₂' ws₁' ws₂' ?cs ∧
      ok-flow-aux-2 s₁ s₂ t₁ t₂ f f' vs₁ vs₁' ?cs ∧
      ok-flow-aux-3 s₁ t₁ f f' vs₁ vs₁' ws₁ ws₁' ws₂ ws₂' ?cs
      (is ?P1 ∧ ?P2 ∧ ?P3)
      using M and N by fastforce
    hence ?P1 and ?P2 and ?P3 by auto
    show ∃ c₂' t₂ vs₂' ws₂'.
      ok-flow-aux-1 c' c'' c₂' s₁ t₁ t₂ f f'
        vs₁ vs₁' vs₂ vs₂' ws₁' ws₂' ?cs ∧
      ok-flow-aux-2 s₁ s₂ t₁ t₂ f f' vs₁ vs₁' ?cs ∧
      ok-flow-aux-3 s₁ t₁ f f' vs₁ vs₁' ws₁ ws₁' ws₂ ws₂' ?cs
    proof (rule exI [of - c₂';; c₂], rule exI [of - t₂],
     rule exI [of - vs₂'], rule exI [of - ws₂'])
      {
        fix S
        assume S ≠ {} and
         S ⊆ {x. s₁ = t₁ (⊆ sources-aux ?cs vs₁ s₁ f x)} and
         f = f' (⊆ vs₁, vs₁', ⋃ {tags-aux ?cs vs₁ s₁ f x | x. x ∈ S})
        hence
         (c', t₁, f', vs₁', ws₁') →* (c₂';; c₂, t₂, f', vs₂', ws₂') ∧
          map fst [p←drop (length vs₁) vs₂. fst p ∈ S] =
            map fst [p←drop (length vs₁') vs₂'. fst p ∈ S]
          using I and ‹?P1› by (blast intro: star-seq2)
      }
      thus
       ok-flow-aux-1 c' c'' (c₂';; c₂) s₁ t₁ t₂ f f'
         vs₁ vs₁' vs₂ vs₂' ws₁' ws₂' ?cs ∧
        ok-flow-aux-2 s₁ s₂ t₁ t₂ f f' vs₁ vs₁' ?cs ∧
        ok-flow-aux-3 s₁ t₁ f f' vs₁ vs₁' ws₁ ws₁' ws₂ ws₂' ?cs
        using K and ‹?P2› and ‹?P3› by simp
    qed
  qed (simp add: M N)
next
  fix p cfs' cfs''
  assume (c₁', s₁, f, vs₁, ws₁) →*{cfs'} (SKIP, p)
  moreover from this obtain s₁' and vs and ws where
    K: p = (s₁', f, vs, ws)
```

```
  by (blast dest: small-stepsl-stream)
ultimately have
  L: (c_1', s_1, f, vs_1, ws_1) →*{cfs'} (SKIP, s_1', f, vs, ws)
  by simp
assume (c_2, p) →*{cfs''} (c'', s_2, f, vs_2, ws_2)
with K have
  M: (c_2, s_1', f, vs, ws) →*{cfs''} (c'', s_2, f, vs_2, ws_2)
  by simp
assume N: flow cfs_2 = flow cfs' @ flow cfs''
  (is (?cs :: flow) = ?cs' @ ?cs'')
have O: ok-flow-aux U c_1' SKIP s_1 s_1' f vs_1 vs ws_1 ws ?cs'
  using B [OF G D J L] .
have (c_1, s, f, vs_0, ws_0) →*{cfs @ cfs'} (SKIP, s_1', f, vs, ws)
  using J and L by (simp add: small-stepsl-append)
hence (c_1, s, f, vs_0, ws_0) ⇒ (s_1', f, vs, ws)
  by (auto dest: small-stepsl-steps simp: big-iff-small)
hence P: s_1' ∈ Univ B (⊆ state ∩ Y)
  using G and D by (rule ctyping2-approx)
have Q: ok-flow-aux U c_2 c'' s_1' s_2 f vs vs_2 ws ws_2 ?cs''
  using C [OF G - H P - M, of vs ws []] by simp
show ?thesis
proof (rule conjI, clarify)
  fix t_1 f' vs_1' ws_1'
  obtain c_1'' and t_1' and vs_1'' and ws_1'' where
   ok-flow-aux-1 c_1' SKIP c_1'' s_1 t_1 t_1' f f'
      vs_1 vs_1' vs vs_1'' ws_1' ws_1'' ?cs' ∧
    ok-flow-aux-2 s_1 s_1' t_1 t_1' f f' vs_1 vs_1' ?cs' ∧
    ok-flow-aux-3 s_1 t_1 f f' vs_1 vs_1' ws_1 ws_1' ws ws_1'' ?cs'
    (is - ∧ ?P2 ∧ ?P3)
    using O by fastforce
  hence
   ok-flow-aux-1 c_1' SKIP SKIP s_1 t_1 t_1' f f'
      vs_1 vs_1' vs vs_1'' ws_1' ws_1'' ?cs'
    (is ?P1) and ?P2 and ?P3 by auto
  obtain c_2' and t_2 and vs_2' and ws_2' where
   ok-flow-aux-1 c_2 c'' c_2' s_1' t_1' t_2 f f'
     vs vs_1'' vs_2 vs_2' ws_1'' ws_2' ?cs'' ∧
    ok-flow-aux-2 s_1' s_2 t_1' t_2 f f' vs vs_1'' ?cs'' ∧
    ok-flow-aux-3 s_1' t_1' f f' vs vs_1'' ws ws_1'' ws_2 ws_2' ?cs''
    (is ?P1' ∧ ?P2' ∧ ?P3')
    using Q by fastforce
  hence ?P1' and ?P2' and ?P3' by auto
  show ∃ c_2' t_2 vs_2' ws_2'.
    ok-flow-aux-1 c' c'' c_2' s_1 t_1 t_2 f f'
      vs_1 vs_1' vs_2 vs_2' ws_1' ws_2' ?cs ∧
    ok-flow-aux-2 s_1 s_2 t_1 t_2 f f' vs_1 vs_1' ?cs ∧
    ok-flow-aux-3 s_1 t_1 f f' vs_1 vs_1' ws_1 ws_1' ws_2 ws_2' ?cs
  proof (rule exI [of - c_2'], rule exI [of - t_2],
   rule exI [of - vs_2'], rule exI [of - ws_2'])
```

{
fix *S*
assume
R: $S \neq \{\}$ **and**
S: $S \subseteq \{x.\ s_1 = t_1\ (\subseteq \mathit{sources\text{-}aux}\ (?cs'\ @\ ?cs'')\ vs_1\ s_1\ f\ x)\}$ **and**
T: $f = f'\ (\subseteq vs_1,\ vs_1{}',$
$\bigcup\ \{\mathit{tags\text{-}aux}\ (?cs'\ @\ ?cs'')\ vs_1\ s_1\ f\ x \mid x.\ x \in S\})$
(**is** - = - ($\subseteq$ -, -, *?T*))
have $\forall x.\ \mathit{sources\text{-}aux}\ ?cs'\ vs_1\ s_1\ f\ x \subseteq$
$\mathit{sources\text{-}aux}\ (?cs'\ @\ ?cs'')\ vs_1\ s_1\ f\ x$
by (*blast intro*: *subsetD* [*OF sources-aux-append*])
hence $S \subseteq \{x.\ s_1 = t_1\ (\subseteq \mathit{sources\text{-}aux}\ ?cs'\ vs_1\ s_1\ f\ x)\}$
using *S* **by** *blast*
moreover have $\bigcup\ \{\mathit{tags\text{-}aux}\ ?cs'\ vs_1\ s_1\ f\ x \mid x.\ x \in S\} \subseteq\ ?T$
(**is** *?T′* $\subseteq$ -)
by (*blast intro*: *subsetD* [*OF tags-aux-append*])
with *T* **have** $f = f'\ (\subseteq vs_1,\ vs_1{}',\ ?T')$
by (*rule eq-streams-subset*)
ultimately have
$(c_1{}',\ t_1,\ f',\ vs_1{}',\ ws_1{}') \rightarrow\!*\ (\mathit{SKIP},\ t_1{}',\ f',\ vs_1{}'',\ ws_1{}'') \wedge$
map fst [*p*←*drop* (*length* vs_1) *vs*. *fst p* $\in$ *S*] =
map fst [*p*←*drop* (*length* $vs_1{}'$) $vs_1{}''$. *fst p* $\in$ *S*]
(**is** *?Q1* $\wedge$ *?Q2*)
using *R* **and** ‹*?P1*› **by** *simp*
hence *?Q1* **and** *?Q2* **by** *auto*
have $S \subseteq \{x.\ s_1{}' = t_1{}'\ (\subseteq \mathit{sources\text{-}aux}\ ?cs''\ vs\ s_1{}'\ f\ x)\}$
by (*rule sources-aux-rhs* [*OF S T L* ‹*?P2*›])
moreover have $f = f'\ (\subseteq vs,\ vs_1{}'',$
$\bigcup\ \{\mathit{tags\text{-}aux}\ ?cs''\ vs\ s_1{}'\ f\ x \mid x.\ x \in S\})$
by (*rule tags-aux-rhs* [*OF S T L* ‹*?Q1*› ‹*?P1*›])
ultimately have
$(c_2,\ t_1{}',\ f',\ vs_1{}'',\ ws_1{}'') \rightarrow\!*\ (c_2{}',\ t_2,\ f',\ vs_2{}',\ ws_2{}') \wedge$
$(c'' = \mathit{SKIP}) = (c_2{}' = \mathit{SKIP}) \wedge$
map fst [*p*←*drop* (*length vs*) vs_2. *fst p* $\in$ *S*] =
map fst [*p*←*drop* (*length* $vs_1{}''$) $vs_2{}'$. *fst p* $\in$ *S*]
(**is** *?Q1′* $\wedge$ *?R2* $\wedge$ *?Q2′*)
using *R* **and** ‹*?P1′*› **by** *simp*
hence *?Q1′* **and** *?R2* **and** *?Q2′* **by** *auto*
from *I* **and** ‹*?Q1*› **and** ‹*?Q1′*› **have**
$(c',\ t_1,\ f',\ vs_1{}',\ ws_1{}') \rightarrow\!*\ (c_2{}',\ t_2,\ f',\ vs_2{}',\ ws_2{}')$
(**is** *?R1*)
by (*blast intro*: *star-seq2 star-trans*)
moreover have
map fst [*p*←*drop* (*length* vs_1) vs_2. *fst p* $\in$ *S*] =
map fst [*p*←*drop* (*length* $vs_1{}'$) $vs_2{}'$. *fst p* $\in$ *S*]
by (*rule small-steps-inputs* [*OF L M* ‹*?Q1*› ‹*?Q1′*› ‹*?Q2*› ‹*?Q2′*›])
ultimately have *?R1* $\wedge$ *?R2* $\wedge$ *?this*
using ‹*?R2*› **by** *simp*
}

```
moreover {
  fix S
  assume
    R: S ≠ {} and
    S: S ⊆ {x. s1 = t1 (⊆ sources (?cs' @ ?cs'') vs1 s1 f x)} and
    T: f = f' (⊆ vs1, vs1',
      ⋃ {tags (?cs' @ ?cs'') vs1 s1 f x | x. x ∈ S})
      (is - = - (⊆ -, -, ?T))
  have ∀x. sources-aux (?cs' @ ?cs'') vs1 s1 f x ⊆
    sources (?cs' @ ?cs'') vs1 s1 f x
    by (blast intro: subsetD [OF sources-aux-sources])
  moreover have ∀x. sources-aux ?cs' vs1 s1 f x ⊆
    sources-aux (?cs' @ ?cs'') vs1 s1 f x
    by (blast intro: subsetD [OF sources-aux-append])
  ultimately have U: S ⊆ {x. s1 = t1 (⊆ sources-aux ?cs' vs1 s1 f x)}
    using S by blast
  have ⋃ {tags-aux (?cs' @ ?cs'') vs1 s1 f x | x. x ∈ S} ⊆ ?T
    (is ?T' ⊆ -)
    by (blast intro: subsetD [OF tags-aux-tags])
  moreover have ⋃ {tags-aux ?cs' vs1 s1 f x | x. x ∈ S} ⊆ ?T'
    (is ?T'' ⊆ -)
    by (blast intro: subsetD [OF tags-aux-append])
  ultimately have ?T'' ⊆ ?T
    by simp
  with T have f = f' (⊆ vs1, vs1', ?T'')
    by (rule eq-streams-subset)
  hence V: (c1', t1, f', vs1', ws1') →* (SKIP, t1', f', vs1'', ws1'')
    using R and U and ‹?P1› by simp
  have S ⊆ {x. s1' = t1' (⊆ sources ?cs'' vs s1' f x)}
    by (rule sources-rhs [OF S T L ‹?P2›])
  moreover have f = f' (⊆ vs, vs1'',
    ⋃ {tags ?cs'' vs s1' f x | x. x ∈ S})
    by (rule tags-rhs [OF S T L V ‹?P1›])
  ultimately have s2 = t2 (⊆ S)
    using ‹?P2'› by blast
}
moreover {
  fix S
  assume
    R: S ≠ {} and
    S: S ⊆ {x. s1 = t1 (⊆ sources-out (?cs' @ ?cs'') vs1 s1 f x)} and
    T: f = f' (⊆ vs1, vs1',
      ⋃ {tags-out (?cs' @ ?cs'') vs1 s1 f x | x. x ∈ S})
      (is - = - (⊆ -, -, ?T))
  have U: ∀x. sources-aux (?cs' @ ?cs'') vs1 s1 f x ⊆
    sources-out (?cs' @ ?cs'') vs1 s1 f x
    by (blast intro: subsetD [OF sources-aux-sources-out])
  moreover have ∀x. sources-aux ?cs' vs1 s1 f x ⊆
    sources-aux (?cs' @ ?cs'') vs1 s1 f x
```

by (*blast intro*: *subsetD* [*OF sources-aux-append*])
ultimately have V: $S \subseteq \{x.\ s_1 = t_1\ (\subseteq \textit{sources-aux ?cs}'\ vs_1\ s_1\ f\ x)\}$
using S **by** *blast*
have W: $\bigcup\ \{\textit{tags-aux}\ (\textit{?cs}'\ @\ \textit{?cs}'')\ vs_1\ s_1\ f\ x \mid x.\ x \in S\} \subseteq \textit{?T}$
(**is** $\textit{?T}' \subseteq$ -)
by (*blast intro*: *subsetD* [*OF tags-aux-tags-out*])
moreover have $\bigcup\ \{\textit{tags-aux ?cs}'\ vs_1\ s_1\ f\ x \mid x.\ x \in S\} \subseteq \textit{?T}'$
(**is** $\textit{?T}'' \subseteq$ -)
by (*blast intro*: *subsetD* [*OF tags-aux-append*])
ultimately have $\textit{?T}'' \subseteq \textit{?T}$
by *simp*
with T **have** $f = f'\ (\subseteq vs_1,\ vs_1',\ \textit{?T}'')$
by (*rule eq-streams-subset*)
hence X: $(c_1',\ t_1,\ f',\ vs_1',\ ws_1') \rightarrow*\ (\textit{SKIP},\ t_1',\ f',\ vs_1'',\ ws_1'')$
using R **and** V **and** ‹*?P1*› **by** *simp*
have Y: $S \subseteq \{x.\ s_1 = t_1\ (\subseteq \textit{sources-aux}\ (\textit{?cs}'\ @\ \textit{?cs}'')\ vs_1\ s_1\ f\ x)\}$
using S **and** U **by** *blast*
have Z: $f = f'\ (\subseteq vs_1,\ vs_1',\ \textit{?T}')$
using T **and** W **by** (*rule eq-streams-subset*)
have $S \subseteq \{x.\ s_1' = t_1'\ (\subseteq \textit{sources-aux ?cs}''\ vs\ s_1'\ f\ x)\}$
by (*rule sources-aux-rhs* [*OF Y Z L* ‹*?P2*›])
moreover have $f = f'\ (\subseteq vs,\ vs_1'',$
$\bigcup\ \{\textit{tags-aux ?cs}''\ vs\ s_1'\ f\ x \mid x.\ x \in S\})$
by (*rule tags-aux-rhs* [*OF Y Z L X* ‹*?P1*›])
ultimately have AA:
$(c_2,\ t_1',\ f',\ vs_1'',\ ws_1'') \rightarrow*\ (c_2',\ t_2,\ f',\ vs_2',\ ws_2')$
using R **and** ‹*?P1'*› **by** *simp*
have $\forall x.\ \textit{sources-out ?cs}'\ vs_1\ s_1\ f\ x \subseteq$
$\textit{sources-out}\ (\textit{?cs}'\ @\ \textit{?cs}'')\ vs_1\ s_1\ f\ x$
by (*blast intro*: *subsetD* [*OF sources-out-append*])
hence $S \subseteq \{x.\ s_1 = t_1\ (\subseteq \textit{sources-out ?cs}'\ vs_1\ s_1\ f\ x)\}$
using S **by** *blast*
moreover have $\bigcup\ \{\textit{tags-out ?cs}'\ vs_1\ s_1\ f\ x \mid x.\ x \in S\} \subseteq \textit{?T}$
(**is** $\textit{?T}' \subseteq$ -)
by (*blast intro*: *subsetD* [*OF tags-out-append*])
with T **have** $f = f'\ (\subseteq vs_1,\ vs_1',\ \textit{?T}')$
by (*rule eq-streams-subset*)
ultimately have AB: $[p \leftarrow \textit{drop}\ (\textit{length}\ ws_1)\ ws.\ \textit{fst}\ p \in S] =$
$[p \leftarrow \textit{drop}\ (\textit{length}\ ws_1')\ ws_1''.\ \textit{fst}\ p \in S]$
using R **and** ‹*?P3*› **by** *simp*
have $S \subseteq \{x.\ s_1' = t_1'\ (\subseteq \textit{sources-out ?cs}''\ vs\ s_1'\ f\ x)\}$
by (*rule sources-out-rhs* [*OF S T L* ‹*?P2*›])
moreover have $f = f'\ (\subseteq vs,\ vs_1'',$
$\bigcup\ \{\textit{tags-out ?cs}''\ vs\ s_1'\ f\ x \mid x.\ x \in S\})$
by (*rule tags-out-rhs* [*OF S T L X* ‹*?P1*›])
ultimately have $[p \leftarrow \textit{drop}\ (\textit{length}\ ws)\ ws_2.\ \textit{fst}\ p \in S] =$
$[p \leftarrow \textit{drop}\ (\textit{length}\ ws_1'')\ ws_2'.\ \textit{fst}\ p \in S]$
using R **and** ‹*?P3'*› **by** *simp*
hence $[p \leftarrow \textit{drop}\ (\textit{length}\ ws_1)\ ws_2.\ \textit{fst}\ p \in S] =$

```
              [p←drop (length ws1') ws2'. fst p ∈ S]
              by (rule small-steps-outputs [OF L M X AA AB])
          }
          ultimately show
           ok-flow-aux-1 c' c'' c2' s1 t1 t2 f f'
              vs1 vs1' vs2 vs2' ws1' ws2' ?cs ∧
            ok-flow-aux-2 s1 s2 t1 t2 f f' vs1 vs1' ?cs ∧
            ok-flow-aux-3 s1 t1 f f' vs1 vs1' ws1 ws1' ws2 ws2' ?cs
            using N by auto
        qed
      qed (simp add: no-upd-append N O Q)
    qed
  next
    fix s' p cfs cfs'
    assume I: (c1, s, f, vs0, ws0) →∗{cfs} (SKIP, s', p)
    hence (c1, s, f, vs0, ws0) ⇒ (s', p)
      by (auto dest: small-stepsl-steps simp: big-iff-small)
    hence J: s' ∈ Univ B (⊆ state ∩ Y)
      using G and D by (rule ctyping2-approx)
    assume (c2, s', p) →∗{cfs'} (c', s1, f, vs1, ws1)
    moreover obtain vs and ws where p = (f, vs, ws)
      using I by (blast dest: small-stepsl-stream)
    ultimately have K: (c2, s', f, vs, ws) →∗{cfs'} (c', s1, f, vs1, ws1)
      by simp
    show ?thesis
      using C [OF G - H J K F] by simp
  qed
qed

lemma ctyping2-correct-aux-or:
  assumes
    A: (U, v) |= c1 OR c2 (⊆ A, X) = Some (C, Y) and
    B: ⋀C Y c' c'' s s1 s2 vs0 vs1 vs2 ws0 ws1 ws2 cfs1 cfs2.
      (U, v) |= c1 (⊆ A, X) = Some (C, Y) ⟹
      s ∈ Univ A (⊆ state ∩ X) ⟹
      (c1, s, f, vs0, ws0) →∗{cfs1} (c', s1, f, vs1, ws1) ⟹
      (c', s1, f, vs1, ws1) →∗{cfs2} (c'', s2, f, vs2, ws2) ⟹
        ok-flow-aux U c' c'' s1 s2 f vs1 vs2 ws1 ws2 (flow cfs2) and
    C: ⋀C Y c' c'' s s1 s2 vs0 vs1 vs2 ws0 ws1 ws2 cfs1 cfs2.
      (U, v) |= c2 (⊆ A, X) = Some (C, Y) ⟹
      s ∈ Univ A (⊆ state ∩ X) ⟹
      (c2, s, f, vs0, ws0) →∗{cfs1} (c', s1, f, vs1, ws1) ⟹
      (c', s1, f, vs1, ws1) →∗{cfs2} (c'', s2, f, vs2, ws2) ⟹
        ok-flow-aux U c' c'' s1 s2 f vs1 vs2 ws1 ws2 (flow cfs2) and
    D: s ∈ Univ A (⊆ state ∩ X) and
    E: (c1 OR c2, s, f, vs0, ws0) →∗{cfs1} (c', s1, f, vs1, ws1) and
    F: (c', s1, f, vs1, ws1) →∗{cfs2} (c'', s2, f, vs2, ws2)
  shows ok-flow-aux U c' c'' s1 s2 f vs1 vs2 ws1 ws2 (flow cfs2)
proof –
```

```
from $A$ obtain $C_1$ and $Y_1$ and $C_2$ and $Y_2$ where
  $G$: $(U, v) \models c_1 (\subseteq A, X) = Some\ (C_1, Y_1)$ and
  $H$: $(U, v) \models c_2 (\subseteq A, X) = Some\ (C_2, Y_2)$
  by (auto split: option.split-asm)
have
 $(c', s_1, f, vs_1, ws_1) = (c_1\ OR\ c_2, s, f, vs_0, ws_0) \vee$
  $(c_1, s, f, vs_0, ws_0) \rightarrow*\{tl\ cfs_1\}\ (c', s_1, f, vs_1, ws_1) \vee$
  $(c_2, s, f, vs_0, ws_0) \rightarrow*\{tl\ cfs_1\}\ (c', s_1, f, vs_1, ws_1)$
  (is $?P \vee ?Q \vee ?R$)
  using $E$ by (blast dest: small-stepsl-or)
thus ?thesis
proof (rule disjE, erule-tac [2] disjE)
  assume ?P
  hence $(c_1\ OR\ c_2, s, f, vs_0, ws_0) \rightarrow*\{cfs_2\}\ (c'', s_2, f, vs_2, ws_2)$
    using $F$ by simp
  hence
   $(c'', s_2, f, vs_2, ws_2) = (c_1\ OR\ c_2, s, f, vs_0, ws_0) \wedge$
     $flow\ cfs_2 = [] \vee$
   $(c_1, s, f, vs_0, ws_0) \rightarrow*\{tl\ cfs_2\}\ (c'', s_2, f, vs_2, ws_2) \wedge$
     $flow\ cfs_2 = flow\ (tl\ cfs_2) \vee$
   $(c_2, s, f, vs_0, ws_0) \rightarrow*\{tl\ cfs_2\}\ (c'', s_2, f, vs_2, ws_2) \wedge$
     $flow\ cfs_2 = flow\ (tl\ cfs_2)$
   (is $?P' \vee$ -)
   by (rule small-stepsl-or)
  thus ?thesis
  proof (rule disjE, erule-tac [2] disjE, erule-tac [2−3] conjE)
    assume ?P'
    with ‹?P› show ?thesis
      by fastforce
  next
    assume
      $I$: $(c_1, s, f, vs_0, ws_0) \rightarrow*\{tl\ cfs_2\}\ (c'', s_2, f, vs_2, ws_2)$ and
      $J$: $flow\ cfs_2 = flow\ (tl\ cfs_2)$
        (is $?cs = ?cs'$)
    have $K$: $(c_1, s, f, vs_0, ws_0) \rightarrow*\{[]\}\ (c_1, s, f, vs_0, ws_0)$
      by simp
    hence $L$: $ok\text{-}flow\text{-}aux\ U\ c_1\ c''\ s\ s_2\ f\ vs_0\ vs_2\ ws_0\ ws_2\ ?cs'$
      by (rule $B$ [OF $G$ $D$ - $I$])
    show ?thesis
    proof (rule conjI, clarify)
      fix $t_1\ f'\ {vs_1}'\ {ws_1}'$
      obtain ${c_2}'$ and $t_2$ and ${vs_2}'$ and ${ws_2}'$ where
       $ok\text{-}flow\text{-}aux\text{-}1\ c_1\ c''\ {c_2}'\ s\ t_1\ t_2\ f\ f'$
         $vs_0\ {vs_1}'\ vs_2\ {vs_2}'\ {ws_1}'\ {ws_2}'\ ?cs' \wedge$
        $ok\text{-}flow\text{-}aux\text{-}2\ s\ s_2\ t_1\ t_2\ f\ f'\ vs_0\ {vs_1}'\ ?cs' \wedge$
        $ok\text{-}flow\text{-}aux\text{-}3\ s\ t_1\ f\ f'\ vs_0\ {vs_1}'\ ws_0\ {ws_1}'\ ws_2\ {ws_2}'\ ?cs'$
        (is $?P1 \wedge ?P2 \wedge ?P3$)
        using $L$ by fastforce
      hence ?P1 and ?P2 and ?P3 by auto
```

show $\exists\, c_2{}'\ t_2\ vs_2{}'\ ws_2{}'.$
  ok-flow-aux-1 $c'\ c''\ c_2{}'\ s_1\ t_1\ t_2\ f\ f'$
    $vs_1\ vs_1{}'\ vs_2\ vs_2{}'\ ws_1{}'\ ws_2{}'$ *?cs* $\wedge$
  ok-flow-aux-2 $s_1\ s_2\ t_1\ t_2\ f\ f'\ vs_1\ vs_1{}'$ *?cs* $\wedge$
  ok-flow-aux-3 $s_1\ t_1\ f\ f'\ vs_1\ vs_1{}'\ ws_1\ ws_1{}'\ ws_2\ ws_2{}'$ *?cs*
proof (*rule exI* [*of* - $c_2{}'$], *rule exI* [*of* - t_2],
 rule exI [*of* - $vs_2{}'$], *rule exI* [*of* - $ws_2{}'$])
  {
    fix S
    assume
      $S \neq \{\}$ **and**
      $S \subseteq \{x.\ s_1 = t_1\ (\subseteq \textit{sources-aux ?cs}'\ vs_1\ s_1\ f\ x)\}$ **and**
      $f = f'\ (\subseteq vs_1,\ vs_1{}',\ \bigcup\ \{\textit{tags-aux ?cs}'\ vs_1\ s_1\ f\ x \mid x.\ x \in S\})$
    hence
      $(c_1,\ t_1,\ f',\ vs_1{}',\ ws_1{}') \rightarrow\!*\ (c_2{}',\ t_2,\ f',\ vs_2{}',\ ws_2{}') \wedge$
       $(c'' = \textit{SKIP}) = (c_2{}' = \textit{SKIP}) \wedge$
       $\textit{map fst}\ [p \leftarrow \textit{drop}\ (\textit{length}\ vs_1)\ vs_2.\ \textit{fst}\ p \in S] =$
         $\textit{map fst}\ [p \leftarrow \textit{drop}\ (\textit{length}\ vs_1{}')\ vs_2{}'.\ \textit{fst}\ p \in S]$
      (**is** *?Q1* $\wedge$ *?Q2* $\wedge$ *?Q3*)
      using ‹*?P*› **and** ‹*?P1*› **by** *simp*
    hence *?Q1* **and** *?Q2* **and** *?Q3* **by** *auto*
    moreover have $(c_1\ \textit{OR}\ c_2,\ t_1,\ f',\ vs_1{}',\ ws_1{}') \rightarrow$
      $(c_1,\ t_1,\ f',\ vs_1{}',\ ws_1{}')$ **..**
    hence $(c',\ t_1,\ f',\ vs_1{}',\ ws_1{}') \rightarrow\!*\ (c_2{}',\ t_2,\ f',\ vs_2{}',\ ws_2{}')$
      using ‹*?P*› **and** ‹*?Q1*› **by** (*blast intro*: *star-trans*)
    ultimately have *?this* $\wedge$ *?Q2* $\wedge$ *?Q3*
      by *simp*
  }
  moreover {
    fix S
    assume
      $S \subseteq \{x.\ s_1 = t_1\ (\subseteq \textit{sources ?cs}'\ vs_1\ s_1\ f\ x)\}$ **and**
      $f = f'\ (\subseteq vs_1,\ vs_1{}',\ \bigcup\ \{\textit{tags ?cs}'\ vs_1\ s_1\ f\ x \mid x.\ x \in S\})$
    hence $s_2 = t_2\ (\subseteq S)$
      using ‹*?P*› **and** ‹*?P2*› **by** *blast*
  }
  moreover {
    fix S
    assume
      $S \neq \{\}$ **and**
      $S \subseteq \{x.\ s_1 = t_1\ (\subseteq \textit{sources-out ?cs}'\ vs_1\ s_1\ f\ x)\}$ **and**
      $f = f'\ (\subseteq vs_1,\ vs_1{}',\ \bigcup\ \{\textit{tags-out ?cs}'\ vs_1\ s_1\ f\ x \mid x.\ x \in S\})$
    hence $[p \leftarrow \textit{drop}\ (\textit{length}\ ws_1)\ ws_2.\ \textit{fst}\ p \in S] =$
      $[p \leftarrow \textit{drop}\ (\textit{length}\ ws_1{}')\ ws_2{}'.\ \textit{fst}\ p \in S]$
      using ‹*?P*› **and** ‹*?P3*› **by** *simp*
  }
  ultimately show
   ok-flow-aux-1 $c'\ c''\ c_2{}'\ s_1\ t_1\ t_2\ f\ f'$
     $vs_1\ vs_1{}'\ vs_2\ vs_2{}'\ ws_1{}'\ ws_2{}'$ *?cs* $\wedge$

ok-flow-aux-2 s_1 s_2 t_1 t_2 f f' vs_1 vs_1' *?cs* ∧
ok-flow-aux-3 s_1 t_1 f f' vs_1 vs_1' ws_1 ws_1' ws_2 ws_2' *?cs*
using *J* **by** *auto*
qed
qed (*simp add*: *B* [*OF G D K I*] *J*)
next
assume
I: (c_2, *s*, *f*, vs_0, ws_0) →∗{*tl* cfs_2} (c'', s_2, *f*, vs_2, ws_2) **and**
J: *flow* cfs_2 = *flow* (*tl* cfs_2)
(**is** *?cs* = *?cs'*)
have *K*: (c_2, *s*, *f*, vs_0, ws_0) →∗{[]} (c_2, *s*, *f*, vs_0, ws_0)
by *simp*
hence *L*: *ok-flow-aux U* c_2 c'' *s* s_2 *f* vs_0 vs_2 ws_0 ws_2 *?cs'*
by (*rule C* [*OF H D - I*])
show *?thesis*
proof (*rule conjI*, *clarify*)
fix t_1 f' vs_1' ws_1'
obtain c_2' **and** t_2 **and** vs_2' **and** ws_2' **where**
ok-flow-aux-1 c_2 c'' c_2' *s* t_1 t_2 f f'
vs_0 vs_1' vs_2 vs_2' ws_1' ws_2' *?cs'* ∧
ok-flow-aux-2 *s* s_2 t_1 t_2 f f' vs_0 vs_1' *?cs'* ∧
ok-flow-aux-3 *s* t_1 f f' vs_0 vs_1' ws_0 ws_1' ws_2 ws_2' *?cs'*
(**is** *?P1* ∧ *?P2* ∧ *?P3*)
using *L* **by** *fastforce*
hence *?P1* **and** *?P2* **and** *?P3* **by** *auto*
show ∃ c_2' t_2 vs_2' ws_2'.
ok-flow-aux-1 c' c'' c_2' s_1 t_1 t_2 f f'
vs_1 vs_1' vs_2 vs_2' ws_1' ws_2' *?cs* ∧
ok-flow-aux-2 s_1 s_2 t_1 t_2 f f' vs_1 vs_1' *?cs* ∧
ok-flow-aux-3 s_1 t_1 f f' vs_1 vs_1' ws_1 ws_1' ws_2 ws_2' *?cs*
proof (*rule exI* [*of* - c_2'], *rule exI* [*of* - t_2],
rule exI [*of* - vs_2'], *rule exI* [*of* - ws_2'])
{
fix *S*
assume
$S \neq \{\}$ **and**
$S \subseteq \{x.\ s_1 = t_1$ (⊆ *sources-aux ?cs'* vs_1 s_1 *f x*)} **and**
$f = f'$ (⊆ vs_1, vs_1', ⋃ {*tags-aux ?cs'* vs_1 s_1 *f x* | *x*. $x \in S$})
hence
(c_2, t_1, f', vs_1', ws_1') →∗ (c_2', t_2, f', vs_2', ws_2') ∧
(c'' = *SKIP*) = (c_2' = *SKIP*) ∧
map fst [*p*←*drop* (*length* vs_1) vs_2. *fst p* ∈ *S*] =
map fst [*p*←*drop* (*length* vs_1') vs_2'. *fst p* ∈ *S*]
(**is** *?Q1* ∧ *?Q2* ∧ *?Q3*)
using ‹*?P*› **and** ‹*?P1*› **by** *simp*
hence *?Q1* **and** *?Q2* **and** *?Q3* **by** *auto*
moreover have (c_1 *OR* c_2, t_1, f', vs_1', ws_1') →
(c_2, t_1, f', vs_1', ws_1') **..**
hence (c', t_1, f', vs_1', ws_1') →∗ (c_2', t_2, f', vs_2', ws_2')

```
          using ‹?P› and ‹?Q1› by (blast intro: star-trans)
        ultimately have ?this ∧ ?Q2 ∧ ?Q3
          by simp
      }
      moreover {
        fix S
        assume
         $S \subseteq \{x.\ s_1 = t_1\ (\subseteq sources\ ?cs'\ vs_1\ s_1\ f\ x)\}$ and
         $f = f'\ (\subseteq vs_1,\ vs_1',\ \bigcup\ \{tags\ ?cs'\ vs_1\ s_1\ f\ x \mid x.\ x \in S\})$
        hence $s_2 = t_2\ (\subseteq S)$
          using ‹?P› and ‹?P2› by blast
      }
      moreover {
        fix S
        assume
         $S \neq \{\}$ and
         $S \subseteq \{x.\ s_1 = t_1\ (\subseteq sources\text{-}out\ ?cs'\ vs_1\ s_1\ f\ x)\}$ and
         $f = f'\ (\subseteq vs_1,\ vs_1',\ \bigcup\ \{tags\text{-}out\ ?cs'\ vs_1\ s_1\ f\ x \mid x.\ x \in S\})$
        hence $[p \leftarrow drop\ (length\ ws_1)\ ws_2.\ fst\ p \in S] =$
          $[p \leftarrow drop\ (length\ ws_1')\ ws_2'.\ fst\ p \in S]$
          using ‹?P› and ‹?P3› by simp
      }
      ultimately show
       $ok\text{-}flow\text{-}aux\text{-}1\ c'\ c''\ c_2'\ s_1\ t_1\ t_2\ f\ f'$
          $vs_1\ vs_1'\ vs_2\ vs_2'\ ws_1'\ ws_2'\ ?cs\ \wedge$
        $ok\text{-}flow\text{-}aux\text{-}2\ s_1\ s_2\ t_1\ t_2\ f\ f'\ vs_1\ vs_1'\ ?cs\ \wedge$
        $ok\text{-}flow\text{-}aux\text{-}3\ s_1\ t_1\ f\ f'\ vs_1\ vs_1'\ ws_1\ ws_1'\ ws_2\ ws_2'\ ?cs$
        using J by auto
    qed
  qed (simp add: C [OF H D K I] J)
 qed
next
 assume ?Q
 thus ?thesis
   by (rule B [OF G D - F])
next
 assume ?R
 thus ?thesis
   by (rule C [OF H D - F])
qed
qed
```

```
lemma ctyping2-correct-aux-if:
 assumes
  A: $(U,\ v) \models IF\ b\ THEN\ c_1\ ELSE\ c_2\ (\subseteq A,\ X) = Some\ (C,\ Y)$ and
  B: $\bigwedge U'\ p\ B_1\ B_2\ C_1\ Y_1\ c'\ c''\ s\ s_1\ s_2\ vs_0\ vs_1\ vs_2\ ws_0\ ws_1\ ws_2\ cfs_1\ cfs_2.$
   $(U',\ p) = (insert\ (Univ?\ A\ X,\ bvars\ b)\ U,\ \models b\ (\subseteq A,\ X)) \Longrightarrow$
   $(B_1,\ B_2) = p \Longrightarrow$
   $(U',\ v) \models c_1\ (\subseteq B_1,\ X) = Some\ (C_1,\ Y_1) \Longrightarrow$
```

$s \in \mathit{Univ}\ B_1\ (\subseteq \mathit{state} \cap X) \Longrightarrow$
$(c_1, s, f, \mathit{vs}_0, \mathit{ws}_0) \rightarrow*\{\mathit{cfs}_1\}\ (c', s_1, f, \mathit{vs}_1, \mathit{ws}_1) \Longrightarrow$
$(c', s_1, f, \mathit{vs}_1, \mathit{ws}_1) \rightarrow*\{\mathit{cfs}_2\}\ (c'', s_2, f, \mathit{vs}_2, \mathit{ws}_2) \Longrightarrow$
$\mathit{ok\text{-}flow\text{-}aux}\ U'\ c'\ c''\ s_1\ s_2\ f\ \mathit{vs}_1\ \mathit{vs}_2\ \mathit{ws}_1\ \mathit{ws}_2\ (\mathit{flow}\ \mathit{cfs}_2)$ **and**
C: $\bigwedge U'\ p\ B_1\ B_2\ C_2\ Y_2\ c'\ c''\ s\ s_1\ s_2\ \mathit{vs}_0\ \mathit{vs}_1\ \mathit{vs}_2\ \mathit{ws}_0\ \mathit{ws}_1\ \mathit{ws}_2\ \mathit{cfs}_1\ \mathit{cfs}_2.$
$(U', p) = (\mathit{insert}\ (\mathit{Univ?}\ A\ X, \mathit{bvars}\ b)\ U, \models b\ (\subseteq A, X)) \Longrightarrow$
$(B_1, B_2) = p \Longrightarrow$
$(U', v) \models c_2\ (\subseteq B_2, X) = \mathit{Some}\ (C_2, Y_2) \Longrightarrow$
$s \in \mathit{Univ}\ B_2\ (\subseteq \mathit{state} \cap X) \Longrightarrow$
$(c_2, s, f, \mathit{vs}_0, \mathit{ws}_0) \rightarrow*\{\mathit{cfs}_1\}\ (c', s_1, f, \mathit{vs}_1, \mathit{ws}_1) \Longrightarrow$
$(c', s_1, f, \mathit{vs}_1, \mathit{ws}_1) \rightarrow*\{\mathit{cfs}_2\}\ (c'', s_2, f, \mathit{vs}_2, \mathit{ws}_2) \Longrightarrow$
$\mathit{ok\text{-}flow\text{-}aux}\ U'\ c'\ c''\ s_1\ s_2\ f\ \mathit{vs}_1\ \mathit{vs}_2\ \mathit{ws}_1\ \mathit{ws}_2\ (\mathit{flow}\ \mathit{cfs}_2)$ **and**
D: $s \in \mathit{Univ}\ A\ (\subseteq \mathit{state} \cap X)$ **and**
E: $(\mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2, s, f, \mathit{vs}_0, \mathit{ws}_0) \rightarrow*\{\mathit{cfs}_1\}$
$(c', s_1, f, \mathit{vs}_1, \mathit{ws}_1)$ **and**
F: $(c', s_1, f, \mathit{vs}_1, \mathit{ws}_1) \rightarrow*\{\mathit{cfs}_2\}\ (c'', s_2, f, \mathit{vs}_2, \mathit{ws}_2)$
shows $\mathit{ok\text{-}flow\text{-}aux}\ U\ c'\ c''\ s_1\ s_2\ f\ \mathit{vs}_1\ \mathit{vs}_2\ \mathit{ws}_1\ \mathit{ws}_2\ (\mathit{flow}\ \mathit{cfs}_2)$
proof −
let $?U' = \mathit{insert}\ (\mathit{Univ?}\ A\ X, \mathit{bvars}\ b)\ U$
from A **obtain** B_1 **and** B_2 **and** C_1 **and** C_2 **and** Y_1 **and** Y_2 **where**
G: $\models b\ (\subseteq A, X) = (B_1, B_2)$ **and**
H: $(?U', v) \models c_1\ (\subseteq B_1, X) = \mathit{Some}\ (C_1, Y_1)$ **and**
I: $(?U', v) \models c_2\ (\subseteq B_2, X) = \mathit{Some}\ (C_2, Y_2)$
by (*auto split*: *option.split-asm prod.split-asm*)
have
$(c', s_1, f, \mathit{vs}_1, \mathit{ws}_1) = (\mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2, s, f, \mathit{vs}_0, \mathit{ws}_0)\ \vee$
$\mathit{bval}\ b\ s \wedge (c_1, s, f, \mathit{vs}_0, \mathit{ws}_0) \rightarrow*\{\mathit{tl}\ \mathit{cfs}_1\}\ (c', s_1, f, \mathit{vs}_1, \mathit{ws}_1)\ \vee$
$\neg\ \mathit{bval}\ b\ s \wedge (c_2, s, f, \mathit{vs}_0, \mathit{ws}_0) \rightarrow*\{\mathit{tl}\ \mathit{cfs}_1\}\ (c', s_1, f, \mathit{vs}_1, \mathit{ws}_1)$
(**is** $?P \vee$ -)
using E **by** (*blast dest*: *small-stepsl-if*)
thus *?thesis*
proof (*rule disjE*, *erule-tac* [*2*] *disjE*, *erule-tac* [*2−3*] *conjE*)
assume *?P*
hence $(\mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2, s, f, \mathit{vs}_0, \mathit{ws}_0) \rightarrow*\{\mathit{cfs}_2\}$
$(c'', s_2, f, \mathit{vs}_2, \mathit{ws}_2)$
using F **by** *simp*
hence
$(c'', s_2, f, \mathit{vs}_2, \mathit{ws}_2) = (\mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2, s, f, \mathit{vs}_0, \mathit{ws}_0)\ \wedge$
$\mathit{flow}\ \mathit{cfs}_2 = []\ \vee$
$\mathit{bval}\ b\ s \wedge (c_1, s, f, \mathit{vs}_0, \mathit{ws}_0) \rightarrow*\{\mathit{tl}\ \mathit{cfs}_2\}\ (c'', s_2, f, \mathit{vs}_2, \mathit{ws}_2)\ \wedge$
$\mathit{flow}\ \mathit{cfs}_2 = \langle \mathit{bvars}\ b\rangle\ \#\ \mathit{flow}\ (\mathit{tl}\ \mathit{cfs}_2)\ \vee$
$\neg\ \mathit{bval}\ b\ s \wedge (c_2, s, f, \mathit{vs}_0, \mathit{ws}_0) \rightarrow*\{\mathit{tl}\ \mathit{cfs}_2\}\ (c'', s_2, f, \mathit{vs}_2, \mathit{ws}_2)\ \wedge$
$\mathit{flow}\ \mathit{cfs}_2 = \langle \mathit{bvars}\ b\rangle\ \#\ \mathit{flow}\ (\mathit{tl}\ \mathit{cfs}_2)$
(**is** $?P' \vee$ -)
by (*rule small-stepsl-if*)
thus *?thesis*
proof (*rule disjE*, *erule-tac* [*2*] *disjE*, (*erule-tac* [*2−3*] *conjE*)+)
assume $?P'$
with ‹*?P*› **show** *?thesis*

by *fastforce*
next
assume
J: *bval b s* **and**
K: $(c_1,\ s,\ f,\ vs_0,\ ws_0) \rightarrow*\{tl\ cfs_2\}\ (c'',\ s_2,\ f,\ vs_2,\ ws_2)$ **and**
L: $flow\ cfs_2 = \langle bvars\ b\rangle\ \#\ flow\ (tl\ cfs_2)$
(**is** *?cs* = - # *?cs′*)
have *M*: $s \in Univ\ B_1\ (\subseteq state \cap X)$
using *J* **by** (*insert btyping2-approx* [*OF G D*], *simp*)
have *N*: $(c_1,\ s,\ f,\ vs_0,\ ws_0) \rightarrow*\{[]\}\ (c_1,\ s,\ f,\ vs_0,\ ws_0)$
by *simp*
show *?thesis*
proof (*rule conjI*, *clarify*)
fix $t_1\ f'\ vs_1{}'\ ws_1{}'$
show $\exists\, c_2{}'\ t_2\ vs_2{}'\ ws_2{}'.$
ok-flow-aux-1 $c'\ c''\ c_2{}'\ s_1\ t_1\ t_2\ f\ f'$
$vs_1\ vs_1{}'\ vs_2\ vs_2{}'\ ws_1{}'\ ws_2{}'$ *?cs* ∧
ok-flow-aux-2 $s_1\ s_2\ t_1\ t_2\ f\ f'\ vs_1\ vs_1{}'$ *?cs* ∧
ok-flow-aux-3 $s_1\ t_1\ f\ f'\ vs_1\ vs_1{}'\ ws_1\ ws_1{}'\ ws_2\ ws_2{}'$ *?cs*
proof (*cases bval b* t_1)
assume *O*: *bval b* t_1
have *ok-flow-aux ?U′* $c_1\ c''\ s\ s_2\ f\ vs_0\ vs_2\ ws_0\ ws_2$ *?cs′*
using *B* [*OF - - H M N K*] **and** *G* **by** *simp*
then obtain $c_2{}'$ **and** t_2 **and** $vs_2{}'$ **and** $ws_2{}'$ **where**
ok-flow-aux-1 $c_1\ c''\ c_2{}'\ s\ t_1\ t_2\ f\ f'$
$vs_0\ vs_1{}'\ vs_2\ vs_2{}'\ ws_1{}'\ ws_2{}'$ *?cs′* ∧
ok-flow-aux-2 $s\ s_2\ t_1\ t_2\ f\ f'\ vs_0\ vs_1{}'$ *?cs′* ∧
ok-flow-aux-3 $s\ t_1\ f\ f'\ vs_0\ vs_1{}'\ ws_0\ ws_1{}'\ ws_2\ ws_2{}'$ *?cs′*
(**is** *?P1* ∧ *?P2* ∧ *?P3*)
by *fastforce*
hence *?P1* **and** *?P2* **and** *?P3* **by** *auto*
show *?thesis*
proof (*rule exI* [*of* - $c_2{}'$], *rule exI* [*of* - t_2],
rule exI [*of* - $vs_2{}'$], *rule exI* [*of* - $ws_2{}'$])
{
fix *S*
assume
P: $S \neq \{\}$ **and**
Q: $S \subseteq \{x.\ s_1 = t_1$
$(\subseteq sources\text{-}aux\ (\langle bvars\ b\rangle\ \#\ ?cs')\ vs_1\ s_1\ f\ x)\}$ **and**
R: $f = f'\ (\subseteq vs_1,\ vs_1{}',$
$\bigcup\ \{tags\text{-}aux\ (\langle bvars\ b\rangle\ \#\ ?cs')\ vs_1\ s_1\ f\ x \mid x.\ x \in S\})$
have $\forall\, x.\ sources\text{-}aux\ ?cs'\ vs_1\ s_1\ f\ x \subseteq$
$sources\text{-}aux\ (\langle bvars\ b\rangle\ \#\ ?cs')\ vs_1\ s_1\ f\ x$
by (*blast intro*: *subsetD* [*OF sources-aux-observe-tl*])
hence $S \subseteq \{x.\ s_1 = t_1\ (\subseteq sources\text{-}aux\ ?cs'\ vs_1\ s_1\ f\ x)\}$
using *Q* **by** *blast*
moreover have $f = f'\ (\subseteq vs_1,\ vs_1{}',$
$\bigcup\ \{tags\text{-}aux\ ?cs'\ vs_1\ s_1\ f\ x \mid x.\ x \in S\})$

using R **by** $(\mathit{simp\ add}\colon \mathit{tags\text{-}aux\text{-}observe\text{-}tl})$
ultimately have
$(c_1,\ t_1,\ f',\ \mathit{vs_1}',\ \mathit{ws_1}') \to\!* (c_2',\ t_2,\ f',\ \mathit{vs_2}',\ \mathit{ws_2}') \wedge$
$(c'' = \mathit{SKIP}) = (c_2' = \mathit{SKIP}) \wedge$
$\mathit{map\ fst}\ [p{\leftarrow}\mathit{drop}\ (\mathit{length}\ \mathit{vs_1})\ \mathit{vs_2}.\ \mathit{fst}\ p \in S] =$
$\mathit{map\ fst}\ [p{\leftarrow}\mathit{drop}\ (\mathit{length}\ \mathit{vs_1}')\ \mathit{vs_2}'.\ \mathit{fst}\ p \in S]$
(**is** $?Q1 \wedge ?Q2 \wedge ?Q3$)
using P **and** $\langle ?P \rangle$ **and** $\langle ?P1 \rangle$ **by** *simp*
hence $?Q1$ **and** $?Q2$ **and** $?Q3$ **by** *auto*
moreover have $(\mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2,\ t_1,\ f',\ \mathit{vs_1}',\ \mathit{ws_1}') \to$
$(c_1,\ t_1,\ f',\ \mathit{vs_1}',\ \mathit{ws_1}')$
using O **..**
hence $(c',\ t_1,\ f',\ \mathit{vs_1}',\ \mathit{ws_1}') \to\!* (c_2',\ t_2,\ f',\ \mathit{vs_2}',\ \mathit{ws_2}')$
using $\langle ?P \rangle$ **and** $\langle ?Q1 \rangle$ **by** $(\mathit{blast\ intro}\colon \mathit{star\text{-}trans})$
ultimately have $?\mathit{this} \wedge ?Q2 \wedge ?Q3$
by *simp*
}
moreover {
fix S
assume
$P\colon S \subseteq \{x.\ s_1 = t_1$
$(\subseteq \mathit{sources}\ (\langle \mathit{bvars}\ b\rangle\ \#\ ?\mathit{cs}')\ \mathit{vs_1}\ s_1\ f\ x)\}$ **and**
$Q\colon f = f'\ (\subseteq \mathit{vs_1},\ \mathit{vs_1}',$
$\bigcup\ \{\mathit{tags}\ (\langle \mathit{bvars}\ b\rangle\ \#\ ?\mathit{cs}')\ \mathit{vs_1}\ s_1\ f\ x \mid x.\ x \in S\})$
have $\forall x.\ \mathit{sources}\ ?\mathit{cs}'\ \mathit{vs_1}\ s_1\ f\ x \subseteq$
$\mathit{sources}\ (\langle \mathit{bvars}\ b\rangle\ \#\ ?\mathit{cs}')\ \mathit{vs_1}\ s_1\ f\ x$
by $(\mathit{blast\ intro}\colon \mathit{subsetD}\ [\mathit{OF\ sources\text{-}observe\text{-}tl}])$
hence $S \subseteq \{x.\ s_1 = t_1\ (\subseteq \mathit{sources}\ ?\mathit{cs}'\ \mathit{vs_1}\ s_1\ f\ x)\}$
using P **by** *blast*
moreover have $f = f'\ (\subseteq \mathit{vs_1},\ \mathit{vs_1}',$
$\bigcup\ \{\mathit{tags}\ ?\mathit{cs}'\ \mathit{vs_1}\ s_1\ f\ x \mid x.\ x \in S\})$
using Q **by** $(\mathit{simp\ add}\colon \mathit{tags\text{-}observe\text{-}tl})$
ultimately have $s_2 = t_2\ (\subseteq S)$
using $\langle ?P \rangle$ **and** $\langle ?P2 \rangle$ **by** *blast*
}
moreover {
fix S
assume
$P\colon S \neq \{\}$ **and**
$Q\colon S \subseteq \{x.\ s_1 = t_1$
$(\subseteq \mathit{sources\text{-}out}\ (\langle \mathit{bvars}\ b\rangle\ \#\ ?\mathit{cs}')\ \mathit{vs_1}\ s_1\ f\ x)\}$ **and**
$R\colon f = f'\ (\subseteq \mathit{vs_1},\ \mathit{vs_1}',$
$\bigcup\ \{\mathit{tags\text{-}out}\ (\langle \mathit{bvars}\ b\rangle\ \#\ ?\mathit{cs}')\ \mathit{vs_1}\ s_1\ f\ x \mid x.\ x \in S\})$
have $\forall x.\ \mathit{sources\text{-}out}\ ?\mathit{cs}'\ \mathit{vs_1}\ s_1\ f\ x \subseteq$
$\mathit{sources\text{-}out}\ (\langle \mathit{bvars}\ b\rangle\ \#\ ?\mathit{cs}')\ \mathit{vs_1}\ s_1\ f\ x$
by $(\mathit{blast\ intro}\colon \mathit{subsetD}\ [\mathit{OF\ sources\text{-}out\text{-}observe\text{-}tl}])$
hence $S \subseteq \{x.\ s_1 = t_1\ (\subseteq \mathit{sources\text{-}out}\ ?\mathit{cs}'\ \mathit{vs_1}\ s_1\ f\ x)\}$
using Q **by** *blast*
moreover have $f = f'\ (\subseteq \mathit{vs_1},\ \mathit{vs_1}',$

```
        ⋃ {tags-out ?cs′ vs₁ s₁ f x | x. x ∈ S})
        using R by (simp add: tags-out-observe-tl)
      ultimately have [p←drop (length ws₁) ws₂. fst p ∈ S] =
        [p←drop (length ws₁′) ws₂′. fst p ∈ S]
        using P and ‹?P› and ‹?P3› by simp
    }
    ultimately show
     ok-flow-aux-1 c′ c″ c₂′ s₁ t₁ t₂ f f′
        vs₁ vs₁′ vs₂ vs₂′ ws₁′ ws₂′ ?cs ∧
      ok-flow-aux-2 s₁ s₂ t₁ t₂ f f′ vs₁ vs₁′ ?cs ∧
      ok-flow-aux-3 s₁ t₁ f f′ vs₁ vs₁′ ws₁ ws₁′ ws₂ ws₂′ ?cs
      using L by auto
  qed
next
  assume O: ¬ bval b t₁
  show ?thesis
  proof (cases ∃ S ≠ {}. S ⊆ {x. s₁ = t₁
   (⊆ sources-aux (⟨bvars b⟩ # ?cs′) vs₁ s₁ f x)})
    from O have (IF b THEN c₁ ELSE c₂, t₁, f′, vs₁′, ws₁′) →
      (c₂, t₁, f′, vs₁′, ws₁′) ..
    moreover assume ∃ S ≠ {}. S ⊆ {x. s₁ = t₁
      (⊆ sources-aux (⟨bvars b⟩ # ?cs′) vs₁ s₁ f x)}
    then obtain S where
      P: S ≠ {} and
      Q: S ⊆ {x. s = t₁
        (⊆ sources-aux (⟨bvars b⟩ # ?cs′) vs₁ s f x)}
      using ‹?P› by blast
    have R: Univ? A X: bvars b ⇝| S
      using Q and D by (rule sources-aux-bval, insert J O, simp)
    have ∃ p. (c₂, t₁, f′, vs₁′, ws₁′) ⇒ p
      using I by (rule ctyping2-term, insert P R, auto)
    then obtain t₂ and f″ and vs₂′ and ws₂′ where
      S: (c₂, t₁, f′, vs₁′, ws₁′) →∗ (SKIP, t₂, f″, vs₂′, ws₂′)
      by (auto simp: big-iff-small)
    ultimately have
     (c′, t₁, f′, vs₁′, ws₁′) →∗ (SKIP, t₂, f′, vs₂′, ws₂′)
      (is ?Q1)
      using ‹?P› by (blast dest: small-steps-stream
       intro: star-trans)
    have T: (c₂, t₁, f′, vs₁′, ws₁′) ⇒ (t₂, f″, vs₂′, ws₂′)
      using S by (simp add: big-iff-small)
    show ?thesis
    proof (cases c″ = SKIP)
      assume c″ = SKIP
        (is ?Q2)
      show ?thesis
      proof (rule exI [of - SKIP], rule exI [of - t₂],
       rule exI [of - vs₂′], rule exI [of - ws₂′])
        {
```

```
  fix S
  assume S ⊆ {x. s = t₁
    (⊆ sources-aux (⟨bvars b⟩ # ?cs') vs₁ s f x)}
  hence U: Univ? A X: bvars b ⇝| S
    using D by (rule sources-aux-bval, insert J O, simp)
  hence [p←drop (length vs₁') vs₂'. fst p ∈ S] = []
    using I and T by (blast dest: ctyping2-confine)
  moreover have no-upd ?cs' S
    using B [OF - - H M N K] and G and U by simp
  hence [p←in-flow ?cs' vs₁ f. fst p ∈ S] = []
    by (rule no-upd-in-flow)
  moreover have vs₂ = vs₀ @ in-flow ?cs' vs₀ f
    using K by (rule small-stepsl-in-flow)
  ultimately have [p←drop (length vs₁) vs₂. fst p ∈ S] =
    [p←drop (length vs₁') vs₂'. fst p ∈ S]
    using ‹?P› by simp
  hence ?Q1 ∧ ?Q2 ∧ ?this
    using ‹?Q1› and ‹?Q2› by simp
}
moreover {
  fix S
  assume U: S ⊆ {x. s = t₁
    (⊆ sources (⟨bvars b⟩ # ?cs') vs₁ s f x)}
  moreover have
   ∀x. sources-aux (⟨bvars b⟩ # ?cs') vs₁ s f x ⊆
      sources (⟨bvars b⟩ # ?cs') vs₁ s f x
    by (blast intro: subsetD [OF sources-aux-sources])
  ultimately have S ⊆ {x. s = t₁
    (⊆ sources-aux (⟨bvars b⟩ # ?cs') vs₁ s f x)}
    by blast
  hence V: Univ? A X: bvars b ⇝| S
    using D by (rule sources-aux-bval, insert J O, simp)
  hence t₁ = t₂ (⊆ S)
    using I and T by (blast dest: ctyping2-confine)
  moreover have W: no-upd ?cs' S
    using B [OF - - H M N K] and G and V by simp
  hence run-flow ?cs' vs₀ s f = s (⊆ S)
    by (rule no-upd-run-flow)
  moreover have s₂ = run-flow ?cs' vs₀ s f
    using K by (rule small-stepsl-run-flow)
  moreover have
   ∀x ∈ S. x ∈ sources (⟨bvars b⟩ # ?cs') vs₁ s f x
    by (rule no-upd-sources, simp add: W)
  hence s = t₁ (⊆ S)
    using U by blast
  ultimately have s₂ = t₂ (⊆ S)
    by simp
}
moreover {
```

fix *S*
assume $S \subseteq \{x.\ s = t_1$
($\subseteq$ *sources-out* ($\langle$*bvars b*$\rangle$ # *?cs'*) vs_1 *s f x*)}
moreover have
$\forall\, x.$ *sources-aux* ($\langle$*bvars b*$\rangle$ # *?cs'*) vs_1 *s f x* $\subseteq$
sources-out ($\langle$*bvars b*$\rangle$ # *?cs'*) vs_1 *s f x*
by (*blast intro*: *subsetD* [*OF sources-aux-sources-out*])
ultimately have $S \subseteq \{x.\ s = t_1$
($\subseteq$ *sources-aux* ($\langle$*bvars b*$\rangle$ # *?cs'*) vs_1 *s f x*)}
by *blast*
hence *U*: *Univ? A X*: *bvars b* $\rightsquigarrow|$ *S*
using *D* **by** (*rule sources-aux-bval*, *insert J O*, *simp*)
hence [*p*←*drop* (*length* ws_1') ws_2'. *fst p* $\in$ *S*] = []
using *I* **and** *T* **by** (*blast dest*: *ctyping2-confine*)
moreover have *no-upd ?cs' S*
using *B* [*OF* - - *H M N K*] **and** *G* **and** *U* **by** *simp*
hence [*p*←*out-flow ?cs'* vs_1 *s f*. *fst p* $\in$ *S*] = []
by (*rule no-upd-out-flow*)
moreover have $ws_2 = ws_0$ @ *out-flow ?cs'* vs_0 *s f*
using *K* **by** (*rule small-stepsl-out-flow*)
ultimately have
[*p*←*drop* (*length* ws_1) ws_2. *fst p* $\in$ *S*] =
[*p*←*drop* (*length* ws_1') ws_2'. *fst p* $\in$ *S*]
using ‹*?P*› **by** *simp*
}
ultimately show
ok-flow-aux-1 c' c'' SKIP s_1 t_1 t_2 *f f'*
vs_1 vs_1' vs_2 vs_2' ws_1' ws_2' *?cs* $\wedge$
ok-flow-aux-2 s_1 s_2 t_1 t_2 *f f'* vs_1 vs_1' *?cs* $\wedge$
ok-flow-aux-3 s_1 t_1 *f f'* vs_1 vs_1' ws_1 ws_1' ws_2 ws_2' *?cs*
using *L* **and** ‹*?P*› **by** *auto*
qed
next
assume $c'' \neq$ *SKIP*
(**is** *?Q2*)
show *?thesis*
proof (*rule exI* [*of* - *c'*], *rule exI* [*of* - t_1],
rule exI [*of* - vs_1'], *rule exI* [*of* - ws_1'])
{
fix *S*
assume $S \subseteq \{x.\ s = t_1$
($\subseteq$ *sources-aux* ($\langle$*bvars b*$\rangle$ # *?cs'*) vs_1 *s f x*)}
hence *Univ? A X*: *bvars b* $\rightsquigarrow|$ *S*
using *D* **by** (*rule sources-aux-bval*, *insert J O*, *simp*)
hence *no-upd ?cs' S*
using *B* [*OF* - - *H M N K*] **and** *G* **by** *simp*
hence [*p*←*in-flow ?cs'* vs_1 *f*. *fst p* $\in$ *S*] = []
by (*rule no-upd-in-flow*)
moreover have $vs_2 = vs_0$ @ *in-flow ?cs'* vs_0 *f*

```
    using K by (rule small-stepsl-in-flow)
  ultimately have
   [p←drop (length vs_1) vs_2. fst p ∈ S] = []
    using ‹?P› by simp
  hence ?Q2 ∧ ?this
    using ‹?Q2› by simp
}
moreover {
  fix S
  assume U: S ⊆ {x. s = t_1
    (⊆ sources (⟨bvars b⟩ # ?cs') vs_1 s f x)}
  moreover have
   ∀x. sources-aux (⟨bvars b⟩ # ?cs') vs_1 s f x ⊆
      sources (⟨bvars b⟩ # ?cs') vs_1 s f x
    by (blast intro: subsetD [OF sources-aux-sources])
  ultimately have S ⊆ {x. s = t_1
    (⊆ sources-aux (⟨bvars b⟩ # ?cs') vs_1 s f x)}
    by blast
  hence Univ? A X: bvars b ⇝| S
    using D by (rule sources-aux-bval, insert J O, simp)
  hence V: no-upd ?cs' S
    using B [OF - - H M N K] and G by simp
  hence run-flow ?cs' vs_0 s f = s (⊆ S)
    by (rule no-upd-run-flow)
  moreover have s_2 = run-flow ?cs' vs_0 s f
    using K by (rule small-stepsl-run-flow)
  moreover have
   ∀x ∈ S. x ∈ sources (⟨bvars b⟩ # ?cs') vs_1 s f x
    by (rule no-upd-sources, simp add: V)
  hence s = t_1 (⊆ S)
    using U by blast
  ultimately have s_2 = t_1 (⊆ S)
    by simp
}
moreover {
  fix S
  assume S ⊆ {x. s = t_1
    (⊆ sources-out (⟨bvars b⟩ # ?cs') vs_1 s f x)}
  moreover have
   ∀x. sources-aux (⟨bvars b⟩ # ?cs') vs_1 s f x ⊆
      sources-out (⟨bvars b⟩ # ?cs') vs_1 s f x
    by (blast intro: subsetD [OF sources-aux-sources-out])
  ultimately have S ⊆ {x. s = t_1
    (⊆ sources-aux (⟨bvars b⟩ # ?cs') vs_1 s f x)}
    by blast
  hence Univ? A X: bvars b ⇝| S
    using D by (rule sources-aux-bval, insert J O, simp)
  hence no-upd ?cs' S
    using B [OF - - H M N K] and G by simp
```

```
              hence [p←out-flow ?cs' vs1 s f. fst p ∈ S] = []
                by (rule no-upd-out-flow)
              moreover have ws2 = ws0 @ out-flow ?cs' vs0 s f
                using K by (rule small-stepsl-out-flow)
              ultimately have
               [p←drop (length ws1) ws2. fst p ∈ S] = []
                using ‹?P› by simp
            }
            ultimately show
             ok-flow-aux-1 c' c'' c' s1 t1 t1 f f'
                vs1 vs1' vs2 vs1' ws1' ws1' ?cs ∧
              ok-flow-aux-2 s1 s2 t1 t1 f f' vs1 vs1' ?cs ∧
              ok-flow-aux-3 s1 t1 f f' vs1 vs1' ws1 ws1' ws2 ws1' ?cs
              using L and ‹?P› by auto
          qed
        qed
      next
        assume ∄S. S ≠ {} ∧ S ⊆ {x. s1 = t1
          (⊆ sources-aux (⟨bvars b⟩ # ?cs') vs1 s1 f x)}
        hence O: ∀c2' t2 vs2' ws2'.
          ok-flow-aux-1 c' c'' c2' s1 t1 t2 f f'
            vs1 vs1' vs2 vs2' ws1' ws2' ?cs ∧
          ok-flow-aux-2 s1 s2 t1 t2 f f' vs1 vs1' ?cs ∧
          ok-flow-aux-3 s1 t1 f f' vs1 vs1' ws1 ws1' ws2 ws2' ?cs
          using L by (auto intro!: ok-flow-aux-degen)
        show ?thesis
          by (rule exI [of - SKIP], rule exI [of - λx. 0],
           rule exI [of - []], rule exI [of - []],
           simp add: O [rule-format, of SKIP λx. 0 [] []])
      qed
    qed
  qed (simp add: B [OF - - H M N K] G L)
next
  assume
    J: ¬ bval b s and
    K: (c2, s, f, vs0, ws0) →*{tl cfs2} (c'', s2, f, vs2, ws2) and
    L: flow cfs2 = ⟨bvars b⟩ # flow (tl cfs2)
      (is ?cs = - # ?cs')
  have M: s ∈ Univ B2 (⊆ state ∩ X)
    using J by (insert btyping2-approx [OF G D], simp)
  have N: (c2, s, f, vs0, ws0) →*{[]} (c2, s, f, vs0, ws0)
    by simp
  show ?thesis
  proof (rule conjI, clarify)
    fix t1 f' vs1' ws1'
    show ∃c2' t2 vs2' ws2'.
      ok-flow-aux-1 c' c'' c2' s1 t1 t2 f f'
        vs1 vs1' vs2 vs2' ws1' ws2' ?cs ∧
      ok-flow-aux-2 s1 s2 t1 t2 f f' vs1 vs1' ?cs ∧
```

```
  ok-flow-aux-3 s1 t1 f f' vs1 vs1' ws1 ws1' ws2 ws2' ?cs
proof (cases bval b t1, cases ∃ S ≠ {}.
 S ⊆ {x. s1 = t1 (⊆ sources-aux (⟨bvars b⟩ # ?cs') vs1 s1 f x)})
  assume O: ¬ bval b t1
  have ok-flow-aux ?U' c2 c'' s s2 f vs0 vs2 ws0 ws2 ?cs'
    using C [OF - - I M N K] and G by simp
  then obtain c2' and t2 and vs2' and ws2' where
   ok-flow-aux-1 c2 c'' c2' s t1 t2 f f'
     vs0 vs1' vs2 vs2' ws1' ws2' ?cs' ∧
    ok-flow-aux-2 s s2 t1 t2 f f' vs0 vs1' ?cs' ∧
    ok-flow-aux-3 s t1 f f' vs0 vs1' ws0 ws1' ws2 ws2' ?cs'
    (is ?P1 ∧ ?P2 ∧ ?P3)
    by fastforce
  hence ?P1 and ?P2 and ?P3 by auto
  show ?thesis
  proof (rule exI [of - c2'], rule exI [of - t2],
   rule exI [of - vs2'], rule exI [of - ws2'])
    {
      fix S
      assume
        P: S ≠ {} and
        Q: S ⊆ {x. s1 = t1
          (⊆ sources-aux (⟨bvars b⟩ # ?cs') vs1 s1 f x)} and
        R: f = f' (⊆ vs1, vs1',
          ⋃ {tags-aux (⟨bvars b⟩ # ?cs') vs1 s1 f x | x. x ∈ S})
      have ∀ x. sources-aux ?cs' vs1 s1 f x ⊆
        sources-aux (⟨bvars b⟩ # ?cs') vs1 s1 f x
        by (blast intro: subsetD [OF sources-aux-observe-tl])
      hence S ⊆ {x. s1 = t1 (⊆ sources-aux ?cs' vs1 s1 f x)}
        using Q by blast
      moreover have f = f' (⊆ vs1, vs1',
        ⋃ {tags-aux ?cs' vs1 s1 f x | x. x ∈ S})
        using R by (simp add: tags-aux-observe-tl)
      ultimately have
       (c2, t1, f', vs1', ws1') →* (c2', t2, f', vs2', ws2') ∧
        (c'' = SKIP) = (c2' = SKIP) ∧
        map fst [p←drop (length vs1) vs2. fst p ∈ S] =
          map fst [p←drop (length vs1') vs2'. fst p ∈ S]
        (is ?Q1 ∧ ?Q2 ∧ ?Q3)
        using P and ‹?P› and ‹?P1› by simp
      hence ?Q1 and ?Q2 and ?Q3 by auto
      moreover have (IF b THEN c1 ELSE c2, t1, f', vs1', ws1') →
        (c2, t1, f', vs1', ws1')
        using O ..
      hence (c', t1, f', vs1', ws1') →* (c2', t2, f', vs2', ws2')
        using ‹?P› and ‹?Q1› by (blast intro: star-trans)
      ultimately have ?this ∧ ?Q2 ∧ ?Q3
        by simp
    }
```

```
      moreover {
        fix S
        assume
          P: S ⊆ {x. s1 = t1
            (⊆ sources (⟨bvars b⟩ # ?cs') vs1 s1 f x)} and
          Q: f = f' (⊆ vs1, vs1',
            ⋃ {tags (⟨bvars b⟩ # ?cs') vs1 s1 f x | x. x ∈ S})
        have ∀x. sources ?cs' vs1 s1 f x ⊆
          sources (⟨bvars b⟩ # ?cs') vs1 s1 f x
          by (blast intro: subsetD [OF sources-observe-tl])
        hence S ⊆ {x. s1 = t1 (⊆ sources ?cs' vs1 s1 f x)}
          using P by blast
        moreover have f = f' (⊆ vs1, vs1',
          ⋃ {tags ?cs' vs1 s1 f x | x. x ∈ S})
          using Q by (simp add: tags-observe-tl)
        ultimately have s2 = t2 (⊆ S)
          using ‹?P› and ‹?P2› by blast
      }
      moreover {
        fix S
        assume
          P: S ≠ {} and
          Q: S ⊆ {x. s1 = t1
            (⊆ sources-out (⟨bvars b⟩ # ?cs') vs1 s1 f x)} and
          R: f = f' (⊆ vs1, vs1',
            ⋃ {tags-out (⟨bvars b⟩ # ?cs') vs1 s1 f x | x. x ∈ S})
        have ∀x. sources-out ?cs' vs1 s1 f x ⊆
          sources-out (⟨bvars b⟩ # ?cs') vs1 s1 f x
          by (blast intro: subsetD [OF sources-out-observe-tl])
        hence S ⊆ {x. s1 = t1 (⊆ sources-out ?cs' vs1 s1 f x)}
          using Q by blast
        moreover have f = f' (⊆ vs1, vs1',
          ⋃ {tags-out ?cs' vs1 s1 f x | x. x ∈ S})
          using R by (simp add: tags-out-observe-tl)
        ultimately have [p←drop (length ws1) ws2. fst p ∈ S] =
          [p←drop (length ws1') ws2'. fst p ∈ S]
          using P and ‹?P› and ‹?P3› by simp
      }
      ultimately show
       ok-flow-aux-1 c' c'' c2' s1 t1 t2 f f'
          vs1 vs1' vs2 vs2' ws1' ws2' ?cs ∧
        ok-flow-aux-2 s1 s2 t1 t2 f f' vs1 vs1' ?cs ∧
        ok-flow-aux-3 s1 t1 f f' vs1 vs1' ws1 ws1' ws2 ws2' ?cs
        using L by auto
    qed
  next
    assume O: bval b t1
    hence (IF b THEN c1 ELSE c2, t1, f', vs1', ws1') →
      (c1, t1, f', vs1', ws1') ..
```

```
moreover assume ∃ S ≠ {}.
  S ⊆ {x. s_1 = t_1 (⊆ sources-aux (⟨bvars b⟩ # ?cs′) vs_1 s_1 f x)}
then obtain S where
  P: S ≠ {} and
  Q: S ⊆ {x. s = t_1 (⊆ sources-aux (⟨bvars b⟩ # ?cs′) vs_1 s f x)}
  using ‹?P› by blast
have R: Univ? A X: bvars b ⇝| S
  using Q and D by (rule sources-aux-bval, insert J O, simp)
have ∃ p. (c_1, t_1, f′, vs_1′, ws_1′) ⇒ p
  using H by (rule ctyping2-term, insert P R, auto)
then obtain t_2 and f″ and vs_2′ and ws_2′ where
  S: (c_1, t_1, f′, vs_1′, ws_1′) →* (SKIP, t_2, f″, vs_2′, ws_2′)
  by (auto simp: big-iff-small)
ultimately have
 (c′, t_1, f′, vs_1′, ws_1′) →* (SKIP, t_2, f′, vs_2′, ws_2′)
  (is ?Q1)
  using ‹?P› by (blast dest: small-steps-stream intro: star-trans)
have T: (c_1, t_1, f′, vs_1′, ws_1′) ⇒ (t_2, f″, vs_2′, ws_2′)
  using S by (simp add: big-iff-small)
show ?thesis
proof (cases c″ = SKIP)
  assume c″ = SKIP
    (is ?Q2)
  show ?thesis
  proof (rule exI [of - SKIP], rule exI [of - t_2],
   rule exI [of - vs_2′], rule exI [of - ws_2′])
    {
      fix S
      assume S ⊆ {x. s = t_1
        (⊆ sources-aux (⟨bvars b⟩ # ?cs′) vs_1 s f x)}
      hence U: Univ? A X: bvars b ⇝| S
        using D by (rule sources-aux-bval, insert J O, simp)
      hence [p←drop (length vs_1′) vs_2′. fst p ∈ S] = []
        using H and T by (blast dest: ctyping2-confine)
      moreover have no-upd ?cs′ S
        using C [OF - - I M N K] and G and U by simp
      hence [p←in-flow ?cs′ vs_1 f. fst p ∈ S] = []
        by (rule no-upd-in-flow)
      moreover have vs_2 = vs_0 @ in-flow ?cs′ vs_0 f
        using K by (rule small-stepsl-in-flow)
      ultimately have [p←drop (length vs_1) vs_2. fst p ∈ S] =
        [p←drop (length vs_1′) vs_2′. fst p ∈ S]
        using ‹?P› by simp
      hence ?Q1 ∧ ?Q2 ∧ ?this
        using ‹?Q1› and ‹?Q2› by simp
    }
    moreover {
      fix S
      assume U: S ⊆ {x. s = t_1
```

(⊆ *sources* (⟨*bvars b*⟩ # *?cs'*) vs_1 *s f x*)}
moreover have ∀ *x*. *sources-aux* (⟨*bvars b*⟩ # *?cs'*) vs_1 *s f x* ⊆
sources (⟨*bvars b*⟩ # *?cs'*) vs_1 *s f x*
by (*blast intro*: *subsetD* [*OF sources-aux-sources*])
ultimately have $S \subseteq \{x.\ s = t_1$
(⊆ *sources-aux* (⟨*bvars b*⟩ # *?cs'*) vs_1 *s f x*)}
by *blast*
hence *V*: *Univ? A X*: *bvars b* ⇝| *S*
using *D* **by** (*rule sources-aux-bval*, *insert J O*, *simp*)
hence $t_1 = t_2$ (⊆ *S*)
using *H* **and** *T* **by** (*blast dest*: *ctyping2-confine*)
moreover have *W*: *no-upd ?cs' S*
using *C* [*OF* - - *I M N K*] **and** *G* **and** *V* **by** *simp*
hence *run-flow ?cs'* vs_0 *s f* = *s* (⊆ *S*)
by (*rule no-upd-run-flow*)
moreover have s_2 = *run-flow ?cs'* vs_0 *s f*
using *K* **by** (*rule small-stepsl-run-flow*)
moreover have ∀ *x* ∈ *S*. *x* ∈ *sources* (⟨*bvars b*⟩ # *?cs'*) vs_1 *s f x*
by (*rule no-upd-sources*, *simp add*: *W*)
hence $s = t_1$ (⊆ *S*)
using *U* **by** *blast*
ultimately have $s_2 = t_2$ (⊆ *S*)
by *simp*
}
moreover {
fix *S*
assume $S \subseteq \{x.\ s = t_1$
(⊆ *sources-out* (⟨*bvars b*⟩ # *?cs'*) vs_1 *s f x*)}
moreover have ∀ *x*. *sources-aux* (⟨*bvars b*⟩ # *?cs'*) vs_1 *s f x* ⊆
sources-out (⟨*bvars b*⟩ # *?cs'*) vs_1 *s f x*
by (*blast intro*: *subsetD* [*OF sources-aux-sources-out*])
ultimately have $S \subseteq \{x.\ s = t_1$
(⊆ *sources-aux* (⟨*bvars b*⟩ # *?cs'*) vs_1 *s f x*)}
by *blast*
hence *U*: *Univ? A X*: *bvars b* ⇝| *S*
using *D* **by** (*rule sources-aux-bval*, *insert J O*, *simp*)
hence [*p*←*drop* (*length* ws_1') ws_2'. *fst p* ∈ *S*] = []
using *H* **and** *T* **by** (*blast dest*: *ctyping2-confine*)
moreover have *no-upd ?cs' S*
using *C* [*OF* - - *I M N K*] **and** *G* **and** *U* **by** *simp*
hence [*p*←*out-flow ?cs'* vs_1 *s f*. *fst p* ∈ *S*] = []
by (*rule no-upd-out-flow*)
moreover have $ws_2 = ws_0$ @ *out-flow ?cs'* vs_0 *s f*
using *K* **by** (*rule small-stepsl-out-flow*)
ultimately have
[*p*←*drop* (*length* ws_1) ws_2. *fst p* ∈ *S*] =
[*p*←*drop* (*length* ws_1') ws_2'. *fst p* ∈ *S*]
using ‹*?P*› **by** *simp*
}

```
    ultimately show
     ok-flow-aux-1 c' c'' SKIP s1 t1 t2 f f'
        vs1 vs1' vs2 vs2' ws1' ws2' ?cs ∧
      ok-flow-aux-2 s1 s2 t1 t2 f f' vs1 vs1' ?cs ∧
      ok-flow-aux-3 s1 t1 f f' vs1 vs1' ws1 ws1' ws2 ws2' ?cs
      using L and ‹?P› by auto
  qed
next
  assume c'' ≠ SKIP
    (is ?Q2)
  show ?thesis
  proof (rule exI [of - c'], rule exI [of - t1],
   rule exI [of - vs1'], rule exI [of - ws1'])
    {
      fix S
      assume S ⊆ {x. s = t1
        (⊆ sources-aux (⟨bvars b⟩ # ?cs') vs1 s f x)}
      hence Univ? A X: bvars b ↝| S
        using D by (rule sources-aux-bval, insert J O, simp)
      hence no-upd ?cs' S
        using C [OF - - I M N K] and G by simp
      hence [p←in-flow ?cs' vs1 f. fst p ∈ S] = []
        by (rule no-upd-in-flow)
      moreover have vs2 = vs0 @ in-flow ?cs' vs0 f
        using K by (rule small-stepsl-in-flow)
      ultimately have [p←drop (length vs1) vs2. fst p ∈ S] = []
        using ‹?P› by simp
      hence ?Q2 ∧ ?this
        using ‹?Q2› by simp
    }
    moreover {
      fix S
      assume U: S ⊆ {x. s = t1
        (⊆ sources (⟨bvars b⟩ # ?cs') vs1 s f x)}
      moreover have ∀x. sources-aux (⟨bvars b⟩ # ?cs') vs1 s f x ⊆
        sources (⟨bvars b⟩ # ?cs') vs1 s f x
        by (blast intro: subsetD [OF sources-aux-sources])
      ultimately have S ⊆ {x. s = t1
        (⊆ sources-aux (⟨bvars b⟩ # ?cs') vs1 s f x)}
        by blast
      hence Univ? A X: bvars b ↝| S
        using D by (rule sources-aux-bval, insert J O, simp)
      hence V: no-upd ?cs' S
        using C [OF - - I M N K] and G by simp
      hence run-flow ?cs' vs0 s f = s (⊆ S)
        by (rule no-upd-run-flow)
      moreover have s2 = run-flow ?cs' vs0 s f
        using K by (rule small-stepsl-run-flow)
      moreover have ∀x ∈ S. x ∈ sources (⟨bvars b⟩ # ?cs') vs1 s f x
```

```
          by (rule no-upd-sources, simp add: V)
        hence s = t_1 (⊆ S)
          using U by blast
        ultimately have s_2 = t_1 (⊆ S)
          by simp
      }
      moreover {
        fix S
        assume S ⊆ {x. s = t_1
          (⊆ sources-out (⟨bvars b⟩ # ?cs') vs_1 s f x)}
        moreover have ∀x. sources-aux (⟨bvars b⟩ # ?cs') vs_1 s f x ⊆
          sources-out (⟨bvars b⟩ # ?cs') vs_1 s f x
          by (blast intro: subsetD [OF sources-aux-sources-out])
        ultimately have S ⊆ {x. s = t_1
          (⊆ sources-aux (⟨bvars b⟩ # ?cs') vs_1 s f x)}
          by blast
        hence Univ? A X: bvars b ⇝| S
          using D by (rule sources-aux-bval, insert J O, simp)
        hence no-upd ?cs' S
          using C [OF - - I M N K] and G by simp
        hence [p←out-flow ?cs' vs_1 s f. fst p ∈ S] = []
          by (rule no-upd-out-flow)
        moreover have ws_2 = ws_0 @ out-flow ?cs' vs_0 s f
          using K by (rule small-stepsl-out-flow)
        ultimately have [p←drop (length ws_1) ws_2. fst p ∈ S] = []
          using ‹?P› by simp
      }
      ultimately show
       ok-flow-aux-1 c' c'' c' s_1 t_1 t_1 f f'
          vs_1 vs_1' vs_2 vs_1' ws_1' ws_1' ?cs ∧
        ok-flow-aux-2 s_1 s_2 t_1 t_1 f f' vs_1 vs_1' ?cs ∧
        ok-flow-aux-3 s_1 t_1 f f' vs_1 vs_1' ws_1 ws_1' ws_2 ws_1' ?cs
        using L and ‹?P› by auto
    qed
  qed
next
  assume ∄S. S ≠ {} ∧
    S ⊆ {x. s_1 = t_1 (⊆ sources-aux (⟨bvars b⟩ # ?cs') vs_1 s_1 f x)}
  hence O: ∀c_2' t_2 vs_2' ws_2'.
    ok-flow-aux-1 c' c'' c_2' s_1 t_1 t_2 f f'
      vs_1 vs_1' vs_2 vs_2' ws_1' ws_2' ?cs ∧
    ok-flow-aux-2 s_1 s_2 t_1 t_2 f f' vs_1 vs_1' ?cs ∧
    ok-flow-aux-3 s_1 t_1 f f' vs_1 vs_1' ws_1 ws_1' ws_2 ws_2' ?cs
    using L by (auto intro!: ok-flow-aux-degen)
  show ?thesis
    by (rule exI [of - SKIP], rule exI [of - λx. 0],
     rule exI [of - []], rule exI [of - []],
     simp add: O [rule-format, of SKIP λx. 0 [] []])
qed
```

```
      qed (simp add: C [OF - - I M N K] G L)
    qed
  next
    assume bval b s
    hence J: $s \in Univ\ B_1\ (\subseteq state \cap X)$
      by (insert btyping2-approx [OF G D], simp)
    assume K: $(c_1, s, f, vs_0, ws_0) \rightarrow*\{tl\ cfs_1\}\ (c', s_1, f, vs_1, ws_1)$
    show ?thesis
      using B [OF - - H J K F] and G by simp
  next
    assume ¬ bval b s
    hence J: $s \in Univ\ B_2\ (\subseteq state \cap X)$
      by (insert btyping2-approx [OF G D], simp)
    assume K: $(c_2, s, f, vs_0, ws_0) \rightarrow*\{tl\ cfs_1\}\ (c', s_1, f, vs_1, ws_1)$
    show ?thesis
      using C [OF - - I J K F] and G by simp
  qed
qed

lemma ctyping2-correct-aux-while:
  assumes
    A: (U, v) ⊨ WHILE b DO c (⊆ A, X) = Some (B, W) and
    B: $\bigwedge B_1\ B_2\ C\ Y\ B_1'\ B_2'\ D\ Z\ c_1\ c_2\ s\ s_1\ s_2\ vs_0\ vs_1\ vs_2\ ws_0\ ws_1\ ws_2\ cfs_1\ cfs_2.$
      $(B_1, B_2) = \models b\ (\subseteq A, X) \Longrightarrow$
      $(C, Y) = \vdash c\ (\subseteq B_1, X) \Longrightarrow$
      $(B_1', B_2') = \models b\ (\subseteq C, Y) \Longrightarrow$
      ∀(B, W) ∈ insert (Univ? A X ∪ Univ? C Y, bvars b) U.
        B: W ⇝ UNIV ⟹
      $(\{\}, False) \models c\ (\subseteq B_1, X) = Some\ (D, Z) \Longrightarrow$
      $s \in Univ\ B_1\ (\subseteq state \cap X) \Longrightarrow$
      $(c, s, f, vs_0, ws_0) \rightarrow*\{cfs_1\}\ (c_1, s_1, f, vs_1, ws_1) \Longrightarrow$
      $(c_1, s_1, f, vs_1, ws_1) \rightarrow*\{cfs_2\}\ (c_2, s_2, f, vs_2, ws_2) \Longrightarrow$
        $ok\text{-}flow\text{-}aux\ \{\}\ c_1\ c_2\ s_1\ s_2\ f\ vs_1\ vs_2\ ws_1\ ws_2\ (flow\ cfs_2)$ and
    C: $\bigwedge B_1\ B_2\ C\ Y\ B_1'\ B_2'\ D'\ Z'\ c_1\ c_2\ s\ s_1\ s_2\ vs_0\ vs_1\ vs_2\ ws_0\ ws_1\ ws_2\ cfs_1\ cfs_2.$
      $(B_1, B_2) = \models b\ (\subseteq A, X) \Longrightarrow$
      $(C, Y) = \vdash c\ (\subseteq B_1, X) \Longrightarrow$
      $(B_1', B_2') = \models b\ (\subseteq C, Y) \Longrightarrow$
      ∀(B, W) ∈ insert (Univ? A X ∪ Univ? C Y, bvars b) U.
        B: W ⇝ UNIV ⟹
      $(\{\}, False) \models c\ (\subseteq B_1', Y) = Some\ (D', Z') \Longrightarrow$
      $s \in Univ\ B_1'\ (\subseteq state \cap Y) \Longrightarrow$
      $(c, s, f, vs_0, ws_0) \rightarrow*\{cfs_1\}\ (c_1, s_1, f, vs_1, ws_1) \Longrightarrow$
      $(c_1, s_1, f, vs_1, ws_1) \rightarrow*\{cfs_2\}\ (c_2, s_2, f, vs_2, ws_2) \Longrightarrow$
        $ok\text{-}flow\text{-}aux\ \{\}\ c_1\ c_2\ s_1\ s_2\ f\ vs_1\ vs_2\ ws_1\ ws_2\ (flow\ cfs_2)$ and
    D: s ∈ Univ A (⊆ state ∩ X) and
    E: $(WHILE\ b\ DO\ c, s, f, vs_0, ws_0) \rightarrow*\{cfs_1\}\ (c_1, s_1, f, vs_1, ws_1)$ and
    F: $(c_1, s_1, f, vs_1, ws_1) \rightarrow*\{cfs_2\}\ (c_2, s_2, f, vs_2, ws_2)$
 shows $ok\text{-}flow\text{-}aux\ U\ c_1\ c_2\ s_1\ s_2\ f\ vs_1\ vs_2\ ws_1\ ws_2\ (flow\ cfs_2)$
proof –
```

```
from A obtain B_1 B_2 C Y B_1' B_2' D Z D' Z' where
  G: ⊨ b (⊆ A, X) = (B_1, B_2) and
  H: ⊢ c (⊆ B_1, X) = (C, Y) and
  I: ⊨ b (⊆ C, Y) = (B_1', B_2') and
  J: ({}, False) ⊨ c (⊆ B_1, X) = Some (D, Z) and
  K: ({}, False) ⊨ c (⊆ B_1', Y) = Some (D', Z') and
  L: ∀ (B, W) ∈ insert (Univ? A X ∪ Univ? C Y, bvars b) U. B: W ⇝ UNIV
  by (fastforce split: if-split-asm option.split-asm prod.split-asm)
from UnI1 [OF D, of Univ C (⊆ state ∩ Y)] and E and F show ?thesis
proof (induction cfs_1 @ cfs_2 arbitrary: cfs_1 cfs_2 s vs_0 ws_0 c_1 s_1 vs_1 ws_1
 rule: length-induct)
  fix cfs_1 cfs_2 s vs_0 ws_0 c_1 s_1 vs_1 ws_1
  assume
    M: ∀ cfs. length cfs < length (cfs_1 @ cfs_2) ⟶
      (∀ cfs' cfs''. cfs = cfs' @ cfs'' ⟶
        (∀ s. s ∈ Univ A (⊆ state ∩ X) ∪ Univ C (⊆ state ∩ Y) ⟶
          (∀ vs_0 ws_0 c_1 s_1 vs_1 ws_1.
            (WHILE b DO c, s, f, vs_0, ws_0) →*{cfs'} (c_1, s_1, f, vs_1, ws_1) ⟶
            (c_1, s_1, f, vs_1, ws_1) →*{cfs''} (c_2, s_2, f, vs_2, ws_2) ⟶
              ok-flow-aux U c_1 c_2 s_1 s_2 f vs_1 vs_2 ws_1 ws_2 (flow cfs'')))) and
    N: s ∈ Univ A (⊆ state ∩ X) ∪ Univ C (⊆ state ∩ Y) and
    O: (c_1, s_1, f, vs_1, ws_1) →*{cfs_2} (c_2, s_2, f, vs_2, ws_2)
  assume (WHILE b DO c, s, f, vs_0, ws_0) →*{cfs_1} (c_1, s_1, f, vs_1, ws_1)
  hence
   (c_1, s_1, f, vs_1, ws_1) = (WHILE b DO c, s, f, vs_0, ws_0) ∧
      flow cfs_1 = [] ∨
    bval b s ∧
      (c;; WHILE b DO c, s, f, vs_0, ws_0) →*{tl cfs_1} (c_1, s_1, f, vs_1, ws_1) ∧
      flow cfs_1 = ⟨bvars b⟩ # flow (tl cfs_1) ∨
    ¬ bval b s ∧
      (c_1, s_1, f, vs_1, ws_1) = (SKIP, s, f, vs_0, ws_0) ∧
      flow cfs_1 = [⟨bvars b⟩]
    by (rule small-stepsl-while)
  thus ok-flow-aux U c_1 c_2 s_1 s_2 f vs_1 vs_2 ws_1 ws_2 (flow cfs_2)
  proof (rule disjE, erule-tac [2] disjE, erule-tac conjE,
   (erule-tac [2−3] conjE)+)
    assume P: (c_1, s_1, f, vs_1, ws_1) = (WHILE b DO c, s, f, vs_0, ws_0)
    hence (WHILE b DO c, s, f, vs_0, ws_0) →*{cfs_2} (c_2, s_2, f, vs_2, ws_2)
      using O by simp
    hence
     (c_2, s_2, f, vs_2, ws_2) = (WHILE b DO c, s, f, vs_0, ws_0) ∧
        flow cfs_2 = [] ∨
      bval b s ∧
        (c;; WHILE b DO c, s, f, vs_0, ws_0) →*{tl cfs_2}
          (c_2, s_2, f, vs_2, ws_2) ∧
        flow cfs_2 = ⟨bvars b⟩ # flow (tl cfs_2) ∨
      ¬ bval b s ∧
        (c_2, s_2, f, vs_2, ws_2) = (SKIP, s, f, vs_0, ws_0) ∧
        flow cfs_2 = [⟨bvars b⟩]
```

by (*rule small-stepsl-while*)

thus *?thesis*

proof (*rule disjE*, *erule-tac* [*2*] *disjE*, *erule-tac conjE*, (*erule-tac* [*2*−*3*] *conjE*)+)

assume

$(c_2, s_2, f, vs_2, ws_2) = (WHILE\ b\ DO\ c, s, f, vs_0, ws_0)$ **and**

flow cfs_2 = []

thus *?thesis*

using *P* **by** *fastforce*

next

assume

Q: *bval b s* **and**

R: *flow* cfs_2 = ⟨*bvars b*⟩ # *flow* (*tl* cfs_2)

(**is** *?cs* = - # *?cs′*)

assume (*c*;; *WHILE b DO c*, *s*, *f*, vs_0, ws_0) →∗{*tl* cfs_2}

(c_2, s_2, *f*, vs_2, ws_2)

hence

(∃ *c′ cfs*.

c_2 = *c′*;; *WHILE b DO c* ∧

(*c*, *s*, *f*, vs_0, ws_0) →∗{*cfs*} (*c′*, s_2, *f*, vs_2, ws_2) ∧

?cs′ = *flow cfs*) ∨

(∃ *p cfs′ cfs″*.

length cfs″ < *length* (*tl* cfs_2) ∧

(*c*, *s*, *f*, vs_0, ws_0) →∗{*cfs′*} (*SKIP*, *p*) ∧

(*WHILE b DO c*, *p*) →∗{*cfs″*} (c_2, s_2, *f*, vs_2, ws_2) ∧

?cs′ = *flow cfs′* @ *flow cfs″*)

by (*rule small-stepsl-seq*)

thus *?thesis*

apply (*rule disjE*)

apply (*erule exE*)+

apply (*erule conjE*)+

subgoal for *c′ cfs*

proof −

assume

S: c_2 = *c′*;; *WHILE b DO c* **and**

T: (*c*, *s*, *f*, vs_0, ws_0) →∗{*cfs*} (*c′*, s_2, *f*, vs_2, ws_2) **and**

U: *?cs′* = *flow cfs*

have *V*: (*c*, *s*, *f*, vs_0, ws_0) →∗{[]} (*c*, *s*, *f*, vs_0, ws_0)

by *simp*

from *N* **have**

ok-flow-aux {} *c c′ s* s_2 *f* vs_0 vs_2 ws_0 ws_2 (*flow cfs*)

proof

assume *W*: *s* ∈ *Univ A* (⊆ *state* ∩ *X*)

have *X*: *s* ∈ *Univ* B_1 (⊆ *state* ∩ *X*)

using *Q* **by** (*insert btyping2-approx* [*OF G W*], *simp*)

show *?thesis*

by (*rule B* [*OF G* [*symmetric*] *H* [*symmetric*] *I* [*symmetric*] *L J X V T*])

next

```
  assume W: s ∈ Univ C (⊆ state ∩ Y)
  have X: s ∈ Univ B1' (⊆ state ∩ Y)
    using Q by (insert btyping2-approx [OF I W], simp)
  show ?thesis
    by (rule C [OF G [symmetric] H [symmetric] I [symmetric]
     L K X V T])
qed
hence W: ok-flow-aux {} c c' s1 s2 f vs1 vs2 ws1 ws2 ?cs'
  using P and U by simp
show ?thesis
proof (rule conjI, clarify)
  fix t1 f' vs1' ws1'
  obtain c2' and t2 and vs2' and ws2' where
   ok-flow-aux-1 c c' c2' s1 t1 t2 f f'
      vs1 vs1' vs2 vs2' ws1' ws2' ?cs' ∧
    ok-flow-aux-2 s1 s2 t1 t2 f f' vs1 vs1' ?cs' ∧
    ok-flow-aux-3 s1 t1 f f' vs1 vs1' ws1 ws1' ws2 ws2' ?cs'
    (is ?P1 ∧ ?P2 ∧ ?P3)
    using W by fastforce
  hence ?P1 and ?P2 and ?P3 by auto
  show ∃ c2' t2 vs2' ws2'.
    ok-flow-aux-1 c1 c2 c2' s1 t1 t2 f f'
      vs1 vs1' vs2 vs2' ws1' ws2' ?cs ∧
    ok-flow-aux-2 s1 s2 t1 t2 f f' vs1 vs1' ?cs ∧
    ok-flow-aux-3 s1 t1 f f' vs1 vs1' ws1 ws1' ws2 ws2' ?cs
  proof (rule exI [of - c2';; WHILE b DO c], rule exI [of - t2],
   rule exI [of - vs2'], rule exI [of - ws2'])
    {
      fix S
      assume
        X: S ≠ {} and
        Y: S ⊆ {x. s1 = t1
          (⊆ sources-aux (⟨bvars b⟩ # ?cs') vs1 s1 f x)} and
        Z: f = f' (⊆ vs1, vs1',
          ⋃ {tags-aux (⟨bvars b⟩ # ?cs') vs1 s1 f x | x. x ∈ S})
      have ∀ x. sources-aux ?cs' vs1 s1 f x ⊆
        sources-aux (⟨bvars b⟩ # ?cs') vs1 s1 f x
        by (blast intro: subsetD [OF sources-aux-observe-tl])
      hence S ⊆ {x. s1 = t1 (⊆ sources-aux ?cs' vs1 s1 f x)}
        using Y by blast
      moreover have f = f' (⊆ vs1, vs1',
        ⋃ {tags-aux ?cs' vs1 s1 f x | x. x ∈ S})
        using Z by (simp add: tags-aux-observe-tl)
      ultimately have
       (c, t1, f', vs1', ws1') →* (c2', t2, f', vs2', ws2') ∧
        map fst [p←drop (length vs1) vs2. fst p ∈ S] =
          map fst [p←drop (length vs1') vs2'. fst p ∈ S]
        (is ?Q1 ∧ ?Q2)
        using X and ‹?P1› by simp
```

```
 hence ?Q1 and ?Q2 by auto
 have s1 = t1 (⊆ bvars b)
  by (rule eq-states-while [OF Y X], insert L N P, simp+)
 hence bval b t1
  using P and Q by (blast dest: bvars-bval)
 hence (WHILE b DO c, t1, f', vs1', ws1') →
  (c;; WHILE b DO c, t1, f', vs1', ws1') ..
 hence (c1, t1, f', vs1', ws1') →*
  (c2';; WHILE b DO c, t2, f', vs2', ws2')
  using P and ‹?Q1› by (blast intro: star-seq2 star-trans)
 hence ?this ∧ ?Q2
  using ‹?Q2› by simp
}
moreover {
 fix S
 assume
  X: S ⊆ {x. s1 = t1
   (⊆ sources (⟨bvars b⟩ # ?cs') vs1 s1 f x)} and
  Y: f = f' (⊆ vs1, vs1',
   ⋃ {tags (⟨bvars b⟩ # ?cs') vs1 s1 f x | x. x ∈ S})
 have ∀ x. sources ?cs' vs1 s1 f x ⊆
  sources (⟨bvars b⟩ # ?cs') vs1 s1 f x
  by (blast intro: subsetD [OF sources-observe-tl])
 hence S ⊆ {x. s1 = t1 (⊆ sources ?cs' vs1 s1 f x)}
  using X by blast
 moreover have f = f' (⊆ vs1, vs1',
  ⋃ {tags ?cs' vs1 s1 f x | x. x ∈ S})
  using Y by (simp add: tags-observe-tl)
 ultimately have s2 = t2 (⊆ S)
  using ‹?P2› by blast
}
moreover {
 fix S
 assume
  X: S ≠ {} and
  Y: S ⊆ {x. s1 = t1
   (⊆ sources-out (⟨bvars b⟩ # ?cs') vs1 s1 f x)} and
  Z: f = f' (⊆ vs1, vs1',
   ⋃ {tags-out (⟨bvars b⟩ # ?cs') vs1 s1 f x | x. x ∈ S})
 have ∀ x. sources-out ?cs' vs1 s1 f x ⊆
  sources-out (⟨bvars b⟩ # ?cs') vs1 s1 f x
  by (blast intro: subsetD [OF sources-out-observe-tl])
 hence S ⊆ {x. s1 = t1 (⊆ sources-out ?cs' vs1 s1 f x)}
  using Y by blast
 moreover have f = f' (⊆ vs1, vs1',
  ⋃ {tags-out ?cs' vs1 s1 f x | x. x ∈ S})
  using Z by (simp add: tags-out-observe-tl)
 ultimately have [p←drop (length ws1) ws2. fst p ∈ S] =
  [p←drop (length ws1') ws2'. fst p ∈ S]
```

```
          using X and ‹?P3› by simp
      }
      ultimately show
       ok-flow-aux-1 c1 c2 (c2';; WHILE b DO c) s1 t1 t2 f f'
          vs1 vs1' vs2 vs2' ws1' ws2' ?cs ∧
        ok-flow-aux-2 s1 s2 t1 t2 f f' vs1 vs1' ?cs ∧
        ok-flow-aux-3 s1 t1 f f' vs1 vs1' ws1 ws1' ws2 ws2' ?cs
        using R and S by auto
    qed
  qed (insert L, auto simp: no-upd-empty)
qed
apply (erule exE)+
apply (erule conjE)+
subgoal for p cfs' cfs''
proof -
  assume (c, s, f, vs0, ws0) →*{cfs'} (SKIP, p)
  moreover from this obtain s1' and vs and ws where
    S: p = (s1', f, vs, ws)
    by (blast dest: small-stepsl-stream)
  ultimately have
    T: (c, s1, f, vs1, ws1) →*{cfs'} (SKIP, s1', f, vs, ws)
    using P by simp
  assume (WHILE b DO c, p) →*{cfs''} (c2, s2, f, vs2, ws2)
  with S have
    U: (WHILE b DO c, s1', f, vs, ws) →*{cfs''}
     (c2, s2, f, vs2, ws2)
    by simp
  assume V: ?cs' = flow cfs' @ flow cfs''
    (is (- :: flow) = ?cs1' @ ?cs2')
  have W: (c, s1, f, vs1, ws1) →*{[]} (c, s1, f, vs1, ws1)
    by simp
  from N have ok-flow-aux {} c SKIP s1 s1' f vs1 vs ws1 ws ?cs1'
  proof
    assume X: s ∈ Univ A (⊆ state ∩ X)
    have Y: s1 ∈ Univ B1 (⊆ state ∩ X)
      using P and Q by (insert btyping2-approx [OF G X], simp)
    show ?thesis
      by (rule B [OF G [symmetric] H [symmetric] I [symmetric]
       L J Y W T])
  next
    assume X: s ∈ Univ C (⊆ state ∩ Y)
    have Y: s1 ∈ Univ B1' (⊆ state ∩ Y)
      using P and Q by (insert btyping2-approx [OF I X], simp)
    show ?thesis
      by (rule C [OF G [symmetric] H [symmetric] I [symmetric]
       L K Y W T])
  qed
  hence X: ok-flow-aux {} c SKIP s1 s1' f vs1 vs ws1 ws ?cs1'
    using P by simp
```

assume $\mathit{length}\ \mathit{cfs}'' < \mathit{length}\ (\mathit{tl}\ \mathit{cfs}_2)$
hence $\mathit{length}\ ([]\ @\ \mathit{cfs}'') < \mathit{length}\ (\mathit{cfs}_1\ @\ \mathit{cfs}_2)$
by *simp*
moreover have $[]\ @\ \mathit{cfs}'' = []\ @\ \mathit{cfs}''$ **..**
moreover from T **have** $(c,\ s,\ f,\ \mathit{vs}_0,\ \mathit{ws}_0) \Rightarrow (s_1',\ f,\ \mathit{vs},\ \mathit{ws})$
using P **by** (*auto dest*: *small-stepsl-steps simp*: *big-iff-small*)
hence $s_1' \in \mathit{Univ}\ A\ (\subseteq \mathit{state} \cap X) \cup \mathit{Univ}\ C\ (\subseteq \mathit{state} \cap Y)$
by (*rule univ-states-while* [OF - $G\ H\ I\ J\ K\ Q\ N$])
moreover have $(\mathit{WHILE}\ b\ \mathit{DO}\ c,\ s_1',\ f,\ \mathit{vs},\ \mathit{ws}) \rightarrow *\{[]\}$
$(\mathit{WHILE}\ b\ \mathit{DO}\ c,\ s_1',\ f,\ \mathit{vs},\ \mathit{ws})$
by *simp*
ultimately have
Y: $\mathit{ok\text{-}flow\text{-}aux}\ U\ (\mathit{WHILE}\ b\ \mathit{DO}\ c)\ c_2\ s_1'\ s_2\ f\ \mathit{vs}\ \mathit{vs}_2\ \mathit{ws}\ \mathit{ws}_2\ ?\mathit{cs}_2'$
using U **by** (*rule* M [*rule-format*])
show *?thesis*
proof (*rule conjI*, *clarify*)
fix $t_1\ f'\ \mathit{vs}_1'\ \mathit{ws}_1'$
obtain c_1'' **and** t_1' **and** vs_1'' **and** ws_1'' **where**
$\mathit{ok\text{-}flow\text{-}aux\text{-}1}\ c\ \mathit{SKIP}\ c_1''\ s_1\ t_1\ t_1'\ f\ f'$
$\mathit{vs}_1\ \mathit{vs}_1'\ \mathit{vs}\ \mathit{vs}_1''\ \mathit{ws}_1'\ \mathit{ws}_1''\ ?\mathit{cs}_1' \wedge$
$\mathit{ok\text{-}flow\text{-}aux\text{-}2}\ s_1\ s_1'\ t_1\ t_1'\ f\ f'\ \mathit{vs}_1\ \mathit{vs}_1'\ ?\mathit{cs}_1' \wedge$
$\mathit{ok\text{-}flow\text{-}aux\text{-}3}\ s_1\ t_1\ f\ f'\ \mathit{vs}_1\ \mathit{vs}_1'\ \mathit{ws}_1\ \mathit{ws}_1'\ \mathit{ws}\ \mathit{ws}_1''\ ?\mathit{cs}_1'$
(**is** - $\wedge$ *?P2* $\wedge$ *?P3*)
using X **by** *fastforce*
hence
$\mathit{ok\text{-}flow\text{-}aux\text{-}1}\ c\ \mathit{SKIP}\ \mathit{SKIP}\ s_1\ t_1\ t_1'\ f\ f'$
$\mathit{vs}_1\ \mathit{vs}_1'\ \mathit{vs}\ \mathit{vs}_1''\ \mathit{ws}_1'\ \mathit{ws}_1''\ ?\mathit{cs}_1'$
(**is** *?P1*) **and** *?P2* **and** *?P3* **by** *auto*
obtain c_2' **and** t_2 **and** vs_2' **and** ws_2' **where**
$\mathit{ok\text{-}flow\text{-}aux\text{-}1}\ (\mathit{WHILE}\ b\ \mathit{DO}\ c)\ c_2\ c_2'\ s_1'\ t_1'\ t_2\ f\ f'$
$\mathit{vs}\ \mathit{vs}_1''\ \mathit{vs}_2\ \mathit{vs}_2'\ \mathit{ws}_1''\ \mathit{ws}_2'\ ?\mathit{cs}_2' \wedge$
$\mathit{ok\text{-}flow\text{-}aux\text{-}2}\ s_1'\ s_2\ t_1'\ t_2\ f\ f'\ \mathit{vs}\ \mathit{vs}_1''\ ?\mathit{cs}_2' \wedge$
$\mathit{ok\text{-}flow\text{-}aux\text{-}3}\ s_1'\ t_1'\ f\ f'\ \mathit{vs}\ \mathit{vs}_1''\ \mathit{ws}\ \mathit{ws}_1''\ \mathit{ws}_2\ \mathit{ws}_2'\ ?\mathit{cs}_2'$
(**is** *?P1'* $\wedge$ *?P2'* $\wedge$ *?P3'*)
using Y **by** *fastforce*
hence *?P1'* **and** *?P2'* **and** *?P3'* **by** *auto*
show $\exists\, c_2'\ t_2\ \mathit{vs}_2'\ \mathit{ws}_2'.$
$\mathit{ok\text{-}flow\text{-}aux\text{-}1}\ c_1\ c_2\ c_2'\ s_1\ t_1\ t_2\ f\ f'$
$\mathit{vs}_1\ \mathit{vs}_1'\ \mathit{vs}_2\ \mathit{vs}_2'\ \mathit{ws}_1'\ \mathit{ws}_2'\ ?\mathit{cs} \wedge$
$\mathit{ok\text{-}flow\text{-}aux\text{-}2}\ s_1\ s_2\ t_1\ t_2\ f\ f'\ \mathit{vs}_1\ \mathit{vs}_1'\ ?\mathit{cs} \wedge$
$\mathit{ok\text{-}flow\text{-}aux\text{-}3}\ s_1\ t_1\ f\ f'\ \mathit{vs}_1\ \mathit{vs}_1'\ \mathit{ws}_1\ \mathit{ws}_1'\ \mathit{ws}_2\ \mathit{ws}_2'\ ?\mathit{cs}$
proof (*rule exI* [*of* - c_2'], *rule exI* [*of* - t_2],
rule exI [*of* - vs_2'], *rule exI* [*of* - ws_2'])
{
fix S
assume
Z: $S \neq \{\}$ **and**
AA: $S \subseteq \{x.\ s_1 = t_1\ (\subseteq \mathit{sources\text{-}aux}$
$(\langle \mathit{bvars}\ b\rangle\ \#\ ?\mathit{cs}_1'\ @\ ?\mathit{cs}_2')\ \mathit{vs}_1\ s_1\ f\ x)\}$ **and**

AB: $f = f'$ ($\subseteq$ vs_1, vs_1', $\bigcup$ {*tags-aux*
(⟨*bvars b*⟩ # $?cs_1'$ @ $?cs_2'$) vs_1 s_1 *f x* | *x*. *x* ∈ *S*})
have ∀ *x*. *sources-aux* ($?cs_1'$ @ $?cs_2'$) vs_1 s_1 *f x* ⊆
sources-aux (⟨*bvars b*⟩ # $?cs_1'$ @ $?cs_2'$) vs_1 s_1 *f x*
by (*blast intro*: *subsetD* [*OF sources-aux-observe-tl*])
hence *AC*: *S* ⊆ {*x*. $s_1 = t_1$
($\subseteq$ *sources-aux* ($?cs_1'$ @ $?cs_2'$) vs_1 s_1 *f x*)}
using *AA* **by** *blast*
moreover have ∀ *x*. *sources-aux* $?cs_1'$ vs_1 s_1 *f x* ⊆
sources-aux ($?cs_1'$ @ $?cs_2'$) vs_1 s_1 *f x*
by (*blast intro*: *subsetD* [*OF sources-aux-append*])
ultimately have
AD: *S* ⊆ {*x*. $s_1 = t_1$ ($\subseteq$ *sources-aux* $?cs_1'$ vs_1 s_1 *f x*)}
by *blast*
have *AE*: $f = f'$ ($\subseteq$ vs_1, vs_1',
$\bigcup$ {*tags-aux* ($?cs_1'$ @ $?cs_2'$) vs_1 s_1 *f x* | *x*. *x* ∈ *S*})
(**is** - = - ($\subseteq$ -, -, *?T*))
using *AB* **by** (*simp add*: *tags-aux-observe-tl*)
moreover have
$\bigcup$ {*tags-aux* $?cs_1'$ vs_1 s_1 *f x* | *x*. *x* ∈ *S*} ⊆ *?T*
(**is** *?T'* ⊆ -)
by (*blast intro*: *subsetD* [*OF tags-aux-append*])
ultimately have $f = f'$ ($\subseteq$ vs_1, vs_1', *?T'*)
by (*rule eq-streams-subset*)
hence
(*c*, t_1, f', vs_1', ws_1') →∗ (*SKIP*, t_1', f', vs_1'', ws_1'') ∧
map fst [*p*←*drop* (*length* vs_1) *vs*. *fst p* ∈ *S*] =
map fst [*p*←*drop* (*length* vs_1') vs_1''. *fst p* ∈ *S*]
(**is** *?Q1* ∧ *?Q2*)
using *Z* **and** *AD* **and** ‹*?P1*› **by** *simp*
hence *?Q1* **and** *?Q2* **by** *auto*
have *S* ⊆ {*x*. $s_1' = t_1'$ ($\subseteq$ *sources-aux* $?cs_2'$ *vs* s_1' *f x*)}
by (*rule sources-aux-rhs* [*OF AC AE T* ‹*?P2*›])
moreover have $f = f'$ ($\subseteq$ *vs*, vs_1'',
$\bigcup$ {*tags-aux* $?cs_2'$ *vs* s_1' *f x* | *x*. *x* ∈ *S*})
by (*rule tags-aux-rhs* [*OF AC AE T* ‹*?Q1*› ‹*?P1*›])
ultimately have
(*WHILE b DO c*, t_1', f', vs_1'', ws_1'') →∗
(c_2', t_2, f', vs_2', ws_2') ∧
(c_2 = *SKIP*) = (c_2' = *SKIP*) ∧
map fst [*p*←*drop* (*length vs*) vs_2. *fst p* ∈ *S*] =
map fst [*p*←*drop* (*length* vs_1'') vs_2'. *fst p* ∈ *S*]
(**is** *?Q1'* ∧ *?R2* ∧ *?Q2'*)
using *Z* **and** ‹*?P1'*› **by** *simp*
hence *?Q1'* **and** *?R2* **and** *?Q2'* **by** *auto*
have $s_1 = t_1$ ($\subseteq$ *bvars b*)
by (*rule eq-states-while* [*OF AA Z*], *insert L N P*, *simp+*)
hence *bval b* t_1
using *P* **and** *Q* **by** (*blast dest*: *bvars-bval*)

hence (*WHILE b DO c*, t_1, f', vs_1', ws_1') →
(*c*;; *WHILE b DO c*, t_1, f', vs_1', ws_1') **..**
moreover have (*c*;; *WHILE b DO c*, t_1, f', vs_1', ws_1') →*
(c_2', t_2, f', vs_2', ws_2')
using ‹*?Q1*› **and** ‹*?Q1'*›
by (*blast intro*: *star-seq2 star-trans*)
ultimately have
(c_1, t_1, f', vs_1', ws_1') →* (c_2', t_2, f', vs_2', ws_2')
(**is** *?R1*)
using *P* **by** (*blast intro*: *star-trans*)
moreover have
map fst [*p*←*drop* (*length* vs_1) vs_2. *fst p* ∈ *S*] =
map fst [*p*←*drop* (*length* vs_1') vs_2'. *fst p* ∈ *S*]
by (*rule small-steps-inputs*
[*OF T U* ‹*?Q1*› ‹*?Q1'*› ‹*?Q2*› ‹*?Q2'*›])
ultimately have *?R1* ∧ *?R2* ∧ *?this*
using ‹*?R2*› **by** *simp*
}
moreover {
fix *S*
assume
Z: *S* ≠ {} **and**
AA: *S* ⊆ {*x*. s_1 = t_1 (⊆ *sources*
(⟨*bvars b*⟩ # *?*cs_1' @ *?*cs_2') vs_1 s_1 *f x*)} **and**
AB: *f* = f' (⊆ vs_1, vs_1', ⋃ {*tags*
(⟨*bvars b*⟩ # *?*cs_1' @ *?*cs_2') vs_1 s_1 *f x* | *x*. *x* ∈ *S*})
have ∀ *x*. *sources* (*?*cs_1' @ *?*cs_2') vs_1 s_1 *f x* ⊆
sources (⟨*bvars b*⟩ # *?*cs_1' @ *?*cs_2') vs_1 s_1 *f x*
by (*blast intro*: *subsetD* [*OF sources-observe-tl*])
hence *AC*: *S* ⊆ {*x*. s_1 = t_1
(⊆ *sources* (*?*cs_1' @ *?*cs_2') vs_1 s_1 *f x*)}
using *AA* **by** *blast*
have ∀ *x*. *sources-aux* (*?*cs_1' @ *?*cs_2') vs_1 s_1 *f x* ⊆
sources (*?*cs_1' @ *?*cs_2') vs_1 s_1 *f x*
by (*blast intro*: *subsetD* [*OF sources-aux-sources*])
moreover have ∀ *x*. *sources-aux ?*cs_1' vs_1 s_1 *f x* ⊆
sources-aux (*?*cs_1' @ *?*cs_2') vs_1 s_1 *f x*
by (*blast intro*: *subsetD* [*OF sources-aux-append*])
ultimately have
AD: *S* ⊆ {*x*. s_1 = t_1 (⊆ *sources-aux ?*cs_1' vs_1 s_1 *f x*)}
using *AC* **by** *blast*
have *AE*: *f* = f' (⊆ vs_1, vs_1',
⋃ {*tags* (*?*cs_1' @ *?*cs_2') vs_1 s_1 *f x* | *x*. *x* ∈ *S*})
(**is** - = - (⊆ -, -, *?T*))
using *AB* **by** (*simp add*: *tags-observe-tl*)
have
⋃ {*tags-aux* (*?*cs_1' @ *?*cs_2') vs_1 s_1 *f x* | *x*. *x* ∈ *S*} ⊆ *?T*
(**is** *?T'* ⊆ -)
by (*blast intro*: *subsetD* [*OF tags-aux-tags*])

moreover have
$\bigcup\ \{\mathit{tags\text{-}aux}\ ?cs_1{}'\ vs_1\ s_1\ f\ x \mid x.\ x \in S\} \subseteq\ ?T'$
(**is** $?T'' \subseteq$ -)
by (*blast intro*: *subsetD* [*OF tags-aux-append*])
ultimately have $?T'' \subseteq\ ?T$
by *simp*
with *AE* **have** $f = f'\ (\subseteq vs_1,\ vs_1{}',\ ?T'')$
by (*rule eq-streams-subset*)
hence *AF*: $(c,\ t_1,\ f',\ vs_1{}',\ ws_1{}') \rightarrow\!*$
$(\mathit{SKIP},\ t_1{}',\ f',\ vs_1{}'',\ ws_1{}'')$
using *Z* **and** *AD* **and** ‹*?P1*› **by** *simp*
have $S \subseteq \{x.\ s_1{}' = t_1{}'\ (\subseteq \mathit{sources}\ ?cs_2{}'\ vs\ s_1{}'\ f\ x)\}$
by (*rule sources-rhs* [*OF AC AE T* ‹*?P2*›])
moreover have $f = f'\ (\subseteq vs,\ vs_1{}'',$
$\bigcup\ \{\mathit{tags}\ ?cs_2{}'\ vs\ s_1{}'\ f\ x \mid x.\ x \in S\})$
by (*rule tags-rhs* [*OF AC AE T AF* ‹*?P1*›])
ultimately have $s_2 = t_2\ (\subseteq S)$
using ‹*?P2'*› **by** *blast*
}
moreover {
fix *S*
assume
Z: $S \neq \{\}$ **and**
AA: $S \subseteq \{x.\ s_1 = t_1\ (\subseteq \mathit{sources\text{-}out}$
$(\langle \mathit{bvars}\ b\rangle\ \#\ ?cs_1{}'\ @\ ?cs_2{}')\ vs_1\ s_1\ f\ x)\}$ **and**
AB: $f = f'\ (\subseteq vs_1,\ vs_1{}',\ \bigcup\ \{\mathit{tags\text{-}out}$
$(\langle \mathit{bvars}\ b\rangle\ \#\ ?cs_1{}'\ @\ ?cs_2{}')\ vs_1\ s_1\ f\ x \mid x.\ x \in S\})$
have $\forall x.\ \mathit{sources\text{-}out}\ (?cs_1{}'\ @\ ?cs_2{}')\ vs_1\ s_1\ f\ x \subseteq$
$\mathit{sources\text{-}out}\ (\langle \mathit{bvars}\ b\rangle\ \#\ ?cs_1{}'\ @\ ?cs_2{}')\ vs_1\ s_1\ f\ x$
by (*blast intro*: *subsetD* [*OF sources-out-observe-tl*])
hence *AC*: $S \subseteq \{x.\ s_1 = t_1$
$(\subseteq \mathit{sources\text{-}out}\ (?cs_1{}'\ @\ ?cs_2{}')\ vs_1\ s_1\ f\ x)\}$
using *AA* **by** *blast*
have *AD*: $\forall x.\ \mathit{sources\text{-}aux}\ (?cs_1{}'\ @\ ?cs_2{}')\ vs_1\ s_1\ f\ x \subseteq$
$\mathit{sources\text{-}out}\ (?cs_1{}'\ @\ ?cs_2{}')\ vs_1\ s_1\ f\ x$
by (*blast intro*: *subsetD* [*OF sources-aux-sources-out*])
moreover have $\forall x.\ \mathit{sources\text{-}aux}\ ?cs_1{}'\ vs_1\ s_1\ f\ x \subseteq$
$\mathit{sources\text{-}aux}\ (?cs_1{}'\ @\ ?cs_2{}')\ vs_1\ s_1\ f\ x$
by (*blast intro*: *subsetD* [*OF sources-aux-append*])
ultimately have
AE: $S \subseteq \{x.\ s_1 = t_1\ (\subseteq \mathit{sources\text{-}aux}\ ?cs_1{}'\ vs_1\ s_1\ f\ x)\}$
using *AC* **by** *blast*
have *AF*: $f = f'\ (\subseteq vs_1,\ vs_1{}',$
$\bigcup\ \{\mathit{tags\text{-}out}\ (?cs_1{}'\ @\ ?cs_2{}')\ vs_1\ s_1\ f\ x \mid x.\ x \in S\})$
(**is** - = - ($\subseteq$ -, -, $?T$))
using *AB* **by** (*simp add*: *tags-out-observe-tl*)
have *AG*:
$\bigcup\ \{\mathit{tags\text{-}aux}\ (?cs_1{}'\ @\ ?cs_2{}')\ vs_1\ s_1\ f\ x \mid x.\ x \in S\} \subseteq\ ?T$
(**is** $?T' \subseteq$ -)

by (*blast intro*: *subsetD* [*OF tags-aux-tags-out*])
moreover have
$\bigcup\ \{\mathit{tags\text{-}aux}\ ?cs_1'\ vs_1\ s_1\ f\ x \mid x.\ x \in S\} \subseteq\ ?T'$
(**is** $?T'' \subseteq$ -)
by (*blast intro*: *subsetD* [*OF tags-aux-append*])
ultimately have $?T'' \subseteq\ ?T$
by *simp*
with AF **have** $f = f'\ (\subseteq vs_1,\ vs_1',\ ?T'')$
by (*rule eq-streams-subset*)
hence AH: $(c,\ t_1,\ f',\ vs_1',\ ws_1') \rightarrow*$
$(\mathit{SKIP},\ t_1',\ f',\ vs_1'',\ ws_1'')$
using Z **and** AE **and** ‹*?P1*› **by** *simp*
have AI: $S \subseteq \{x.\ s_1 = t_1$
$(\subseteq \mathit{sources\text{-}aux}\ (?cs_1'\ @\ ?cs_2')\ vs_1\ s_1\ f\ x)\}$
using AC **and** AD **by** *blast*
have AJ: $f = f'\ (\subseteq vs_1,\ vs_1',\ ?T')$
using AF **and** AG **by** (*rule eq-streams-subset*)
have $S \subseteq \{x.\ s_1' = t_1'\ (\subseteq \mathit{sources\text{-}aux}\ ?cs_2'\ vs\ s_1'\ f\ x)\}$
by (*rule sources-aux-rhs* [*OF AI AJ T* ‹*?P2*›])
moreover have $f = f'\ (\subseteq vs,\ vs_1'',$
$\bigcup\ \{\mathit{tags\text{-}aux}\ ?cs_2'\ vs\ s_1'\ f\ x \mid x.\ x \in S\})$
by (*rule tags-aux-rhs* [*OF AI AJ T AH* ‹*?P1*›])
ultimately have AK:
$(\mathit{WHILE}\ b\ \mathit{DO}\ c,\ t_1',\ f',\ vs_1'',\ ws_1'') \rightarrow*$
$(c_2',\ t_2,\ f',\ vs_2',\ ws_2')$
using Z **and** ‹*?P1'*› **by** *simp*
have $\forall x.\ \mathit{sources\text{-}out}\ ?cs_1'\ vs_1\ s_1\ f\ x \subseteq$
$\mathit{sources\text{-}out}\ (?cs_1'\ @\ ?cs_2')\ vs_1\ s_1\ f\ x$
by (*blast intro*: *subsetD* [*OF sources-out-append*])
hence $S \subseteq \{x.\ s_1 = t_1\ (\subseteq \mathit{sources\text{-}out}\ ?cs_1'\ vs_1\ s_1\ f\ x)\}$
using AC **by** *blast*
moreover have
$\bigcup\ \{\mathit{tags\text{-}out}\ ?cs_1'\ vs_1\ s_1\ f\ x \mid x.\ x \in S\} \subseteq\ ?T$
(**is** $?T' \subseteq$ -)
by (*blast intro*: *subsetD* [*OF tags-out-append*])
with AF **have** $f = f'\ (\subseteq vs_1,\ vs_1',\ ?T')$
by (*rule eq-streams-subset*)
ultimately have AL:
$[p \leftarrow \mathit{drop}\ (\mathit{length}\ ws_1)\ ws.\ \mathit{fst}\ p \in S] =$
$[p \leftarrow \mathit{drop}\ (\mathit{length}\ ws_1')\ ws_1''.\ \mathit{fst}\ p \in S]$
using Z **and** ‹*?P3*› **by** *simp*
have $S \subseteq \{x.\ s_1' = t_1'\ (\subseteq \mathit{sources\text{-}out}\ ?cs_2'\ vs\ s_1'\ f\ x)\}$
by (*rule sources-out-rhs* [*OF AC AF T* ‹*?P2*›])
moreover have $f = f'\ (\subseteq vs,\ vs_1'',$
$\bigcup\ \{\mathit{tags\text{-}out}\ ?cs_2'\ vs\ s_1'\ f\ x \mid x.\ x \in S\})$
by (*rule tags-out-rhs* [*OF AC AF T AH* ‹*?P1*›])
ultimately have $[p \leftarrow \mathit{drop}\ (\mathit{length}\ ws)\ ws_2.\ \mathit{fst}\ p \in S] =$
$[p \leftarrow \mathit{drop}\ (\mathit{length}\ ws_1'')\ ws_2'.\ \mathit{fst}\ p \in S]$
using Z **and** ‹*?P3'*› **by** *simp*

```
          hence [p←drop (length ws1) ws2. fst p ∈ S] =
            [p←drop (length ws1') ws2'. fst p ∈ S]
            by (rule small-steps-outputs [OF T U AH AK AL])
        }
        ultimately show
         ok-flow-aux-1 c1 c2 c2' s1 t1 t2 f f'
           vs1 vs1' vs2 vs2' ws1' ws2' ?cs ∧
          ok-flow-aux-2 s1 s2 t1 t2 f f' vs1 vs1' ?cs ∧
          ok-flow-aux-3 s1 t1 f f' vs1 vs1' ws1 ws1' ws2 ws2' ?cs
          using R and V by auto
      qed
    qed (insert L, auto simp: no-upd-empty)
  qed
  done
next
  assume
    Q: ¬ bval b s and
    R: flow cfs2 = [⟨bvars b⟩]
      (is ?cs = -)
  assume (c2, s2, f, vs2, ws2) = (SKIP, s, f, vs0, ws0)
  hence S: (c2, s2, f, vs2, ws2) = (SKIP, s1, f, vs1, ws1)
    using P by simp
  show ?thesis
  proof (rule conjI, clarify)
    fix t1 f' vs1' ws1'
    show ∃ c2' t2 vs2' ws2'.
      ok-flow-aux-1 c1 c2 c2' s1 t1 t2 f f'
        vs1 vs1' vs2 vs2' ws1' ws2' ?cs ∧
      ok-flow-aux-2 s1 s2 t1 t2 f f' vs1 vs1' ?cs ∧
      ok-flow-aux-3 s1 t1 f f' vs1 vs1' ws1 ws1' ws2 ws2' ?cs
    proof (rule exI [of - SKIP], rule exI [of - t1],
     rule exI [of - vs1'], rule exI [of - ws1'])
      {
        fix S
        assume
          S ⊆ {x. s1 = t1 (⊆ sources-aux [⟨bvars b⟩] vs1 s1 f x)} and
          S ≠ {}
        hence s1 = t1 (⊆ bvars b)
          by (rule eq-states-while, insert L N P, simp+)
        hence ¬ bval b t1
          using P and Q by (blast dest: bvars-bval)
        hence (c1, t1, f', vs1', ws1') →* (SKIP, t1, f', vs1', ws1')
          using P by simp
      }
      moreover {
        fix S
        assume S ⊆ {x. s1 = t1 (⊆ sources [⟨bvars b⟩] vs1 s1 f x)}
        moreover have ∀ x. sources [] vs1 s1 f x ⊆
          sources [⟨bvars b⟩] vs1 s1 f x
```

```
            by (blast intro!: sources-observe-tl)
          ultimately have s1 = t1 (⊆ S)
            by auto
        }
        ultimately show
         ok-flow-aux-1 c1 c2 SKIP s1 t1 t1 f f'
           vs1 vs1' vs2 vs1' ws1' ws1' ?cs ∧
          ok-flow-aux-2 s1 s2 t1 t1 f f' vs1 vs1' ?cs ∧
          ok-flow-aux-3 s1 t1 f f' vs1 vs1' ws1 ws1' ws2 ws1' ?cs
          using R and S by auto
      qed
    qed (insert L, auto simp: no-upd-empty)
  qed
next
  assume P: bval b s
  assume (c;; WHILE b DO c, s, f, vs0, ws0) →*{tl cfs1}
    (c1, s1, f, vs1, ws1)
  hence
   (∃ c' cfs.
      c1 = c';; WHILE b DO c ∧
      (c, s, f, vs0, ws0) →*{cfs} (c', s1, f, vs1, ws1) ∧
      flow (tl cfs1) = flow cfs) ∨
    (∃ p cfs' cfs''.
      length cfs'' < length (tl cfs1) ∧
      (c, s, f, vs0, ws0) →*{cfs'} (SKIP, p) ∧
      (WHILE b DO c, p) →*{cfs''} (c1, s1, f, vs1, ws1) ∧
      flow (tl cfs1) = flow cfs' @ flow cfs'')
    by (rule small-stepsl-seq)
  thus ?thesis
    apply (rule disjE)
     apply (erule exE)+
     apply (erule conjE)+
    subgoal for c' cfs
    proof −
      assume
        Q: (c, s, f, vs0, ws0) →*{cfs} (c', s1, f, vs1, ws1) and
        R: c1 = c';; WHILE b DO c
      hence (c';; WHILE b DO c, s1, f, vs1, ws1) →*{cfs2}
        (c2, s2, f, vs2, ws2)
        using O by simp
      hence
       (∃ c'' cfs'.
          c2 = c'';; WHILE b DO c ∧
          (c', s1, f, vs1, ws1) →*{cfs'} (c'', s2, f, vs2, ws2) ∧
          flow cfs2 = flow cfs') ∨
        (∃ p cfs' cfs''.
          length cfs'' < length cfs2 ∧
          (c', s1, f, vs1, ws1) →*{cfs'} (SKIP, p) ∧
          (WHILE b DO c, p) →*{cfs''} (c2, s2, f, vs2, ws2) ∧
```

```
    flow cfs_2 = flow cfs' @ flow cfs'')
  by (rule small-stepsl-seq)
thus ?thesis
  apply (rule disjE)
   apply (erule exE)+
   apply (erule conjE)+
  subgoal for c'' cfs'
  proof -
    assume
      S: c_2 = c'';; WHILE b DO c and
      T: (c', s_1, f, vs_1, ws_1) →*{cfs'} (c'', s_2, f, vs_2, ws_2) and
      U: flow cfs_2 = flow cfs'
        (is ?cs = ?cs')
    from N have ok-flow-aux {} c' c'' s_1 s_2 f vs_1 vs_2 ws_1 ws_2 ?cs'
    proof
      assume V: s ∈ Univ A (⊆ state ∩ X)
      have W: s ∈ Univ B_1 (⊆ state ∩ X)
        using P by (insert btyping2-approx [OF G V], simp)
      show ?thesis
        by (rule B [OF G [symmetric] H [symmetric] I [symmetric]
         L J W Q T])
    next
      assume V: s ∈ Univ C (⊆ state ∩ Y)
      have W: s ∈ Univ B_1' (⊆ state ∩ Y)
        using P by (insert btyping2-approx [OF I V], simp)
      show ?thesis
        by (rule C [OF G [symmetric] H [symmetric] I [symmetric]
         L K W Q T])
    qed
    hence V: ok-flow-aux {} c' c'' s_1 s_2 f vs_1 vs_2 ws_1 ws_2 ?cs
      using U by simp
    show ?thesis
    proof (rule conjI, clarify)
      fix t_1 f' vs_1' ws_1'
      obtain c_2' and t_2 and vs_2' and ws_2' where
       ok-flow-aux-1 c' c'' c_2' s_1 t_1 t_2 f f'
          vs_1 vs_1' vs_2 vs_2' ws_1' ws_2' ?cs ∧
        ok-flow-aux-2 s_1 s_2 t_1 t_2 f f' vs_1 vs_1' ?cs ∧
        ok-flow-aux-3 s_1 t_1 f f' vs_1 vs_1' ws_1 ws_1' ws_2 ws_2' ?cs
        (is ?P1 ∧ ?P2 ∧ ?P3)
        using V by fastforce
      hence ?P1 and ?P2 and ?P3 by auto
      show ∃ c_2' t_2 vs_2' ws_2'.
        ok-flow-aux-1 c_1 c_2 c_2' s_1 t_1 t_2 f f'
          vs_1 vs_1' vs_2 vs_2' ws_1' ws_2' ?cs ∧
        ok-flow-aux-2 s_1 s_2 t_1 t_2 f f' vs_1 vs_1' ?cs ∧
        ok-flow-aux-3 s_1 t_1 f f' vs_1 vs_1' ws_1 ws_1' ws_2 ws_2' ?cs
      proof (rule exI [of - c_2';; WHILE b DO c],
       rule exI [of - t_2], rule exI [of - vs_2'], rule exI [of - ws_2'])
```

```
      {
        fix S
        assume S ≠ {} and
         S ⊆ {x. s1 = t1 (⊆ sources-aux ?cs vs1 s1 f x)} and
         f = f' (⊆ vs1, vs1',
           ⋃ {tags-aux ?cs vs1 s1 f x | x. x ∈ S})
        hence
         (c1, t1, f', vs1', ws1') →∗
           (c2';; WHILE b DO c, t2, f', vs2', ws2') ∧
          map fst [p←drop (length vs1) vs2. fst p ∈ S] =
            map fst [p←drop (length vs1') vs2'. fst p ∈ S]
          using R and ‹?P1› by (blast intro: star-seq2)
      }
      thus
       ok-flow-aux-1 c1 c2 (c2';; WHILE b DO c) s1 t1 t2 f f'
         vs1 vs1' vs2 vs2' ws1' ws2' ?cs ∧
        ok-flow-aux-2 s1 s2 t1 t2 f f' vs1 vs1' ?cs ∧
        ok-flow-aux-3 s1 t1 f f' vs1 vs1' ws1 ws1' ws2 ws2' ?cs
        using S and ‹?P2› and ‹?P3› by simp
    qed
  qed (insert L, auto simp: no-upd-empty)
qed
apply (erule exE)+
apply (erule conjE)+
subgoal for p cfs' cfs''
proof −
  assume (c', s1, f, vs1, ws1) →∗{cfs'} (SKIP, p)
  moreover from this obtain s1' and vs and ws where
    S: p = (s1', f, vs, ws)
    by (blast dest: small-stepsl-stream)
  ultimately have
    T: (c', s1, f, vs1, ws1) →∗{cfs'} (SKIP, s1', f, vs, ws)
    by simp
  assume (WHILE b DO c, p) →∗{cfs''} (c2, s2, f, vs2, ws2)
  with S have
    U: (WHILE b DO c, s1', f, vs, ws) →∗{cfs''}
      (c2, s2, f, vs2, ws2)
    by simp
  assume V: flow cfs2 = flow cfs' @ flow cfs''
    (is (?cs :: flow) = ?cs1 @ ?cs2)
  from N have
    W: ok-flow-aux {} c' SKIP s1 s1' f vs1 vs ws1 ws ?cs1
  proof
    assume X: s ∈ Univ A (⊆ state ∩ X)
    have Y: s ∈ Univ B1 (⊆ state ∩ X)
      using P by (insert btyping2-approx [OF G X], simp)
    show ?thesis
      by (rule B [OF G [symmetric] H [symmetric] I [symmetric]
       L J Y Q T])
```

next
assume *X*: $s \in$ *Univ C* ($\subseteq$ *state* $\cap$ *Y*)
have *Y*: $s \in$ *Univ* B_1' ($\subseteq$ *state* $\cap$ *Y*)
using *P* **by** (*insert btyping2-approx* [*OF I X*], *simp*)
show *?thesis*
by (*rule C* [*OF G* [*symmetric*] *H* [*symmetric*] *I* [*symmetric*]
L K Y Q T])
qed
assume *length* $cfs'' <$ *length* cfs_2
hence *length* ([] @ cfs'') $<$ *length* (cfs_1 @ cfs_2)
by *simp*
moreover have [] @ cfs'' = [] @ cfs'' **..**
moreover have
(c, s, f, vs_0, ws_0) $\rightarrow*$\{*cfs* @ *cfs'*\} (*SKIP*, s_1', f, *vs*, *ws*)
using *Q* **and** *T* **by** (*rule small-stepsl-append*)
hence (c, s, f, vs_0, ws_0) $\Rightarrow$ (s_1', f, *vs*, *ws*)
by (*auto dest*: *small-stepsl-steps simp*: *big-iff-small*)
hence $s_1' \in$ *Univ A* ($\subseteq$ *state* $\cap$ *X*) $\cup$ *Univ C* ($\subseteq$ *state* $\cap$ *Y*)
by (*rule univ-states-while* [*OF - G H I J K P N*])
moreover have (*WHILE b DO c*, s_1', f, *vs*, *ws*) $\rightarrow*$\{[]\}
(*WHILE b DO c*, s_1', f, *vs*, *ws*)
by *simp*
ultimately have *X*:
ok-flow-aux U (*WHILE b DO c*) c_2 s_1' s_2 f *vs* vs_2 *ws* ws_2 $?cs_2$
using *U* **by** (*rule M* [*rule-format*])
show *?thesis*
proof (*rule conjI*, *clarify*)
fix t_1 f' vs_1' ws_1'
obtain c_1'' **and** t_1' **and** vs_1'' **and** ws_1'' **where**
ok-flow-aux-1 c' *SKIP* c_1'' s_1 t_1 t_1' f f'
vs_1 vs_1' *vs* vs_1'' ws_1' ws_1'' $?cs_1$ $\wedge$
ok-flow-aux-2 s_1 s_1' t_1 t_1' f f' vs_1 vs_1' $?cs_1$ $\wedge$
ok-flow-aux-3 s_1 t_1 f f' vs_1 vs_1' ws_1 ws_1' *ws* ws_1'' $?cs_1$
(**is** - $\wedge$ *?P2* $\wedge$ *?P3*)
using *W* **by** *fastforce*
hence
ok-flow-aux-1 c' *SKIP SKIP* s_1 t_1 t_1' f f'
vs_1 vs_1' *vs* vs_1'' ws_1' ws_1'' $?cs_1$
(**is** *?P1*) **and** *?P2* **and** *?P3* **by** *auto*
obtain c_2' **and** t_2 **and** vs_2' **and** ws_2' **where**
ok-flow-aux-1 (*WHILE b DO c*) c_2 c_2' s_1' t_1' t_2 f f'
vs vs_1'' vs_2 vs_2' ws_1'' ws_2' $?cs_2$ $\wedge$
ok-flow-aux-2 s_1' s_2 t_1' t_2 f f' *vs* vs_1'' $?cs_2$ $\wedge$
ok-flow-aux-3 s_1' t_1' f f' *vs* vs_1'' *ws* ws_1'' ws_2 ws_2' $?cs_2$
(**is** *?P1'* $\wedge$ *?P2'* $\wedge$ *?P3'*)
using *X* **by** *fastforce*
hence *?P1'* **and** *?P2'* **and** *?P3'* **by** *auto*
show $\exists$ c_2' t_2 vs_2' ws_2'.
ok-flow-aux-1 c_1 c_2 c_2' s_1 t_1 t_2 f f'

```
    vs₁ vs₁' vs₂ vs₂' ws₁' ws₂' ?cs ∧
  ok-flow-aux-2 s₁ s₂ t₁ t₂ f f' vs₁ vs₁' ?cs ∧
  ok-flow-aux-3 s₁ t₁ f f' vs₁ vs₁' ws₁ ws₁' ws₂ ws₂' ?cs
proof (rule exI [of - c₂'], rule exI [of - t₂],
 rule exI [of - vs₂'], rule exI [of - ws₂'])
  {
    fix S
    assume
      Y: S ≠ {} and
      Z: S ⊆ {x. s₁ = t₁
        (⊆ sources-aux (?cs₁ @ ?cs₂) vs₁ s₁ f x)} and
      AA: f = f' (⊆ vs₁, vs₁',
        ⋃ {tags-aux (?cs₁ @ ?cs₂) vs₁ s₁ f x | x. x ∈ S})
        (is - = - (⊆ -, -, ?T))
    have ∀ x. sources-aux ?cs₁ vs₁ s₁ f x ⊆
      sources-aux (?cs₁ @ ?cs₂) vs₁ s₁ f x
      by (blast intro: subsetD [OF sources-aux-append])
    hence S ⊆ {x. s₁ = t₁ (⊆ sources-aux ?cs₁ vs₁ s₁ f x)}
      using Z by blast
    moreover have
     ⋃ {tags-aux ?cs₁ vs₁ s₁ f x | x. x ∈ S} ⊆ ?T
      (is ?T' ⊆ -)
      by (blast intro: subsetD [OF tags-aux-append])
    with AA have f = f' (⊆ vs₁, vs₁', ?T')
      by (rule eq-streams-subset)
    ultimately have
     (c', t₁, f', vs₁', ws₁') →*
        (SKIP, t₁', f', vs₁'', ws₁'') ∧
      map fst [p←drop (length vs₁) vs. fst p ∈ S] =
        map fst [p←drop (length vs₁') vs₁''. fst p ∈ S]
      (is ?Q1 ∧ ?Q2)
      using Y and ‹?P1› by simp
    hence ?Q1 and ?Q2 by auto
    have S ⊆ {x. s₁' = t₁' (⊆ sources-aux ?cs₂ vs s₁' f x)}
      by (rule sources-aux-rhs [OF Z AA T ‹?P2›])
    moreover have f = f' (⊆ vs, vs₁'',
      ⋃ {tags-aux ?cs₂ vs s₁' f x | x. x ∈ S})
      by (rule tags-aux-rhs [OF Z AA T ‹?Q1› ‹?P1›])
    ultimately have
     (WHILE b DO c, t₁', f', vs₁'', ws₁'') →*
        (c₂', t₂, f', vs₂', ws₂') ∧
      (c₂ = SKIP) = (c₂' = SKIP) ∧
      map fst [p←drop (length vs) vs₂. fst p ∈ S] =
        map fst [p←drop (length vs₁'') vs₂'. fst p ∈ S]
      (is ?Q1' ∧ ?R2 ∧ ?Q2')
      using Y and ‹?P1'› by simp
    hence ?Q1' and ?R2 and ?Q2' by auto
    from R and ‹?Q1› and ‹?Q1'› have
     (c₁, t₁, f', vs₁', ws₁') →* (c₂', t₂, f', vs₂', ws₂')
```

```
    (is ?R1)
    by (blast intro: star-seq2 star-trans)
  moreover have
    map fst [p←drop (length vs_1) vs_2. fst p ∈ S] =
      map fst [p←drop (length vs_1') vs_2'. fst p ∈ S]
    by (rule small-steps-inputs
     [OF T U ‹?Q1› ‹?Q1'› ‹?Q2› ‹?Q2'›])
  ultimately have ?R1 ∧ ?R2 ∧ ?this
    using ‹?R2› by simp
}
moreover {
  fix S
  assume
    Y: S ≠ {} and
    Z: S ⊆ {x. s_1 = t_1
      (⊆ sources (?cs_1 @ ?cs_2) vs_1 s_1 f x)} and
    AA: f = f' (⊆ vs_1, vs_1',
      ⋃ {tags (?cs_1 @ ?cs_2) vs_1 s_1 f x | x. x ∈ S})
      (is - = - (⊆ -, -, ?T))
  have ∀x. sources-aux (?cs_1 @ ?cs_2) vs_1 s_1 f x ⊆
    sources (?cs_1 @ ?cs_2) vs_1 s_1 f x
    by (blast intro: subsetD [OF sources-aux-sources])
  moreover have ∀x. sources-aux ?cs_1 vs_1 s_1 f x ⊆
    sources-aux (?cs_1 @ ?cs_2) vs_1 s_1 f x
    by (blast intro: subsetD [OF sources-aux-append])
  ultimately have
    AB: S ⊆ {x. s_1 = t_1 (⊆ sources-aux ?cs_1 vs_1 s_1 f x)}
    using Z by blast
  have
   ⋃ {tags-aux (?cs_1 @ ?cs_2) vs_1 s_1 f x | x. x ∈ S} ⊆ ?T
   (is ?T' ⊆ -)
    by (blast intro: subsetD [OF tags-aux-tags])
  moreover have
   ⋃ {tags-aux ?cs_1 vs_1 s_1 f x | x. x ∈ S} ⊆ ?T'
   (is ?T'' ⊆ -)
    by (blast intro: subsetD [OF tags-aux-append])
  ultimately have ?T'' ⊆ ?T
    by simp
  with AA have f = f' (⊆ vs_1, vs_1', ?T'')
    by (rule eq-streams-subset)
  hence AC: (c', t_1, f', vs_1', ws_1') →*
    (SKIP, t_1', f', vs_1'', ws_1'')
    using Y and AB and ‹?P1› by simp
  have S ⊆ {x. s_1' = t_1' (⊆ sources ?cs_2 vs s_1' f x)}
    by (rule sources-rhs [OF Z AA T ‹?P2›])
  moreover have f = f' (⊆ vs, vs_1'',
    ⋃ {tags ?cs_2 vs s_1' f x | x. x ∈ S})
    by (rule tags-rhs [OF Z AA T AC ‹?P1›])
  ultimately have s_2 = t_2 (⊆ S)
```

using ‹*?P2'*› **by** *blast*
}
moreover {
fix S
assume
Y: $S \neq \{\}$ **and**
Z: $S \subseteq \{x.\ s_1 = t_1$
$(\subseteq \textit{sources-out}\ (\textit{?cs}_1\ @\ \textit{?cs}_2)\ \textit{vs}_1\ s_1\ f\ x)\}$ **and**
AA: $f = f'\ (\subseteq \textit{vs}_1,\ \textit{vs}_1',$
$\bigcup\ \{\textit{tags-out}\ (\textit{?cs}_1\ @\ \textit{?cs}_2)\ \textit{vs}_1\ s_1\ f\ x \mid x.\ x \in S\})$
(**is** - = - $(\subseteq$ -, -, $?T))$
have AB: $\forall x.\ \textit{sources-aux}\ (\textit{?cs}_1\ @\ \textit{?cs}_2)\ \textit{vs}_1\ s_1\ f\ x \subseteq$
$\textit{sources-out}\ (\textit{?cs}_1\ @\ \textit{?cs}_2)\ \textit{vs}_1\ s_1\ f\ x$
by (*blast intro*: *subsetD* [*OF sources-aux-sources-out*])
moreover have $\forall x.\ \textit{sources-aux}\ \textit{?cs}_1\ \textit{vs}_1\ s_1\ f\ x \subseteq$
$\textit{sources-aux}\ (\textit{?cs}_1\ @\ \textit{?cs}_2)\ \textit{vs}_1\ s_1\ f\ x$
by (*blast intro*: *subsetD* [*OF sources-aux-append*])
ultimately have
AC: $S \subseteq \{x.\ s_1 = t_1\ (\subseteq \textit{sources-aux}\ \textit{?cs}_1\ \textit{vs}_1\ s_1\ f\ x)\}$
using Z **by** *blast*
have AD:
$\bigcup\ \{\textit{tags-aux}\ (\textit{?cs}_1\ @\ \textit{?cs}_2)\ \textit{vs}_1\ s_1\ f\ x \mid x.\ x \in S\} \subseteq ?T$
(**is** $?T' \subseteq$ -)
by (*blast intro*: *subsetD* [*OF tags-aux-tags-out*])
moreover have
$\bigcup\ \{\textit{tags-aux}\ \textit{?cs}_1\ \textit{vs}_1\ s_1\ f\ x \mid x.\ x \in S\} \subseteq ?T'$
(**is** $?T'' \subseteq$ -)
by (*blast intro*: *subsetD* [*OF tags-aux-append*])
ultimately have $?T'' \subseteq ?T$
by *simp*
with AA **have** $f = f'\ (\subseteq \textit{vs}_1,\ \textit{vs}_1',\ ?T'')$
by (*rule eq-streams-subset*)
hence AE: $(c',\ t_1,\ f',\ \textit{vs}_1',\ \textit{ws}_1') \rightarrow\!*$
$(\textit{SKIP},\ t_1',\ f',\ \textit{vs}_1'',\ \textit{ws}_1'')$
using Y **and** AC **and** ‹*?P1*› **by** *simp*
have AF: $S \subseteq \{x.\ s_1 = t_1$
$(\subseteq \textit{sources-aux}\ (\textit{?cs}_1\ @\ \textit{?cs}_2)\ \textit{vs}_1\ s_1\ f\ x)\}$
using Z **and** AB **by** *blast*
have AG: $f = f'\ (\subseteq \textit{vs}_1,\ \textit{vs}_1',\ ?T')$
using AA **and** AD **by** (*rule eq-streams-subset*)
have $S \subseteq \{x.\ s_1' = t_1'\ (\subseteq \textit{sources-aux}\ \textit{?cs}_2\ \textit{vs}\ s_1'\ f\ x)\}$
by (*rule sources-aux-rhs* [*OF AF AG T* ‹*?P2*›])
moreover have $f = f'\ (\subseteq \textit{vs},\ \textit{vs}_1'',$
$\bigcup\ \{\textit{tags-aux}\ \textit{?cs}_2\ \textit{vs}\ s_1'\ f\ x \mid x.\ x \in S\})$
by (*rule tags-aux-rhs* [*OF AF AG T AE* ‹*?P1*›])
ultimately have
AH: $(\textit{WHILE}\ b\ \textit{DO}\ c,\ t_1',\ f',\ \textit{vs}_1'',\ \textit{ws}_1'') \rightarrow\!*$
$(c_2',\ t_2,\ f',\ \textit{vs}_2',\ \textit{ws}_2')$
using Y **and** ‹*?P1'*› **by** *simp*

```
          have ∀ x. sources-out ?cs1 vs1 s1 f x ⊆
            sources-out (?cs1 @ ?cs2) vs1 s1 f x
            by (blast intro: subsetD [OF sources-out-append])
          hence S ⊆ {x. s1 = t1 (⊆ sources-out ?cs1 vs1 s1 f x)}
            using Z by blast
          moreover have
           ⋃ {tags-out ?cs1 vs1 s1 f x | x. x ∈ S} ⊆ ?T
            (is ?T′ ⊆ -)
            by (blast intro: subsetD [OF tags-out-append])
          with AA have f = f′ (⊆ vs1, vs1′, ?T′)
            by (rule eq-streams-subset)
          ultimately have AI:
           [p←drop (length ws1) ws. fst p ∈ S] =
            [p←drop (length ws1′) ws1″. fst p ∈ S]
            using Y and ‹?P3› by simp
          have S ⊆ {x. s1′ = t1′ (⊆ sources-out ?cs2 vs s1′ f x)}
            by (rule sources-out-rhs [OF Z AA T ‹?P2›])
          moreover have f = f′ (⊆ vs, vs1″,
            ⋃ {tags-out ?cs2 vs s1′ f x | x. x ∈ S})
            by (rule tags-out-rhs [OF Z AA T AE ‹?P1›])
          ultimately have [p←drop (length ws) ws2. fst p ∈ S] =
            [p←drop (length ws1″) ws2′. fst p ∈ S]
            using Y and ‹?P3′› by simp
          hence [p←drop (length ws1) ws2. fst p ∈ S] =
            [p←drop (length ws1′) ws2′. fst p ∈ S]
            by (rule small-steps-outputs [OF T U AE AH AI])
        }
        ultimately show
         ok-flow-aux-1 c1 c2 c2′ s1 t1 t2 f f′
           vs1 vs1′ vs2 vs2′ ws1′ ws2′ ?cs ∧
          ok-flow-aux-2 s1 s2 t1 t2 f f′ vs1 vs1′ ?cs ∧
          ok-flow-aux-3 s1 t1 f f′ vs1 vs1′ ws1 ws1′ ws2 ws2′ ?cs
          using V by auto
      qed
    qed (insert L, auto simp: no-upd-empty)
  qed
  done
qed
apply (erule exE)+
apply (erule conjE)+
subgoal for p cfs′ cfs″
proof −
  assume (c, s, f, vs0, ws0) →∗{cfs′} (SKIP, p)
  moreover from this obtain s1′ and vs and ws where
    Q: p = (s1′, f, vs, ws)
    by (blast dest: small-stepsl-stream)
  ultimately have
    R: (c, s, f, vs0, ws0) →∗{cfs′} (SKIP, s1′, f, vs, ws)
    by simp
```

```
          assume (WHILE b DO c, p) →*{cfs''} (c1, s1, f, vs1, ws1)
          with Q have S:
           (WHILE b DO c, s1', f, vs, ws) →*{cfs''} (c1, s1, f, vs1, ws1)
            by simp
          assume length cfs'' < length (tl cfs1)
          hence length (cfs'' @ cfs2) < length (cfs1 @ cfs2)
            by simp
          moreover have cfs'' @ cfs2 = cfs'' @ cfs2 ..
          moreover have (c, s, f, vs0, ws0) ⇒ (s1', f, vs, ws)
            using R by (auto dest: small-stepsl-steps simp: big-iff-small)
          hence s1' ∈ Univ A (⊆ state ∩ X) ∪ Univ C (⊆ state ∩ Y)
            by (rule univ-states-while [OF - G H I J K P N])
          ultimately show ?thesis
            using S and O by (rule M [rule-format])
        qed
        done
    next
      assume (c1, s1, f, vs1, ws1) = (SKIP, s, f, vs0, ws0)
      moreover from this have
       (c2, s2, f, vs2, ws2) = (SKIP, s1, f, vs1, ws1) ∧ flow cfs2 = []
        using O by (blast intro!: small-stepsl-skip)
      ultimately show ?thesis
        by (insert L, fastforce)
    qed
  qed
qed

lemma ctyping2-correct-aux:
⟦(U, v) ⊨ c (⊆ A, X) = Some (B, Y); s ∈ Univ A (⊆ state ∩ X);
    (c, s, f, vs0, ws0) →*{cfs1} (c1, s1, f, vs1, ws1);
    (c1, s1, f, vs1, ws1) →*{cfs2} (c2, s2, f, vs2, ws2)⟧ ⟹
  ok-flow-aux U c1 c2 s1 s2 f vs1 vs2 ws1 ws2 (flow cfs2)
  apply (induction (U, v) c A X arbitrary: B Y U v c1 c2 s s1 s2
   vs0 vs1 vs2 ws0 ws1 ws2 cfs1 cfs2 rule: ctyping2.induct)
         apply fastforce
        apply (erule ctyping2-correct-aux-assign, assumption+)
       apply (erule ctyping2-correct-aux-input, assumption+)
      apply (erule ctyping2-correct-aux-output, assumption+)
     apply (erule ctyping2-correct-aux-seq, assumption+)
    apply (erule ctyping2-correct-aux-or, assumption+)
   apply (erule ctyping2-correct-aux-if, assumption+)
  apply (erule ctyping2-correct-aux-while, assumption+)
  done


theorem ctyping2-correct:
  assumes A: (U, v) ⊨ c (⊆ A, X) = Some (B, Y)
  shows correct c A X
proof (subst correct-def, clarify)
```

fix $s\ t\ c_1\ c_2\ s_1\ s_2\ f\ vs\ vs_1\ vs_2\ ws\ ws_1\ ws_2\ cfs_2\ t_1\ f'\ vs_1'\ ws_1'$
let $?cs = \mathit{flow}\ cfs_2$
assume $t \in A$ **and** $s = t\ (\subseteq \mathit{state} \cap X)$
hence $s \in \mathit{Univ}\ A\ (\subseteq \mathit{state} \cap X)$
  by *blast*
moreover assume $(c, s, f, vs, ws) \to\!* (c_1, s_1, f, vs_1, ws_1)$
then obtain cfs_1 **where** $(c, s, f, vs, ws) \to\!*\{cfs_1\}\ (c_1, s_1, f, vs_1, ws_1)$
  by (*blast dest*: *small-steps-stepsl*)
moreover assume $(c_1, s_1, f, vs_1, ws_1) \to\!*\{cfs_2\}\ (c_2, s_2, f, vs_2, ws_2)$
ultimately have $\mathit{ok\text{-}flow\text{-}aux}\ U\ c_1\ c_2\ s_1\ s_2\ f\ vs_1\ vs_2\ ws_1\ ws_2\ ?cs$
  by (*rule ctyping2-correct-aux* [*OF A*])
then obtain c_2' **and** t_2 **and** vs_2' **and** ws_2' **where**
 $\mathit{ok\text{-}flow\text{-}aux\text{-}1}\ c_1\ c_2\ c_2'\ s_1\ t_1\ t_2\ f\ f'\ vs_1\ vs_1'\ vs_2\ vs_2'\ ws_1'\ ws_2'\ ?cs\ \wedge$
  $\mathit{ok\text{-}flow\text{-}aux\text{-}2}\ s_1\ s_2\ t_1\ t_2\ f\ f'\ vs_1\ vs_1'\ ?cs\ \wedge$
  $\mathit{ok\text{-}flow\text{-}aux\text{-}3}\ s_1\ t_1\ f\ f'\ vs_1\ vs_1'\ ws_1\ ws_1'\ ws_2\ ws_2'\ ?cs$
  (**is** $?P1 \wedge ?P2 \wedge ?P3$)
  by *fastforce*
hence $?P1$ **and** $?P2$ **and** $?P3$ **by** *auto*
show $\exists\, c_2'\ t_2\ vs_2'\ ws_2'.$
  $\mathit{ok\text{-}flow\text{-}1}\ c_1\ c_2\ c_2'\ s_1\ s_2\ t_1\ t_2\ f\ f'\ vs_1\ vs_1'\ vs_2\ vs_2'\ ws_1'\ ws_2'\ ?cs\ \wedge$
  $\mathit{ok\text{-}flow\text{-}2}\ c_1\ c_2\ c_2'\ s_1\ t_1\ t_2\ f\ f'\ vs_1\ vs_1'\ vs_2\ vs_2'\ ws_1\ ws_1'\ ws_2\ ws_2'\ ?cs$
proof (*rule exI* [*of* - c_2'], *rule exI* [*of* - t_2],
 rule exI [*of* - vs_2'], *rule exI* [*of* - ws_2'])
 {
  fix S
  assume
   B: $S \neq \{\}$ **and**
   C: $S \subseteq \{x.\ s_1 = t_1\ (\subseteq \mathit{sources}\ ?cs\ vs_1\ s_1\ f\ x)\}$ **and**
   D: $f = f'\ (\subseteq vs_1, vs_1', \bigcup\ \{\mathit{tags}\ ?cs\ vs_1\ s_1\ f\ x \mid x.\ x \in S\})$
    (**is** - = - ($\subseteq$ -, -, $?T$))
  have $\forall x.\ \mathit{sources\text{-}aux}\ ?cs\ vs_1\ s_1\ f\ x \subseteq \mathit{sources}\ ?cs\ vs_1\ s_1\ f\ x$
   by (*blast intro*: *subsetD* [*OF sources-aux-sources*])
  hence $S \subseteq \{x.\ s_1 = t_1\ (\subseteq \mathit{sources\text{-}aux}\ ?cs\ vs_1\ s_1\ f\ x)\}$
   using C **by** *blast*
  moreover have $\bigcup\ \{\mathit{tags\text{-}aux}\ ?cs\ vs_1\ s_1\ f\ x \mid x.\ x \in S\} \subseteq ?T$
   (**is** $?T' \subseteq$ -)
   by (*blast intro*: *subsetD* [*OF tags-aux-tags*])
  with D **have** $f = f'\ (\subseteq vs_1, vs_1', ?T')$
   by (*rule eq-streams-subset*)
  ultimately have
   $(c_1, t_1, f', vs_1', ws_1') \to\!* (c_2', t_2, f', vs_2', ws_2')\ \wedge$
   $(c_2 = \mathit{SKIP}) = (c_2' = \mathit{SKIP})\ \wedge$
   $\mathit{map}\ \mathit{fst}\ [p \leftarrow \mathit{drop}\ (\mathit{length}\ vs_1)\ vs_2.\ \mathit{fst}\ p \in S] =$
    $\mathit{map}\ \mathit{fst}\ [p \leftarrow \mathit{drop}\ (\mathit{length}\ vs_1')\ vs_2'.\ \mathit{fst}\ p \in S]$
   (**is** $?Q$)
   using B **and** ‹$?P1$› **by** *simp*
  moreover have $s_2 = t_2\ (\subseteq S)$
   using B **and** C **and** D **and** ‹$?P2$› **by** *simp*
  ultimately have $?Q \wedge ?this$ **..**

```
  }
  moreover {
    fix S
    assume
      B: S ≠ {} and
      C: S ⊆ {x. s1 = t1 (⊆ sources-out ?cs vs1 s1 f x)} and
      D: f = f' (⊆ vs1, vs1', ⋃ {tags-out ?cs vs1 s1 f x | x. x ∈ S})
        (is - = - (⊆ -, -, ?T))
    have ∀x. sources-aux ?cs vs1 s1 f x ⊆ sources-out ?cs vs1 s1 f x
      by (blast intro: subsetD [OF sources-aux-sources-out])
    hence S ⊆ {x. s1 = t1 (⊆ sources-aux ?cs vs1 s1 f x)}
      using C by blast
    moreover have ⋃ {tags-aux ?cs vs1 s1 f x | x. x ∈ S} ⊆ ?T
      (is ?T' ⊆ -)
      by (blast intro: subsetD [OF tags-aux-tags-out])
    with D have f = f' (⊆ vs1, vs1', ?T')
      by (rule eq-streams-subset)
    ultimately have
     (c1, t1, f', vs1', ws1') →* (c2', t2, f', vs2', ws2') ∧
      (c2 = SKIP) = (c2' = SKIP) ∧
      map fst [p←drop (length vs1) vs2. fst p ∈ S] =
        map fst [p←drop (length vs1') vs2'. fst p ∈ S]
      (is ?Q)
      using B and ‹?P1› by simp
    moreover have
     [p←drop (length ws1) ws2. fst p ∈ S] =
        [p←drop (length ws1') ws2'. fst p ∈ S]
      using B and C and D and ‹?P3› by simp
    ultimately have ?Q ∧ ?this ..
  }
  ultimately show
   ok-flow-1 c1 c2 c2' s1 s2 t1 t2 f f' vs1 vs1' vs2 vs2' ws1' ws2' ?cs ∧
    ok-flow-2 c1 c2 c2' s1 t1 t2 f f' vs1 vs1' vs2 vs2' ws1 ws1' ws2 ws2' ?cs
    by auto
 qed
qed

end

end
```

References

[1] C. Ballarin. *Tutorial to Locales and Locale Interpretation.* https://isabelle.in.tum.de/website-Isabelle2024/dist/Isabelle2024/doc/locales.pdf.

[2] A. Krauss. *Defining Recursive Functions in Isabelle/HOL.*

https://isabelle.in.tum.de/website-Isabelle2024/dist/Isabelle2024/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*, May 2024. https://isabelle.in.tum.de/website-Isabelle2024/dist/Isabelle2024/doc/prog-prove.pdf.

[5] T. Nipkow and G. Klein. Theory HOL-IMP.Big_Step (included in the Isabelle2024 distribution). https://isabelle.in.tum.de/website-Isabelle2024/dist/library/HOL/HOL-IMP/Big_Step.html.

[6] T. Nipkow and G. Klein. Theory HOL-IMP.Com (included in the Isabelle2024 distribution). https://isabelle.in.tum.de/website-Isabelle2024/dist/library/HOL/HOL-IMP/Com.html.

[7] T. Nipkow and G. Klein. Theory HOL-IMP.Small_Step (included in the Isabelle2024 distribution). https://isabelle.in.tum.de/website-Isabelle2024/dist/library/HOL/HOL-IMP/Small_Step.html.

[8] T. Nipkow and G. Klein. *Concrete Semantics with Isabelle/HOL*. Springer-Verlag, Feb. 2023. (Current version: http://www.concrete-semantics.org/concrete-semantics.pdf).

[9] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, May 2024. https://isabelle.in.tum.de/website-Isabelle2024/dist/Isabelle2024/doc/tutorial.pdf.

[10] P. Noce. Information Flow Control via Stateful Intransitive Noninterference in Language IMP. *Archive of Formal Proofs*, Feb. 2024. https://isa-afp.org/entries/IMP_Noninterference.html, Formal proof development.

[11] D. Volpano and G. Smith. Eliminating Covert Flows with Minimum Typings. In *Proc. 10th IEEE Computer Security Foundations Workshop*, June 1997.

[12] D. Volpano, G. Smith, and C. Irvine. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, Jan. 1996.