Information Flow Control via Stateful Intransitive Noninterference 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

The scope of information flow control via static type systems is in principle much broader than information flow security, since this concept promises to cope with information flow correctness in full generality. Such a correctness policy can be expressed by extending the notion of a single stateless level-based interference relation applying throughout a program – addressed by the static security type systems described by Volpano, Smith, and Irvine, and formalized in Nipkow and Klein's book on formal programming language semantics (in the version of February 2023) – to that of a stateful interference function mapping program states to (generally) intransitive interference relations.

This paper studies information flow control via stateful intransitive noninterference. First, the notion of termination-sensitive information flow security with respect to a level-based interference relation is generalized to that of termination-sensitive information flow correctness with respect to such a correctness policy. Then, a static type system is specified and is proven to be capable of enforcing such policies. Finally, the information flow correctness notion and the static type system introduced here are proven to degenerate to the counterparts formalized in Nipkow and Klein's book in case of a stateless level-based information flow correctness policy. Although the operational semantics 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 Underlying concepts and formal definitions

theory *Definitions*
imports *HOL−IMP.Small-Step*
begin
⟨*ML*⟩

In a passage of his book *Clean Architecture: A Craftsman's Guide to Software Structure and Design* (Prentice Hall, 2017), Robert C. Martin defines a computer program as "a detailed description of the policy by which inputs are transformed into outputs", remarking that "indeed, at its core, that's all a computer program actually is". Accordingly, the scope of information flow control via static type systems is in principle much broader than language-based information flow security, since this concept promises to cope with information flow correctness in full generality.

This is already shown by a basic program implementing the Euclidean algorithm, in Donald Knuth's words "the granddaddy of all algorithms, because it is the oldest nontrivial algorithm that has survived to the present day" (from *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, third edition, Addison-Wesley, 1997). Here below is a sample such C program, where variables `a` and `b` initially contain two positive integers and `a` will finally contain the output, namely the greatest common divisor of those integers.

```
1 do
2 {
3     r = a % b;
4     a = b;
5     b = r;
6 } while (b);
```

Even in a so basic program, information is not allowed to indistinctly flow from any variable to any other one, on pain of the program being incorrect. If an incautious programmer swapped `a` for `b` in the assignment at line 4, the greatest common divisor output for any two inputs `a` and `b` would invariably match `a`, whereas swapping the sides of the assignment at line 5 would give rise to an endless loop. Indeed, despite the marked differences in the resulting program behavior, both of these potential errors originate in information flowing between variables along paths other than the demanded ones. A sound implementation of the Euclidean algorithm does not provide for any information flow from `a` to `b`, or from `b` to `r`.

The static security type systems addressed in [11], [10], and [7] restrict the information flows occurring in a program based on a mapping of each of its variables to a *domain* along with an *interference relation* between such domains, including any pair of domains such that the former may interfere with the latter. Accordingly, if function *dom* stands for such a mapping, and infix notation $u \rightsquigarrow v$ denotes the inclusion of any pair of domains (u, v) in such a relation (both notations are borrowed from [9]), the above errors would be detected at compile time by a static type system enforcing an interference relation such that:

- $dom\ a \rightsquigarrow dom\ r$, $dom\ b \rightsquigarrow dom\ r$ (line 3),
- $dom\ b \rightsquigarrow dom\ a$ (line 4),
- $dom\ r \rightsquigarrow dom\ b$ (line 5),

and ruling out any other pair of distinct domains. Such an interference relation would also embrace the implicit information flow from `b` to the other two variables arising from the loop's termination condition (line 6).

Remarkably, as $dom\ a \rightsquigarrow dom\ r$ and $dom\ r \rightsquigarrow dom\ b$ but $\neg\ dom\ a \rightsquigarrow dom\ b$, this interference relation turns out to be intransitive. Therefore, unlike the security static type systems studied in [11] and [10], which deal with *level-based*, and then *transitive*, interference relations, a static type system aimed at enforcing information flow correctness in full generality must be capable of dealing with *intransitive* interference relations as well. This should come as no surprise, since [9] shows that this is the general case already for interference relations expressing information flow security policies.

But the bar can be raised further. Considering the above program again, the information flows needed for its operation, as listed above, need not be allowed throughout the program. Indeed, information needs to flow from `a` and `b` to `r` at line 3, from `b` to `a` at line 4, from `r` to `b` at line 5, and then (implicitly) from `b` to the other two variables at line 6. Based on this observation, error detection at compile time can be made finer-grained by rewriting the program as follows, where `i` is a further integer variable introduced for this purpose.

```
1  do
2  {
3      i = 0;
4      r = a % b;
5      i = 1;
6      a = b;
7      i = 2;
8      b = r;
9      i = 3;
10 } while (b);
```

In this program, `i` serves as a state variable whose value in every execution step can be determined already at compile time. Since a distinct set of information flows is allowed for each of its values, a finer-grained information flow correctness policy for this program can be expressed by extending the concept of a single, *stateless* interference relation applying throughout the program to that of a *stateful interference function* mapping program states to interference relations (in this case, according to the value of `i`). As a result of this extension, for each program state, a distinct interference relation – that is, the one to which the applied interference function maps that state – can be enforced at compile time by a suitable static type system.

If mixfix notation s: $u \rightsquigarrow v$ denotes the inclusion of any pair of domains (u, v) in the interference relation associated with any state s, a finer-grained information flow correctness policy for this program can then be expressed as an interference function such that:

- s: $\mathit{dom}\ a \rightsquigarrow \mathit{dom}\ r$, s: $\mathit{dom}\ b \rightsquigarrow \mathit{dom}\ r$ for any s where $\texttt{i} = 0$ (line 4),
- s: $\mathit{dom}\ b \rightsquigarrow \mathit{dom}\ a$ for any s where $\texttt{i} = 1$ (line 6),
- s: $\mathit{dom}\ r \rightsquigarrow \mathit{dom}\ b$ for any s where $\texttt{i} = 2$ (line 8),
- s: $\mathit{dom}\ b \rightsquigarrow \mathit{dom}\ a$, s: $\mathit{dom}\ b \rightsquigarrow \mathit{dom}\ r$, s: $\mathit{dom}\ b \rightsquigarrow \mathit{dom}\ i$ for any s where $\texttt{i} = 3$ (line 10),

and ruling out any other pair of distinct domains in any state.

Notably, to enforce such an interference function, a static type system would not need to keep track of the full program state in every program execution step (which would be unfeasible, as the values of `a`, `b`, and `r` cannot be determined at compile time), but only of the values of some specified state variables (in this case, of `i` alone). Accordingly, term *state variable* will henceforth refer to any program variable whose value may affect that of the interference function expressing the information flow correctness policy in force, namely the interference relation to be applied.

Needless to say, there would be something artificial about the introduction of such a state variable into the above sample program, since it is indeed so basic as not to provide for a state machine on its own, so that `i` would be aimed exclusively at enabling the enforcement of such an information flow correctness policy. Yet, real-world imperative programs, for which error detection at compile time is truly meaningful, *do* typically provide for state machines such that only a subset of all the potential information flows is allowed in each state; and even for those which do not, the addition of some *ad hoc* state variable to enforce such a policy could likely be an acceptable trade-off.

Accordingly, the goal of this paper is to study information flow control via stateful intransitive noninterference. First, the notion of termination-sensitive information flow security with respect to a level-based interference relation, as defined in [7], section 9.2.6, is generalized to that of termination-sensitive information flow correctness with respect to a stateful interference function having (generally) intransitive interference relations as values. Then, a static type system is specified and is proven to be capable of enforcing such information flow correctness policies. Finally, the information flow correctness notion and the static type system introduced here are proven to degenerate to the counterparts addressed in [7], section 9.2.6, in case of a stateless level-based information flow correctness policy.

Although the operational semantics of the didactic imperative programming language IMP employed in [7] is used for this purpose, the introduced concepts are applicable to larger, real-world imperative programming languages as well, by just affording the additional type system complexity arising from richer language constructs. Accordingly, the informal explanations accompanying formal content in what follows will keep making use of sample C code snippets.

For further information about the formal definitions and proofs contained in this paper, see Isabelle documentation, particularly [8], [4], [2], [3], and [1].

1.1 Global context definitions

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

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

type-synonym *flow* = *com-flow list*
type-synonym *config* = *state set* × *vname set*
type-synonym *scope* = *config set* × *bool*

abbreviation *eq-states* :: *state* ⇒ *state* ⇒ *vname set* ⇒ *bool*
(‹(- = - ′(⊆ -′))› [*51*, *51*] *50*) **where**
$s = t\ (\subseteq X) \equiv \forall x \in X.\ s\ x = t\ x$

abbreviation *univ-states* :: *state set* ⇒ *vname set* ⇒ *state set*
(‹(*Univ* - ′(⊆ -′))› [*51*] *75*) **where**
$Univ\ A\ (\subseteq X) \equiv \{s.\ \exists t \in A.\ s = t\ (\subseteq X)\}$

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*)

fun *run-flow* :: *flow* ⇒ *state* ⇒ *state* **where**
run-flow (*x* ::= *a* # *cs*) *s* = *run-flow cs* (*s*(*x* := *aval a s*)) |
run-flow (- # *cs*) *s* = *run-flow cs s* |
run-flow - *s* = *s*

primrec *no-upd* :: *flow* ⇒ *vname* ⇒ *bool* **where**
no-upd (*c* # *cs*) *x* =
((*case c of y* ::= - ⇒ *y* ≠ *x* | - ⇒ *True*) ∧ *no-upd cs x*) |
no-upd [] - = *True*

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

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

fun *flow-aux* :: *com list* ⇒ *flow* **where**
flow-aux ((*x* ::= *a*) # *cs*) = (*x* ::= *a*) # *flow-aux cs* |
flow-aux ((*IF b THEN - ELSE -*) # *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* × *state*) *list* ⇒ *flow* **where**
flow cfs = *flow-aux* (*map fst cfs*)

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

⟨*proof*⟩
termination ⟨*proof*⟩

lemmas *small-stepsl-induct* = *small-stepsl.induct* [*split-format*(*complete*)]

1.2 Local context definitions

In what follows, stateful intransitive noninterference will be formalized within the local context defined by means of a *locale* [1], named *noninterf*. Later on, this will enable to prove the degeneracy of the following definitions to the stateless level-based counterparts addressed in [11], [10], and [7], and formalized in [5] and [6], via a suitable locale interpretation.

Locale *noninterf* contains three parameters, as follows.

- A stateful interference function *interf* mapping program states to *interference predicates* of two domains, intended to be true just in case the former domain is allowed to interfere with the latter.
- A function *dom* mapping program variables to their respective domains.
- A set *state* collecting all state variables.

As the type of the domains is modeled using a type variable, it may be assigned arbitrarily by any locale interpretation, which will enable to set it to *nat* upon proving degeneracy. Moreover, the above mixfix notation *s*: *u* ⇝ *v* is adopted to express the fact that any two domains *u*, *v* satisfy the interference predicate *interf s* associated with any state *s*, namely the fact that *u* is allowed to interfere with *v* in state *s*.

Locale *noninterf* also contains an assumption, named *interf-state*, which serves the purpose of supplying parameter *state* with its intended semantics, namely standing for the set of all state variables. The assumption is that function *interf* maps any two program states agreeing on the values of all the variables in set *state* to the same interference predicate. Correspondingly, any locale interpretation instantiating parameter *state* as the empty set must instantiate parameter *interf* as a function mapping any two program states, even if differing in the values of all variables, to the same interference predicate – namely, as a constant function. Hence, any such locale interpretation refers to a single, stateless interference predicate applying throughout the program. Unsurprisingly, this is the way how those parameters will be instantiated upon proving degeneracy.

The one just mentioned is the only locale assumption. Particularly, the following formalization does not rely upon the assumption that the interference predicates returned by function *interf* be *reflexive*, although this will be the case for any meaningful real-world information flow correctness policy.

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

Locale parameters *interf* and *dom* are provided with their intended semantics by the definitions of functions *sources* and *correct*, which are formalized here below based on the following underlying ideas.

As long as a stateless transitive interference relation between domains is considered, the condition for the correctness of the value of a variable resulting from a full or partial program execution need not take into account the execution flow producing it, but rather the initial program state only. In fact, this is what happens with the stateless level-based correctness condition addressed in [11], [10], and [7]: the resulting value of a variable of level l is correct if the same value is produced for any initial state agreeing with the given one on the value of every variable of level not higher than l.

Things are so simple because, for any variables x, y, and z, if *dom z* ⇝ *dom y* and *dom y* ⇝ *dom x*, transitivity entails *dom z* ⇝ *dom x*, and these interference relationships hold statelessly. Therefore, z may be counted among the variables whose initial values are allowed to affect `x` independently of whether some intermediate value of `y` may affect `x` within the actual execution flow.

Unfortunately, switching to stateful intransitive interference relations puts an end to that happy circumstance – indeed, even statefulness or intransitivity alone would suffice for this sad ending. In this context, deciding about the correctness of the resulting value of a variable `x` still demands the detection of the variables whose initial values are allowed to interfere with `x`, but the execution flow leading from the initial program state to the resulting one needs to be considered to perform such detection.

This is precisely the task of function *sources*, so named after its finite state machine counterpart defined in [9]. It takes as inputs an execution flow cs, an initial program state s, and a variable `x`, and outputs the set of the variables whose values in s are allowed to affect the value of `x` in the state s' into which cs turns s, according to cs as well as to the information flow correctness policy expressed by parameters *interf* and *dom*.

In more detail, execution flows are modeled as lists comprising items of two possible kinds, namely an assignment of the value of an arithmetic expression a to a variable `z` or else an *observation* of the values of the variables in a set X, denoted through notations $z ::= a$ (same as with assignment commands) and $\langle X \rangle$ and keeping track of explicit and implicit information flows, respectively. Particularly, item $\langle X \rangle$ refers to the act of observing the values of the variables in X leaving the program state unaltered. During the execution of an IMP program, this happens upon any evaluation of a boolean expression containing all and only the variables in X.

Function *sources* is defined along with an auxiliary function *sources-aux* by means of mutual recursion. Based on this definition, *sources* $cs\ s\ x$ 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 $i \leq n$, c_i is an assignment $y_{i+1} ::= a_i$ where:
  - $y_i \in avars\ a_i$,
  - *run-flow* cs_i s: *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 to variable y_{i+1} (as assignment c_i would otherwise be irrelevant),

  or else an observation $\langle X_i \rangle$ where:

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

- cs_1 does not comprise any assignment to variable y.

In addition, *sources cs s x* contains variable x also if cs does not comprise any assignment to variable x.

function
  sources :: *flow* ⇒ *state* ⇒ *vname* ⇒ *vname set* **and**
  sources-aux :: *flow* ⇒ *state* ⇒ *vname* ⇒ *vname set* **where**

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

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

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

sources-aux [] - - = {}

⟨*proof*⟩

termination ⟨*proof*⟩

lemmas *sources-induct* = *sources-sources-aux.induct*

Predicate *correct* takes as inputs a program c, a set of program states A, and a set of variables X. Its truth value equals that of the following termination-sensitive information flow correctness condition: for any state s agreeing with a state in A on the values of the state variables in X, if the *small-step* program semantics turns configuration (c, s) into configuration (c_1, s_1), and (c_1, s_1) into configuration (c_2, s_2), then for any state t_1 agreeing with s_1 on the values of the variables in *sources cs* s_1 x, where cs is the execution flow leading from (c_1, s_1) to (c_2, s_2), the small-step semantics turns (c_1, t_1) into some configuration (c_2', t_2) such that:

- $c_2' = SKIP$ (namely, (c_2', t_2) is a *final* configuration) just in case $c_2 = SKIP$, and

- the value of variable `x` in state t_2 is the same as in state s_2.

Here below are some comments about this definition.

- As *sources cs* s_1 x is the set of the variables whose values in s_1 are allowed to affect the value of `x` in s_2, this definition requires any state t_1 indistinguishable from s_1 in the values of those variables to produce a state where variable `x` has the same value as in s_2 in the continuation of program execution.

- Configuration (c_2', t_2) must be the same one for *any* variable `x` such that s_1 and t_1 agree on the values of any variable in *sources cs* s_1 x. Otherwise, even if states s_2 and t_2 agreed on the value of `x`, they could be distinguished all the same based on a discrepancy between the respective values of some other variable. Likewise, if state t_2 alone had to be the same for any such `x`, while command c_2' were allowed to vary, state t_1 could be distinguished from s_1 based on the continuation of program execution. This is the reason why the universal quantification over x is nested within the existential quantification over both c_2' and t_2.

- The state machine for a program typically provides for a set of initial states from which its execution is intended to start. In any such case, information flow correctness need not be assessed for arbitrary initial states, but just for those complying with the settled tuples of initial values for state variables. The values of any other variables do not matter, as they do not affect function *interf*'s ones. This is the motivation for parameter A, which then needs to contain just one state for each of such tuples, while parameter X enables to exclude the state variables, if any, whose initial values are not settled.

- If locale parameter *state* matches the empty set, s will be any state agreeing with some state in A on the value of possibly even no variable at all, that is, a fully arbitrary state provided that A is nonempty. This makes s range over all possible states, as required for establishing the degeneracy of the present definition to the stateless level-based counterpart addressed in [7], section 9.2.6.

Why express information flow correctness in terms of the small-step program semantics, instead of resorting to the big-step one as happens with the stateless level-based correctness condition in [7], section 9.2.6? The answer is provided by the following sample C programs, where `i` is a state variable.

```
y = i;
i = (i) ? 1 : 0;
```

```
x = i + y;
```

```
x = 0;
if (i == 10)
{
  x = 10;
}
i = (i) ? 1 : 0;
x += i;
```

Let `i` be allowed to interfere with `x` just in case `i` matches 0 or 1, and `y` be never allowed to do so. If s_1 were constrained to be the initial state, for both programs `i` would be included among the variables on which t_1 needs to agree with s_1 in order to be indistinguishable from s_1 in the value of `x` resulting from the final assignment. Thus, both programs would fail to be labeled as wrong ones, although in both of them the information flow blatantly bypasses the sanitization of the initial value of `i`, respectively due to an illegal explicit flow and an illegal implicit flow. On the contrary, the present information flow correctness definition detects any such illegal information flow by checking every partial program execution on its own.

abbreviation *ok-flow* :: *com* ⇒ *com* ⇒ *state* ⇒ *state* ⇒ *flow* ⇒ *bool* **where**
$\textit{ok-flow}\ c_1\ c_2\ s_1\ s_2\ cs \equiv$
$\quad \forall t_1.\ \exists c_2'\ t_2.\ \forall x.$
$\quad\quad s_1 = t_1\ (\subseteq \textit{sources}\ cs\ s_1\ x) \longrightarrow$
$\quad\quad\quad (c_1,\ t_1) \rightarrow\!*\ (c_2',\ t_2) \wedge (c_2 = \textit{SKIP}) = (c_2' = \textit{SKIP}) \wedge s_2\ x = t_2\ x$

definition *correct* :: *com* ⇒ *state set* ⇒ *vname set* ⇒ *bool* **where**
$\textit{correct}\ c\ A\ X \equiv$
$\quad \forall s \in \textit{Univ}\ A\ (\subseteq \textit{state} \cap X).\ \forall c_1\ c_2\ s_1\ s_2\ \textit{cfs}.$
$\quad\quad (c,\ s) \rightarrow\!*\ (c_1,\ s_1) \wedge (c_1,\ s_1) \rightarrow\!*\{\textit{cfs}\}\ (c_2,\ s_2) \longrightarrow$
$\quad\quad\quad \textit{ok-flow}\ c_1\ c_2\ s_1\ s_2\ (\textit{flow}\ \textit{cfs})$

abbreviation *interf-set* :: *state set* ⇒ *'d set* ⇒ *'d set* ⇒ *bool*
$\quad$ (‹(-: - ⇝ -)› [51, 51, 51] 50) **where**
$A\colon U \leadsto W \equiv \forall s \in A.\ \forall u \in U.\ \forall w \in W.\ s\colon u \leadsto w$

abbreviation *ok-flow-aux* ::
$\quad$ *config set* ⇒ *com* ⇒ *com* ⇒ *state* ⇒ *state* ⇒ *flow* ⇒ *bool* **where**
$\textit{ok-flow-aux}\ U\ c_1\ c_2\ s_1\ s_2\ cs \equiv$
$\quad (\forall t_1.\ \exists c_2'\ t_2.\ \forall x.$
$\quad\quad (s_1 = t_1\ (\subseteq \textit{sources-aux}\ cs\ s_1\ x) \longrightarrow$
$\quad\quad\quad (c_1,\ t_1) \rightarrow\!*\ (c_2',\ t_2) \wedge (c_2 = \textit{SKIP}) = (c_2' = \textit{SKIP})) \wedge$
$\quad\quad (s_1 = t_1\ (\subseteq \textit{sources}\ cs\ s_1\ x) \longrightarrow s_2\ x = t_2\ x)) \wedge$

$$(\forall\, x.\ (\exists\, p \in U.\ \mathit{case}\ p\ \mathit{of}\ (B,\ Y) \Rightarrow$$
$$\exists\, s \in B.\ \exists\, y \in Y.\ \neg\ s\colon \mathit{dom}\ y \rightsquigarrow \mathit{dom}\ x) \longrightarrow \mathit{no\text{-}upd}\ \mathit{cs}\ x)$$

The next step is defining a static type system guaranteeing that well-typed programs satisfy this information flow correctness criterion. Whenever defining a function, and the pursued type system is obviously no exception, the primary question that one has to answer is: which inputs and outputs should it provide for? The type system formalized in [6] simply makes a pass/fail decision on an input program, based on an input security level, and outputs the verdict as a boolean value. Is this still enough in the present case? The answer can be found by considering again the above C program that computes the greatest common divisor of two positive integers `a`, `b` using a state variable `i`, along with its associated stateful interference function. For the reader's convenience, the program is reported here below.

```
1  do
2  {
3      i = 0;
4      r = a % b;
5      i = 1;
6      a = b;
7      i = 2;
8      b = r;
9      i = 3;
10 } while (b);
```

As s: *dom a* $\rightsquigarrow$ *dom r* only for a state s where `i` $= 0$, the type system cannot determine that the assignment `r = a % b` at line 4 is well-typed without knowing that `i` $= 0$ whenever that step is executed. Consequently, upon checking the assignment `i = 0` at line 3, the type system must output information indicating that `i` $= 0$ as a result of its execution. This information will then be input to the type system when it is recursively invoked to check line 4, so as to enable the well-typedness of the next assignment to be ascertained.

Therefore, in addition to the program under scrutiny, the type system needs to take a set of program states as input, and as long as the program is well-typed, the output must include a set of states covering any change to the values of the state variables possibly triggered by the input program. In other words, the type system has to *simulate* the execution of the input program at compile time as regards the values of its state variables. In the following formalization, this results in making the type system take an input of type *state set* and output a value of the same type. Yet, since state variables alone are relevant, a real-world implementation of the type system would not need to work with full *state* values, but just with tuples of state variables' values.

Is the input/output of a set of program states sufficient to keep track of the possible values of the state variables at each execution step? Here below is a sample C program helping find an answer, which determines the minimum of two integers `a`, `b` and assigns it to variable `a` using a state variable `i`.

```
1 i = (a > b) ? 1 : 0;
2 if (i > 0)
3 {
4   a = b;
5 }
```

Assuming that the initial value of `i` is 0, the information flow correctness policy for this program will be such that:

- s: *dom a* $\leadsto$ *dom i*, s: *dom b* $\leadsto$ *dom i* for any program state s where `i` $= 0$ (line 1),
- s: *dom i* $\leadsto$ *dom a* for any s where `i` $= 0$ or `i` $= 1$ (line 2, more on this later),
- s: *dom b* $\leadsto$ *dom a* for any s where `i` $= 1$ (line 4),

ruling out any other pair of distinct domains in any state.

So far, everything has gone smoothly. However, what happens if the program is changed as follows?

```
1 i = a - b;
2 if (i > 0)
3 {
4   a = b;
5 }
```

Upon simulating the execution of the former program, the type system can determine the set $\{0, 1\}$ of the possible values of variable `i` arising from the conditional assignment `i = (a > b) ? 1 : 0` at line 1. On the contrary, in the case of the latter program, the possible values of `i` after the assignment `i = a - b` at line 1 must be marked as being *indeterminate*, since they depend on the initial values of variables `a` and `b`, which are unknown at compile time. Hence, the type system needs to provide for an additional input/output parameter of type *vname set*, whose input and output values shall collect the variables whose possible values before and after the execution of the input program are *determinate*.

The correctness of the simulation of program execution by the type system can be expressed as the following condition. Suppose that the type system outputs a *state set* A' and a *vname set* X' when it is input a program c, a *state set* A, and a *vname set* X. Then, for any state s agreeing with some state in A on the value of every state variable in X, if $(c,\ s) \Rightarrow s'$, s' must agree with some state in A' on the value of every state variable in X'. This can be summarized by saying that the type system must *overapproximate* program semantics, since any algorithm simulating program execution cannot but be imprecise (see [7], *incipit* of chapter 13).

In turn, if the outputs for c, A', X' are A'', X'' and $(c,\ s') \Rightarrow s''$, s'' must agree with some state in A'' on the value of every state variable in X''. But if c is a loop and $(c,\ s) \Rightarrow s'$, then $(c,\ s') \Rightarrow s''$ just in case $s' = s''$, so that the type system is guaranteed to overapproximate the semantics of c only if states consistent with A', X' are also consistent with A'', X'' and vice versa. Thus, the type system needs to be *idempotent* if c is a loop, that is, it must be such that $A' = A''$ and $X' = X''$ in this case. Since idempotence is not required for control structures other than loops, the main type system *ctyping2* formalized in what follows will delegate the simulation of the execution of loop bodies to an auxiliary, idempotent type system *ctyping1*.

This type system keeps track of the program state updates possibly occurring in its input program using sets of lists of functions of type *vname* $\Rightarrow$ *val option option*. Command *SKIP* is mapped to a singleton made of the empty list, as no state update takes place. An assignment to a variable `x` is mapped to a singleton made of a list comprising a single function, whose value is *Some* (*Some* i) or *Some None* for `x` if it is a state variable and the right-hand side is a constant $N\ i$ or a non-constant expression, respectively, and *None* otherwise. That is, *None* stands for *unchanged/non-state variable* (remember, only state variable updates need to be tracked), whereas *Some None* stands for *indeterminate variable*, since the value of a non-constant expression in a loop iteration (remember, *ctyping1* is meant for simulating the execution of loop bodies) is in general unknown at compile time.

At first glance, a conditional statement could simply be mapped to the union of the sets tracking the program state updates possibly occurring in its branches. However, things are not so simple, as shown by the sample C loop here below, which has a conditional statement as its body.

```
for (i = 0; i < 2; i++)
{
  if (n % 2)
  {
```

```
    a = 1;
    b = 1;
    n++;
  }
  else
  {
    a = 2;
    c = 2;
    n++;
  }
}
```

If the initial value of the integer variable n is even, the final values of variables a, b, and c will be 1, 1, 2, whereas if the initial value of n is odd, the final values of the aforesaid variables will be 2, 1, 2. Assuming that their initial value is 0, the potential final values tracked by considering each branch individually are 1, 1, 0 and 2, 0, 2 instead. These are exactly the values generated by a single loop iteration; if they are fed back into the loop body along with the increased value of n, the actual final values listed above are produced.

As a result, a mere union of the sets tracking the program state updates possibly occurring in each branch would not be enough for the type system to be idempotent. The solution is to rather construct every possible alternate concatenation without repetitions of the lists contained in each set, which is referred to as *merging* those sets in the following formalization. In fact, alternating the state updates performed by each branch in the previous example produces the actual final values listed above. Since the latest occurrence of a state update makes any previous occurrence irrelevant for the final state, repetitions need not be taken into account, which ensures the finiteness of the construction provided that the sets being merged are finite. In the special case where the boolean condition can be evaluated at compile time, considering the picked branch alone is of course enough.

Another case trickier than what one could expect at first glance is that of sequential composition. This is shown by the sample C loop here below, whose body consists of the sequential composition of some assignments with a conditional statement.

```
for (i = 0; i < 2; i++)
{
  a = 1;
  b = 1;
  if (n % 2)
  {
    a = 2;
```

```
    c = 2;
    n++;
  }
  else
  {
    b = 3;
    d = 3;
    n++;
  }
}
```

If the initial value of the integer variable `n` is even, the final values of variables `a`, `b`, `c`, and `d` will be 2, 1, 2, 3, whereas if the initial value of `n` is odd, the final values of the aforesaid variables will be 1, 3, 2, 3. Assuming that their initial value is 0, the potential final values tracked by considering the sequences of the state updates triggered by the starting assignments with the updates, simulated as described above, possibly triggered by the conditional statement rather are:

- 2, 1, 2, 0,
- 1, 3, 0, 3,
- 2, 3, 2, 3.

The first two tuples of values match the ones generated by a single loop iteration, and produce the actual final values listed above if they are fed back into the loop body along with the increased value of `n`.

Hence, concatenating the lists tracking the state updates possibly triggered by the first command in the sequence (the starting assignment sequence in the previous example) with the lists tracking the updates possibly triggered by the second command in the sequence (the conditional statement in the previous example) would not suffice for the type system to be idempotent. The solution is to rather append the latter lists to those constructed by *merging* the sets tracking the state updates possibly performed by each command in the sequence. Again, provided that such sets are finite, this construction is finite, too. In the special case where the latter set is a singleton, the aforesaid merging is unnecessary, as it would merely insert a preceding occurrence of the single appended list into the resulting concatenated lists, and such repetitions are irrelevant as observed above.

Surprisingly enough, the case of loops is actually simpler than possible first-glance expectations. A loop defines two branches, namely its body and an implicit alternative branch doing nothing. Thus, it can simply be mapped to the union of the set tracking the state updates possibly occurring in its body with a singleton made of the empty list. As happens with conditional statements, in the special case where the boolean condition can be evaluated at compile time, considering the selected branch alone is obviously enough.

Type system *ctyping1* uses the set of lists resulting from this recursion over the input command to construct a set F of functions of type *vname* $\Rightarrow$ *val option option*, as follows: for each list *ys* in the former set, F contains the function mapping any variable `x` to the rightmost occurrence, if any, of pattern *Some* v to which `x` is mapped by any function in *ys* (that is, to the latest update, if any, of `x` tracked in *ys*), or else to *None*. Then, if A, X are the input *state set* and *vname set*, and B, Y the output ones:

- B is the set of the program states constructed by picking a function f and a state s from F and A, respectively, and mapping any variable `x` to i if $f\ x = Some\ (Some\ i)$, or else to $s\ x$ if $f\ x = None$ (namely, to its value in the initial state s if f marks it as being unchanged).

- Y is *UNIV* if $A = \{\}$ (more on this later), or else the set of the variables not mapped to *Some None* (that is, not marked as being indeterminate) by any function in F, and contained in X (namely, being initially determinate) if mapped to *None* (that is, if marked as being unchanged) by some function in F.

When can *ctyping1* evaluate the boolean condition of a conditional statement or a loop, so as to possibly detect and discard some "dead" branch? This question can be answered by examining the following sample C loop, where `n` is a state variable, while integer `j` is unknown at compile time.

```
for (i = 0; i != j; i++)
{
  if (n == 1)
  {
    n = 2;
  }
  else if (n == 0)
  {
    n = 1;
  }
}
```

Assuming that the initial value of `n` is 0, its final value will be 0, 1, or 2 according to whether `j` matches 0, 1, or any other positive integer, respectively, whereas the loop will not even terminate if `j` is negative. Consequently, the type system cannot avoid tracking the state updates possibly triggered in every branch, on pain of failing to be idempotent. As a result, evaluating the boolean conditions in the conditional statement at compile time so as to discard some branch is not possible, even though they only depend on an initially determinate state variable. The conclusion is that *ctyping1* may generally evaluate boolean conditions just in case they contain constants alone, namely only if they are trivial enough to be possibly eliminated by program optimization. This is exactly what *ctyping1* does by passing any boolean condition found in the input program to the type system *btyping1* for boolean expressions, defined here below as well.

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

$\vdash$ *Bc v* = *Some v* |

$\vdash$ *Not b* = (*case* $\vdash$ *b of*
Some v ⇒ *Some* (¬ *v*) | *-* ⇒ *None*) |

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

$\vdash$ *Less* a_1 a_2 = (*if avars* $a_1 \cup$ *avars* a_2 = {}
then Some (*aval* a_1 ($\lambda x.\ 0$) < *aval* a_2 ($\lambda x.\ 0$)) *else None*)

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

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

xs ∈ *A* ⟹ [(*xs*, *True*)] ∈ *A* $\bigsqcup$ *B* |

ys ∈ *B* ⟹ [(*ys*, *False*)] ∈ *A* $\bigsqcup$ *B* |

⟦*ws* ∈ *A* $\bigsqcup$ *B*; ¬ *snd* (*last ws*); *xs* ∈ *A*; (*xs*, *True*) ∉ *set ws*⟧ ⟹
ws @ [(*xs*, *True*)] ∈ *A* $\bigsqcup$ *B* |

⟦*ws* ∈ *A* $\bigsqcup$ *B*; *snd* (*last ws*); *ys* ∈ *B*; (*ys*, *False*) ∉ *set ws*⟧ ⟹
ws @ [(*ys*, *False*)] ∈ *A* $\bigsqcup$ *B*

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

definition *ctyping1-append* ::
state-upd list set ⇒ *state-upd list set* ⇒ *state-upd list set*
(**infixl** ‹@› *55*) **where**
A @ *B* ≡ {*xs* @ *ys* | *xs ys*. *xs* ∈ *A* ∧ *ys* ∈ *B*}

definition *ctyping1-merge* ::
state-upd list set ⇒ *state-upd list set* ⇒ *state-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-upd list set ⇒ *state-upd list set* ⇒ *state-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* ⇒ *state-upd list set*
(‹(⊢ -)› [*51*] *60*) **where**

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

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

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

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

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

definition *ctyping1-seq* :: *state-upd* ⇒ *state-upd* ⇒ *state-upd*
(**infixl** ‹;;› *55*) **where**
$S;;\ T \equiv \lambda x.\ case\ T\ x\ of\ None \Rightarrow S\ x \mid Some\ v \Rightarrow Some\ v$

definition *ctyping1* :: *com* ⇒ *state set* ⇒ *vname set* ⇒ *config*
(‹(⊢ - '(⊆ -, -'))› [*51*] *55*) **where**
$\vdash c\ (\subseteq A,\ X) \equiv let\ F = \{\lambda x.\ foldl\ (;;)\ (\lambda x.\ None)\ ys\ x \mid ys.\ ys \in \vdash c\}\ in$
$(\{\lambda x.\ case\ f\ x\ of\ None \Rightarrow s\ x \mid Some\ None \Rightarrow t\ x \mid Some\ (Some\ i) \Rightarrow i \mid$
$f\ s\ t.\ f \in F \wedge s \in A\},$
$Univ??\ A\ \{x.\ \forall f \in F.\ f\ x \neq Some\ None \wedge (f\ x = None \longrightarrow x \in X)\})$

A further building block propaedeutic to the definition of the main type system *ctyping2* is the definition of its own companion type system *btyping2* for boolean expressions. The goal of *btyping2* is splitting, whenever feasible at compile time, an input *state set* into two complementary subsets, respectively comprising the program states making the input boolean expression true or false. This enables *ctyping2* to apply its information flow correctness checks to conditional branches by considering only the program states in which those branches are executed.

As opposed to *btyping1*, *btyping2* may evaluate its input boolean expression even if it contains variables, provided that all of their values are known at compile time, namely that all of them are determinate state variables – hence *btyping2*, like *ctyping2*, needs to take a *vname set* collecting determinate variables as an additional input. In fact, in the case of a loop body, the dirty work of covering any nested branch by skipping the evaluation of non-constant boolean conditions is already done by *ctyping1*, so that any *state set* and *vname set* input to *btyping2* already encompass every possible execution flow.

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* b_1 b_2 (⊆ *A*, *X*) = (*case* (⊫ b_1 (⊆ *A*, *X*), ⊫ b_2 (⊆ *A*, *X*)) *of*
(*Some* B_1, *Some* B_2) ⇒ *Some* (B_1 ∩ B_2) | - ⇒ *None*) |

⊫ *Less* a_1 a_2 (⊆ *A*, *X*) = (*if avars* a_1 ∪ *avars* a_2 ⊆ *state* ∩ *X*
then Some {*s*. *s* ∈ *A* ∧ *aval* a_1 *s* < *aval* a_2 *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*)

It is eventually time to define the main type system *ctyping2*. Its output consists of the *state set* of the final program states and the *vname set* of the finally determinate variables produced by simulating the execution of the input program, based on the *state set* of initial program states and the *vname set* of initially determinate variables taken as inputs, if information flow correctness checks are passed; otherwise, the output is *None*.

An additional input is the counterpart of the level input to the security type systems formalized in [6], in that it specifies the *scope* in which information flow correctness is validated. It consists of a set of *state set* × *vname set* pairs and a boolean flag. The set keeps track of the variables contained in the boolean conditions, if any, nesting the input program, in association with the program states in which they are evaluated. The flag is *False* if the input program is nested in a loop, in which case state variables set to non-constant expressions are marked as being indeterminate (as observed previously, the value of a non-constant expression in a loop iteration is in general unknown at compile time).

In the recursive definition of *ctyping2*, the equations dealing with conditional branches, namely those applying to conditional statements and loops, construct the output *state set* and *vname set* respectively as the *union* and the *intersection* of the sets computed for each branch. In fact, a possible final state is any one resulting from either branch, and a variable is finally determinate just in case it is such regardless of the branch being picked. Yet, a "dead" branch should have no impact on the determinateness of variables, as it only depends on the other branch. Accordingly, provided that information flow correctness checks are passed, the cases where the output is constructed non-recursively, namely those of *SKIP*, assignments, and loops, return *UNIV* as *vname set* if the input *state set* is empty. In the case of a loop, the *state set* and the *vname set* resulting from one or more iterations of its body are computed using the auxiliary type system *ctyping1*. This explains why *ctyping1* returns *UNIV* as *vname set* if the input *state set* is empty, as mentioned previously.

As happens with the syntax-directed security type system formalized in [6], the cases performing non-recursive information flow correctness checks are those of assignments and loops. In the former case, *ctyping2* verifies that the sets of variables contained in the scope, as well as any variable occurring in the expression on the right-hand side of the assignment, are allowed to interfere with the variable on the left-hand side, respectively in their associated sets of states and in the input *state set*. In the latter case, *ctyping2* verifies that the sets of variables contained in the scope, as well as any variable occurring in the boolean condition of the loop, are allowed to interfere with *every* variable, respectively in their associated sets of states and in the states in which the boolean condition is evaluated. In both cases, if the applying interference relation is unknown as some state variable is indeterminate, each of those checks must be passed for *any* possible state (unless the respective set of states is empty).

Why do the checks performed for loops test interference with *every* variable? The answer is provided by the following sample C program, which sets variables `a` and `b` to the terms in the zero-based positions `j` and `j` + 1 of the Fibonacci sequence.

```
a = 0;
b = 1;
for (i = 0; i != j; i++)
{
  c = b;
  b += a;
  a = c;
}
```

The loop in this program terminates for any nonnegative value of `j`. For any variable `x`, suppose that `j` is not allowed to interfere with `x` in such an initial state, say s. According to the above information flow correctness definition, any initial state t differing from s in the value of `j` must make execution terminate all the same in order for the program to be correct. However, this is not the case, since execution does not terminate for any negative value of `j`. Thus, the type system needs to verify that `j` may interfere with `x`, on pain of returning a wrong *pass* verdict.

The cases that change the scope upon recursively calling the type system are those of conditional statements and loops. In the latter case, the boolean flag is set to *False*, and the set of *state set* $\times$ *vname set* pairs is empty as the whole scope nesting the loop body, including any variable occurring in the boolean condition of the loop, must be allowed to interfere with every variable. In the former case, for both branches, the boolean flag is left unchanged, whereas the set of pairs is extended with the pair composed of the input *state set* (or of *UNIV* if some state variable is indeterminate, unless the input *state set* is empty) and of the set of the variables, if any, occurring in the boolean condition of the statement.

Why is the scope extended with the whole input *state set* for both branches, rather than just with the set of states in which each single branch is selected? Once more, the question can be answered by considering a sample C program, namely a previous one determining the minimum of two integers `a` and `b` using a state variable `i`. For the reader's convenience, the program is reported here below.

```
i = (a > b) ? 1 : 0;
if (i > 0)
{
  a = b;
}
```

Since the branch changing the value of variable `a` is executed just in case `i` $= 1$, suppose that in addition to `b`, `i` also is not allowed to interfere with `a` for `i` $= 0$, and let s be any initial state where `a` $\leq$ `b`. Based on the above information flow correctness definition, any initial state t differing from s in the value of `b` (not bound by the interference of `i` with `a`) must produce the same final value of `a` in order for the program to be correct. However, this is not the case, as the final value of `a` will change for any state t where `a` $>$ `b`. Therefore, the type system needs to verify that `i` may interfere with `a` for `i` $= 0$, too, on pain of returning a wrong *pass* verdict. This is the reason why, as mentioned previously, an information flow correctness policy for this program should be such that s: $\mathit{dom}\ i \rightsquigarrow \mathit{dom}\ a$ even for any state s where `i` $= 0$.

An even simpler example explains why, in the case of an assignment or a loop, the information flow correctness checks described above need to be applied to the set of *state set* $\times$ *vname set* pairs in the scope even if the input *state set* is empty, namely even if the assignment or the loop are nested in a "dead" branch. Here below is a sample C program showing this.

```c
if (i)
{
  a = 1;
}
```

Assuming that the initial value of `i` is 0, the assignment nested within the conditional statement is not executed, so that the final value of `a` matches the initial one, say 0. Suppose that `i` is not allowed to interfere with `a` in such an initial state, say s. According to the above information flow correctness definition, any initial state t differing from s in the value of `i` must produce the same final value of `a` in order for the program to be correct. However, this is not the case, as the final value of `a` is 1 for any nonzero value of `i`. Therefore, the type system needs to verify that `i` may interfere with `a` in state s even though the conditional branch is not executed in that state, on pain of returning a wrong *pass* verdict.

abbreviation *atyping* :: *bool* ⇒ *aexp* ⇒ *vname set* ⇒ *bool*
  (‹(- ⊨ - ′(⊆ -′))› [*51*, *51*] *50*) **where**
$v \models a\ (\subseteq X) \equiv \mathit{avars}\ a = \{\} \lor \mathit{avars}\ a \subseteq \mathit{state} \cap X \land v$

definition *univ-states-if* :: *state set* ⇒ *vname set* ⇒ *state set*
  (‹(*Univ?* - -)› [*51*, *75*] *75*) **where**
$\mathit{Univ?}\ A\ X \equiv \mathit{if}\ \mathit{state} \subseteq X\ \mathit{then}\ A\ \mathit{else}\ \mathit{Univ}\ A\ (\subseteq \{\})$

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

$- \models \mathit{SKIP}\ (\subseteq A,\ X) = \mathit{Some}\ (A,\ \mathit{Univ??}\ A\ X)\ |$

$(U,\ v) \models x ::= a\ (\subseteq A,\ X) =$
 $(\mathit{if}\ (\forall (B,\ Y) \in \mathit{insert}\ (\mathit{Univ?}\ A\ X,\ \mathit{avars}\ a)\ U.\ B\text{: } \mathit{dom}\ `\ Y \rightsquigarrow \{\mathit{dom}\ x\})$
  $\mathit{then}\ \mathit{Some}\ (\mathit{if}\ x \in \mathit{state} \land A \neq \{\}$
   $\mathit{then}\ \mathit{if}\ v \models a\ (\subseteq X)$
    $\mathit{then}\ (\{s(x := \mathit{aval}\ a\ s) \mid s.\ s \in A\},\ \mathit{insert}\ x\ X)\ \mathit{else}\ (A,\ X - \{x\})$
   $\mathit{else}\ (A,\ \mathit{Univ??}\ A\ X))$
  $\mathit{else}\ \mathit{None})\ |$

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

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

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

end

end

2 Idempotence of the auxiliary type system meant for loop bodies

theory *Idempotence*
imports *Definitions*
begin

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 composed of *state set* B and *vname set* Y, then the same output is returned if B and Y are fed back into the type system (lemma *ctyping1-idem*).

2.1 Global context proofs

lemma *remdups-filter-last*:
$last\ [x \leftarrow remdups\ xs.\ P\ x] = last\ [x \leftarrow xs.\ P\ x]$
$\langle proof \rangle$

lemma *remdups-append*:
$set\ xs \subseteq set\ ys \Longrightarrow remdups\ (xs\ @\ ys) = remdups\ ys$
$\langle proof \rangle$

lemma *remdups-concat-1*:
$remdups\ (concat\ (remdups\ []))\ =\ remdups\ (concat\ [])$
⟨*proof*⟩

lemma *remdups-concat-2*:
$remdups\ (concat\ (remdups\ xs))\ =\ remdups\ (concat\ xs)\ \Longrightarrow$
$\quad remdups\ (concat\ (remdups\ (x\ \#\ xs)))\ =\ remdups\ (concat\ (x\ \#\ xs))$
⟨*proof*⟩

lemma *remdups-concat*:
$remdups\ (concat\ (remdups\ xs))\ =\ remdups\ (concat\ xs)$
⟨*proof*⟩

2.2 Local context proofs

context *noninterf*
begin

lemma *ctyping1-seq-last*:
$foldl\ (;;)\ S\ xs\ =\ (\lambda x.\ let\ xs'\ =\ [T{\leftarrow}xs.\ T\ x\ \neq\ None]\ in$
$\quad if\ xs'\ =\ []\ then\ S\ x\ else\ last\ xs'\ x)$
⟨*proof*⟩

lemma *ctyping1-seq-remdups*:
$foldl\ (;;)\ S\ (remdups\ xs)\ =\ foldl\ (;;)\ S\ xs$
⟨*proof*⟩

lemma *ctyping1-seq-remdups-concat*:
$foldl\ (;;)\ S\ (concat\ (remdups\ xs))\ =\ foldl\ (;;)\ S\ (concat\ xs)$
⟨*proof*⟩

lemma *ctyping1-seq-eq*:
assumes A: $foldl\ (;;)\ (\lambda x.\ None)\ xs\ =\ foldl\ (;;)\ (\lambda x.\ None)\ ys$
shows $foldl\ (;;)\ S\ xs\ =\ foldl\ (;;)\ S\ ys$
⟨*proof*⟩

lemma *ctyping1-merge-aux-butlast*:
$[\![ws\ \in\ A\ \bigsqcup\ B;\ butlast\ ws\ \neq\ []]\!]\ \Longrightarrow$
$\quad snd\ (last\ (butlast\ ws))\ =\ (\neg\ snd\ (last\ ws))$
⟨*proof*⟩

lemma *ctyping1-merge-aux-distinct*:
$ws\ \in\ A\ \bigsqcup\ B\ \Longrightarrow\ distinct\ ws$
⟨*proof*⟩

lemma *ctyping1-merge-aux-nonempty*:
$ws\ \in\ A\ \bigsqcup\ B\ \Longrightarrow\ ws\ \neq\ []$

⟨*proof*⟩

lemma *ctyping1-merge-aux-item*:
$[\![ws \in A \bigsqcup B;\ w \in set\ ws]\!] \Longrightarrow fst\ w \in (if\ snd\ w\ then\ A\ else\ B)$
⟨*proof*⟩

lemma *ctyping1-merge-aux-take-1* [*elim*]:
$[\![take\ n\ ws \in A \bigsqcup B;\ \neg\ snd\ (last\ ws);\ xs \in A;\ (xs,\ True) \notin set\ ws]\!] \Longrightarrow$
$take\ n\ ws\ @\ take\ (n - length\ ws)\ [(xs,\ True)] \in A \bigsqcup B$
⟨*proof*⟩

lemma *ctyping1-merge-aux-take-2* [*elim*]:
$[\![take\ n\ ws \in A \bigsqcup B;\ snd\ (last\ ws);\ ys \in B;\ (ys,\ False) \notin set\ ws]\!] \Longrightarrow$
$take\ n\ ws\ @\ take\ (n - length\ ws)\ [(ys,\ False)] \in A \bigsqcup B$
⟨*proof*⟩

lemma *ctyping1-merge-aux-take*:
$[\![ws \in A \bigsqcup B;\ 0 < n]\!] \Longrightarrow take\ n\ ws \in A \bigsqcup B$
⟨*proof*⟩

lemma *ctyping1-merge-aux-drop-1* [*elim*]:
assumes
A: $xs \in A$ **and**
B: $ys \in B$
shows $drop\ n\ [(xs,\ True)]\ @\ [(ys,\ False)] \in A \bigsqcup B$
⟨*proof*⟩

lemma *ctyping1-merge-aux-drop-2* [*elim*]:
assumes
A: $xs \in A$ **and**
B: $ys \in B$
shows $drop\ n\ [(ys,\ False)]\ @\ [(xs,\ True)] \in A \bigsqcup B$
⟨*proof*⟩

lemma *ctyping1-merge-aux-drop-3*:
assumes
A: $\bigwedge xs\ v.\ (xs,\ True) \notin set\ (drop\ n\ ws) \Longrightarrow$
$xs \in A \Longrightarrow v \Longrightarrow drop\ n\ ws\ @\ [(xs,\ True)] \in A \bigsqcup B$ **and**
B: $xs \in A$ **and**
C: $ys \in B$ **and**
D: $(xs,\ True) \notin set\ ws$ **and**
E: $(ys,\ False) \notin set\ (drop\ n\ ws)$
shows $drop\ n\ ws\ @\ drop\ (n - length\ ws)\ [(xs,\ True)]\ @$
$[(ys,\ False)] \in A \bigsqcup B$
⟨*proof*⟩

lemma *ctyping1-merge-aux-drop-4*:
assumes

A: $\bigwedge ys\ v.\ (ys,\ False) \notin set\ (drop\ n\ ws) \Longrightarrow$
$ys \in B \Longrightarrow \neg\ v \Longrightarrow drop\ n\ ws\ @\ [(ys,\ False)] \in A \bigsqcup B$ **and**
B: $ys \in B$ **and**
C: $xs \in A$ **and**
D: $(ys,\ False) \notin set\ ws$ **and**
E: $(xs,\ True) \notin set\ (drop\ n\ ws)$
shows $drop\ n\ ws\ @\ drop\ (n - length\ ws)\ [(ys,\ False)]\ @$
$[(xs,\ True)] \in A \bigsqcup B$
$\langle proof \rangle$

lemma *ctyping1-merge-aux-drop*:
$[\![ws \in A \bigsqcup B;\ w \notin set\ (drop\ n\ ws);$
$fst\ w \in (if\ snd\ w\ then\ A\ else\ B);\ snd\ w = (\neg\ snd\ (last\ ws))]\!] \Longrightarrow$
$drop\ n\ ws\ @\ [w] \in A \bigsqcup B$
$\langle proof \rangle$

lemma *ctyping1-merge-aux-closed-1*:
assumes
A: $\forall\, vs.\ length\ vs \leq length\ us \longrightarrow$
$(\forall\, ls\ rs.\ vs = ls\ @\ rs \longrightarrow ls \in A \bigsqcup B \longrightarrow rs \in A \bigsqcup B \longrightarrow$
$(\exists\, ws \in A \bigsqcup B.\ foldl\ (;;)\ (\lambda x.\ None)\ (concat\ (map\ fst\ ws)) =$
$foldl\ (;;)\ (\lambda x.\ None)\ (concat\ (map\ fst\ (ls\ @\ rs))) \wedge$
$length\ ws \leq length\ (ls\ @\ rs) \wedge snd\ (last\ ws) = snd\ (last\ rs)))$
(**is** $\forall$ -. - $\longrightarrow$ $(\forall\, ls\ rs.$ - $\longrightarrow$ - $\longrightarrow$ - $\longrightarrow$ $(\exists\, ws \in$ -. $?P\ ws\ ls\ rs)))$ **and**
B: $us \in A \bigsqcup B$ **and**
C: $fst\ v \in (if\ snd\ v\ then\ A\ else\ B)$ **and**
D: $snd\ v = (\neg\ snd\ (last\ us))$
shows $\exists\, ws \in A \bigsqcup B.\ foldl\ (;;)\ (\lambda x.\ None)\ (concat\ (map\ fst\ ws)) =$
$foldl\ (;;)\ (\lambda x.\ None)\ (concat\ (map\ fst\ (us\ @\ [v]))) \wedge$
$length\ ws \leq Suc\ (length\ us) \wedge snd\ (last\ ws) = snd\ v$
$\langle proof \rangle$

lemma *ctyping1-merge-aux-closed*:
assumes
A: $\forall\, xs \in A.\ \forall\, ys \in A.\ \exists\, zs \in A.$
$foldl\ (;;)\ (\lambda x.\ None)\ zs = foldl\ (;;)\ (\lambda x.\ None)\ (xs\ @\ ys)$ **and**
B: $\forall\, xs \in B.\ \forall\, ys \in B.\ \exists\, zs \in B.$
$foldl\ (;;)\ (\lambda x.\ None)\ zs = foldl\ (;;)\ (\lambda x.\ None)\ (xs\ @\ ys)$
shows $[\![us \in A \bigsqcup B;\ vs \in A \bigsqcup B]\!] \Longrightarrow$
$\exists\, ws \in A \bigsqcup B.\ foldl\ (;;)\ (\lambda x.\ None)\ (concat\ (map\ fst\ ws)) =$
$foldl\ (;;)\ (\lambda x.\ None)\ (concat\ (map\ fst\ (us\ @\ vs))) \wedge$
$length\ ws \leq length\ (us\ @\ vs) \wedge snd\ (last\ ws) = snd\ (last\ vs)$
(**is** $[\![$-; -$]\!] \Longrightarrow \exists\, ws \in$ -. $?P\ ws\ us\ vs)$
$\langle proof \rangle$

lemma *ctyping1-merge-closed*:
assumes

A: $\forall\, xs \in A.\ \forall\, ys \in A.\ \exists\, zs \in A.$
  $foldl\ (;;)\ (\lambda x.\ None)\ zs = foldl\ (;;)\ (\lambda x.\ None)\ (xs\ @\ ys)$ **and**
B: $\forall\, xs \in B.\ \forall\, ys \in B.\ \exists\, zs \in B.$
  $foldl\ (;;)\ (\lambda x.\ None)\ zs = foldl\ (;;)\ (\lambda x.\ None)\ (xs\ @\ ys)$ **and**
C: $xs \in A \sqcup B$ **and**
D: $ys \in A \sqcup B$
shows $\exists\, zs \in A \sqcup B.\ foldl\ (;;)\ (\lambda x.\ None)\ zs =$
  $foldl\ (;;)\ (\lambda x.\ None)\ (xs\ @\ ys)$
$\langle proof \rangle$

lemma *ctyping1-merge-append-closed*:
assumes
A: $\forall\, xs \in A.\ \forall\, ys \in A.\ \exists\, zs \in A.$
  $foldl\ (;;)\ (\lambda x.\ None)\ zs = foldl\ (;;)\ (\lambda x.\ None)\ (xs\ @\ ys)$ **and**
B: $\forall\, xs \in B.\ \forall\, ys \in B.\ \exists\, zs \in B.$
  $foldl\ (;;)\ (\lambda x.\ None)\ zs = foldl\ (;;)\ (\lambda x.\ None)\ (xs\ @\ ys)$ **and**
C: $xs \in A \sqcup_{@} B$ **and**
D: $ys \in A \sqcup_{@} B$
shows $\exists\, zs \in A \sqcup_{@} B.\ foldl\ (;;)\ (\lambda x.\ None)\ zs =$
  $foldl\ (;;)\ (\lambda x.\ None)\ (xs\ @\ ys)$
$\langle proof \rangle$

lemma *ctyping1-aux-closed*:
$[\![xs \in \vdash c;\ ys \in \vdash c]\!] \Longrightarrow \exists\, zs \in \vdash c.\ foldl\ (;;)\ (\lambda x.\ None)\ zs =$
  $foldl\ (;;)\ (\lambda x.\ None)\ (xs\ @\ ys)$
$\langle proof \rangle$

lemma *ctyping1-idem-1*:
assumes
A: $s \in A$ **and**
B: $xs \in \vdash c$ **and**
C: $ys \in \vdash c$
shows $\exists f\ r.$
$(\exists\, t.$
  $(\lambda x.\ case\ foldl\ (;;)\ (\lambda x.\ None)\ ys\ x\ of$
    $None \Rightarrow case\ foldl\ (;;)\ (\lambda x.\ None)\ xs\ x\ of$
      $None \Rightarrow s\ x \mid Some\ None \Rightarrow t'\ x \mid Some\ (Some\ i) \Rightarrow i \mid$
    $Some\ None \Rightarrow t''\ x \mid Some\ (Some\ i) \Rightarrow i) =$
  $(\lambda x.\ case\ f\ x\ of$
    $None \Rightarrow r\ x \mid Some\ None \Rightarrow t\ x \mid Some\ (Some\ i) \Rightarrow i)) \wedge$
$(\exists\, zs.\ f = foldl\ (;;)\ (\lambda x.\ None)\ zs \wedge zs \in \vdash c) \wedge$
$r \in A$
$\langle proof \rangle$

lemma *ctyping1-idem-2*:
assumes
A: $s \in A$ **and**
B: $xs \in \vdash c$

shows $\exists f\ r.$
$(\exists\, t.$
$(\lambda x.\ \mathit{case\ foldl}\ (;;)\ (\lambda x.\ \mathit{None})\ \mathit{xs}\ x\ \mathit{of}$
$\mathit{None} \Rightarrow s\ x \mid \mathit{Some\ None} \Rightarrow t'\ x \mid \mathit{Some}\ (\mathit{Some}\ i) \Rightarrow i) =$
$(\lambda x.\ \mathit{case}\ f\ x\ \mathit{of}$
$\mathit{None} \Rightarrow r\ x \mid \mathit{Some\ None} \Rightarrow t\ x \mid \mathit{Some}\ (\mathit{Some}\ i) \Rightarrow i)) \wedge$
$(\exists\, \mathit{xs}.\ f = \mathit{foldl}\ (;;)\ (\lambda x.\ \mathit{None})\ \mathit{xs} \wedge \mathit{xs} \in \vdash c) \wedge$
$(\exists f\ s.$
$(\exists\, t.\ r = (\lambda x.\ \mathit{case}\ f\ x\ \mathit{of}$
$\mathit{None} \Rightarrow s\ x \mid \mathit{Some\ None} \Rightarrow t\ x \mid \mathit{Some}\ (\mathit{Some}\ i) \Rightarrow i)) \wedge$
$(\exists\, \mathit{xs}.\ f = \mathit{foldl}\ (;;)\ (\lambda x.\ \mathit{None})\ \mathit{xs} \wedge \mathit{xs} \in \vdash c) \wedge$
$s \in A)$
$\langle \mathit{proof} \rangle$

lemma *ctyping1-idem*:
$\vdash c\ (\subseteq A,\ X) = (B,\ Y) \Longrightarrow \vdash c\ (\subseteq B,\ Y) = (B,\ Y)$
$\langle \mathit{proof} \rangle$

end

end

3 Overapproximation of program semantics by the type system

theory *Overapproximation*
imports *Idempotence*
begin

The purpose of this section is to prove that type system *ctyping2* overapproximates program semantics, namely that if (a) $(c,\ s) \Rightarrow t$, (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 a state in A on the value of every state variable in X, then t must agree with some state in B on the value of every state variable in Y (lemma *ctyping2-approx*).

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

3.1 Global context proofs

lemma *avars-aval*:
$s = t\ (\subseteq \mathit{avars}\ a) \Longrightarrow \mathit{aval}\ a\ s = \mathit{aval}\ a\ t$
$\langle \mathit{proof} \rangle$

3.2 Local context proofs

context *noninterf*
begin

lemma *interf-set-mono*:
$[\![A' \subseteq A;\ X \subseteq X';\ \forall (B',\ Y') \in U'.\ \exists (B,\ Y) \in U.\ B' \subseteq B \wedge Y' \subseteq Y;$
$\quad \forall (B,\ Y) \in insert\ (Univ?\ A\ X,\ Z)\ U.\ B\colon dom\ `\ Y \rightsquigarrow W]\!] \Longrightarrow$
$\ \forall (B,\ Y) \in insert\ (Univ?\ A'\ X',\ Z)\ U'.\ B\colon dom\ `\ Y \rightsquigarrow W$
$\langle proof \rangle$

lemma *btyping1-btyping2-aux-1* [*elim*]:
assumes
A: $avars\ a_1 = \{\}$ **and**
B: $avars\ a_2 = \{\}$ **and**
C: $aval\ a_1\ (\lambda x.\ 0) < aval\ a_2\ (\lambda x.\ 0)$
shows $aval\ a_1\ s < aval\ a_2\ s$
$\langle proof \rangle$

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

lemma *btyping1-btyping2-aux*:
$\vdash b = Some\ v \Longrightarrow \models b\ (\subseteq A,\ X) = Some\ (if\ v\ then\ A\ else\ \{\})$
$\langle proof \rangle$

lemma *btyping1-btyping2*:
$\vdash b = Some\ v \Longrightarrow \models b\ (\subseteq A,\ X) = (if\ v\ then\ (A,\ \{\})\ else\ (\{\},\ A))$
$\langle proof \rangle$

lemma *btyping2-aux-subset*:
$\models b\ (\subseteq A,\ X) = Some\ A' \Longrightarrow A' = \{s.\ s \in A \wedge bval\ b\ s\}$
$\langle proof \rangle$

lemma *btyping2-aux-diff*:
$[\![\models b\ (\subseteq A,\ X) = Some\ B;\ \models b\ (\subseteq A',\ X') = Some\ B';\ A' \subseteq A;\ B' \subseteq B]\!] \Longrightarrow$
$\quad A' - B' \subseteq A - B$
$\langle proof \rangle$

lemma *btyping2-aux-mono*:
$[\![\models b\ (\subseteq A,\ X) = Some\ B;\ A' \subseteq A;\ X \subseteq X']\!] \Longrightarrow$
$\quad \exists B'.\ \models b\ (\subseteq A',\ X') = Some\ B' \wedge B' \subseteq B$

$\langle proof \rangle$

lemma *btyping2-mono*:
$[\![\models b\ (\subseteq A,\ X) = (B_1,\ B_2);\ \models b\ (\subseteq A',\ X') = (B_1',\ B_2');\ A' \subseteq A;\ X \subseteq X']\!] \Longrightarrow$
$B_1' \subseteq B_1 \wedge B_2' \subseteq B_2$
$\langle proof \rangle$

lemma *btyping2-un-eq*:
$\models b\ (\subseteq A,\ X) = (B_1,\ B_2) \Longrightarrow B_1 \cup B_2 = A$
$\langle proof \rangle$

lemma *btyping2-fst-empty*:
$\models b\ (\subseteq \{\},\ X) = (\{\},\ \{\})$
$\langle proof \rangle$

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

lemma *ctyping1-merge-in*:
$xs \in A \cup B \Longrightarrow xs \in A \sqcup B$
$\langle proof \rangle$

lemma *ctyping1-merge-append-in*:
$[\![xs \in A;\ ys \in B]\!] \Longrightarrow xs\ @\ ys \in A \sqcup_{@} B$
$\langle proof \rangle$

lemma *ctyping1-aux-nonempty*:
$\vdash c \neq \{\}$
$\langle proof \rangle$

lemma *ctyping1-mono*:
$[\![(B,\ Y) = \vdash c\ (\subseteq A,\ X);\ (B',\ Y') = \vdash c\ (\subseteq A',\ X');\ A' \subseteq A;\ X \subseteq X']\!] \Longrightarrow$
$B' \subseteq B \wedge Y \subseteq Y'$
$\langle proof \rangle$

lemma *ctyping2-fst-empty*:
$\mathit{Some}\ (B,\ Y) = (U,\ v) \models c\ (\subseteq \{\},\ X) \Longrightarrow (B,\ Y) = (\{\},\ \mathit{UNIV})$
$\langle proof \rangle$

lemma *ctyping2-mono-assign* [*elim*!]:
$[\![(U,\ \mathit{False}) \models x ::= a\ (\subseteq A,\ X) = \mathit{Some}\ (C,\ Z);\ A' \subseteq A;\ X \subseteq X';$
$\forall (B',\ Y') \in U'.\ \exists (B,\ Y) \in U.\ B' \subseteq B \wedge Y' \subseteq Y]\!] \Longrightarrow$
$\exists C'\ Z'.\ (U',\ \mathit{False}) \models x ::= a\ (\subseteq A',\ X') = \mathit{Some}\ (C',\ Z') \wedge$
$C' \subseteq C \wedge Z \subseteq Z'$
$\langle proof \rangle$

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: $\bigwedge p\ B\ Y\ B'\ C\ Y'\ Z\ U'.$
$(U,\ False) \models c_1\ (\subseteq A,\ X) = Some\ p \Longrightarrow (B,\ Y) = p \Longrightarrow$
$(U,\ False) \models c_2\ (\subseteq B,\ Y) = Some\ (C,\ Z) \Longrightarrow B' \subseteq B \Longrightarrow Y \subseteq Y' \Longrightarrow$
$\forall (B',\ Y') \in U'.\ \exists (B,\ Y) \in U.\ B' \subseteq B \wedge Y' \subseteq Y \Longrightarrow$
$\exists C'\ Z'.\ (U',\ False) \models c_2\ (\subseteq B',\ Y') = Some\ (C',\ Z') \wedge$
$C' \subseteq C \wedge Z \subseteq Z'$ **and**
C: $(U,\ False) \models c_1;;\ c_2\ (\subseteq A,\ X) = Some\ (C,\ Z)$ **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 $\exists C'\ Z'.\ (U',\ False) \models c_1;;\ c_2\ (\subseteq A',\ X') = Some\ (C',\ Z') \wedge$
$C' \subseteq C \wedge Z \subseteq Z'$
$\langle proof \rangle$

lemma *ctyping2-mono-if*:
assumes
A: $\bigwedge W\ p\ B_1\ B_2\ B_1{}'\ C_1\ X'\ Y_1\ W'.\ (W,\ p) =$
$(insert\ (Univ?\ A\ X,\ bvars\ b)\ U,\ \models b\ (\subseteq A,\ X)) \Longrightarrow (B_1,\ B_2) = p \Longrightarrow$
$(W,\ False) \models c_1\ (\subseteq B_1,\ X) = Some\ (C_1,\ Y_1) \Longrightarrow B_1{}' \subseteq B_1 \Longrightarrow$
$X \subseteq X' \Longrightarrow \forall (B',\ Y') \in W'.\ \exists (B,\ Y) \in W.\ B' \subseteq B \wedge Y' \subseteq Y \Longrightarrow$
$\exists C_1{}'\ Y_1{}'.\ (W',\ False) \models c_1\ (\subseteq B_1{}',\ X') = Some\ (C_1{}',\ Y_1{}') \wedge$
$C_1{}' \subseteq C_1 \wedge Y_1 \subseteq Y_1{}'$ **and**
B: $\bigwedge W\ p\ B_1\ B_2\ B_2{}'\ C_2\ X'\ Y_2\ W'.\ (W,\ p) =$
$(insert\ (Univ?\ A\ X,\ bvars\ b)\ U,\ \models b\ (\subseteq A,\ X)) \Longrightarrow (B_1,\ B_2) = p \Longrightarrow$
$(W,\ False) \models c_2\ (\subseteq B_2,\ X) = Some\ (C_2,\ Y_2) \Longrightarrow B_2{}' \subseteq B_2 \Longrightarrow$
$X \subseteq X' \Longrightarrow \forall (B',\ Y') \in W'.\ \exists (B,\ Y) \in W.\ B' \subseteq B \wedge Y' \subseteq Y \Longrightarrow$
$\exists C_2{}'\ Y_2{}'.\ (W',\ False) \models c_2\ (\subseteq B_2{}',\ X') = Some\ (C_2{}',\ Y_2{}') \wedge$
$C_2{}' \subseteq C_2 \wedge Y_2 \subseteq Y_2{}'$ **and**
C: $(U,\ False) \models IF\ b\ THEN\ c_1\ ELSE\ c_2\ (\subseteq A,\ X) = Some\ (C,\ Y)$ **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 $\exists C'\ Y'.\ (U',\ False) \models IF\ b\ THEN\ c_1\ ELSE\ c_2\ (\subseteq A',\ X') =$
$Some\ (C',\ Y') \wedge C' \subseteq C \wedge Y \subseteq Y'$
$\langle proof \rangle$

lemma *ctyping2-mono-while*:
assumes
A: $\bigwedge B_1\ B_2\ C\ Y\ B_1{}'\ B_2{}'\ D_1\ E\ X'\ V\ U'.\ (B_1,\ B_2) = \models b\ (\subseteq A,\ X) \Longrightarrow$
$(C,\ Y) = \vdash c\ (\subseteq B_1,\ X) \Longrightarrow (B_1{}',\ B_2{}') = \models b\ (\subseteq C,\ Y) \Longrightarrow$
$\forall (B,\ W) \in insert\ (Univ?\ A\ X \cup Univ?\ C\ Y,\ bvars\ b)\ U.$
B: $dom\ `\ W \leadsto UNIV \Longrightarrow$

$(\{\}, \mathit{False}) \models c\ (\subseteq B_1,\ X) = \mathit{Some}\ (E,\ V) \Longrightarrow D_1 \subseteq B_1 \Longrightarrow$
$X \subseteq X' \Longrightarrow \forall (B',\ Y') \in U'.\ \exists (B,\ Y) \in \{\}.\ B' \subseteq B \wedge Y' \subseteq Y \Longrightarrow$
$\exists E'\ V'.\ (U',\ \mathit{False}) \models c\ (\subseteq D_1,\ X') = \mathit{Some}\ (E',\ V') \wedge$
$E' \subseteq E \wedge V \subseteq V'$ **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: dom ‘ $W \rightsquigarrow \mathit{UNIV} \Longrightarrow$
$(\{\}, \mathit{False}) \models c\ (\subseteq B_1{}',\ Y) = \mathit{Some}\ (F,\ W) \Longrightarrow D_1{}' \subseteq B_1{}' \Longrightarrow$
$Y \subseteq Y' \Longrightarrow \forall (B',\ Y') \in U'.\ \exists (B,\ Y) \in \{\}.\ B' \subseteq B \wedge Y' \subseteq Y \Longrightarrow$
$\exists F'\ W'.\ (U',\ \mathit{False}) \models c\ (\subseteq D_1{}',\ Y') = \mathit{Some}\ (F',\ W') \wedge$
$F' \subseteq F \wedge W \subseteq W'$ **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 $\exists B'\ Z'.\ (U',\ \mathit{False}) \models \mathit{WHILE}\ b\ \mathit{DO}\ c\ (\subseteq A',\ X') = \mathit{Some}\ (B',\ Z') \wedge$
$B' \subseteq B \wedge Z \subseteq Z'$
$\langle \mathit{proof} \rangle$

lemma *ctyping2-mono*:
$[\![(U,\ \mathit{False}) \models c\ (\subseteq A,\ X) = \mathit{Some}\ (C,\ Z);\ A' \subseteq A;\ X \subseteq X';$
$\forall (B',\ Y') \in U'.\ \exists (B,\ Y) \in U.\ B' \subseteq B \wedge Y' \subseteq Y]\!] \Longrightarrow$
$\exists C'\ Z'.\ (U',\ \mathit{False}) \models c\ (\subseteq A',\ X') = \mathit{Some}\ (C',\ Z') \wedge C' \subseteq C \wedge Z \subseteq Z'$
$\langle \mathit{proof} \rangle$

lemma *ctyping1-ctyping2-fst-assign* [*elim*!]:
assumes
A: $(C,\ Z) = \vdash x ::= a\ (\subseteq A,\ X)$ **and**
B: $\mathit{Some}\ (C',\ Z') = (U,\ \mathit{False}) \models x ::= a\ (\subseteq A,\ X)$
shows $C' \subseteq C$
$\langle \mathit{proof} \rangle$

lemma *ctyping1-ctyping2-fst-seq*:
assumes
A: $\bigwedge B\ B'\ Y\ Y'.\ (B,\ Y) = \vdash c_1\ (\subseteq A,\ X) \Longrightarrow$
$\mathit{Some}\ (B',\ Y') = (U,\ \mathit{False}) \models c_1\ (\subseteq A,\ X) \Longrightarrow B' \subseteq B$ **and**
B: $\bigwedge p\ B\ Y\ C\ C'\ Z\ Z'.\ (U,\ \mathit{False}) \models c_1\ (\subseteq A,\ X) = \mathit{Some}\ p \Longrightarrow$
$(B,\ Y) = p \Longrightarrow (C,\ Z) = \vdash c_2\ (\subseteq B,\ Y) \Longrightarrow$
$\mathit{Some}\ (C',\ Z') = (U,\ \mathit{False}) \models c_2\ (\subseteq B,\ Y) \Longrightarrow C' \subseteq C$ **and**
C: $(C,\ Z) = \vdash c_1;;\ c_2\ (\subseteq A,\ X)$ **and**
D: $\mathit{Some}\ (C',\ Z') = (U,\ \mathit{False}) \models c_1;;\ c_2\ (\subseteq A,\ X)$
shows $C' \subseteq C$
$\langle \mathit{proof} \rangle$

lemma *ctyping1-ctyping2-fst-if*:
assumes
A: $\bigwedge U'\ p\ B_1\ B_2\ C_1\ C_1{}'\ Y_1\ Y_1{}'.$

$(U', p) = (insert\ (Univ?\ A\ X,\ bvars\ b)\ U, \models b\ (\subseteq A,\ X)) \Longrightarrow$
$(B_1,\ B_2) = p \Longrightarrow (C_1,\ Y_1) = \vdash c_1\ (\subseteq B_1,\ X) \Longrightarrow$
$Some\ (C_1{}',\ Y_1{}') = (U',\ False) \models c_1\ (\subseteq B_1,\ X) \Longrightarrow C_1{}' \subseteq C_1$ **and**
B: $\bigwedge U'\ p\ B_1\ B_2\ C_2\ C_2{}'\ Y_2\ Y_2{}'.$
$(U', p) = (insert\ (Univ?\ A\ X,\ bvars\ b)\ U, \models b\ (\subseteq A,\ X)) \Longrightarrow$
$(B_1,\ B_2) = p \Longrightarrow (C_2,\ Y_2) = \vdash c_2\ (\subseteq B_2,\ X) \Longrightarrow$
$Some\ (C_2{}',\ Y_2{}') = (U',\ False) \models c_2\ (\subseteq B_2,\ X) \Longrightarrow C_2{}' \subseteq C_2$ **and**
C: $(C,\ Y) = \vdash IF\ b\ THEN\ c_1\ ELSE\ c_2\ (\subseteq A,\ X)$ **and**
D: $Some\ (C',\ Y') = (U,\ False) \models IF\ b\ THEN\ c_1\ ELSE\ c_2\ (\subseteq A,\ X)$
shows $C' \subseteq C$
⟨*proof*⟩

lemma *ctyping1-ctyping2-fst-while*:
assumes
A: $(C,\ Y) = \vdash WHILE\ b\ DO\ c\ (\subseteq A,\ X)$ **and**
B: $Some\ (C',\ Y') = (U,\ False) \models WHILE\ b\ DO\ c\ (\subseteq A,\ X)$
shows $C' \subseteq C$
⟨*proof*⟩

lemma *ctyping1-ctyping2-fst*:
$[\![(C,\ Z) = \vdash c\ (\subseteq A,\ X);\ Some\ (C',\ Z') = (U,\ False) \models c\ (\subseteq A,\ X)]\!] \Longrightarrow$
$C' \subseteq C$
⟨*proof*⟩

lemma *ctyping1-ctyping2-snd-assign* [*elim*!]:
$[\![(C,\ Z) = \vdash x ::= a\ (\subseteq A,\ X);$
$Some\ (C',\ Z') = (U,\ False) \models x ::= a\ (\subseteq A,\ X)]\!] \Longrightarrow Z \subseteq Z'$
⟨*proof*⟩

lemma *ctyping1-ctyping2-snd-seq*:
assumes
A: $\bigwedge B\ B'\ Y\ Y'.\ (B,\ Y) = \vdash c_1\ (\subseteq A,\ X) \Longrightarrow$
$Some\ (B',\ Y') = (U,\ False) \models c_1\ (\subseteq A,\ X) \Longrightarrow Y \subseteq Y'$ **and**
B: $\bigwedge p\ B\ Y\ C\ C'\ Z\ Z'.\ (U,\ False) \models c_1\ (\subseteq A,\ X) = Some\ p \Longrightarrow$
$(B,\ Y) = p \Longrightarrow (C,\ Z) = \vdash c_2\ (\subseteq B,\ Y) \Longrightarrow$
$Some\ (C',\ Z') = (U,\ False) \models c_2\ (\subseteq B,\ Y) \Longrightarrow Z \subseteq Z'$ **and**
C: $(C,\ Z) = \vdash c_1;;\ c_2\ (\subseteq A,\ X)$ **and**
D: $Some\ (C',\ Z') = (U,\ False) \models c_1;;\ c_2\ (\subseteq A,\ X)$
shows $Z \subseteq Z'$
⟨*proof*⟩

lemma *ctyping1-ctyping2-snd-if*:
assumes
A: $\bigwedge U'\ p\ B_1\ B_2\ C_1\ C_1{}'\ Y_1\ Y_1{}'.$
$(U', p) = (insert\ (Univ?\ A\ X,\ bvars\ b)\ U, \models b\ (\subseteq A,\ X)) \Longrightarrow$
$(B_1,\ B_2) = p \Longrightarrow (C_1,\ Y_1) = \vdash c_1\ (\subseteq B_1,\ X) \Longrightarrow$
$Some\ (C_1{}',\ Y_1{}') = (U',\ False) \models c_1\ (\subseteq B_1,\ X) \Longrightarrow Y_1 \subseteq Y_1{}'$ **and**
B: $\bigwedge U'\ p\ B_1\ B_2\ C_2\ C_2{}'\ Y_2\ Y_2{}'.$

$(U', p) = (\mathit{insert}\ (\mathit{Univ?}\ A\ X,\ \mathit{bvars}\ b)\ U,\ \models b\ (\subseteq A,\ X)) \Longrightarrow$
$(B_1,\ B_2) = p \Longrightarrow (C_2,\ Y_2) = \vdash c_2\ (\subseteq B_2,\ X) \Longrightarrow$
$\mathit{Some}\ (C_2',\ Y_2') = (U',\ \mathit{False}) \models c_2\ (\subseteq B_2,\ X) \Longrightarrow Y_2 \subseteq Y_2'$ **and**
C: $(C,\ Y) = \vdash \mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2\ (\subseteq A,\ X)$ **and**
D: $\mathit{Some}\ (C',\ Y') = (U,\ \mathit{False}) \models \mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2\ (\subseteq A,\ X)$
shows $Y \subseteq Y'$
⟨*proof*⟩

lemma *ctyping1-ctyping2-snd-while*:
assumes
A: $(C,\ Y) = \vdash \mathit{WHILE}\ b\ \mathit{DO}\ c\ (\subseteq A,\ X)$ **and**
B: $\mathit{Some}\ (C',\ Y') = (U,\ \mathit{False}) \models \mathit{WHILE}\ b\ \mathit{DO}\ c\ (\subseteq A,\ X)$
shows $Y \subseteq Y'$
⟨*proof*⟩

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

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

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*⟩

lemma *btyping2-aux-approx-2* [*elim*]:
assumes
A: $\mathit{avars}\ a_1 \subseteq \mathit{state}$ **and**
B: $\mathit{avars}\ a_2 \subseteq \mathit{state}$ **and**
C: $\mathit{avars}\ a_1 \subseteq X$ **and**
D: $\mathit{avars}\ a_2 \subseteq X$ **and**
E: $\mathit{aval}\ a_1\ s < \mathit{aval}\ a_2\ s$ **and**
F: $r \in A$ **and**
G: $s = r\ (\subseteq \mathit{state} \cap X)$
shows $\exists\, r'.\ r' \in A \wedge \mathit{aval}\ a_1\ r' < \mathit{aval}\ a_2\ r' \wedge r = r'\ (\subseteq \mathit{state} \cap X)$
⟨*proof*⟩

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*⟩

lemma *btyping2-aux-approx*:
$[\![\models b\ (\subseteq A,\ X) = Some\ A';\ s \in Univ\ A\ (\subseteq state \cap X)]\!] \Longrightarrow$
  $s \in Univ\ (if\ bval\ b\ s\ then\ A'\ else\ A - A')\ (\subseteq state \cap X)$
⟨*proof*⟩

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

lemma *ctyping2-approx-assign* [*elim*!]:
$[\![\forall t'.\ aval\ a\ s = t'\ x \longrightarrow (\forall s.\ t' = s(x := aval\ a\ s) \longrightarrow s \notin A) \vee$
  $(\exists y \in state \cap X.\ y \neq x \wedge t\ y \neq t'\ y);$
 $v \models a\ (\subseteq X);\ t \in A;\ s = t\ (\subseteq state \cap X)]\!] \Longrightarrow False$
⟨*proof*⟩

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

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

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

$\langle proof \rangle$

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

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: $(\{\},\ False) \models c\ (\subseteq B_1,\ X) = Some\ (D,\ Z)$ **and**
E: $(\{\},\ False) \models c\ (\subseteq B_1',\ Y) = Some\ (D',\ Z')$ **and**
F: $r_1 \in A$ **and**
G: $s_1 = r_1\ (\subseteq state \cap X)$ **and**
H: $bval\ b\ s_1$ **and**
I: $\bigwedge C\ B\ Y\ W\ U.\ (case \models b\ (\subseteq C,\ Y)\ of\ (B_1',\ B_2') \Rightarrow$
$case \vdash c\ (\subseteq B_1',\ Y)\ of\ (C',\ Y') \Rightarrow$
$case \models b\ (\subseteq C',\ Y')\ of\ (B_1'',\ B_2'') \Rightarrow$
$if\ (\forall s \in Univ?\ C\ Y \cup Univ?\ C'\ Y'.\ \forall x \in bvars\ b.\ All\ (interf\ s\ (dom\ x))) \wedge$
$(\forall p \in U.\ case\ p\ of\ (B,\ W) \Rightarrow \forall s \in B.\ \forall x \in W.\ All\ (interf\ s\ (dom\ x)))$
$then\ case\ (\{\},\ False) \models c\ (\subseteq B_1',\ Y)\ of$
$None \Rightarrow None \mid Some\ \text{-} \Rightarrow case\ (\{\},\ False) \models c\ (\subseteq B_1'',\ Y')\ of$
$None \Rightarrow None \mid Some\ \text{-} \Rightarrow Some\ (B_2' \cup B_2'',\ Univ??\ B_2'\ Y \cap Y')$
$else\ None) = Some\ (B,\ W) \Longrightarrow$
$\exists r \in C.\ s_2 = r\ (\subseteq state \cap Y) \Longrightarrow \exists r \in B.\ s_3 = r\ (\subseteq state \cap W)$
(**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: $\forall s \in Univ?\ A\ X \cup Univ?\ C\ Y.\ \forall x \in bvars\ b.\ All\ (interf\ s\ (dom\ x))$ **and**
L: $\forall p \in U.\ \forall B\ W.\ p\ = (B,\ W) \longrightarrow$
$(\forall s \in B.\ \forall x \in W.\ All\ (interf\ s\ (dom\ x)))$
shows $\exists r \in B_2 \cup B_2'.\ s_3 = r\ (\subseteq state \cap Univ??\ B_2\ X \cap Y)$
$\langle proof \rangle$

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

lemma *ctyping2-approx-while-4*:
$[\![\models b\ (\subseteq A,\ X) = (B_1,\ B_2);$
$\vdash c\ (\subseteq B_1,\ X) = (C,\ Y);$
$\models b\ (\subseteq C,\ Y) = (B_1',\ B_2');$
$(\{\},\ False) \models c\ (\subseteq B_1,\ X) = Some\ (D,\ Z);$
$(\{\},\ False) \models c\ (\subseteq B_1',\ Y) = Some\ (D',\ Z');$
$r_1 \in A;\ s_1 = r_1\ (\subseteq state \cap X);\ bval\ b\ s_1;$
$\bigwedge C\ B\ Y\ W\ U.\ (case \models b\ (\subseteq C,\ Y)\ of\ (B_1',\ B_2') \Rightarrow$
$case \vdash c\ (\subseteq B_1',\ Y)\ of\ (C',\ Y') \Rightarrow$
$case \models b\ (\subseteq C',\ Y')\ of\ (B_1'',\ B_2'') \Rightarrow$

$if\ (\forall s \in Univ?\ C\ Y \cup Univ?\ C'\ Y'.\ \forall x \in bvars\ b.\ All\ (interf\ s\ (dom\ x))) \wedge$
$(\forall p \in U.\ case\ p\ of\ (B,\ W) \Rightarrow \forall s \in B.\ \forall x \in W.\ All\ (interf\ s\ (dom\ x)))$
$then\ case\ (\{\},\ False) \models c\ (\subseteq B_1{}',\ Y)\ of$
$None \Rightarrow None \mid Some\ \text{-} \Rightarrow case\ (\{\},\ False) \models c\ (\subseteq B_1{}'',\ Y')\ of$
$None \Rightarrow None \mid Some\ \text{-} \Rightarrow Some\ (B_2{}' \cup B_2{}'',\ Univ??\ B_2{}'\ Y \cap Y')$
$else\ None) = Some\ (B,\ W) \Longrightarrow$
$\exists r \in C.\ s_2 = r\ (\subseteq state \cap Y) \Longrightarrow \exists r \in B.\ s_3 = r\ (\subseteq state \cap W);$
$\bigwedge A\ B\ X\ Y\ U\ v.\ (U,\ v) \models c\ (\subseteq A,\ X) = Some\ (B,\ Y) \Longrightarrow$
$\exists r \in A.\ s_1 = r\ (\subseteq state \cap X) \Longrightarrow \exists r \in B.\ s_2 = r\ (\subseteq state \cap Y);$
$\forall s \in Univ?\ A\ X \cup Univ?\ C\ Y.\ \forall x \in bvars\ b.\ All\ (interf\ s\ (dom\ x));$
$\forall p \in U.\ \forall B\ W.\ p = (B,\ W) \longrightarrow (\forall s \in B.\ \forall x \in W.\ All\ (interf\ s\ (dom\ x)));$
$\forall r \in B_2 \cup B_2{}'.\ \exists x \in state \cap (X \cap Y).\ s_3\ x \neq r\ x\rrbracket \Longrightarrow$
$False$
$\langle proof \rangle$

lemma *ctyping2-approx*:
$\llbracket(c,\ s) \Rightarrow t;\ (U,\ v) \models c\ (\subseteq A,\ X) = Some\ (B,\ Y);$
$s \in Univ\ A\ (\subseteq state \cap X)\rrbracket \Longrightarrow t \in Univ\ B\ (\subseteq state \cap Y)$
$\langle proof \rangle$

end

end

4 Sufficiency of well-typedness for information flow correctness

theory *Correctness*
imports *Overapproximation*
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 lemmas *ctyping1-idem* and *ctyping2-approx* proven in the previous sections.

4.1 Global context proofs

lemma *flow-append-1*:
assumes A: $\bigwedge cfs' :: (com \times state)\ list.$
$c\ \#\ map\ fst\ (cfs :: (com \times state)\ list) = map\ fst\ cfs' \Longrightarrow$
$flow\text{-}aux\ (map\ fst\ cfs'\ @\ map\ fst\ cfs'') =$

$\quad$ *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''*)
⟨*proof*⟩

lemma *flow-append*:
flow (*cfs* @ *cfs'*) = *flow cfs* @ *flow cfs'*
⟨*proof*⟩

lemma *flow-cons*:
flow (*cf* # *cfs*) = *flow-aux* (*fst cf* # []) @ *flow cfs*
⟨*proof*⟩

lemma *small-stepsl-append*:
$[\![(c, s) \to\!*\{cfs\}\ (c', s'); (c', s') \to\!*\{cfs'\}\ (c'', s'')]\!] \Longrightarrow$
$(c, s) \to\!*\{cfs\ @\ cfs'\}\ (c'', s'')$
⟨*proof*⟩

lemma *small-stepsl-cons-1*:
$(c, s) \to\!*\{[cf]\}\ (c'', s'') \Longrightarrow$
$cf = (c, s) \wedge$
$(\exists\, c'\ s'.\ (c, s) \to (c', s') \wedge (c', s') \to\!*\{[]\}\ (c'', s''))$
⟨*proof*⟩

lemma *small-stepsl-cons-2*:
$[\![(c, s) \to\!*\{cf\ \#\ cfs\}\ (c'', s'') \Longrightarrow$
$cf = (c, s) \wedge$
$(\exists\, c'\ s'.\ (c, s) \to (c', s') \wedge (c', s') \to\!*\{cfs\}\ (c'', s''));$
$(c, s) \to\!*\{cf\ \#\ cfs\ @\ [(c'', s'')]\}\ (c''', s''')]\!] \Longrightarrow$
$cf = (c, s) \wedge$
$(\exists\, c'\ s'.\ (c, s) \to (c', s') \wedge$
$(c', s') \to\!*\{cfs\ @\ [(c'', s'')]\}\ (c''', s'''))$
⟨*proof*⟩

lemma *small-stepsl-cons*:
$(c, s) \to\!*\{cf\ \#\ cfs\}\ (c'', s'') \Longrightarrow$
$cf = (c, s) \wedge$
$(\exists\, c'\ s'.\ (c, s) \to (c', s') \wedge (c', s') \to\!*\{cfs\}\ (c'', s''))$
⟨*proof*⟩

lemma *small-steps-stepsl-1*:
$\exists\, cfs.\ (c, s) \to\!*\{cfs\}\ (c, s)$
⟨*proof*⟩

lemma *small-steps-stepsl-2*:
$[\![(c, s) \to (c', s'); (c', s') \to\!*\{cfs\}\ (c'', s'')]\!] \Longrightarrow$
$\exists\, cfs'.\ (c, s) \to\!*\{cfs'\}\ (c'', s'')$

$\langle proof \rangle$

lemma *small-steps-stepsl*:
$(c, s) \rightarrow* (c', s') \Longrightarrow \exists\, cfs.\ (c, s) \rightarrow*\{cfs\}\ (c', s')$
$\langle proof \rangle$

lemma *small-stepsl-steps*:
$(c, s) \rightarrow*\{cfs\}\ (c', s') \Longrightarrow (c, s) \rightarrow* (c', s')$
$\langle proof \rangle$

lemma *small-stepsl-skip*:
$(SKIP, s) \rightarrow*\{cfs\}\ (c, t) \Longrightarrow$
$(c, t) = (SKIP, s) \wedge flow\ cfs = []$
$\langle proof \rangle$

lemma *small-stepsl-assign-1*:
$(x ::= a, s) \rightarrow*\{[]\}\ (c', s') \Longrightarrow$
$(c', s') = (x ::= a, s) \wedge flow\ [] = [] \vee$
$(c', s') = (SKIP, s(x := aval\ a\ s)) \wedge flow\ [] = [x ::= a]$
$\langle proof \rangle$

lemma *small-stepsl-assign-2*:
$[\![(x ::= a, s) \rightarrow*\{cfs\}\ (c', s') \Longrightarrow$
$(c', s') = (x ::= a, s) \wedge flow\ cfs = [] \vee$
$(c', s') = (SKIP, s(x := aval\ a\ s)) \wedge flow\ cfs = [x ::= a];$
$(x ::= a, s) \rightarrow*\{cfs\ @\ [(c', s')]\}\ (c'', s'')]\!] \Longrightarrow$
$(c'', s'') = (x ::= a, s) \wedge$
$flow\ (cfs\ @\ [(c', s')]) = [] \vee$
$(c'', s'') = (SKIP, s(x := aval\ a\ s)) \wedge$
$flow\ (cfs\ @\ [(c', s')]) = [x ::= a]$
$\langle proof \rangle$

lemma *small-stepsl-assign*:
$(x ::= a, s) \rightarrow*\{cfs\}\ (c, t) \Longrightarrow$
$(c, t) = (x ::= a, s) \wedge flow\ cfs = [] \vee$
$(c, t) = (SKIP, s(x := aval\ a\ s)) \wedge flow\ cfs = [x ::= a]$
$\langle proof \rangle$

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

⟨*proof*⟩

lemma *small-stepsl-seq-2*:
assumes
A: $(c_1;; c_2, s) \to*\{cfs\}\ (c', s') \Longrightarrow$
$(\exists\, c''\ cfs'.\ c' = c'';; c_2 \wedge$
$(c_1, s) \to*\{cfs'\}\ (c'', s') \wedge$
$\mathit{flow}\ cfs = \mathit{flow}\ cfs') \vee$
$(\exists\, s''\ cfs'\ cfs''.\ \mathit{length}\ cfs'' < \mathit{length}\ cfs \wedge$
$(c_1, s) \to*\{cfs'\}\ (\mathit{SKIP}, s'') \wedge$
$(c_2, s'') \to*\{cfs''\}\ (c', s') \wedge$
$\mathit{flow}\ cfs = \mathit{flow}\ cfs'\ @\ \mathit{flow}\ cfs'')$ **and**
B: $(c_1;; c_2, s) \to*\{cfs\ @\ [(c', s')]\}\ (c'', s'')$
shows
$(\exists\, d\ cfs'.\ c'' = d;; c_2 \wedge$
$(c_1, s) \to*\{cfs'\}\ (d, s'') \wedge$
$\mathit{flow}\ (cfs\ @\ [(c', s')]) = \mathit{flow}\ cfs') \vee$
$(\exists\, t\ cfs'\ cfs''.\ \mathit{length}\ cfs'' < \mathit{length}\ (cfs\ @\ [(c', s')]) \wedge$
$(c_1, s) \to*\{cfs'\}\ (\mathit{SKIP}, t) \wedge$
$(c_2, t) \to*\{cfs''\}\ (c'', s'') \wedge$
$\mathit{flow}\ (cfs\ @\ [(c', s')]) = \mathit{flow}\ cfs'\ @\ \mathit{flow}\ cfs'')$
(**is** $?P \vee ?Q$)
⟨*proof*⟩

lemma *small-stepsl-seq*:
$(c_1;; c_2, s) \to*\{cfs\}\ (c, t) \Longrightarrow$
$(\exists\, c'\ cfs'.\ c = c';; c_2 \wedge$
$(c_1, s) \to*\{cfs'\}\ (c', t) \wedge$
$\mathit{flow}\ cfs = \mathit{flow}\ cfs') \vee$
$(\exists\, s'\ cfs'\ cfs''.\ \mathit{length}\ cfs'' < \mathit{length}\ cfs \wedge$
$(c_1, s) \to*\{cfs'\}\ (\mathit{SKIP}, s') \wedge (c_2, s') \to*\{cfs''\}\ (c, t) \wedge$
$\mathit{flow}\ cfs = \mathit{flow}\ cfs'\ @\ \mathit{flow}\ cfs'')$
⟨*proof*⟩

lemma *small-stepsl-if-1*:
$(\mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2, s) \to*\{[]\}\ (c', s') \Longrightarrow$
$(c', s') = (\mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2, s) \wedge$
$\mathit{flow}\ [] = [] \vee$
$\mathit{bval}\ b\ s \wedge (c_1, s) \to*\{\mathit{tl}\ []\}\ (c', s') \wedge$
$\mathit{flow}\ [] = \langle \mathit{bvars}\ b\rangle\ \#\ \mathit{flow}\ (\mathit{tl}\ []) \vee$
$\neg\ \mathit{bval}\ b\ s \wedge (c_2, s) \to*\{\mathit{tl}\ []\}\ (c', s') \wedge$
$\mathit{flow}\ [] = \langle \mathit{bvars}\ b\rangle\ \#\ \mathit{flow}\ (\mathit{tl}\ [])$
⟨*proof*⟩

lemma *small-stepsl-if-2*:
assumes
A: $(\mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2, s) \to*\{cfs\}\ (c', s') \Longrightarrow$
$(c', s') = (\mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2, s) \wedge$

flow cfs = [] ∨
bval b s ∧ $(c_1, s) \rightarrow*\{tl\ cfs\}\ (c', s')$ ∧
flow cfs = ⟨*bvars b*⟩ # *flow* (*tl cfs*) ∨
¬ *bval b s* ∧ $(c_2, s) \rightarrow*\{tl\ cfs\}\ (c', s')$ ∧
flow cfs = ⟨*bvars b*⟩ # *flow* (*tl cfs*) **and**
B: (*IF b THEN* c_1 *ELSE* c_2, *s*) $\rightarrow*\{cfs\ @\ [(c', s')]\}\ (c'', s'')$
shows
(c'', s'') = (*IF b THEN* c_1 *ELSE* c_2, *s*) ∧
flow (*cfs* @ $[(c', s')]$) = [] ∨
bval b s ∧ $(c_1, s) \rightarrow*\{tl\ (cfs\ @\ [(c', s')])\}\ (c'', s'')$ ∧
flow (*cfs* @ $[(c', s')]$) = ⟨*bvars b*⟩ # *flow* (*tl* (*cfs* @ $[(c', s')]$)) ∨
¬ *bval b s* ∧ $(c_2, s) \rightarrow*\{tl\ (cfs\ @\ [(c', s')])\}\ (c'', s'')$ ∧
flow (*cfs* @ $[(c', s')]$) = ⟨*bvars b*⟩ # *flow* (*tl* (*cfs* @ $[(c', s')]$))
(**is** - ∨ *?P*)
⟨*proof*⟩

lemma *small-stepsl-if*:
(*IF b THEN* c_1 *ELSE* c_2, *s*) $\rightarrow*\{cfs\}\ (c, t) \Longrightarrow$
(*c*, *t*) = (*IF b THEN* c_1 *ELSE* c_2, *s*) ∧
flow cfs = [] ∨
bval b s ∧ $(c_1, s) \rightarrow*\{tl\ cfs\}\ (c, t)$ ∧
flow cfs = ⟨*bvars b*⟩ # *flow* (*tl cfs*) ∨
¬ *bval b s* ∧ $(c_2, s) \rightarrow*\{tl\ cfs\}\ (c, t)$ ∧
flow cfs = ⟨*bvars b*⟩ # *flow* (*tl cfs*)
⟨*proof*⟩

lemma *small-stepsl-while-1*:
(*WHILE b DO c*, *s*) $\rightarrow*\{[]\}\ (c', s') \Longrightarrow$
(c', s') = (*WHILE b DO c*, *s*) ∧ *flow* [] = [] ∨
(*IF b THEN c*;; *WHILE b DO c ELSE SKIP*, *s*) $\rightarrow*\{tl\ []\}\ (c', s')$ ∧
flow [] = *flow* (*tl* [])
⟨*proof*⟩

lemma *small-stepsl-while-2*:
assumes
A: (*WHILE b DO c*, *s*) $\rightarrow*\{cfs\}\ (c', s') \Longrightarrow$
(c', s') = (*WHILE b DO c*, *s*) ∧
flow cfs = [] ∨
(*IF b THEN c*;; *WHILE b DO c ELSE SKIP*, *s*) $\rightarrow*\{tl\ cfs\}\ (c', s')$ ∧
flow cfs = *flow* (*tl cfs*) **and**
B: (*WHILE b DO c*, *s*) $\rightarrow*\{cfs\ @\ [(c', s')]\}\ (c'', s'')$
shows
(c'', s'') = (*WHILE b DO c*, *s*) ∧
flow (*cfs* @ $[(c', s')]$) = [] ∨
(*IF b THEN c*;; *WHILE b DO c ELSE SKIP*, *s*)
$\rightarrow*\{tl\ (cfs\ @\ [(c', s')])\}\ (c'', s'')$ ∧
flow (*cfs* @ $[(c', s')]$) = *flow* (*tl* (*cfs* @ $[(c', s')]$))
(**is** - ∨ *?P*)

$\langle proof \rangle$

lemma *small-stepsl-while*:
$(WHILE\ b\ DO\ c,\ s) \to{*}\{cfs\}\ (c',\ s') \Longrightarrow$
$\quad (c',\ s') = (WHILE\ b\ DO\ c,\ s)\ \wedge$
$\quad\quad flow\ cfs = []\ \vee$
$\quad (IF\ b\ THEN\ c;;\ WHILE\ b\ DO\ c\ ELSE\ SKIP,\ s) \to{*}\{tl\ cfs\}\ (c',\ s')\ \wedge$
$\quad\quad flow\ cfs = flow\ (tl\ cfs)$
$\langle proof \rangle$

lemma *bvars-bval*:
$s = t\ (\subseteq bvars\ b) \Longrightarrow bval\ b\ s = bval\ b\ t$
$\langle proof \rangle$

lemma *run-flow-append*:
$run\text{-}flow\ (cs\ @\ cs')\ s = run\text{-}flow\ cs'\ (run\text{-}flow\ cs\ s)$
$\langle proof \rangle$

lemma *no-upd-append*:
$no\text{-}upd\ (cs\ @\ cs')\ x = (no\text{-}upd\ cs\ x \wedge no\text{-}upd\ cs'\ x)$
$\langle proof \rangle$

lemma *no-upd-run-flow*:
$no\text{-}upd\ cs\ x \Longrightarrow run\text{-}flow\ cs\ s\ x = s\ x$
$\langle proof \rangle$

lemma *small-stepsl-run-flow-1*:
$(c,\ s) \to{*}\{[]\}\ (c',\ s') \Longrightarrow s' = run\text{-}flow\ (flow\ [])\ s$
$\langle proof \rangle$

lemma *small-stepsl-run-flow-2*:
$(c,\ s) \to (c',\ s') \Longrightarrow s' = run\text{-}flow\ (flow\text{-}aux\ [c])\ s$
$\langle proof \rangle$

lemma *small-stepsl-run-flow-3*:
$[\![(c,\ s) \to{*}\{cfs\}\ (c',\ s') \Longrightarrow s' = run\text{-}flow\ (flow\ cfs)\ s;$
$\quad (c,\ s) \to{*}\{cfs\ @\ [(c',\ s')]\}\ (c'',\ s'')]\!] \Longrightarrow$
$\ s'' = run\text{-}flow\ (flow\ (cfs\ @\ [(c',\ s')]))\ s$
$\langle proof \rangle$

lemma *small-stepsl-run-flow*:
$(c,\ s) \to{*}\{cfs\}\ (c',\ s') \Longrightarrow s' = run\text{-}flow\ (flow\ cfs)\ s$
$\langle proof \rangle$

4.2 Local context proofs

context *noninterf*
begin

lemma *no-upd-sources*:
$no\text{-}upd\ cs\ x \Longrightarrow x \in sources\ cs\ s\ x$
⟨*proof*⟩

lemma *sources-aux-sources*:
$sources\text{-}aux\ cs\ s\ x \subseteq sources\ cs\ s\ x$
⟨*proof*⟩

lemma *sources-aux-append*:
$sources\text{-}aux\ cs\ s\ x \subseteq sources\text{-}aux\ (cs\ @\ cs')\ s\ x$
⟨*proof*⟩

lemma *sources-aux-observe-hd-1*:
$\forall y \in X.\ s\colon dom\ y \leadsto dom\ x \Longrightarrow X \subseteq sources\text{-}aux\ [\langle X\rangle]\ s\ x$
⟨*proof*⟩

lemma *sources-aux-observe-hd-2*:
$(\forall y \in X.\ s\colon dom\ y \leadsto dom\ x \Longrightarrow X \subseteq sources\text{-}aux\ (\langle X\rangle\ \#\ xs)\ s\ x) \Longrightarrow$
$\quad \forall y \in X.\ s\colon dom\ y \leadsto dom\ x \Longrightarrow X \subseteq sources\text{-}aux\ (\langle X\rangle\ \#\ xs\ @\ [x'])\ s\ x$
⟨*proof*⟩

lemma *sources-aux-observe-hd*:
$\forall y \in X.\ s\colon dom\ y \leadsto dom\ x \Longrightarrow X \subseteq sources\text{-}aux\ (\langle X\rangle\ \#\ cs)\ s\ x$
⟨*proof*⟩

lemma *sources-observe-tl-1*:
 assumes
  A: $\bigwedge z\ a.\ c = (x ::= a :: com\text{-}flow) \Longrightarrow z = x \Longrightarrow$
   $sources\text{-}aux\ cs\ s\ x \subseteq sources\text{-}aux\ (\langle X\rangle\ \#\ cs)\ s\ x$ **and**
  B: $\bigwedge z\ a\ y.\ c = (x ::= a :: com\text{-}flow) \Longrightarrow z = x \Longrightarrow$
   $sources\ cs\ s\ y \subseteq sources\ (\langle X\rangle\ \#\ cs)\ s\ y$ **and**
  C: $\bigwedge z\ a.\ c = (z ::= a :: com\text{-}flow) \Longrightarrow z \neq x \Longrightarrow$
   $sources\ cs\ s\ x \subseteq sources\ (\langle X\rangle\ \#\ cs)\ s\ x$ **and**
  D: $\bigwedge Y\ y.\ c = \langle Y\rangle \Longrightarrow$
   $sources\ cs\ s\ y \subseteq sources\ (\langle X\rangle\ \#\ cs)\ s\ y$ **and**
  E: $z \in (case\ c\ of$
   $z ::= a \Rightarrow if\ z = x$
    $then\ sources\text{-}aux\ cs\ s\ x \cup \bigcup\ \{sources\ cs\ s\ y \mid y.$
     $run\text{-}flow\ cs\ s\colon dom\ y \leadsto dom\ x \wedge y \in avars\ a\}$
    $else\ sources\ cs\ s\ x \mid$
   $\langle X\rangle \Rightarrow$
    $sources\ cs\ s\ x \cup \bigcup\ \{sources\ cs\ s\ y \mid y.$
     $run\text{-}flow\ cs\ s\colon dom\ y \leadsto dom\ x \wedge y \in X\})$
 shows $z \in sources\ (\langle X\rangle\ \#\ cs\ @\ [c])\ s\ x$
⟨*proof*⟩

lemma *sources-observe-tl-2*:
assumes
A: $\bigwedge z\ a.\ c = (z ::= a :: \textit{com-flow}) \Longrightarrow$
$\textit{sources-aux}\ cs\ s\ x \subseteq \textit{sources-aux}\ (\langle X\rangle\ \#\ cs)\ s\ x$ **and**
B: $\bigwedge Y.\ c = \langle Y\rangle \Longrightarrow$
$\textit{sources-aux}\ cs\ s\ x \subseteq \textit{sources-aux}\ (\langle X\rangle\ \#\ cs)\ s\ x$ **and**
C: $\bigwedge Y\ y.\ c = \langle Y\rangle \Longrightarrow$
$\textit{sources}\ cs\ s\ y \subseteq \textit{sources}\ (\langle X\rangle\ \#\ cs)\ s\ y$ **and**
D: $z \in (\textit{case}\ c\ \textit{of}$
$z ::= a \Rightarrow$
$\textit{sources-aux}\ cs\ s\ x\ |$
$\langle X\rangle \Rightarrow$
$\textit{sources-aux}\ cs\ s\ x \cup \bigcup\ \{\textit{sources}\ cs\ s\ y\ |\ y.$
$\textit{run-flow}\ cs\ s\text{: } \textit{dom}\ y \rightsquigarrow \textit{dom}\ x \wedge y \in X\})$
shows $z \in \textit{sources-aux}\ (\langle X\rangle\ \#\ cs\ @\ [c])\ s\ x$
$\langle \textit{proof}\rangle$

lemma *sources-observe-tl*:
$\textit{sources}\ cs\ s\ x \subseteq \textit{sources}\ (\langle X\rangle\ \#\ cs)\ s\ x$
and *sources-aux-observe-tl*:
$\textit{sources-aux}\ cs\ s\ x \subseteq \textit{sources-aux}\ (\langle X\rangle\ \#\ cs)\ s\ x$
$\langle \textit{proof}\rangle$

lemma *sources-member-1*:
assumes
A: $\bigwedge z\ a.\ c = (x ::= a :: \textit{com-flow}) \Longrightarrow z = x \Longrightarrow$
$y \in \textit{sources-aux}\ cs'\ (\textit{run-flow}\ cs\ s)\ x \Longrightarrow$
$\textit{sources}\ cs\ s\ y \subseteq \textit{sources-aux}\ (cs\ @\ cs')\ s\ x$ **and**
B: $\bigwedge z\ a\ w.\ c = (x ::= a :: \textit{com-flow}) \Longrightarrow z = x \Longrightarrow$
$y \in \textit{sources}\ cs'\ (\textit{run-flow}\ cs\ s)\ w \Longrightarrow$
$\textit{sources}\ cs\ s\ y \subseteq \textit{sources}\ (cs\ @\ cs')\ s\ w$ **and**
C: $\bigwedge z\ a.\ c = (z ::= a :: \textit{com-flow}) \Longrightarrow z \neq x \Longrightarrow$
$y \in \textit{sources}\ cs'\ (\textit{run-flow}\ cs\ s)\ x \Longrightarrow$
$\textit{sources}\ cs\ s\ y \subseteq \textit{sources}\ (cs\ @\ cs')\ s\ x$ **and**
D: $\bigwedge Y\ w.\ c = \langle Y\rangle \Longrightarrow$
$y \in \textit{sources}\ cs'\ (\textit{run-flow}\ cs\ s)\ w \Longrightarrow$
$\textit{sources}\ cs\ s\ y \subseteq \textit{sources}\ (cs\ @\ cs')\ s\ w$ **and**
E: $y \in (\textit{case}\ c\ \textit{of}$
$z ::= a \Rightarrow \textit{if}\ z = x$
$\textit{then}\ \textit{sources-aux}\ cs'\ (\textit{run-flow}\ cs\ s)\ x\ \cup$
$\bigcup\ \{\textit{sources}\ cs'\ (\textit{run-flow}\ cs\ s)\ y\ |\ y.$
$\textit{run-flow}\ cs'\ (\textit{run-flow}\ cs\ s)\text{: } \textit{dom}\ y \rightsquigarrow \textit{dom}\ x \wedge y \in \textit{avars}\ a\}$
$\textit{else}\ \textit{sources}\ cs'\ (\textit{run-flow}\ cs\ s)\ x\ |$
$\langle X\rangle \Rightarrow$
$\textit{sources}\ cs'\ (\textit{run-flow}\ cs\ s)\ x\ \cup$
$\bigcup\ \{\textit{sources}\ cs'\ (\textit{run-flow}\ cs\ s)\ y\ |\ y.$
$\textit{run-flow}\ cs'\ (\textit{run-flow}\ cs\ s)\text{: } \textit{dom}\ y \rightsquigarrow \textit{dom}\ x \wedge y \in X\})$ **and**
F: $z \in \textit{sources}\ cs\ s\ y$

shows $z \in$ *sources* $(cs\ @\ cs'\ @\ [c])\ s\ x$
$\langle proof \rangle$

lemma *sources-member-2*:
assumes
A: $\bigwedge z\ a.\ c = (z ::= a :: com\text{-}flow) \Longrightarrow$
$y \in sources\text{-}aux\ cs'\ (run\text{-}flow\ cs\ s)\ x \Longrightarrow$
$sources\ cs\ s\ y \subseteq sources\text{-}aux\ (cs\ @\ cs')\ s\ x$ **and**
B: $\bigwedge Y.\ c = \langle Y \rangle \Longrightarrow$
$y \in sources\text{-}aux\ cs'\ (run\text{-}flow\ cs\ s)\ x \Longrightarrow$
$sources\ cs\ s\ y \subseteq sources\text{-}aux\ (cs\ @\ cs')\ s\ x$ **and**
C: $\bigwedge Y\ w.\ c = \langle Y \rangle \Longrightarrow$
$y \in sources\ cs'\ (run\text{-}flow\ cs\ s)\ w \Longrightarrow$
$sources\ cs\ s\ y \subseteq sources\ (cs\ @\ cs')\ s\ w$ **and**
D: $y \in (case\ c\ of$
$z ::= a \Rightarrow$
$sources\text{-}aux\ cs'\ (run\text{-}flow\ cs\ s)\ x\ |$
$\langle X \rangle \Rightarrow$
$sources\text{-}aux\ cs'\ (run\text{-}flow\ cs\ s)\ x\ \cup$
$\bigcup\ \{sources\ cs'\ (run\text{-}flow\ cs\ s)\ y\ |\ y.$
$run\text{-}flow\ cs'\ (run\text{-}flow\ cs\ s)\text{: } dom\ y \leadsto dom\ x \wedge y \in X\})$ **and**
E: $z \in sources\ cs\ s\ y$
shows $z \in sources\text{-}aux\ (cs\ @\ cs'\ @\ [c])\ s\ x$
$\langle proof \rangle$

lemma *sources-member*:
$y \in sources\ cs'\ (run\text{-}flow\ cs\ s)\ x \Longrightarrow$
$sources\ cs\ s\ y \subseteq sources\ (cs\ @\ cs')\ s\ x$
and *sources-aux-member*:
$y \in sources\text{-}aux\ cs'\ (run\text{-}flow\ cs\ s)\ x \Longrightarrow$
$sources\ cs\ s\ y \subseteq sources\text{-}aux\ (cs\ @\ cs')\ s\ x$
$\langle proof \rangle$

lemma *ctyping2-confine*:
$[\![(c,\ s) \Rightarrow s';\ (U,\ v) \models c\ (\subseteq A,\ X) = Some\ (B,\ Y);$
$\exists (C,\ Z) \in U.\ \neg\ C\text{: } dom\ `\ Z \leadsto \{dom\ x\}]\!] \Longrightarrow s'\ x = s\ x$
$\langle proof \rangle$

lemma *ctyping2-term-if*:
$[\![\bigwedge x'\ y'\ z''\ s.\ x' = x \Longrightarrow y' = y \Longrightarrow z = z'' \Longrightarrow \exists s'.\ (c_1,\ s) \Rightarrow s';$
$\bigwedge x'\ y'\ z''\ s.\ x' = x \Longrightarrow y' = y \Longrightarrow z' = z'' \Longrightarrow \exists s'.\ (c_2,\ s) \Rightarrow s']\!] \Longrightarrow$
$\exists s'.\ (IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s) \Rightarrow s'$
$\langle proof \rangle$

lemma *ctyping2-term*:
$[\![(U,\ v) \models c\ (\subseteq A,\ X) = Some\ (B,\ Y);$
$\exists (C,\ Z) \in U.\ \neg\ C\text{: } dom\ `\ Z \leadsto UNIV]\!] \Longrightarrow \exists s'.\ (c,\ s) \Rightarrow s'$
$\langle proof \rangle$

lemma *ctyping2-correct-aux-skip* [*elim*]:
$[\![(SKIP,\ s) \rightarrow*\{cfs_1\}\ (c_1,\ s_1);\ (c_1,\ s_1) \rightarrow*\{cfs_2\}\ (c_2,\ s_2)]\!] \Longrightarrow$
$(\forall\, t_1.\ \exists\, c_2{}'\ t_2.\ \forall\, x.$
$(s_1 = t_1\ (\subseteq\ sources\text{-}aux\ (flow\ cfs_2)\ s_1\ x) \longrightarrow$
$(c_1,\ t_1) \rightarrow*\ (c_2{}',\ t_2) \wedge (c_2 = SKIP) = (c_2{}' = SKIP)) \wedge$
$(s_1 = t_1\ (\subseteq\ sources\ (flow\ cfs_2)\ s_1\ x) \longrightarrow s_2\ x = t_2\ x)) \wedge$
$(\forall\, x.\ (\exists\, p \in U.\ case\ p\ of\ (B,\ W) \Rightarrow$
$\exists\, s \in B.\ \exists\, y \in W.\ \neg\ s{:}\ dom\ y \rightsquigarrow dom\ x) \longrightarrow no\text{-}upd\ (flow\ cfs_2)\ x)$
$\langle proof \rangle$

lemma *ctyping2-correct-aux-assign* [*elim*]:
assumes
A: $(if\ (\forall\, s \in Univ?\ A\ X.\ \forall\, y \in avars\ a.\ s{:}\ dom\ y \rightsquigarrow dom\ x) \wedge$
$(\forall\, p \in U.\ \forall\, B\ Y.\ p = (B,\ Y) \longrightarrow$
$(\forall\, s \in B.\ \forall\, y \in Y.\ s{:}\ dom\ y \rightsquigarrow dom\ x))$
$then\ Some\ (if\ x \in state \wedge A \neq \{\}$
$then\ if\ v \models a\ (\subseteq X)$
$then\ (\{s(x := aval\ a\ s)\ |s.\ s \in A\},\ insert\ x\ X)$
$else\ (A,\ X - \{x\})$
$else\ (A,\ Univ??\ A\ X))$
$else\ None) = Some\ (B,\ Y)$
(**is** $(if\ ?P\ then\ \text{-}\ else\ \text{-}) = \text{-}$) **and**
B: $(x ::= a,\ s) \rightarrow*\{cfs_1\}\ (c_1,\ s_1)$ **and**
C: $(c_1,\ s_1) \rightarrow*\{cfs_2\}\ (c_2,\ s_2)$ **and**
D: $r \in A$ **and**
E: $s = r\ (\subseteq\ state \cap X)$
shows
$(\forall\, t_1.\ \exists\, c_2{}'\ t_2.\ \forall\, x.$
$(s_1 = t_1\ (\subseteq\ sources\text{-}aux\ (flow\ cfs_2)\ s_1\ x) \longrightarrow$
$(c_1,\ t_1) \rightarrow*\ (c_2{}',\ t_2) \wedge (c_2 = SKIP) = (c_2{}' = SKIP)) \wedge$
$(s_1 = t_1\ (\subseteq\ sources\ (flow\ cfs_2)\ s_1\ x) \longrightarrow s_2\ x = t_2\ x)) \wedge$
$(\forall\, x.\ (\exists\, p \in U.\ case\ p\ of\ (B,\ Y) \Rightarrow$
$\exists\, s \in B.\ \exists\, y \in Y.\ \neg\ s{:}\ dom\ y \rightsquigarrow dom\ x) \longrightarrow no\text{-}upd\ (flow\ cfs_2)\ x)$
$\langle proof \rangle$

lemma *ctyping2-correct-aux-seq*:
assumes
A: $\bigwedge B'\ s\ c'\ c''\ s_1\ s_2\ cfs_1\ cfs_2.\ B = B' \Longrightarrow$
$\exists\, r \in A.\ s = r\ (\subseteq\ state \cap X) \Longrightarrow$
$(c_1,\ s) \rightarrow*\{cfs_1\}\ (c',\ s_1) \Longrightarrow (c',\ s_1) \rightarrow*\{cfs_2\}\ (c'',\ s_2) \Longrightarrow$
$(\forall\, t_1.\ \exists\, c_2{}'\ t_2.\ \forall\, x.$
$(s_1 = t_1\ (\subseteq\ sources\text{-}aux\ (flow\ cfs_2)\ s_1\ x) \longrightarrow$
$(c',\ t_1) \rightarrow*\ (c_2{}',\ t_2) \wedge (c'' = SKIP) = (c_2{}' = SKIP)) \wedge$
$(s_1 = t_1\ (\subseteq\ sources\ (flow\ cfs_2)\ s_1\ x) \longrightarrow s_2\ x = t_2\ x)) \wedge$
$(\forall\, x.\ (\exists\, p \in U.\ case\ p\ of\ (B,\ W) \Rightarrow$
$\exists\, s \in B.\ \exists\, y \in W.\ \neg\ s{:}\ dom\ y \rightsquigarrow dom\ x) \longrightarrow$
$no\text{-}upd\ (flow\ cfs_2)\ x)$ **and**

B: $\bigwedge B'\ B''\ C\ Z\ s\ c'\ c''\ s_1\ s_2\ cfs_1\ cfs_2.\ B = B' \Longrightarrow B'' = B' \Longrightarrow$
$(U,\ v) \models c_2\ (\subseteq B',\ Y) = \mathit{Some}\ (C,\ Z) \Longrightarrow$
$\exists\, r \in B'.\ s = r\ (\subseteq \mathit{state} \cap Y) \Longrightarrow$
$(c_2,\ s) \rightarrow\!*\{cfs_1\}\ (c',\ s_1) \Longrightarrow (c',\ s_1) \rightarrow\!*\{cfs_2\}\ (c'',\ s_2) \Longrightarrow$
$(\forall\, t_1.\ \exists\, c_2{}'\ t_2.\ \forall\, x.$
$(s_1 = t_1\ (\subseteq \mathit{sources\text{-}aux}\ (\mathit{flow}\ cfs_2)\ s_1\ x) \longrightarrow$
$(c',\ t_1) \rightarrow\!*\ (c_2{}',\ t_2) \wedge (c'' = \mathit{SKIP}) = (c_2{}' = \mathit{SKIP})) \wedge$
$(s_1 = t_1\ (\subseteq \mathit{sources}\ (\mathit{flow}\ cfs_2)\ s_1\ x) \longrightarrow s_2\ x = t_2\ x)) \wedge$
$(\forall\, x.\ (\exists\, p \in U.\ \mathit{case}\ p\ \mathit{of}\ (B,\ W) \Rightarrow$
$\exists\, s \in B.\ \exists\, y \in W.\ \neg\ s\colon \mathit{dom}\ y \leadsto \mathit{dom}\ x) \longrightarrow$
$\mathit{no\text{-}upd}\ (\mathit{flow}\ cfs_2)\ x)$ **and**
C: $(U,\ v) \models c_1\ (\subseteq A,\ X) = \mathit{Some}\ (B,\ Y)$ **and**
D: $(U,\ v) \models c_2\ (\subseteq B,\ Y) = \mathit{Some}\ (C,\ Z)$ **and**
E: $(c_1;;\ c_2,\ s) \rightarrow\!*\{cfs_1\}\ (c',\ s_1)$ **and**
F: $(c',\ s_1) \rightarrow\!*\{cfs_2\}\ (c'',\ s_2)$ **and**
G: $r \in A$ **and**
H: $s = r\ (\subseteq \mathit{state} \cap X)$
shows
$(\forall\, t_1.\ \exists\, c_2{}'\ t_2.\ \forall\, x.$
$(s_1 = t_1\ (\subseteq \mathit{sources\text{-}aux}\ (\mathit{flow}\ cfs_2)\ s_1\ x) \longrightarrow$
$(c',\ t_1) \rightarrow\!*\ (c_2{}',\ t_2) \wedge (c'' = \mathit{SKIP}) = (c_2{}' = \mathit{SKIP})) \wedge$
$(s_1 = t_1\ (\subseteq \mathit{sources}\ (\mathit{flow}\ cfs_2)\ s_1\ x) \longrightarrow s_2\ x = t_2\ x)) \wedge$
$(\forall\, x.\ (\exists\, p \in U.\ \mathit{case}\ p\ \mathit{of}\ (B,\ W) \Rightarrow$
$\exists\, s \in B.\ \exists\, y \in W.\ \neg\ s\colon \mathit{dom}\ y \leadsto \mathit{dom}\ x) \longrightarrow \mathit{no\text{-}upd}\ (\mathit{flow}\ cfs_2)\ x)$
$\langle \mathit{proof} \rangle$

lemma *ctyping2-correct-aux-if*:
assumes
A: $\bigwedge U'\ B\ C\ s\ c'\ c''\ s_1\ s_2\ cfs_1\ cfs_2.$
$U' = \mathit{insert}\ (\mathit{Univ?}\ A\ X,\ \mathit{bvars}\ b)\ U \Longrightarrow B = B_1 \Longrightarrow C_1 = C \Longrightarrow$
$\exists\, r \in B_1.\ s = r\ (\subseteq \mathit{state} \cap X) \Longrightarrow$
$(c_1,\ s) \rightarrow\!*\{cfs_1\}\ (c',\ s_1) \Longrightarrow (c',\ s_1) \rightarrow\!*\{cfs_2\}\ (c'',\ s_2) \Longrightarrow$
$(\forall\, t_1.\ \exists\, c_2{}'\ t_2.\ \forall\, x.$
$(s_1 = t_1\ (\subseteq \mathit{sources\text{-}aux}\ (\mathit{flow}\ cfs_2)\ s_1\ x) \longrightarrow$
$(c',\ t_1) \rightarrow\!*\ (c_2{}',\ t_2) \wedge (c'' = \mathit{SKIP}) = (c_2{}' = \mathit{SKIP})) \wedge$
$(s_1 = t_1\ (\subseteq \mathit{sources}\ (\mathit{flow}\ cfs_2)\ s_1\ x) \longrightarrow s_2\ x = t_2\ x)) \wedge$
$(\forall\, x.$
$((\exists\, s \in \mathit{Univ?}\ A\ X.\ \exists\, y \in \mathit{bvars}\ b.\ \neg\ s\colon \mathit{dom}\ y \leadsto \mathit{dom}\ x) \longrightarrow$
$\mathit{no\text{-}upd}\ (\mathit{flow}\ cfs_2)\ x) \wedge$
$((\exists\, p \in U.\ \mathit{case}\ p\ \mathit{of}\ (B,\ W) \Rightarrow$
$\exists\, s \in B.\ \exists\, y \in W.\ \neg\ s\colon \mathit{dom}\ y \leadsto \mathit{dom}\ x) \longrightarrow$
$\mathit{no\text{-}upd}\ (\mathit{flow}\ cfs_2)\ x))$ **and**
B: $\bigwedge U'\ B\ C\ s\ c'\ c''\ s_1\ s_2\ cfs_1\ cfs_2.$
$U' = \mathit{insert}\ (\mathit{Univ?}\ A\ X,\ \mathit{bvars}\ b)\ U \Longrightarrow B = B_1 \Longrightarrow C_2 = C \Longrightarrow$
$\exists\, r \in B_2.\ s = r\ (\subseteq \mathit{state} \cap X) \Longrightarrow$
$(c_2,\ s) \rightarrow\!*\{cfs_1\}\ (c',\ s_1) \Longrightarrow (c',\ s_1) \rightarrow\!*\{cfs_2\}\ (c'',\ s_2) \Longrightarrow$
$(\forall\, t_1.\ \exists\, c_2{}'\ t_2.\ \forall\, x.$
$(s_1 = t_1\ (\subseteq \mathit{sources\text{-}aux}\ (\mathit{flow}\ cfs_2)\ s_1\ x) \longrightarrow$
$(c',\ t_1) \rightarrow\!*\ (c_2{}',\ t_2) \wedge (c'' = \mathit{SKIP}) = (c_2{}' = \mathit{SKIP})) \wedge$

$(s_1 = t_1\ (\subseteq\ \mathit{sources}\ (\mathit{flow}\ \mathit{cfs}_2)\ s_1\ x) \longrightarrow s_2\ x = t_2\ x)) \wedge$
$(\forall\, x.$
$((\exists\, s \in \mathit{Univ?}\ A\ X.\ \exists\, y \in \mathit{bvars}\ b.\ \neg\ s\colon \mathit{dom}\ y \rightsquigarrow \mathit{dom}\ x) \longrightarrow$
$\mathit{no\text{-}upd}\ (\mathit{flow}\ \mathit{cfs}_2)\ x) \wedge$
$((\exists\, p \in U.\ \mathit{case}\ p\ \mathit{of}\ (B,\ W) \Rightarrow$
$\exists\, s \in B.\ \exists\, y \in W.\ \neg\ s\colon \mathit{dom}\ y \rightsquigarrow \mathit{dom}\ x) \longrightarrow$
$\mathit{no\text{-}upd}\ (\mathit{flow}\ \mathit{cfs}_2)\ x))$ **and**
$C\colon \models b\ (\subseteq A,\ X) = (B_1,\ B_2)$ **and**
$D\colon (\mathit{insert}\ (\mathit{Univ?}\ A\ X,\ \mathit{bvars}\ b)\ U,\ v) \models c_1\ (\subseteq B_1,\ X) =$
$\mathit{Some}\ (C_1,\ Y_1)$ **and**
$E\colon (\mathit{insert}\ (\mathit{Univ?}\ A\ X,\ \mathit{bvars}\ b)\ U,\ v) \models c_2\ (\subseteq B_2,\ X) =$
$\mathit{Some}\ (C_2,\ Y_2)$ **and**
$F\colon (\mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2,\ s) \rightarrow\!*\{\mathit{cfs}_1\}\ (c',\ s_1)$ **and**
$G\colon (c',\ s_1) \rightarrow\!*\{\mathit{cfs}_2\}\ (c'',\ s_2)$ **and**
$H\colon r \in A$ **and**
$I\colon s = r\ (\subseteq \mathit{state} \cap X)$
shows
$(\forall\, t_1.\ \exists\, c_2{}'\ t_2.\ \forall\, x.$
$(s_1 = t_1\ (\subseteq\ \mathit{sources\text{-}aux}\ (\mathit{flow}\ \mathit{cfs}_2)\ s_1\ x) \longrightarrow$
$(c',\ t_1) \rightarrow\!*\ (c_2{}',\ t_2) \wedge (c'' = \mathit{SKIP}) = (c_2{}' = \mathit{SKIP})) \wedge$
$(s_1 = t_1\ (\subseteq\ \mathit{sources}\ (\mathit{flow}\ \mathit{cfs}_2)\ s_1\ x) \longrightarrow s_2\ x = t_2\ x)) \wedge$
$(\forall\, x.\ (\exists\, p \in U.\ \mathit{case}\ p\ \mathit{of}\ (B,\ W) \Rightarrow$
$\exists\, s \in B.\ \exists\, y \in W.\ \neg\ s\colon \mathit{dom}\ y \rightsquigarrow \mathit{dom}\ x) \longrightarrow \mathit{no\text{-}upd}\ (\mathit{flow}\ \mathit{cfs}_2)\ x)$
$\langle \mathit{proof} \rangle$

lemma *ctyping2-correct-aux-while*:
assumes
$A\colon \bigwedge B\ C'\ B'\ D'\ s\ c_1\ c_2\ s_1\ s_2\ \mathit{cfs}_1\ \mathit{cfs}_2.$
$B = B_1 \Longrightarrow C' = C \Longrightarrow B' = B_1{}' \Longrightarrow$
$(\forall\, s \in \mathit{Univ?}\ A\ X \cup \mathit{Univ?}\ C\ Y.\ \forall\, x \in \mathit{bvars}\ b.\ \mathit{All}\ (\mathit{interf}\ s\ (\mathit{dom}\ x))) \wedge$
$(\forall\, p \in U.\ \mathit{case}\ p\ \mathit{of}\ (B,\ W) \Rightarrow \forall\, s \in B.\ \forall\, x \in W.\ \mathit{All}\ (\mathit{interf}\ s\ (\mathit{dom}\ x))) \Longrightarrow$
$D = D' \Longrightarrow \exists\, r \in B_1.\ s = r\ (\subseteq \mathit{state} \cap X) \Longrightarrow$
$(c,\ s) \rightarrow\!*\{\mathit{cfs}_1\}\ (c_1,\ s_1) \Longrightarrow (c_1,\ s_1) \rightarrow\!*\{\mathit{cfs}_2\}\ (c_2,\ s_2) \Longrightarrow$
$\forall\, t_1.\ \exists\, c_2{}'\ t_2.\ \forall\, x.$
$(s_1 = t_1\ (\subseteq\ \mathit{sources\text{-}aux}\ (\mathit{flow}\ \mathit{cfs}_2)\ s_1\ x) \longrightarrow$
$(c_1,\ t_1) \rightarrow\!*\ (c_2{}',\ t_2) \wedge (c_2 = \mathit{SKIP}) = (c_2{}' = \mathit{SKIP})) \wedge$
$(s_1 = t_1\ (\subseteq\ \mathit{sources}\ (\mathit{flow}\ \mathit{cfs}_2)\ s_1\ x) \longrightarrow s_2\ x = t_2\ x)$ **and**
$B\colon \bigwedge B\ C'\ B'\ D''\ s\ c_1\ c_2\ s_1\ s_2\ \mathit{cfs}_1\ \mathit{cfs}_2.$
$B = B_1 \Longrightarrow C' = C \Longrightarrow B' = B_1{}' \Longrightarrow$
$(\forall\, s \in \mathit{Univ?}\ A\ X \cup \mathit{Univ?}\ C\ Y.\ \forall\, x \in \mathit{bvars}\ b.\ \mathit{All}\ (\mathit{interf}\ s\ (\mathit{dom}\ x))) \wedge$
$(\forall\, p \in U.\ \mathit{case}\ p\ \mathit{of}\ (B,\ W) \Rightarrow \forall\, s \in B.\ \forall\, x \in W.\ \mathit{All}\ (\mathit{interf}\ s\ (\mathit{dom}\ x))) \Longrightarrow$
$D' = D'' \Longrightarrow \exists\, r \in B_1{}'.\ s = r\ (\subseteq \mathit{state} \cap Y) \Longrightarrow$
$(c,\ s) \rightarrow\!*\{\mathit{cfs}_1\}\ (c_1,\ s_1) \Longrightarrow (c_1,\ s_1) \rightarrow\!*\{\mathit{cfs}_2\}\ (c_2,\ s_2) \Longrightarrow$
$\forall\, t_1.\ \exists\, c_2{}'\ t_2.\ \forall\, x.$
$(s_1 = t_1\ (\subseteq\ \mathit{sources\text{-}aux}\ (\mathit{flow}\ \mathit{cfs}_2)\ s_1\ x) \longrightarrow$
$(c_1,\ t_1) \rightarrow\!*\ (c_2{}',\ t_2) \wedge (c_2 = \mathit{SKIP}) = (c_2{}' = \mathit{SKIP})) \wedge$
$(s_1 = t_1\ (\subseteq\ \mathit{sources}\ (\mathit{flow}\ \mathit{cfs}_2)\ s_1\ x) \longrightarrow s_2\ x = t_2\ x)$ **and**
$C\colon (\mathit{if}\ (\forall\, s \in \mathit{Univ?}\ A\ X \cup \mathit{Univ?}\ C\ Y.\ \forall\, x \in \mathit{bvars}\ b.\ \mathit{All}\ (\mathit{interf}\ s\ (\mathit{dom}\ x))) \wedge$
$(\forall\, p \in U.\ \forall\, B\ W.\ p = (B,\ W) \longrightarrow (\forall\, s \in B.\ \forall\, x \in W.\ \mathit{All}\ (\mathit{interf}\ s\ (\mathit{dom}\ x))))$

$then\ Some\ (B_2 \cup B_2{'},\ Univ??\ B_2\ X \cap Y)\ else\ None) = Some\ (B,\ W)$ **and**
D: $\models b\ (\subseteq A,\ X) = (B_1,\ B_2)$ **and**
E: $\vdash c\ (\subseteq B_1,\ X) = (C,\ Y)$ **and**
F: $\models b\ (\subseteq C,\ Y) = (B_1{'},\ B_2{'})$ **and**
G: $(\{\},\ False) \models c\ (\subseteq B_1,\ X) = Some\ (D,\ Z)$ **and**
H: $(\{\},\ False) \models c\ (\subseteq B_1{'},\ Y) = Some\ (D',\ Z')$
shows
$[\![(WHILE\ b\ DO\ c,\ s) \rightarrow*\{cfs_1\}\ (c_1,\ s_1);$
$(c_1,\ s_1) \rightarrow*\{cfs_2\}\ (c_2,\ s_2);$
$s \in Univ\ A\ (\subseteq state \cap X) \cup Univ\ C\ (\subseteq state \cap Y)]\!] \Longrightarrow$
$(\forall t_1.\ \exists c_2{'}\ t_2.\ \forall x.$
$(s_1 = t_1\ (\subseteq sources\text{-}aux\ (flow\ cfs_2)\ s_1\ x) \longrightarrow$
$(c_1,\ t_1) \rightarrow*\ (c_2{'},\ t_2) \wedge (c_2 = SKIP) = (c_2{'} = SKIP)) \wedge$
$(s_1 = t_1\ (\subseteq sources\ (flow\ cfs_2)\ s_1\ x) \longrightarrow s_2\ x = t_2\ x)) \wedge$
$(\forall x.\ (\exists p \in U.\ case\ p\ of\ (B,\ W) \Rightarrow$
$\exists s \in B.\ \exists y \in W.\ \neg\ s:\ dom\ y \leadsto dom\ x) \longrightarrow no\text{-}upd\ (flow\ cfs_2)\ x)$
$\langle proof \rangle$

lemma *ctyping2-correct-aux*:
$[\![(U,\ v) \models c\ (\subseteq A,\ X) = Some\ (B,\ Y);\ s \in Univ\ A\ (\subseteq state \cap X);$
$(c,\ s) \rightarrow*\{cfs_1\}\ (c_1,\ s_1);\ (c_1,\ s_1) \rightarrow*\{cfs_2\}\ (c_2,\ s_2)]\!] \Longrightarrow$
$ok\text{-}flow\text{-}aux\ U\ c_1\ c_2\ s_1\ s_2\ (flow\ cfs_2)$
$\langle proof \rangle$

theorem *ctyping2-correct*:
assumes A: $(U,\ v) \models c\ (\subseteq A,\ X) = Some\ (B,\ Y)$
shows $correct\ c\ A\ X$
$\langle proof \rangle$

end

end

5 Degeneracy to stateless level-based information flow control

theory *Degeneracy*
imports *Correctness HOL−IMP.Sec-TypingT*
begin

The goal of this concluding section is to prove the degeneracy of the information flow correctness notion and the static type system defined in this paper to the classical counterparts addressed in [7], section 9.2.6, and formalized in [5] and [6], in case of a stateless level-based information flow correctness policy.

First of all, locale *noninterf* is interpreted within the context of the class *sec* defined in [5], as follows.

- Parameter *dom* is instantiated as function *sec*, which also sets the type variable standing for the type of the domains to *nat*.
- Parameter *interf* is instantiated as the predicate such that for any program state, the output is *True* just in case the former input level may interfere with, namely is not larger than, the latter one.
- Parameter *state* is instantiated as the empty set, consistently with the fact that the policy is represented by a single, stateless interference predicate.

Next, the information flow security notion implied by theorem *noninterference* in [6] is formalized as a predicate *secure* taking a program as input. This notion is then proven to be implied, in the degenerate interpretation described above, by the information flow correctness notion formalized as predicate *correct* (theorem *correct-secure*). Particularly:

- This theorem demands the additional assumption that the *state set* A input to *correct* is nonempty, since *correct* is vacuously true for $A = \{\}$.
- In order for this theorem to hold, predicate *secure* needs to slight differ from the information flow security notion implied by theorem *noninterference*, in that it requires state t' to exist if there also exists some variable with a level not larger than l, namely if condition $s = t$ $(\le l)$ is satisfied *nontrivially* – actually, no leakage may arise from two initial states disagreeing on the value of *every* variable. In fact, predicate *correct* requires a nontrivial configuration $({c_2}', t_2)$ to exist in case condition $s_1 = t_1$ $(\subseteq \textit{sources cs } s_1\ x)$ is satisfied *for some variable* x.

Finally, the static type system *ctyping2* is proven to be equivalent to the *sec-type* one defined in [6] in the above degenerate interpretation (theorems *ctyping2-sec-type* and *sec-type-ctyping2*). The former theorem, which proves that a *pass* verdict from *ctyping2* implies the issuance of a *pass* verdict from *sec-type* as well, demands the additional assumptions that (a) the *state set* input to *ctyping2* is nonempty, (b) the input program does not contain any loop with *Bc True* as boolean condition, and (c) the input program has undergone *constant folding*, as addressed in [7], section 3.1.3 for arithmetic expressions and in [7], section 3.2.1 for boolean expressions. Why?

This need arises from the different ways in which the two type systems handle "dead" conditional branches. Type system *sec-type* does not try to detect "dead" branches; it simply applies its full range of information flow security checks to any conditional branch contained in the input program, even if it actually is a "dead" one. On the contrary, type system *ctyping2* detects "dead" branches whenever boolean conditions can be evaluated at compile time, and applies only a subset of its information flow correctness checks to such branches.

As parameter *state* is instantiated as the empty set, boolean conditions containing variables cannot be evaluated at compile time, yet they can if they only contain constants. Thus, assumption (a) prevents *ctyping2* from handling the entire input program as a "dead" branch, while assumptions (b) and (c) ensure that *ctyping2* will not detect any "dead" conditional branch within the program. On the whole, those assumptions guarantee that *ctyping2*, like *sec-type*, applies its full range of checks to *any* conditional branch contained in the input program, as required for theorem *ctyping2-sec-type* to hold.

5.1 Global context definitions and proofs

fun $cgood :: com \Rightarrow bool$ **where**
$cgood\ (c_1;;\ c_2) = (cgood\ c_1 \wedge cgood\ c_2)\ |$
$cgood\ (IF\ \text{-}\ THEN\ c_1\ ELSE\ c_2) = (cgood\ c_1 \wedge cgood\ c_2)\ |$
$cgood\ (WHILE\ b\ DO\ c) = (b \neq Bc\ True \wedge cgood\ c)\ |$
$cgood\ \text{-} = True$

fun $seq :: com \Rightarrow com \Rightarrow com$ **where**
$seq\ SKIP\ c = c\ |$
$seq\ c\ SKIP = c\ |$
$seq\ c_1\ c_2 = c_1;;\ c_2$

fun $ifc :: bexp \Rightarrow com \Rightarrow com \Rightarrow com$ **where**
$ifc\ (Bc\ True)\ c\ \text{-} = c\ |$
$ifc\ (Bc\ False)\ \text{-}\ c = c\ |$
$ifc\ b\ c_1\ c_2 = (if\ c_1 = c_2\ then\ c_1\ else\ IF\ b\ THEN\ c_1\ ELSE\ c_2)$

fun $while :: bexp \Rightarrow com \Rightarrow com$ **where**
$while\ (Bc\ False)\ \text{-} = SKIP\ |$
$while\ b\ c = WHILE\ b\ DO\ c$

primrec $csimp :: com \Rightarrow com$ **where**
$csimp\ SKIP = SKIP\ |$
$csimp\ (x ::= a) = x ::= asimp\ a\ |$
$csimp\ (c_1;;\ c_2) = seq\ (csimp\ c_1)\ (csimp\ c_2)\ |$
$csimp\ (IF\ b\ THEN\ c_1\ ELSE\ c_2) = ifc\ (bsimp\ b)\ (csimp\ c_1)\ (csimp\ c_2)\ |$
$csimp\ (WHILE\ b\ DO\ c) = while\ (bsimp\ b)\ (csimp\ c)$

lemma *not-size*:

$size\ (not\ b) \leq Suc\ (size\ b)$
⟨*proof*⟩

lemma *and-size*:
$size\ (and\ b_1\ b_2) \leq Suc\ (size\ b_1 + size\ b_2)$
⟨*proof*⟩

lemma *less-size*:
$size\ (less\ a_1\ a_2) = 0$
⟨*proof*⟩

lemma *bsimp-size*:
$size\ (bsimp\ b) \leq size\ b$
⟨*proof*⟩

lemma *seq-size*:
$size\ (seq\ c_1\ c_2) \leq Suc\ (size\ c_1 + size\ c_2)$
⟨*proof*⟩

lemma *ifc-size*:
$size\ (ifc\ b\ c_1\ c_2) \leq Suc\ (size\ c_1 + size\ c_2)$
⟨*proof*⟩

lemma *while-size*:
$size\ (while\ b\ c) \leq Suc\ (size\ c)$
⟨*proof*⟩

lemma *csimp-size*:
$size\ (csimp\ c) \leq size\ c$
⟨*proof*⟩

lemma *avars-asimp*:
$avars\ a = \{\} \Longrightarrow \exists\, i.\ asimp\ a = N\ i$
⟨*proof*⟩

lemma *seq-match* [*dest*!]:
$seq\ (csimp\ c_1)\ (csimp\ c_2) = c_1;;\ c_2 \Longrightarrow csimp\ c_1 = c_1 \wedge csimp\ c_2 = c_2$
⟨*proof*⟩

lemma *ifc-match* [*dest*!]:
$ifc\ (bsimp\ b)\ (csimp\ c_1)\ (csimp\ c_2) = IF\ b\ THEN\ c_1\ ELSE\ c_2 \Longrightarrow$
$\quad bsimp\ b = b \wedge (\forall\, v.\ b \neq Bc\ v) \wedge csimp\ c_1 = c_1 \wedge csimp\ c_2 = c_2$
⟨*proof*⟩

lemma *while-match* [*dest*!]:
$while\ (bsimp\ b)\ (csimp\ c) = WHILE\ b\ DO\ c \Longrightarrow$
$\quad bsimp\ b = b \wedge b \neq Bc\ False \wedge csimp\ c = c$

$\langle proof \rangle$

5.2 Local context definitions and proofs

context *sec*
begin

interpretation *noninterf* $\lambda s.$ $(\leq)$ *sec* $\{\}$
$\langle proof \rangle$

notation *interf-set* (‹(-: - $\rightsquigarrow$ -)› [*51*, *51*, *51*] *50*)
notation *univ-states-if* (‹(*Univ?* - -)› [*51*, *75*] *75*)
notation *atyping* (‹(- $\models$ - $'(\subseteq$ -$')$)› [*51*, *51*] *50*)
notation *btyping2-aux* (‹($\models$ - $'(\subseteq$ -, -$')$)› [*51*] *55*)
notation *btyping2* (‹($\models$ - $'(\subseteq$ -, -$')$)› [*51*] *55*)
notation *ctyping1* (‹($\vdash$ - $'(\subseteq$ -, -$')$)› [*51*] *55*)
notation *ctyping2* (‹(- $\models$ - $'(\subseteq$ -, -$')$)› [*51*, *51*] *55*)

abbreviation *eq-le-ext* :: *state* $\Rightarrow$ *state* $\Rightarrow$ *level* $\Rightarrow$ *bool*
(‹(- = - $'(\leq$ -$')$)› [*51*, *51*, *0*] *50*) **where**
$s = t \ (\leq l) \equiv s = t \ (\leq l) \wedge (\exists\, x :: vname.\ sec\ x \leq l)$

definition *secure* :: *com* $\Rightarrow$ *bool* **where**
$secure\ c \equiv \forall\, s\ s'\ t\ l.\ (c,\ s) \Rightarrow s' \wedge s = t\ (\leq l) \longrightarrow$
$(\exists\, t'.\ (c,\ t) \Rightarrow t' \wedge s' = t'\ (\leq l))$

definition *levels* :: *config set* $\Rightarrow$ *level set* **where**
$levels\ U \equiv insert\ 0\ (sec\ `\ \bigcup\ (snd\ `\ \{(B,\ Y) \in U.\ B \neq \{\}\}))$

lemma *avars-finite*:
finite (*avars a*)
$\langle proof \rangle$

lemma *avars-in*:
$n < sec\ a \Longrightarrow sec\ a \in sec\ `\ avars\ a$
$\langle proof \rangle$

lemma *avars-sec*:
$x \in avars\ a \Longrightarrow sec\ x \leq sec\ a$
$\langle proof \rangle$

lemma *avars-ub*:
$sec\ a \leq l = (\forall\, x \in avars\ a.\ sec\ x \leq l)$
$\langle proof \rangle$

lemma *bvars-finite*:
$\mathit{finite}\ (\mathit{bvars}\ b)$
$\langle \mathit{proof} \rangle$

lemma *bvars-in*:
$n < \mathit{sec}\ b \Longrightarrow \mathit{sec}\ b \in \mathit{sec}\ `\ \mathit{bvars}\ b$
$\langle \mathit{proof} \rangle$

lemma *bvars-sec*:
$x \in \mathit{bvars}\ b \Longrightarrow \mathit{sec}\ x \leq \mathit{sec}\ b$
$\langle \mathit{proof} \rangle$

lemma *bvars-ub*:
$\mathit{sec}\ b \leq l = (\forall\, x \in \mathit{bvars}\ b.\ \mathit{sec}\ x \leq l)$
$\langle \mathit{proof} \rangle$

lemma *levels-insert*:
  assumes
    A: $A \neq \{\}$ **and**
    B: $\mathit{finite}\ (\mathit{levels}\ U)$
  shows $\mathit{finite}\ (\mathit{levels}\ (\mathit{insert}\ (A,\ \mathit{bvars}\ b)\ U)) \wedge$
    $\mathit{Max}\ (\mathit{levels}\ (\mathit{insert}\ (A,\ \mathit{bvars}\ b)\ U)) = \mathit{max}\ (\mathit{sec}\ b)\ (\mathit{Max}\ (\mathit{levels}\ U))$
    (**is** $\mathit{finite}\ (\mathit{levels}\ ?U') \wedge\ ?P$)
$\langle \mathit{proof} \rangle$

lemma *sources-le*:
$y \in \mathit{sources}\ \mathit{cs}\ s\ x \Longrightarrow \mathit{sec}\ y \leq \mathit{sec}\ x$
and *sources-aux-le*:
$y \in \mathit{sources\text{-}aux}\ \mathit{cs}\ s\ x \Longrightarrow \mathit{sec}\ y \leq \mathit{sec}\ x$
$\langle \mathit{proof} \rangle$

lemma *bsimp-btyping2-aux-not* [*intro*]:
$[\![\mathit{bsimp}\ b = b \Longrightarrow \forall\, v.\ b \neq \mathit{Bc}\ v \Longrightarrow \models b\ (\subseteq A,\ X) = \mathit{None};$
    $\mathit{not}\ (\mathit{bsimp}\ b) = \mathit{Not}\ b]\!] \Longrightarrow\ \models b\ (\subseteq A,\ X) = \mathit{None}$
$\langle \mathit{proof} \rangle$

lemma *bsimp-btyping2-aux-and* [*intro*]:
  assumes
    A: $[\![\mathit{bsimp}\ b_1 = b_1;\ \forall\, v.\ b_1 \neq \mathit{Bc}\ v]\!] \Longrightarrow\ \models b_1\ (\subseteq A,\ X) = \mathit{None}$ **and**
    B: $\mathit{and}\ (\mathit{bsimp}\ b_1)\ (\mathit{bsimp}\ b_2) = \mathit{And}\ b_1\ b_2$
  shows $\models b_1\ (\subseteq A,\ X) = \mathit{None}$
$\langle \mathit{proof} \rangle$

lemma *bsimp-btyping2-aux-less* [*elim*]:
$[\![\mathit{less}\ (\mathit{asimp}\ a_1)\ (\mathit{asimp}\ a_2) = \mathit{Less}\ a_1\ a_2;$
    $\mathit{avars}\ a_1 = \{\};\ \mathit{avars}\ a_2 = \{\}]\!] \Longrightarrow \mathit{False}$
$\langle \mathit{proof} \rangle$

lemma *bsimp-btyping2-aux*:
$[\![bsimp\ b = b;\ \forall\, v.\ b \neq Bc\ v]\!] \Longrightarrow \models b\ (\subseteq A,\ X) = None$
$\langle proof \rangle$

lemma *bsimp-btyping2*:
$[\![bsimp\ b = b;\ \forall\, v.\ b \neq Bc\ v]\!] \Longrightarrow \models b\ (\subseteq A,\ X) = (A,\ A)$
$\langle proof \rangle$

lemma *csimp-ctyping2-if*:
$[\![\bigwedge U'\ B\ B'.\ U' = U \Longrightarrow B = B_1 \Longrightarrow \{\} = B' \Longrightarrow B_1 \neq \{\} \Longrightarrow False;\ s \in A;$
$\models b\ (\subseteq A,\ X) = (B_1,\ B_2);\ bsimp\ b = b;\ \forall\, v.\ b \neq Bc\ v]\!] \Longrightarrow$
$False$
$\langle proof \rangle$

lemma *csimp-ctyping2-while*:
$[\![(if\ P\ then\ Some\ (B_2 \cup B_2{}',\ Y)\ else\ None) = Some\ (\{\},\ Z);\ s \in A;$
$\models b\ (\subseteq A,\ X) = (B_1,\ B_2);\ bsimp\ b = b;\ b \neq Bc\ True;\ b \neq Bc\ False]\!] \Longrightarrow$
$False$
$\langle proof \rangle$

lemma *csimp-ctyping2*:
$[\![(U,\ v) \models c\ (\subseteq A,\ X) = Some\ (B,\ Y);\ A \neq \{\};\ cgood\ c;\ csimp\ c = c]\!] \Longrightarrow$
$B \neq \{\}$
$\langle proof \rangle$

theorem *correct-secure*:
assumes
A: *correct c A X* **and**
B: $A \neq \{\}$
shows *secure c*
$\langle proof \rangle$

lemma *ctyping2-sec-type-assign* [*elim*]:
assumes
A: $(if\ ((\exists\, s.\ s \in Univ?\ A\ X) \longrightarrow (\forall\, y \in avars\ a.\ sec\ y \leq sec\ x)) \wedge$
$(\forall\, p \in U.\ \forall\, B\ Y.\ p = (B,\ Y) \longrightarrow B = \{\} \vee (\forall\, y \in Y.\ sec\ y \leq sec\ x))$
$then\ Some\ (if\ x \in \{\} \wedge A \neq \{\}$
$then\ if\ v \models a\ (\subseteq X)$
$then\ (\{s(x := aval\ a\ s)\ |s.\ s \in A\},\ insert\ x\ X)\ else\ (A,\ X - \{x\})$
$else\ (A,\ Univ??\ A\ X))$
$else\ None) = Some\ (B,\ Y)$
(**is** (*if* (- $\longrightarrow$ *?P*) $\wedge$ *?Q then* - *else* -) = -) **and**
B: $s \in A$ **and**
C: *finite* (*levels U*)
shows $Max\ (levels\ U) \vdash x ::= a$

$\langle proof \rangle$

lemma *ctyping2-sec-type-seq*:
assumes
A: $\bigwedge B'\ s.\ B = B' \Longrightarrow s \in A \Longrightarrow Max\ (levels\ U) \vdash c_1$ **and**
B: $\bigwedge B'\ B''\ C\ Z\ s'.\ B = B' \Longrightarrow B'' = B' \Longrightarrow$
$(U,\ v) \models c_2\ (\subseteq B',\ Y) = Some\ (C,\ Z) \Longrightarrow$
$s' \in B' \Longrightarrow Max\ (levels\ U) \vdash c_2$ **and**
C: $(U,\ v) \models c_1\ (\subseteq A,\ X) = Some\ (B,\ Y)$ **and**
D: $(U,\ v) \models c_2\ (\subseteq B,\ Y) = Some\ (C,\ Z)$ **and**
E: $s \in A$ **and**
F: $cgood\ c_1$ **and**
G: $csimp\ c_1 = c_1$
shows $Max\ (levels\ U) \vdash c_1;;\ c_2$
$\langle proof \rangle$

lemma *ctyping2-sec-type-if*:
assumes
A: $\bigwedge U'\ B\ C\ s.\ U' = insert\ (Univ?\ A\ X,\ bvars\ b)\ U \Longrightarrow$
$B = B_1 \Longrightarrow C_1 = C \Longrightarrow s \in B_1 \Longrightarrow$
$finite\ (levels\ (insert\ (Univ?\ A\ X,\ bvars\ b)\ U)) \Longrightarrow$
$Max\ (levels\ (insert\ (Univ?\ A\ X,\ bvars\ b)\ U)) \vdash c_1$
(**is** $\bigwedge$- - - -. - = $?U' \Longrightarrow$ - $\Longrightarrow$ - $\Longrightarrow$ - $\Longrightarrow$ - $\Longrightarrow$ -)
assumes
B: $\bigwedge U'\ B\ C\ s.\ U' = ?U' \Longrightarrow B = B_1 \Longrightarrow C_2 = C \Longrightarrow s \in B_2 \Longrightarrow$
$finite\ (levels\ ?U') \Longrightarrow Max\ (levels\ ?U') \vdash c_2$ **and**
C: $\models b\ (\subseteq A,\ X) = (B_1,\ B_2)$ **and**
D: $s \in A$ **and**
E: $bsimp\ b = b$ **and**
F: $\forall v.\ b \neq Bc\ v$ **and**
G: $finite\ (levels\ U)$
shows $Max\ (levels\ U) \vdash IF\ b\ THEN\ c_1\ ELSE\ c_2$
$\langle proof \rangle$

lemma *ctyping2-sec-type-while*:
assumes
A: $\bigwedge B\ C'\ B'\ D'\ s.\ B = B_1 \Longrightarrow C' = C \Longrightarrow B' = B_1{}' \Longrightarrow$
$((\exists s.\ s \in Univ?\ A\ X \lor s \in Univ?\ C\ Y) \longrightarrow$
$(\forall x \in bvars\ b.\ All\ ((\leq)\ (sec\ x)))) \land$
$(\forall p \in U.\ case\ p\ of\ (B,\ W) \Rightarrow (\exists s.\ s \in B) \longrightarrow$
$(\forall x \in W.\ All\ ((\leq)\ (sec\ x)))) \Longrightarrow$
$D = D' \Longrightarrow s \in B_1 \Longrightarrow finite\ (levels\ \{\}) \Longrightarrow Max\ (levels\ \{\}) \vdash c$
(**is** $\bigwedge$- - - - -. - $\Longrightarrow$ - $\Longrightarrow$ - $\Longrightarrow$
$?P \land (\forall p \in$ -. $case\ p\ of$ (-, $W) \Rightarrow$ - $\longrightarrow ?Q\ W) \Longrightarrow$
- $\Longrightarrow$ - $\Longrightarrow$ - $\Longrightarrow$ -)
assumes
B: $(if\ ?P \land (\forall p \in U.\ \forall B\ W.\ p = (B,\ W) \longrightarrow B = \{\} \lor ?Q\ W)$
$then\ Some\ (B_2 \cup B_2{}',\ Univ??\ B_2\ X \cap Y)\ else\ None) = Some\ (B,\ Z)$
(**is** (*if ?R then* - *else* -) = -) **and**

C: $\models b\ (\subseteq A,\ X) = (B_1,\ B_2)$ **and**
D: $s \in A$ **and**
E: $bsimp\ b = b$ **and**
F: $b \neq Bc\ False$ **and**
G: $b \neq Bc\ True$ **and**
H: $finite\ (levels\ U)$
shows $Max\ (levels\ U) \vdash WHILE\ b\ DO\ c$
$\langle proof \rangle$

theorem *ctyping2-sec-type*:
$[\![(U,\ v) \models c\ (\subseteq A,\ X) = Some\ (B,\ Y);$
$s \in A;\ cgood\ c;\ csimp\ c = c;\ finite\ (levels\ U)]\!] \Longrightarrow$
$Max\ (levels\ U) \vdash c$
$\langle proof \rangle$

lemma *sec-type-ctyping2-if*:
assumes
A: $\bigwedge U'\ B_1\ B_2.\ U' = insert\ (Univ?\ A\ X,\ bvars\ b)\ U \Longrightarrow$
$(B_1,\ B_2) = \models b\ (\subseteq A,\ X) \Longrightarrow$
$Max\ (levels\ (insert\ (Univ?\ A\ X,\ bvars\ b)\ U)) \vdash c_1 \Longrightarrow$
$finite\ (levels\ (insert\ (Univ?\ A\ X,\ bvars\ b)\ U)) \Longrightarrow$
$\exists\, C\ Y.\ (insert\ (Univ?\ A\ X,\ bvars\ b)\ U,\ v) \models c_1\ (\subseteq B_1,\ X) =$
$Some\ (C,\ Y)$
(**is** $\bigwedge$- - -. - = $?U' \Longrightarrow$ - $\Longrightarrow$ - $\Longrightarrow$ - $\Longrightarrow$ -)
assumes
B: $\bigwedge U'\ B_1\ B_2.\ U' = ?U' \Longrightarrow (B_1,\ B_2) = \models b\ (\subseteq A,\ X) \Longrightarrow$
$Max\ (levels\ ?U') \vdash c_2 \Longrightarrow finite\ (levels\ ?U') \Longrightarrow$
$\exists\, C\ Y.\ (?U',\ v) \models c_2\ (\subseteq B_2,\ X) = Some\ (C,\ Y)$ **and**
C: $finite\ (levels\ U)$ **and**
D: $max\ (sec\ b)\ (Max\ (levels\ U)) \vdash c_1$ **and**
E: $max\ (sec\ b)\ (Max\ (levels\ U)) \vdash c_2$
shows $\exists\, C\ Y.\ (U,\ v) \models IF\ b\ THEN\ c_1\ ELSE\ c_2\ (\subseteq A,\ X) = Some\ (C,\ Y)$
$\langle proof \rangle$

lemma *sec-type-ctyping2-while*:
assumes
A: $\bigwedge B_1\ B_2\ C\ Y\ B_1'\ B_2'.\ (B_1,\ B_2) = \models b\ (\subseteq A,\ X) \Longrightarrow$
$(C,\ Y) = \vdash c\ (\subseteq B_1,\ X) \Longrightarrow (B_1',\ B_2') = \models b\ (\subseteq C,\ Y) \Longrightarrow$
$((\exists\, s.\ s \in Univ?\ A\ X \vee s \in Univ?\ C\ Y) \longrightarrow$
$(\forall\, x \in bvars\ b.\ All\ ((\le)\ (sec\ x)))) \wedge$
$(\forall\, p \in U.\ case\ p\ of\ (B,\ W) \Rightarrow (\exists\, s.\ s \in B) \longrightarrow$
$(\forall\, x \in W.\ All\ ((\le)\ (sec\ x)))) \Longrightarrow$
$Max\ (levels\ \{\}) \vdash c \Longrightarrow finite\ (levels\ \{\}) \Longrightarrow$
$\exists\, D\ Z.\ (\{\},\ False) \models c\ (\subseteq B_1,\ X) = Some\ (D,\ Z)$
(**is** $\bigwedge$- - $C\ Y$ - -. - $\Longrightarrow$ - $\Longrightarrow$ - $\Longrightarrow$ $?P\ C\ Y \Longrightarrow$ - $\Longrightarrow$ - $\Longrightarrow$ -)
assumes
B: $\bigwedge B_1\ B_2\ C\ Y\ B_1'\ B_2'.\ (B_1,\ B_2) = \models b\ (\subseteq A,\ X) \Longrightarrow$

$(C, Y) = \vdash c (\subseteq B_1, X) \Longrightarrow (B_1', B_2') = \models b (\subseteq C, Y) \Longrightarrow$
$?P\ C\ Y \Longrightarrow Max\ (levels\ \{\}) \vdash c \Longrightarrow finite\ (levels\ \{\}) \Longrightarrow$
$\exists D\ Z.\ (\{\}, False) \models c\ (\subseteq B_1', Y) = Some\ (D, Z)$ **and**
C: $finite\ (levels\ U)$ **and**
D: $Max\ (levels\ U) = 0$ **and**
E: $sec\ b = 0$ **and**
F: $0 \vdash c$
shows $\exists B\ Y.\ (U, v) \models WHILE\ b\ DO\ c\ (\subseteq A, X) = Some\ (B, Y)$
$\langle proof \rangle$

theorem *sec-type-ctyping2*:
$[\![Max\ (levels\ U) \vdash c;\ finite\ (levels\ U)]\!] \Longrightarrow$
$\exists B\ Y.\ (U, v) \models c\ (\subseteq A, X) = Some\ (B, Y)$
$\langle proof \rangle$

end

end

References

[1] C. Ballarin. *Tutorial to Locales and Locale Interpretation.* https://isabelle.in.tum.de/website-Isabelle2023/dist/Isabelle2023/doc/locales.pdf.

[2] A. Krauss. *Defining Recursive Functions in Isabelle/HOL.* https://isabelle.in.tum.de/website-Isabelle2023/dist/Isabelle2023/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*, Sept. 2023. https://isabelle.in.tum.de/website-Isabelle2023/dist/Isabelle2023/doc/prog-prove.pdf.

[5] T. Nipkow and G. Klein. Theory HOL-IMP.Sec_Type_Expr (included in the Isabelle2023 distribution). https://isabelle.in.tum.de/website-Isabelle2023/dist/library/HOL/HOL-IMP/Sec_Type_Expr.html.

[6] T. Nipkow and G. Klein. Theory HOL-IMP.Sec_TypingT (included in the Isabelle2023 distribution). https://isabelle.in.tum.de/website-Isabelle2023/dist/library/HOL/HOL-IMP/Sec_TypingT.html.

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

[8] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, Sept. 2023. https://isabelle.in.tum.de/website-Isabelle2023/dist/Isabelle2023/doc/tutorial.pdf.

[9] J. Rushby. Noninterference, Transitivity, and Channel-Control Security Policies. Technical report, SRI International, Dec. 1992.

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

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