

**A machine-checked soundness proof for
an efficient verification condition
generator: technical report**

Frédéric Vogels Bart Jacobs Frank Piessens

Report CW 568, April 2010



Katholieke Universiteit Leuven
Department of Computer Science

Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

A machine-checked soundness proof for an efficient verification condition generator: technical report

Frédéric Vogels Bart Jacobs Frank Piessens

Report CW 568, April 2010

Department of Computer Science, K.U.Leuven

Abstract

Verification conditions (VCs) are logical formulae whose validity implies the correctness of a program with respect to a specification. The technique of checking software properties by specifying them in a program logic, then generating VCs, and finally feeding these VCs to a theorem prover, is several decades old. It is the underlying technology for state-of-the-art program verifiers such as the Spec \sharp programming system, or ESC/Java. The classic way of computing VCs is by means of Dijkstra's weakest precondition calculus. However, modern verification condition generators (VCgens), including Spec \sharp and ESC/Java's VCgens, are based on an optimized version of this algorithm, that avoids an exponential growth of the VCs in the length of the program to be verified. For this optimized VCgen algorithm, only informal soundness arguments are available. The paper "A machine-checked soundness proof for an efficient verification condition generator" by the same authors describes a fully formal, machine-checked proof of the soundness of such an efficient VCgen algorithm. This technical report elaborates further on the subject by fully detailing all definitions, theorems and proofs.

Contents

1	Introduction	2
1.1	Overview	2
1.2	The Coq script	3
2	The intermediate language	4
2.1	Example	4
2.2	Commands and their behaviour	4
2.3	Example (cont'd)	10
3	Transformation into single assignment form	14
3.1	Single Assignment Form	14
3.2	Sketch of the algorithm	15
3.3	Formal definition of the SA-transformation	16
3.4	Theorems and proofs	20
4	Passification	29
4.1	Algorithm	29
4.2	Failure property	33
4.3	Theorems and proofs	34
5	Weakest preconditions	42
5.1	Stuck states	42
5.2	Conservative and liberal	42
5.3	Soundness of the weakest preconditions	44
5.4	Efficient weakest preconditions	44
5.5	Soundness	55
5.6	Size	56
6	The Coq script	60
6.1	The operational semantics	60
6.2	Size of the weakest preconditions	63
A	Appendix	68

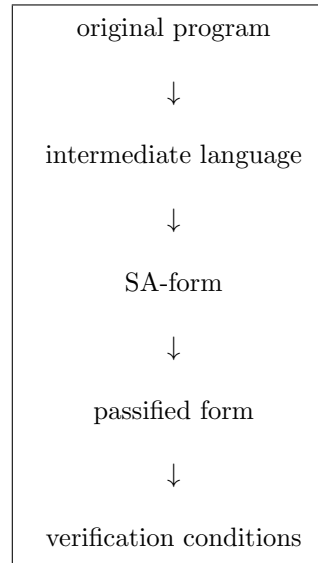


Figure 1: Steps in the verification algorithm

1 Introduction

This technical report is a companion to [9] which, due to a strict page limit, was rather superficial in presenting the algorithm and the proof of its correctness around which the paper revolved. We try to remedy this problem by providing a more complete picture in this document. We tried our best to keep the notations the same, but there are a few deviations, such as how we introduced different arrows (\longrightarrow , \longrightarrow_v and \longrightarrow_p) to make the difference between the different operational semantics more explicit. We believe the benefits (such as clarity) outweigh the disadvantages of changing the notation.

1.1 Overview

Our goal is to provide a machine-checked proof of the soundness of a specific approach to program verification, i.e. the one described by [6] which is essentially a reformulation of [4]. This approach consists of several steps, as shown in Figure 1.

Given a program (written in any programming language, such as Java or C[#]), we first translate it to an intermediate language [4, 6, 1, 7] which we fully formalize in Section 2. We will not be discussing this step, as it lies beyond the scope of this document. For more information, we refer the interested reader to [8].

For reasons explained later, the result of this first translation needs to be transformed, which happens in two separate phases: the single-assignment transformation, which we'll focus on in Section 3, and passification, which is the main subject of Section 4. We will show that both these transformations will preserve some important property of the original code, this property being important to proving the soundness of the verification algorithm.

In the last step, we derive verification conditions, which we discuss in Section 5. We will prove that these verification conditions are sound, i.e. that their validity implies that the original program is “correct”, which we will define more precisely later on. Apart from their soundness, we are also interested in the size of the generated verification conditions: we wish them not to grow too large. This is also dealt with in Section 5.

1.2 The Coq script

Every theorem¹ presented in this document has its counterpart in the Coq script [3]. Each time we state a new theorem, we will mention its Coq name between parentheses.

This text contains only a fraction of all Coq theorems: this document states around 50 theorems, while the Coq script contains approximately 200. Also, only very few (about a dozen) proofs are actually written out here as we only intended to give a high-level view of the Coq script and did not wish to clutter this text with too many details. We do admit that the proofs that are available may contain a bit of hand-waving and may not always convince (or worse, contain errors), but keep in mind a fully machine-checked proof is always available in the Coq script, which can be found at the end of this document. We tried our best to keep the proofs in this text as close as possible to the ones in the Coq script, and while we did sometimes use different identifiers, we believe that they do give a rather good picture of how the actual Coq proof works.

¹Every theorem except one, but this theorem is completely redundant and not needed for proving either the soundness or the size complexity of the verification conditions.

2 The intermediate language

Programs written in the intermediate language, based upon [7, 6, 4], do not represent computations as in regular programming languages. Instead, they represent possible execution paths populated with assertions, i.e. conditions we want to be true regardless of which execution path led to it.

2.1 Example

Let's take a look at a simple example. Figure 2 shows a Java function (a static method to be precise) which computes the average of a series of numbers, contained within an array. For the sake of simplicity, let's focus on preventing runtime errors only. There are several locations where these could occur:

- We access `array`'s `length` field: this is only valid if `array` is not `null`.
- We access the array's elements (`array[i]`): again, `array` must not be `null`, and the index `i` must be within range.
- We perform a division at the end: we need be make sure `array.length` does not equal 0.

We can enforce the constraints on `array` (i.e. that it does not equal `null` and that its length is greater than zero) by using preconditions. Deferring the responsibility to the client is perfectly okay: what would the average of a `null` or empty array be anyway?

To prevent an index-out-of-bounds error, we make use of a loop invariant. A loop invariant must be true when execution reaches the loop, and must be preserved across iterations (i.e. assuming the loop invariant holds, it must still hold after the execution of the loop body). In our case, we need $0 \leq i \leq \text{array.length}$ as loop invariant:

- `i` is initialized to 0, and `array.length` is always non-negative, so the invariant holds when execution arrives at the loop.
- when entering the loop, we know that both the loop invariant $0 \leq i \leq \text{array.length}$ and the loop condition `i != array.length` are true, which combine to $0 \leq i < \text{array.length}$, which is exactly what we need.
- upon leaving the loop, the loop invariant still holds together with the negation of the loop condition, giving us `i == array.length`.

We annotate the code from Figure 2 with this new information in the form of assertions (Figure 3). We take a few liberties syntax-wise, but the intended meaning should be clear. Verification of this piece of code consists of making sure that none of the assertions would fail, no matter what state we execute the function in. This guarantees that no runtime errors (at least none of those we check for) will occur. Code indicated in blue can be inferred by the Java compiler itself² while red code needs to be written by a programmer.

Thus, Figure 3 makes explicit what conditions need to be true at different points in the program (the assertions), as well as what assumptions can be made about the values of certain variables (the preconditions). We now need to translate this code to our intermediate language.

2.2 Commands and their behaviour

A program written in the intermediate language comprises two parts:

The logical part defines constants and functions symbols which we'll be able to refer to in the imperative part. Axioms can also be defined to describe facts about these constants and

²These assertions are actually preconditions of other operations or functions. E.g. one could see the member access as a binary operator `.` which demands its left operand not to be `null`.

```
1 public static double average(double[] array)
2 {
3     double total = 0;
4     int i = 0;
5     while ( i != array.length )
6     {
7         total += array[i];
8         ++i;
9     }
10    return total / array.length;
11 }
```

Figure 2: Java code to be verified

```
1 public static double average(double[] array)
2     requires array != null && array.length != 0
3     ensures true
4 {
5     double total = 0;
6     int i = 0;
7     while ( assert array != null; i != array.length )
8         invariant 0 <= i <= array.length
9     {
10        assert array != null && 0 <= i < array.length;
11        total += array[i];
12        ++i;
13    }
14    assert array != null && array.length != 0;
15    return total / array.length;
16 }
```

Figure 3: Annotated Java code

functions. Axiomatizations for integers, finite maps, booleans, names, etc. are readily available. For example, one could define the function `arraylength` from references to integers, accompanied by an axiom stating that `length` does always return a nonnegative integer. We do not delve any further in this part, more information can be found in [7, 8].

The imperative part consists of global variables which take values in the mathematical structure axiomatized by the logical part and can be used to represent (part of) the program state, and a number of procedures, each of which needs to be verified separately.

Our main interest in this section is the language used to specify the procedure bodies. We will also define what it means to verify a procedure and discuss the approach we take to verify a procedure, our ultimate goal being to prove that the approach is sound. From now on, we will use the term “intermediate language” to refer specifically to the language used to define the procedure bodies.

As explained previously, code written in the intermediate language does not represent computations, but execution paths populated with conditions we expect to be true. Added to this are support for state (in the form of an assign command) and the ability to indicate what assumptions can be made about the variables’ values. The intermediate language provides six commands³ (Figure 4):

- **assert** e states that we wish the expression e to evaluate to **true**. It is the verifier’s responsibility to ensure that all e appearing in assertions evaluate to true, and give us an error message if that is not the case. Assertion commands can be used to enforce that preconditions hold, that no divisions by zero occur, etc.
- **assume** e tells the verifier that e can be assumed to be true. For example, in Java, it is guaranteed that `this` does not equal `null`. To prevent the verifier from complaining about possible `null` dereferences wherever we use `this.someField`, we can just write `assume this != null`. A special case of this is **assert false**, which blocks execution unconditionally so that no further errors can occur. We will use this later to translate loops.
- Assignment, written $x := e$, assigns the result of evaluating e to x .
- Sequencing, written $c_1; c_2$, allows us to combine programs into bigger programs.
- Nondeterministic choice, written $c_1 \parallel c_2$, states that execution can proceed with either c_1 or c_2 . For example, when dealing with an `if` statement, we generally cannot know beforehand (at verification time) which branch will be executed. This problem is solved by just incorporating both branches using the choice command, telling the verifier to deal with both possibilities.
- **skip** is a no-op, added to simplify the definition of the operational semantics (see later).

We now define the behaviour of programs written in the intermediate language. We distinguish two kinds of states:

- An in-progress state, written $\langle c, \mu \rangle$, which consists of a command c and a store μ . An in-progress state corresponds to successful execution.
- A failure state, written `failure` μ , consisting of a store μ . Failure states are the result of failed assertions and hence must be avoided.

We define a store as a total mapping⁴ from identifiers to values (Figure 4), i.e. it holds the values stored in variables. We can also define an expression (as used in the **assert**, **assume**

³Later on we will introduce two new kinds of commands, i.e. the versioned commands and the passified commands. In the future, to indicate more clearly which commands we are talking about, we will also refer to the set of commands defined here as *regular* commands.

⁴Since it is *total* mapping, there are no unbound identifiers and thus all possible variables have values assigned to them. The reason for this oddity will be explained shortly.

```

1 Parameters (value : Set).
2           (T      : value)
3           (F      : value).

4 Definition id      := nat.
5 Definition store := id -> value.
6 Definition expr   := store -> value.

7 Inductive command : Set :=
8 | cAssert   : expr -> command
9 | cAssume   : expr -> command
10 | cAssign   : id -> expr -> command
11 | cSequence : command -> command -> command
12 | cSkip     : command
13 | cChoice   : command -> command -> command.

```

Figure 4: Definition of commands in Coq

and assignment commands) as a total function mapping stores to values. Next, we define the operational semantics (Figure 5):

- **ASSERTTRUE**: given a store μ and an expression e , if e evaluated in μ equals **true**, the state $\langle \mathbf{assert} \ e, \mu \rangle$ is reduced to $\langle \mathbf{skip}, \mu \rangle$, meaning that execution can proceed normally.
- **ASSERTFALSE**: given a store μ and an expression e , if e evaluated in μ does not equal **true**, the state $\langle \mathbf{assert} \ e, \mu \rangle$ is reduced to $\mathbf{failure} \ \mu$, meaning that execution has failed.
- **ASSUME**: given a store μ and an expression e , if e evaluates to **true** in μ , $\langle \mathbf{assume} \ e, \mu \rangle$ reduces to $\langle \mathbf{skip}, \mu \rangle$. However, for the case that e does not evaluate to **true** we do not define a rule and execution will get stuck. We will explain this aspect of the operational semantics shortly.
- **SEQUENCE**: a sequence $c_1; c_2$ is reduced by reducing its left component c_1 as long as possible. Thus, $\langle c_1; c_2, \mu \rangle$ reduces to $\langle c'_1; c_2, \mu' \rangle$ if $\langle c_1, \mu \rangle$ reduces to $\langle c'_1, \mu' \rangle$.
- **SEQUENCESKIP**: if execution reduces the left component of a sequence to **skip**, the next step is to reduce the entire sequence to its right component. In other words, $\langle \mathbf{skip}; c_2, \mu \rangle$ reduces to $\langle c_2, \mu \rangle$.
- **SEQUENCEFAIL**: if reducing the left component of a sequence leads to failure, the entire sequence will also reduce to failure: if $\langle c_1, \mu \rangle \longrightarrow \mathbf{failure} \ \mu'$, then $\langle c_1; c_2, \mu \rangle \longrightarrow \mathbf{failure} \ \mu'$.
- **ASSIGN**: the assignment command $x := e$ changes the store binding for x to the value resulting from evaluating e in the current store.
- **CHOICELEFT**: as mentioned previously, the choice command $c_1 \parallel c_2$ represents nondeterministic choice. This rule lets execution proceed with the left component.
- **CHOICERIGHT**: this rule lets execution continue with the right component of a choice command.

The operational semantics are non-deterministic, meaning there are states for which there is more than one possible reduction, more specifically, these are states where the next command to be executed is a choice (either directly, or nested within a sequence). Thus, instead of having a linear chain of states σ_i where $\sigma_i \longrightarrow \sigma_{i+1}$, we have to deal with an entire execution tree with the following properties:

$$\frac{e(\mu) = \mathbf{true}}{\langle \mathbf{assert} \ e, \mu \rangle \longrightarrow \langle \mathbf{skip}, \mu \rangle} \text{ ASSERTTRUE}$$

$$\frac{e(\mu) \neq \mathbf{true}}{\langle \mathbf{assert} \ e, \mu \rangle \longrightarrow \mathbf{failure} \ \mu} \text{ ASSERTFALSE}$$

$$\frac{e(\mu) = \mathbf{true}}{\langle \mathbf{assume} \ e, \mu \rangle \longrightarrow \langle \mathbf{skip}, \mu \rangle} \text{ ASSUME}$$

$$\frac{\langle c_1, \mu \rangle \longrightarrow \langle c'_1, \mu' \rangle}{\langle c_1; c_2, \mu \rangle \longrightarrow \langle c'_1; c_2, \mu' \rangle} \text{ SEQUENCE}$$

$$\frac{\langle c_1, \mu \rangle \longrightarrow \mathbf{failure} \ \mu'}{\langle c_1; c_2, \mu \rangle \longrightarrow \mathbf{failure} \ \mu'} \text{ SEQUENCEFAIL}$$

$$\frac{}{\langle \mathbf{skip}; c, \mu \rangle \longrightarrow \langle c, \mu \rangle} \text{ SEQUENCESKIP}$$

$$\frac{}{\langle x := e, \mu \rangle \longrightarrow \langle \mathbf{skip}, \mu[x \mapsto e(\mu)] \rangle} \text{ ASSIGN}$$

$$\frac{}{\langle c_1 \parallel c_2, \mu \rangle \longrightarrow \langle c_1, \mu \rangle} \text{ CHOICELEFT}$$

$$\frac{}{\langle c_1 \parallel c_2, \mu \rangle \longrightarrow \langle c_2, \mu \rangle} \text{ CHOICERIGHT}$$

Figure 5: Operational semantics

```

1 Inductive step : state -> state -> Prop :=
2 | stepAssertT : forall e mu,
3   e mu = T ->
4   step (ip (cAssert e) mu) (ip cSkip mu)
5 | stepAssertF : forall e mu,
6   e mu <> T ->
7   step (ip (cAssert e) mu) (failure mu)
8 | stepAssume  : forall e mu,
9   e mu = T ->
10  step (ip (cAssume e) mu) (ip cSkip mu)
11 | stepSeq     : forall c1 c1' c2 mu mu',
12  step (ip c1 mu) (ip c1' mu') ->
13  step (ip (cSequence c1 c2) mu)
14  (ip (cSequence c1' c2) mu')
15 | stepSeqSkip : forall c2 mu,
16  step (ip (cSequence cSkip c2) mu) (ip c2 mu)
17 | stepSeqFail : forall c1 c2 mu mu',
18  step (ip c1 mu) (failure mu') ->
19  step (ip (cSequence c1 c2) mu) (failure mu')
20 | stepAssign  : forall x e mu,
21  step (ip (cAssign x e) mu)
22  (ip cSkip (update_store mu x (e mu)))
23 | stepChoiceL : forall c1 c2 mu,
24  step (ip (cChoice c1 c2) mu) (ip c1 mu)
25 | stepChoiceR : forall c1 c2 mu,
26  step (ip (cChoice c1 c2) mu) (ip c2 mu).

```

Figure 6: Operational semantics in Coq

- every node in the tree corresponds to a state;
- a state σ' is a child of another state σ iff $\sigma \longrightarrow \sigma'$;
- states have either zero, one or two children:
 - zero children (i.e. the leaves of the tree) indicates a stuck state;
 - two children means a choice command was encountered at that point during execution.
- a path from the root to a leaf represents a possible execution path;
- our goal is to prevent the existence of leaves with failure states.

The meaning of the **assume** command becomes clearer when interpreting program execution as a tree: this command can be used to prune a complete branch from the tree, effectively preventing execution along that specific execution path, so that no failure states can be encountered. In other words, the **assume** command allows us to express our wish to ignore certain parts of the execution tree.

One important question remains: what state does belong in the root of the tree? It is clear that it must be an in-progress state, whose command component is the original program. But what store should we choose? The answer to this question is simply that we should consider every possible store. Every possible combination of variable/value bindings must be considered. This means that instead of having a single execution tree, we have an infinite number of them, one for each possible store. None of these trees is allowed to contain a leaf with a failure state, for this would mean that the program fails for certain variable values. Since not every store represents a possible state of the original program (e.g. **this** cannot be bound to **null** in Java, variables of type **int** can only contain integer values, etc.), we need to be able to select a subset of stores where these language-guaranteed facts hold. This is achieved by using **assume** commands, placed at the beginning of the program, so they can cut off the tree close to the root if the store contains invalid bindings. These **assumes** correspond to a program's preconditions.

2.3 Example (cont'd)

We return to our previous example (Figure 2 and Figure 3). We need to translate this code into the intermediate language. For the sake of clarity, we temporarily introduce a new command **havoc** x which destroys all information about the variable x . We will explain later on how to get rid of this new command. Figure 7 shows a possible translation of the average-function.

First of all, we assume the logical part defines the **arraylength** function, of which is guaranteed that its result is nonnegative. Lines 1–2 (Figure 7) correspond to the method's preconditions: we state directly at the beginning we are only interested in considering the cases where **array** is not bound to **null** and where **array** refers to a non-empty array. Lines 3–4 correspond to the variable initializations.

Next, we arrive at the **while** loop. We cannot know beforehand how many times the loop body will be executed, and we certainly cannot encode all cases (0 iterations, 1 iterations, 2, 3, ...) To solve this problem, we use a loop invariant, as explained previously:

1. We verify an arbitrary loop iteration: assuming the invariant holds at the beginning of an iteration we need to prove that it holds at the end of it. We can also assume that the loop condition evaluates to true at the start of an iteration.
2. After the loop, we continue in a state where the loop condition does not hold and where the loop invariant does.

Let us begin by taking a look at how we would generally translate a loop.

while (C) **invariant** I { *body* } *rest*

```

1  assume array != null;
2  assume arraylength(array) != 0;
3  total := 0;
4  i := 0;
5  assert array != null;
6  assert 0 <= i <= arraylength(array);
7  havoc total;
8  havoc i;
9  assume 0 <= i <= arraylength(array);
10 assert array != null;
11 (
12   assume i != arraylength(array);
13   assert array != null && 0 <= i <= arraylength(array);
14   total := total + lookup(array, i);
15   i := i + 1;
16   assert 0 <= i <= length(array);
17   assume false
18 []
19   assume !(i != arraylength(array))
20 );
21 assert array != null && arraylength(array) != 0;
22 result := total / arraylength(array)

```

Figure 7: Translation to the intermediate language (with havoc)

We look at which variables the *body* modifies. Let us call the set of these variables Δ . We do not make any assumptions about the values of the variables in this set and thus destroy any information we have about them. To reconstruct this lost information, we can use the loop invariant. The translation of an arbitrary loop iteration is as follows:

$$\mathbf{havoc} \Delta; \mathbf{assume} C; \mathbf{assume} I; \overline{\mathit{body}}; \mathbf{assert} I$$

where $\overline{\mathit{body}}$ represents the translation of *body*. The second part of the translation looks like

$$\mathbf{havoc} \Delta; \mathbf{assume} \neg C; \mathbf{assume} I; \overline{\mathit{rest}}$$

We combine both using the choice command (and some factoring out of common code):

$$\mathbf{havoc} \Delta; \mathbf{assume} I; (\mathbf{assume} C; \overline{\mathit{body}}; \mathbf{assert} I; \mathbf{assume} \mathbf{false} \parallel \mathbf{assume} \neg C); \overline{\mathit{rest}}$$

We have left out a few details: the loop condition and invariant need to be evaluatable without errors, meaning we need to add a few extra assertions to make sure this is the case.

We return to our `average` function:

- Line 5 ensures we can evaluate the invariant (`array` must not be `null` to be able to fetch the `length` field).
- Line 6 makes sure that upon arriving at the loop, the invariant holds.
- Lines 7–8 destroy whatever information we have about `total` and `i`, the two variables modified by the loop body.
- Line 9 reconstructs the information lost by the `havoc` commands by stating the loop invariant can be assumed to be `true`.

- Line 10 makes sure the loop condition is evaluatable without errors: since the condition accesses a field of `array`, `array` must not be `null`.
- Line 12-17 consider an arbitrary loop iteration, line 19 the exiting of the loop.
- Line 12 states the loop condition holds (which it does, since otherwise the loop body would not be executed).
- Line 13 contains the necessary conditions to evaluate `array[i]`.
- Lines 14–15 correspond directly to the two assignments in the Java-code.
- Line 16 asserts the loop invariant must hold at the end of the loop iteration.
- Line 17 makes an end to the execution path: we do not want it to proceed with the rest of the function.
- Line 19 represents exiting the loop: we can assume the loop condition is false.

We are almost done: only the `return` statement remains. Line 21 makes sure it is possible to evaluate the return-expression, and line 22 performs the necessary computation and stores the result in `result`, a special variable which can be used to simulate `return` statements.

We still need to get rid of the `havoc` statements. Fortunately, this is very easy to achieve. At the beginning of an intermediate language, we know that the store contains bindings for all variables, and that all possible bindings are considered. Thus, initially we have not one bit of information about the variable values. This is exactly what we need: `havoc` needs to remove all known information about a certain variable, meaning we can just replace it by a fresh variable:

$$pre; \mathbf{havoc} \ x; post$$

is equivalent with

$$pre; post[x'/x]$$

where x' is a fresh variable (i.e. not mentioned in either pre or $post$) and $[x'/x]$ stands for substituting x' for x . The final translation is shown in Figure 8.

```
1  assume array != null;
2  assume arraylength(array) != 0;
3  total := 0;
4  i := 0;
5  assert array != null;
6  assert 0 <= i <= arraylength(array);
7  assume 0 <= i' <= arraylength(array);
8  assert array != null;
9  (
10     assume i' != arraylength(array);
11     assert array != null && 0 <= i' <= arraylength(array);
12     total' := total' + lookup(array, i');
13     i' := i' + 1;
14     assert 0 <= i' <= length(array);
15     assume false
16 []
17     assume !(i' != arraylength(array))
18 );
19 assert array != null && arraylength(array) != 0;
20 result := total' / arraylength(array)
```

Figure 8: Translation to the intermediate language (without havoc)

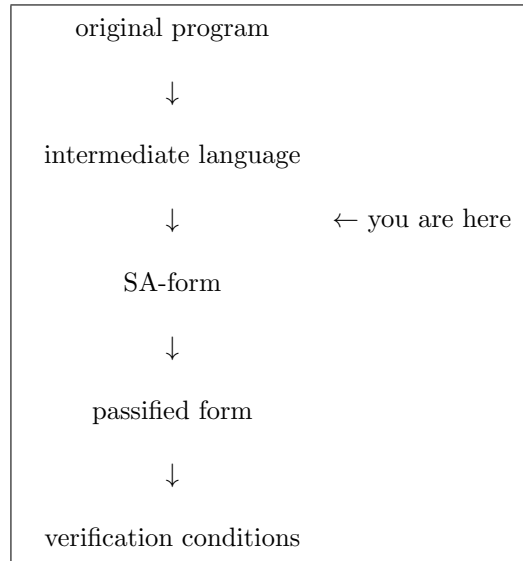


Figure 9: Steps in the verification algorithm

3 Transformation into single assignment form

In this section, we focus on the first transformation, i.e. to single assignment form [6, 4]. First, we define what this single assignment form (SA) is, and sketch how one proceeds to transform an arbitrary IL program to SA. Next, we give a formal definition of the algorithm. We end with the proof of an important property of the SA-transformation, namely that an IL-program and its SA-translation behave “similarly”.

3.1 Single Assignment Form

We say a program written in the IL is in single assignment form if, during each execution path, all reads from the same variable yield the same value. In other words, during an arbitrary execution,

- each variable is at most assigned to once;
- if a variable gets assigned to, it is not read from prior to that point.

Note that it is allowed to have assignments to the same variable in the code, as long as they appear on different execution paths. Thus,

$$x := 1; (y := 5 \parallel y := 6); z := 19$$

is in SA-form. The second condition may need a bit of explanation. Remember that initially, the store assigns arbitrary values to all variables. If an expression (whether in an **assert**, **assume** or in the right side of an assignment) refers to the variable, and that same variable is assigned to at later time (even if only once), the variable seems to change values over time, which we cannot allow. For example,

$$\mathbf{assume} \ x \geq 0; x := 5$$

is *not* in SA-form, even though x is assigned to at most once.

Our goal is to define an algorithm which turns an arbitrary IL-program into an equivalent SA-form. However, we will not prove that the result produced by this algorithm is indeed SA, as it is not this property that actually interests us. What counts is that failure of the original program implies that its SA-translation will also fail. We will delve into this aspect in Section 3.4.

3.2 Sketch of the algorithm

Let us first consider the translation of an IL-program *without* choice commands. This is pretty straightforward: each time a variable x is assigned to, we replace it with an assignment to a fresh variable x' , and we replace all references to x from that point on by references to x' . For example,

$$\mathbf{assert} \ x \leq 6; x := x + 8; \mathbf{assert} \ x \leq 14; x := x + x$$

becomes

$$\mathbf{assert} \ x \leq 6; x' := x + 8; \mathbf{assert} \ x' \leq 14; x'' := x' + x'$$

Choice poses a problem however. Consider the following IL-code:

$$x := 5; y := 10; (x := y + 1 \parallel y := x - 1); \mathbf{assert} \ x - y = 1 \tag{1}$$

There are two possible execution paths for each possible initial store μ :

- $\langle c, \mu \rangle \longrightarrow^* \langle \mathbf{skip}, \mu[x \mapsto 11][y \mapsto 10] \rangle$
- $\langle c, \mu \rangle \longrightarrow^* \langle \mathbf{skip}, \mu[x \mapsto 10][y \mapsto 9] \rangle$

where c stands for the original program (1). We now need to translate it to SA-form. A first attempt would be

$$x := 5; y := 10; (x' := y + 1 \parallel y' := x - 1); \mathbf{assert} \ x' - y' = 1$$

but this is clearly not what we need: either x' or y' was not assigned to prior to the final assertion and hence its value could be anything. While the original program will never fail, this SA-translation certainly does.

The problem resides in the fact that one execution path needs x' and y to be compared, while the other should compare x and y' , meaning there should, in a way, be two different assertions for each choice branch. We can achieve this by using some sort of distributivity:

$$(c_1 \parallel c_2); c_3 \quad \text{behaves similarly to} \quad (c_1; c_3) \parallel (c_2; c_3)$$

Applying this to the original program (1) gives us

$$x := 5; y := 10; (x := y + 1; \mathbf{assert} \ x - y = 1 \parallel y := x - 1; \mathbf{assert} \ x - y = 1)$$

and even further

$$(x := 5; y := 10; x := y + 1; \mathbf{assert} \ x - y = 1) \parallel (x := 5; y := 10; y := x - 1; \mathbf{assert} \ x - y = 1)$$

This way, we expand a single program into two different programs, which can be verified separately. However, each choice command would double the number of programs to be verified, leading to an exponential blowup, making this approach unacceptable. A better solution would be to “synchronize” both branches of a choice command during the SA-translation.

$$x := 5; y := 10; (x' := y + 1; y' := y \parallel y' := x - 1; x' := x); \mathbf{assert} \ x' - y' = 1$$

This is also the approach we will use: during the translation to SA form, both choice branches will be extended by a series of synchronization commands to make sure the rest of the program can refer to a single specific “version” of the variables.

3.3 Formal definition of the SA-transformation

As explained previously, we need to replace each variable on the left side of an assignment by a fresh one, and replace every occurrence of the “old” variable by the “new” one from that point on. Generating fresh variables in an algorithm and proving properties involving them can become quite complex: one needs to collect all variables in a set, create a new one not in that set, and drag that information along in all proofs. We have tried to simplify this situation by making use of the inherent structure present in the problem at hand: our solution consists of adding a version number to each variable, starting with 0. Generating a fresh variable then becomes a simple matter of incrementing this version number. The translation of our program

$$x := 5; y := 10; (x := y + 1 \parallel y := x - 1); \mathbf{assert} \ x - y = 1$$

then becomes

$$x_0 := 5; y_0 := 10; (x_1 := y_0 + 1; y_1 := y_0 \parallel y_1 := x_0 - 1; x_1 := x_0); \mathbf{assert} \ x_1 - y_1 = 1$$

Since we have now versioned variables, we also need versioned expressions, versioned commands, versioned operational semantics, ... We introduce the concept of a *version map* which maps variables to their version numbers and which we can use to translate unversioned objects to their versioned counterpart.

- A versioned identifier (**vid**) consists of a pair whose first component is an (unversioned) identifier and second component a natural number, representing a version. For this text, we also introduce a shorthand notation:

$$x_n \equiv (x, n)$$

- A versioned store (**vstore**) is a total function mapping versioned identifiers to values.
- A versioned expression (**vexpr**) is a total function mapping versioned stores to values.
- The versioned commands are identical to the original ones, except for the fact that **assume**, **assert** and assign take versioned expressions as their arguments, and that a versioned variable appears on the left side of assignment commands.
- Versioned states, written $\langle c_v, \mu_v \rangle_v$ and $\text{failure}_v \ \mu_v$, take versioned commands and versioned stores.
- Versioned operational semantics are identical to the original ones, with the exception they operate on versioned states (Figure 12).

We also need to be able to translate an unversioned expression to its versioned equivalent.

Definition 1. We define the translation of an unversioned expression e to a versioned expression with respect to a version map ν as (see `version_expr` in Figure 11):

$$e_v = \lambda \mu_v. e(\lambda x. \mu_v(x, \nu(x)))$$

The function $\text{SA}_\nu(c)$ which translates the command c to SA-form with respect to version map ν returns two values: the SA-transformation of c , and a version map containing the updated versions for all variables.

Definition 2 (SA transformation). (**transform_sa**) We define the SA transformation of an IL-program with respect to version map ν , written $\text{SA}_\nu(c)$, as follows:

- $\text{SA}_\nu(\mathbf{assert} \ e) = (\mathbf{assert} \ e_v, \nu)$ where e_v is the versioned equivalent of e with respect to ν .
- $\text{SA}_\nu(\mathbf{assume} \ e) = (\mathbf{assume} \ e_v, \nu)$ where e_v is the versioned equivalent of e with respect to ν .

$$\frac{e_v(\mu_v) = \mathbf{true}}{\langle \mathbf{assert} \ e_v, \mu_v \rangle_v \longrightarrow_v \langle \mathbf{skip}, \mu_v \rangle_v} \text{V-ASSERTTRUE}$$

$$\frac{e_v(\mu_v) \neq \mathbf{true}}{\langle \mathbf{assert} \ e_v, \mu_v \rangle_v \longrightarrow_v \mathbf{failure}_v \ \mu_v} \text{V-ASSERTFALSE}$$

$$\frac{e_v(\mu_v) = \mathbf{true}}{\langle \mathbf{assume} \ e_v, \mu_v \rangle_v \longrightarrow_v \langle \mathbf{skip}, \mu_v \rangle_v} \text{V-ASSUME}$$

$$\frac{\langle c_{v,1}, \mu_v \rangle_v \longrightarrow_v \langle c'_{v,1}, \mu'_v \rangle_v}{\langle c_{v,1}; c_{v,2}, \mu_v \rangle_v \longrightarrow_v \langle c'_{v,1}; c_{v,2}, \mu'_v \rangle_v} \text{V-SEQUENCE}$$

$$\frac{\langle c_{v,1}, \mu_v \rangle_v \longrightarrow_v \mathbf{failure}_v \ \mu'_v}{\langle c_{v,1}; c_{v,2}, \mu_v \rangle_v \longrightarrow_v \mathbf{failure}_v \ \mu'_v} \text{V-SEQUENCEFAIL}$$

$$\frac{}{\langle \mathbf{skip}; c, \mu_v \rangle_v \longrightarrow_v \langle c, \mu_v \rangle_v} \text{V-SEQUENCESKIP}$$

$$\frac{}{\langle x_i := e_v, \mu_v \rangle_v \longrightarrow_v \langle \mathbf{skip}, \mu_v[x_i \mapsto e_v(\mu)] \rangle_v} \text{V-ASSIGN}$$

$$\frac{}{\langle c_{v,1} \parallel c_{v,2}, \mu_v \rangle_v \longrightarrow_v \langle c_{v,1}, \mu_v \rangle_v} \text{V-CHOICELEFT}$$

$$\frac{}{\langle c_{v,1} \parallel c_{v,2}, \mu_v \rangle_v \longrightarrow_v \langle c_{v,2}, \mu_v \rangle_v} \text{V-CHOICERIGHT}$$

$$\frac{}{\langle c_v, \mu \rangle_v \longrightarrow_v^* \langle c_v, \mu \rangle_v} \text{V}^*\text{-REFLEXIVITY}$$

$$\frac{\langle c_v, \mu \rangle_v \longrightarrow_v \langle c'_v, \mu' \rangle_v \longrightarrow_v^* \langle c''_v, \mu'' \rangle_v}{\langle c_v, \mu \rangle_v \longrightarrow_v^* \langle c''_v, \mu'' \rangle_v} \text{V}^*\text{-STEP}$$

Figure 10: Versioned operational semantics

- $\text{SA}_\nu(\mathbf{skip}) = (\mathbf{skip}, \nu)$.
- $\text{SA}_\nu(x := e) = (x_{\nu(x)+1} := e_v, \nu[x \mapsto \nu(x) + 1])$ where e_v is the versioned equivalent of e with respect to ν .
- $\text{SA}_\nu(c_1; c_2) = (c'_1; c'_2, \nu'')$ where $(c'_1, \nu') = \text{SA}_\nu(c_1)$ and $(c'_2, \nu'') = \text{SA}_{\nu'}(c_2)$. We quickly draw the reader's attention to the quickly overlooked fact that the SA-transformation of c_2 needs to be done with respect to ν' , not ν .
- $\text{SA}_\nu(c_1 \parallel c_2) = (c'_1; \delta_1 \parallel c'_2; \delta_2, \nu')$ where
 - $(c'_1, \nu_1) = \text{SA}_\nu(c_1)$
 - $(c'_2, \nu_2) = \text{SA}_\nu(c_2)$

```

1 Definition vid      := (id * nat)%type.
2 Definition vstore  := vid -> value.
3 Definition vexpr   := vstore -> value.
4 Definition vmap    := id -> nat.

5 Inductive vstate : Set :=
6 | vip           : vcommand -> vstore -> vstate
7 | vfailure     : vstore -> vstate.

8 Inductive vcommand : Set :=
9 | vcAssert     : vexpr -> vcommand
10 | vcAssume    : vexpr -> vcommand
11 | vcAssign    : vid -> vexpr -> vcommand
12 | vcSequence  : vcommand -> vcommand -> vcommand
13 | vcSkip      : vcommand
14 | vcChoice    : vcommand -> vcommand -> vcommand.

15 Definition version_expr (e : expr) (v : vmap) : vexpr :=
16   fun (vmu : vstore) => e (fun x => vmu (x, v x)).

```

Figure 11: Versioned variants

- $\nu' = \text{join}(\nu_1, \nu_2)$
- δ_1 and δ_2 are synchronization commands for c_2 's and c_1 's targets, from ν_1 and ν_2 to ν' , respectively.

We clearly need to elaborate a bit on the last case. The problem with choice commands is that both branches lead to different variable versionings, i.e. $\text{SA}_\nu(c_1)$ and $\text{SA}_\nu(c_2)$ return different version maps. Let us go back to our example:

$$x := 5; y := 10; (x := y + 1 \parallel y := x - 1); \text{assert } x - y = 1$$

For example, with a version map ν where $\nu(x) = 3$ and $\nu(y) = 8$, we get as translation for both branches:

$$\begin{aligned} \text{SA}_\nu(x := y + 1) &= (x_4 := y_8 + 1, \nu[x \mapsto 4]) \\ \text{SA}_\nu(y := x - 1) &= (y_9 := x_3 - 1, \nu[y \mapsto 9]) \end{aligned}$$

We need to merge these two version maps into one, the obvious choice being

$$\nu[x \mapsto 4][y \mapsto 9]$$

Generally, to “join” two version maps, we take the maximum version for each variable. Remember a version map is a total function from identifiers to natural numbers, so the join-operation becomes (see also Figure 13)

$$\text{join}(\nu_1, \nu_2) = \lambda x. \max(\nu_1(x), \nu_2(x))$$

So, we need two synchronization commands (δ_1 and δ_2) which turn both $\nu[x \mapsto 4]$ and $\nu[y \mapsto 9]$ into $\nu[x \mapsto 4][y \mapsto 9]$. These will obviously be a sequence of assignment commands, which copy the value from the old to the new version:

$$\begin{aligned} \delta_1 &= y_9 := y_8 \\ \delta_2 &= x_4 := x_3 \end{aligned}$$

To find out exactly which variables need such a “version update” for one branch, we look at which variables were assigned to in the other branch by simply scanning the commands and collecting the variables appearing on the left side of assignments. This is sufficient thanks to the fact that aliasing is not made possible by the intermediate language. We now define these steps more formally.

```

1 Inductive vstep : vstate -> vstate -> Prop :=
2 | vstepAssertT : forall e vmu,
3   e vmu = T ->
4   vstep (vip (vcAssert e) vmu) (vip vcSkip vmu)
5 | vstepAssertF : forall e vmu,
6   e vmu <> T ->
7   vstep (vip (vcAssert e) vmu) (vfailure vmu)
8 | vstepAssume  : forall e vmu,
9   e vmu = T ->
10  vstep (vip (vcAssume e) vmu) (vip vcSkip vmu)
11 | vstepSeq     : forall c1 c1' c2 vmu vmu',
12   vstep (vip c1 vmu) (vip c1' vmu') ->
13   vstep (vip (vcSequence c1 c2) vmu)
14   (vip (vcSequence c1' c2) vmu')
15 | vstepSeqSkip : forall c2 vmu,
16   vstep (vip (vcSequence vcSkip c2) vmu) (vip c2 vmu)
17 | vstepSeqFail : forall c1 c2 vmu vmu',
18   vstep (vip c1 vmu) (vfailure vmu') ->
19   vstep (vip (vcSequence c1 c2) vmu) (vfailure vmu')
20 | vstepAssign  : forall x e vmu,
21   vstep (vip (vcAssign x e) vmu)
22   (vip vcSkip (update_vstore vmu x (e vmu)))
23 | vstepChoiceL : forall c1 c2 vmu,
24   vstep (vip (vcChoice c1 c2) vmu) (vip c1 vmu)
25 | vstepChoiceR : forall c1 c2 vmu,
26   vstep (vip (vcChoice c1 c2) vmu) (vip c2 vmu).

```

Figure 12: Versioned operational semantics in Coq

```

1 Definition join (v1 v2 : vmap) :=
2   fun x => max (v1 x) (v2 x).

```

Figure 13: Joining two version maps in Coq

```

1 Fixpoint targets (c : command) : IdSet.t :=
2   match c with
3     | cAssert _ => IdSet.empty
4     | cAssume _ => IdSet.empty
5     | cAssign x _ => IdSet.singleton x
6     | cSequence c1 c2 =>
7       IdSet.union (targets c1) (targets c2)
8     | cChoice c1 c2 =>
9       IdSet.union (targets c1) (targets c2)
10    | cSkip => IdSet.empty
11  end.

```

Figure 14: Syntactic targets in Coq

Definition 3 (Syntactic targets). (**targets**) *The syntactic targets (i.e. the variables assigned to) in a program c are found by the following algorithm (see also Figure 14 for the Coq version):*

$$\begin{aligned}
\text{targets}(\mathbf{assert} \ e) &= \emptyset \\
\text{targets}(\mathbf{assume} \ e) &= \emptyset \\
\text{targets}(x := e) &= \{x\} \\
\text{targets}(c_1; c_2) &= \text{targets}(c_1) \cup \text{targets}(c_2) \\
\text{targets}(c_1 \parallel c_2) &= \text{targets}(c_1) \cup \text{targets}(c_2) \\
\text{targets}(\mathbf{skip}) &= \emptyset
\end{aligned}$$

The generation of synchronization commands (the `sync_vcommand` function in Coq) takes three arguments: the set of variables to be synchronized, and the old and new version map. Given an old version map ν and a new version map ν' , for each variable x in the set, the following assignment needs to be generated:

$$x_{\nu'(x)} := x_{\nu(x)}$$

Since the right hand side must be an expression, i.e. a total function mapping identifiers to values, we need to rewrite it as

$$x_{\nu'(x)} := \lambda \mu_{\nu} . \mu_{\nu}(x_{\nu(x)})$$

We chain these assignments together using sequence commands. The full Coq code can be found in Figure 15.

Definition 4. *Synchronization command (`sync_vcommand`) The synchronization command $\delta = \text{sync}(I, \nu, \nu')$, with ν and ν' version maps and I a finite set of identifiers a, b, c, \dots is the command*

$$a_{\nu'(a)} := a_{\nu(a)}; b_{\nu'(b)} := b_{\nu(b)}; c_{\nu'(c)} := c_{\nu(c)}; \dots; \mathbf{skip}$$

3.4 Theorems and proofs

We have now defined how the SA-transformation works and want to prove a certain property of it. Which property this is becomes clear when we zoom out and take a look at the bigger picture.

The SA-transformation is the first of a series of steps:

- The original IL-program c is transformed into its SA-form c_{sa} .
- The SA-form c_{sa} is passified, giving c_{p} .

```

1 Definition copy_vcnd (x : id) (n m : nat) :=
2   vcAssign (x, m) (fun (vmu : vstore) => vmu (x, n)).

3 Definition insert_copy_vcnd v v' x c :=
4   if decidable_eq_id (v x) (v' x)
5   then c
6   else (vcSequence (copy_vcnd x (v x) (v' x)) c).

7 Definition sync_vcommand (ids : IdSet.t) (v v' : vmap) :=
8   (fset_foldr vcommand (insert_copy_vcnd v v') ids vcSkip).

```

Figure 15: Generation of synchronization commands in Coq

```

1 Fixpoint
2   transform_sa (c : command)
3     (v : vmap)      : (vcommand * vmap) :=
4   match c with
5   | cAssert e =>
6     (vcAssert (version_expr e v), v)
7
8   | cAssume e =>
9     (vcAssume (version_expr e v), v)
10
11  | cAssign x e =>
12    (vcAssign (x, S (v x)) (version_expr e v), inc v x)
13
14  | cSequence c1 c2 =>
15    let (c1', v' ) := transform_sa c1 v in
16    let (c2', v'' ) := transform_sa c2 v' in
17    (vcSequence c1' c2', v'')
18
19  | cSkip => (vcSkip, v)
20
21  | cChoice c1 c2 =>
22    let (c1', v1') := transform_sa c1 v in
23    let (c2', v2') := transform_sa c2 v in
24    let t1 := targets c1 in
25    let t2 := targets c2 in
26    let v' := join v1' v2' in
27    let t := IdSet.union t1 t2 in
28    let d1 := sync_vcommand t v1' v' in
29    let d2 := sync_vcommand t v2' v' in
30    (vcChoice (vcSequence c1' d1)
31              (vcSequence c2' d2), v')
32
33  end.

```

Figure 16: SA-transformation algorithm in Coq

- From the passified form c_p we derive the verification conditions.

We want the validity of the verification conditions to imply that the original program does not fail:

$$\text{VC}(c_p) \Rightarrow \forall \mu \bullet \neg(\exists \mu' \bullet \langle c, \mu \rangle \longrightarrow^* \text{failure } \mu')$$

We can achieve this by splitting the implication up⁵:

$$\begin{aligned} \text{VC}(c_p) &\Rightarrow c_p \text{ does not fail} \\ c_p \text{ does not fail} &\Rightarrow c_{\text{sa}} \text{ does not fail} \\ c_{\text{sa}} \text{ does not fail} &\Rightarrow c \text{ does not fail} \end{aligned}$$

Turning the last arrow around, we get our desired property for the SA-transformation:

$$c \text{ fails} \Rightarrow c_{\text{sa}} \text{ fails}$$

or more formally, if execution of c starting in a store μ leads to failure, so should the execution of c_{sa} when starting in “some equivalent store” μ_ν . We need to better define this store equivalence, especially since it cannot be the same store (due to the fact that execution of c_{sa} needs a *versioned* store). We say the stores⁶ μ and μ_ν are *synchronized with respect to a version map* ν iff

$$\mu \sim^\nu \mu_\nu \quad \equiv \quad \forall x \bullet \mu(x) = \mu_\nu(x, \nu(x))$$

Represented visually, we can represent a regular store by a line and a versioned store by a grid. Store synchronization means that values correspond along a specific line, determined by the version map:

				2	8	5	4		
				5	9	0	4		
				3	4	6	7		
	3	9	0	2	1	0	8		
	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	
					1	2	0	1	ν

The left and right paths represent μ and μ_ν respectively. The extra line of numbers at the bottom right shows the version map contents: $\nu(a) = 1$, $\nu(b) = 2$, ... Thus, we have

$$\mu(a) = 3 = \mu_\nu(a_{\nu(a)}) = \mu_\nu(a_1)$$

Now that we have defined the concept of synchronized stores, we can express our desired SA-property more clearly: given a command c , an initial store μ and a synchronized store μ_ν where $\mu \sim^\nu \mu_\nu$ for some version map ν , we want $\langle c_{\text{sa}}, \mu_\nu \rangle_\nu$ (where c_{sa} is the SA-transformation of c with respect to ν) to fail if $\langle c, \mu \rangle$ fails. More concisely:

$$\left. \begin{array}{l} \langle c, \mu \rangle \longrightarrow^* \text{failure } \mu' \\ \mu \sim^\nu \mu_\nu \end{array} \right\} \Rightarrow \exists \mu'_\nu \bullet \langle c_{\text{sa}}, \mu_\nu \rangle_\nu \longrightarrow^* \text{failure}_\nu \mu'_\nu$$

In order to prove this, we first need to show another property: let (c_{sa}, ν') be the result of the SA-transformation of c with respect to a version map ν , then

$$\begin{array}{ccc} \langle c, \mu \rangle & \longrightarrow^* & \langle \text{skip}, \mu' \rangle \\ \wr & & \wr \\ \langle c_{\text{sa}}, \mu_\nu \rangle & \longrightarrow^*_\nu & \langle \text{skip}, \mu'_\nu \rangle \end{array}$$

Thus, if we start in a state where the stores are synchronized with respect to ν , both will end up in stores synchronized with respect to ν' .

⁵We have reverted to an informal description of “not failing”, since we have not yet defined the operational semantics for passified programs. These differ a bit from the regular and versioned ones, and we do not want to confuse the reader with a sudden change in notation without any further explanation.

⁶To be exact, one regular store and one versioned store. Later we will define synchronization between two versioned stores.

```

1 Definition function (A B : Type) := A -> B.
2 Definition equivalent_functions (A B : Type)
3                               (f g : function A B) :=
4   forall (x : A), f x = g x.
5 Definition store_sync_vstore (mu : store)
6                               (v : vmap)
7                               (vmu : vstore) :=
8   equivalent_functions mu (fun x => vmu (x, v x)).

```

Figure 17: Synchronized stores in Coq

```

1 Definition delta_id (A : Set) (f g : id -> A) (d : IdSet.t) :=
2   forall x : id, f x = g x \ / IdSet.In x d.
3 Definition vmap_delta := delta_id nat.

```

Figure 18: vmap_delta

Lemma 1. (*sync_store*) *Let e_v be the versioned equivalent of an arbitrary expression e with respect to a version map ν :*

$$e_v = \lambda \mu_v. e(\lambda x. \mu_v(x, \nu(x)))$$

Given two ν -synchronized stores μ and μ_v , then evaluation of e in μ yields the same result as evaluating e_v in μ_v :

$$\mu \sim^\nu \mu_v \quad \Rightarrow \quad e(\mu) = e_v(\mu_v)$$

Lemma 2. (*multistep_seq_skip*) *Given*

$$\langle c_1; c_2, \mu \rangle \longrightarrow^* \langle \mathbf{skip}, \mu'' \rangle$$

then there exists a store μ' for which

$$\begin{aligned} \langle c_1, \mu \rangle &\longrightarrow^* \langle \mathbf{skip}, \mu' \rangle \\ \langle c_2, \mu' \rangle &\longrightarrow^* \langle \mathbf{skip}, \mu'' \rangle \end{aligned}$$

Lemma 3. (*sync_vcommand_goes_to_skip*) *For any set of ids I , version maps ν and ν' and versioned store μ_v , there exists a μ'_v so that*

$$\langle \text{sync}(I, \nu, \nu'), \mu_v \rangle_\nu \longrightarrow^* \langle \mathbf{skip}, \mu'_v \rangle_\nu$$

Definition 5 (Delta set). (*vmap_delta*) *We define the delta set of two functions f and g with the same domain D as the subset Δ of that domain D so that*

$$\forall x \in D \bullet f(x) = g(x) \vee x \in \Delta$$

We denote this property as $\text{delta}(f, g, \Delta)$.

Definition 6 (Synchronized versioned stores). (*vstore_sync_vstore*) *We say two versioned stores μ_v and μ'_v are synchronized with respect to version maps ν and ν' when*

$$\forall x \bullet \mu_v(x_{\nu(x)}) = \mu'_v(x_{\nu'(x)})$$

Notation: $\mu_v \stackrel{\nu \nu'}{\sim} \mu'_v$.

```

1 Definition vstore_sync_vstore (mu : vstore)
2                               (v v' : vmap)
3                               (mu' : vstore) :=
4   forall x : id, mu (x, v x) = mu' (x, v' x).

```

Figure 19: vstore_sync_vstore

Lemma 4. (*sync_vcommand_determinism*) *Given a synchronization command $c = \text{sync}(I, \nu, \nu')$ for some set of identifiers I and version maps ν and ν' , and given that*

$$\langle c, \mu_v \rangle_v \longrightarrow_v^* \langle \mathbf{skip}, \mu'_{v,1} \rangle_v$$

and

$$\langle c, \mu \rangle_v \longrightarrow_v^* \langle \mathbf{skip}, \mu'_{v,2} \rangle_v$$

then

$$\mu'_{v,1} = \mu'_{v,2}$$

Lemma 5. (*sync_vcommand_works*) *For any set of identifiers I , version maps ν and ν' , and versioned stores μ_v and μ'_v , if*

$$\text{delta}(\nu, \nu', I)$$

and

$$\langle \text{sync}(I, \nu, \nu'), \mu_v \rangle_v \longrightarrow_v^* \langle \mathbf{skip}, \mu'_v \rangle_v$$

then

$$\mu \stackrel{\nu, \nu'}{\sim} \mu'$$

Lemma 6. (*combine_vmaps*) *Given*

$$\mu \sim^\nu \mu_v \stackrel{\nu, \nu'}{\sim} \mu'_v$$

then

$$\mu \sim^{\nu'} \mu'_v$$

Theorem 1. (*sa_transformation_skip*) *For any command c , stores μ and μ' , versioned store μ_v and version map ν , let $(c_{\text{sa}}, \nu') = \text{SA}_\nu(c)$, then, if*

$$\langle c, \mu \rangle \longrightarrow^* \langle \mathbf{skip}, \mu' \rangle$$

and

$$\mu \sim^\nu \mu_v$$

then there exists some μ'_v for which

$$\langle c', \mu_v \rangle_v \longrightarrow_v^* \langle \mathbf{skip}, \mu'_v \rangle$$

and

$$\mu' \sim^{\nu'} \mu'_v$$

Proof. By structural induction on c .

- $c = \mathbf{assert} \ e$: We know that $c_{\text{sa}} = \mathbf{assert} \ e_v$ and $\nu' = \nu$. The only possible path for

$$\langle \mathbf{assert} \ e, \mu \rangle \longrightarrow^* \langle \mathbf{skip}, \mu' \rangle$$

is

$$\langle \mathbf{assert} \ e, \mu \rangle \longrightarrow \langle \mathbf{skip}, \mu \rangle$$

We choose $\mu'_v = \mu_v$, so that we must prove that

$$\langle \mathbf{assert} \ e_v, \mu_v \rangle_v \longrightarrow_v^* \langle \mathbf{skip}, \mu_v \rangle_v \quad (2)$$

and

$$\mu \sim^\nu \mu_v \quad (3)$$

hold. (3) is trivially true. (2) can be shown by proving that e_v is true when evaluated in the store μ_v , which is the case, see Lemma 1.

- $c = \mathbf{assume} \ e$: similar to the previous case.
- $c = \mathbf{skip}$: trivial.
- $c = x := e$: We choose $\mu'_v = \mu_v[x \mapsto e_v(\mu_v)]$ so that we get

$$\langle x := e_v, \mu_v \rangle_v \longrightarrow_v \langle \mathbf{skip}, \mu_v[x \mapsto e_v(\mu_v)] \rangle_v$$

We know that

$$\langle x := e, \mu \rangle \longrightarrow_v \langle \mathbf{skip}, \mu[x \mapsto e(\mu)] \rangle$$

meaning we need to prove that

$$\mu[x \mapsto e(\mu)] \sim^\nu \mu_v[x \mapsto e_v(\mu_v)]$$

which, apart from some technicalities dealt with in the Coq script (`store_sync_vstore_assignment`), is clearly true.

- $c = c_1; c_2$: From the SA-transformation, we get

$$\begin{aligned} c_{sa} &= c_{1,sa}; c_{2,sa} \\ (c_{1,sa}, \nu_1) &= \mathbf{SA}_{\nu'}(c_1) \\ (c_{2,sa}, \nu_2) &= \mathbf{SA}_{\nu_1}(c_2) \\ \nu' &= \nu_2 \end{aligned}$$

So, we have

$$\langle c_1; c_2, \mu \rangle \longrightarrow^* \langle \mathbf{skip}, \mu' \rangle \quad \wedge \quad \mu \sim^\nu \mu_v$$

and we need to prove that there exists some μ'_v so that

$$\langle c_{1,sa}; c_{2,sa}, \mu_v \rangle_v \longrightarrow_v^* \langle \mathbf{skip}, \mu'_v \rangle_v \quad \wedge \quad \mu' \sim^{\nu'} \mu'_v$$

From Lemma 2 we know that there exists a μ'' so that

$$\begin{aligned} \langle c_1, \mu \rangle &\longrightarrow^* \langle \mathbf{skip}, \mu'' \rangle \\ \langle c_2, \mu'' \rangle &\longrightarrow^* \langle \mathbf{skip}, \mu' \rangle \end{aligned}$$

The induction hypothesis states that there exists a μ''_v so that

$$\begin{aligned} \langle c_{1,sa}, \mu_v \rangle_v &\longrightarrow_v^* \langle \mathbf{skip}, \mu''_v \rangle_v \\ \langle c_{2,sa}, \mu''_v \rangle_v &\longrightarrow_v^* \langle \mathbf{skip}, \mu'_v \rangle_v \end{aligned}$$

$$\begin{aligned} \mu'' &\sim^{\nu_1} \mu''_v \\ \mu' &\sim^{\nu'} \mu'_v \end{aligned}$$

Finally, we can build the following chain:

$$\langle c_{1,sa}; c_{2,sa}, \mu_v \rangle_v \longrightarrow_v^* \langle \mathbf{skip}; c_{2,sa}, \mu''_v \rangle_v \longrightarrow_v \langle c_{2,sa}, \mu''_v \rangle_v \longrightarrow_v^* \langle \mathbf{skip}, \mu'_v \rangle_v$$

which transitivity of \longrightarrow_v^* (`vmultistep_trans`) turns into

$$\langle c_{1,sa}; c_{2,sa}, \mu_v \rangle_v \longrightarrow_v^* \langle \mathbf{skip}, \mu'_v \rangle_v$$

- $c = c_1 \parallel c_2$: We know from the premisses that we reach **skip**, which means that one of the following holds:

$$\begin{aligned} \langle c_1, \mu \rangle &\longrightarrow^* \langle \mathbf{skip}, \mu' \rangle \\ \langle c_2, \mu \rangle &\longrightarrow^* \langle \mathbf{skip}, \mu' \rangle \end{aligned}$$

We only consider the first of these two possibilities, as the other can be dealt with similarly. The SA-translation gives us

$$\begin{aligned} (c_{1,\text{sa}}, \nu_1) &= \text{SA}_\nu(c_1) \\ (c_{2,\text{sa}}, \nu_2) &= \text{SA}_\nu(c_2) \\ \nu' &= \text{join}(\nu_1, \nu_2) \\ c_{\text{sa}} &= c_{1,\text{sa}}; \delta_1 \parallel c_{2,\text{sa}}; \delta_2 \end{aligned}$$

From the induction hypothesis, we know that there exists a store μ''_v so that

$$\langle c_{1,\text{sa}}, \mu_v \rangle_v \longrightarrow_v^* \langle \mathbf{skip}, \mu''_v \rangle_v \quad \wedge \quad \mu' \sim^{\nu_1} \mu''_v$$

We put our current situation in a diagram:

$$\begin{array}{ccccccc} \langle c_1 \parallel c_2, \mu \rangle & \longrightarrow & \langle c_1, \mu \rangle & \longrightarrow^* & \langle \mathbf{skip}, \mu' \rangle & \longrightarrow^* & \langle \mathbf{skip}, \mu' \rangle \\ \zeta & & \zeta & & \zeta & & \zeta \end{array}$$

$$\langle c_{1,\text{sa}}; \delta_1 \parallel c_{2,\text{sa}}; \delta_2, \mu_v \rangle_v \longrightarrow_v \langle c_{1,\text{sa}}; \delta_1, \mu_v \rangle_v \longrightarrow_v^* \langle \mathbf{skip}; \delta_1, \mu''_v \rangle_v \longrightarrow_v^* \langle \mathbf{skip}, \mu'_v \rangle_v$$

Note the upper left corner, representing what we need to prove: we know that

$$\mu' \sim^{\nu_1} \mu''_v$$

but we need that the same store μ' is synchronized with

$$\mu' \sim^{\nu'} \mu'_v$$

This is δ_1 's responsibility. From Lemma 3, Lemma 4 and Lemma 5 we know that

$$\langle \delta_1, \mu''_v \rangle_v \longrightarrow_v^* \langle \mathbf{skip}, \mu'_v \rangle_v$$

with

$$\mu''_v \sim^{\nu_1 \nu'} \mu'_v$$

Combining the different store synchronizations we get from Lemma 6

$$\mu' \sim^{\nu'} \mu'_v$$

□

Lemma 7. (multistep_seq_fail) *Given*

$$\langle c_1; c_2, \mu \rangle \longrightarrow^* \text{failure } \mu'$$

then, either

$$\langle c_1, \mu \rangle \longrightarrow^* \text{failure } \mu'$$

or there exists a store μ'' so that

$$\langle c_1, \mu \rangle \longrightarrow^* \langle \mathbf{skip}, \mu'' \rangle \quad \langle c_2, \mu'' \rangle \longrightarrow^* \text{failure } \mu'$$

Lemma 8. (`vmultistep_lift_seq_fail`) *Given*

$$\langle c_1, \mu_v \rangle_v \longrightarrow_v^* \text{failure}_v \mu'_v$$

then we can lift this inside a sequence:

$$\langle c_1; c_2, \mu_v \rangle_v \longrightarrow_v^* \text{failure}_v \mu'_v$$

Theorem 2. (`sa_transformation_fail`) *For any command c , stores μ and μ' , versioned store μ_v and version map ν , let $(c_{\text{sa}}, \nu') = \text{SA}_\nu(c, \nu)$, then, if*

$$\langle c, \mu \rangle \longrightarrow^* \text{failure}_v \mu'$$

and

$$\mu \sim^\nu \mu_v$$

then there exists some μ'_v for which

$$\langle c', \mu_v \rangle_v \longrightarrow_v^* \text{failure}_v \mu'_v$$

Proof. By structural induction on c .

- $c = \mathbf{assert} \ e$: The only possible path for

$$\langle \mathbf{assert} \ e, \mu \rangle \longrightarrow^* \text{failure} \mu'$$

is

$$\langle \mathbf{assert} \ e, \mu \rangle \longrightarrow \text{failure} \mu \tag{4}$$

We take $\mu'_v = \mu_v$, so that we need to show that

$$\langle \mathbf{assert} \ e_v, \mu_v \rangle_v \longrightarrow_v^* \text{failure} \mu_v$$

This is clearly the case, since we know from (4) that e does not evaluate to **true** in μ , so by Lemma 1 neither will e_v in μ_v .

- $c = \mathbf{assume} \ e$: cannot lead to failure.
- $c = \mathbf{skip}$: cannot lead to failure.
- $c = x := e$: cannot lead to failure.
- $c = c_1; c_2$: The SA-translation gives us

$$(c_{\text{sa},1}; c_{\text{sa},2}, \nu') = \text{SA}_\nu(c)$$

where

$$(c_{\text{sa},1}, \nu'') = \text{SA}_\nu(c_1) \quad (c_{\text{sa},2}, \nu') = \text{SA}_{\nu''}(c_1)$$

Lemma 7 lets us distinguish two cases:

- First case: c_1 fails

$$\langle c_1, \mu \rangle \longrightarrow^* \text{failure} \mu'$$

We need to prove that

$$\langle c_{1,\text{sa}}, \mu_v \rangle_v \longrightarrow_v^* \text{failure}_v \mu'_v$$

From the induction hypothesis, we know that μ'_v exists such that

$$\langle c_{1,\text{sa}}, \mu_v \rangle_v \longrightarrow_v^* \text{failure}_v \mu'_v$$

We lift this using Lemma 8:

$$\langle c_{1,\text{sa}}; c_{2,\text{sa}}, \mu_v \rangle_v \longrightarrow_v^* \text{failure}_v \mu'_v$$

– Second case: c_1 skips, c_2 fails

$$\langle c_1, \mu \rangle \longrightarrow^* \langle \mathbf{skip}, \mu'' \rangle \quad \langle c_2, \mu'' \rangle \longrightarrow^* \mathbf{failure} \mu'$$

From Theorem 1 we know that there exists a μ''_v so that

$$\begin{aligned} \langle c_{1,sa}, \mu_v \rangle_v &\longrightarrow^* \langle \mathbf{skip}, \mu''_v \rangle_v \\ \mu'' &\sim^{\nu''} \mu''_v \end{aligned}$$

This crosses the first bridge. The second bridge is crossed by applying the induction hypothesis:

$$\langle c_{2,sa}, \mu''_v, \longrightarrow^* \rangle_v \mathbf{failure} \mu'_v$$

- $c = c_1 \parallel c_2$: either failure is reached through c_1 or through c_2 . We only discuss with the former case as the latter is dealt with similarly. Thus, we assume that

$$\langle c_1, \mu \rangle \longrightarrow^* \mathbf{failure} \mu'$$

The SA-transformation gives us

$$\text{SA}_\nu(c_1 \parallel c_2) = c_{1,sa}; \delta_1 \parallel c_{2,sa}; \delta_2$$

We need to show that this command fails. This is the case if

$$\langle c_{1,sa}; \delta_1, \mu_v \rangle_v \longrightarrow^* \mathbf{failure}_v \mu'_v$$

and this is true if

$$\langle c_{1,sa}, \mu_v \rangle_v \longrightarrow^* \mathbf{failure}_v \mu'_v$$

We can prove this by applying the induction hypothesis.

□

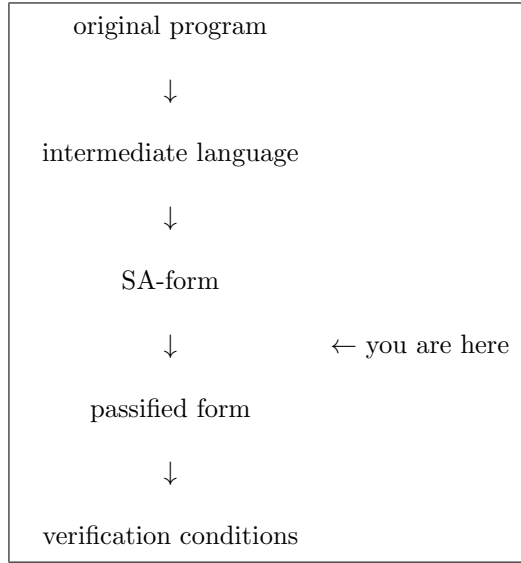


Figure 20: Steps in the verification algorithm

4 Passification

In this section, we provide details about passification [6, 4], the second program transformation. It consists of removing all assignments from a program and replacing them by (behaviour-wise) equivalent commands. As with the SA-transformation, we must then prove that the original program and its passified equivalent do indeed behave similarly.

4.1 Algorithm

We say a program is in passified form if it contains no assignments. This means the store never changes and we can factor it out: instead of it being part of the state, it becomes a component of the step relation. Thus, instead of writing

$$\langle c, \mu \rangle \longrightarrow \langle c', \mu \rangle$$

we can write

$$\langle c \rangle_{\mathbf{p}} \xrightarrow{\mu} \langle c' \rangle_{\mathbf{p}}$$

We define a new set of operational semantics, as we did in the previous section for the SA-transformation. Fortunately, we do not need to define new variants of identifiers, variants, ... as we did in the previous section.

- We can reuse versioned identifiers, versioned commands, and versioned stores.
- Our new set of commands (`pcommand`, see Figure 22) is identical to the versioned variant (`vcommand`) defined in the previous section, with the exception that we leave out the assignment command.
- We still have two states (Figure 21):
 - The failure state `pfailure`, which contrary to the previously defined failure states `failure` and `vfailure` does *not* contain a store component⁷. We denote failure by `failurep`.

⁷We defined `pfailure` storeless simply because we don't need the store information. However, `failure` and `vfailure` do need it for some theorems which build the bridges between the different operational semantics.

```

1 Inductive pstate : Set :=
2 | pip      : pcommand -> pstate
3 | pfailure : pstate.

```

Figure 21: Passified stores

```

1 Inductive pcommand : Set :=
2 | pcAssert  : vexpr -> pcommand
3 | pcAssume  : vexpr -> pcommand
4 | pcSequence : pcommand -> pcommand -> pcommand
5 | pcSkip    : pcommand
6 | pcChoice  : pcommand -> pcommand -> pcommand.

```

Figure 22: Passified commands

- The in-progress state `pip`, whose only component is a passified command (`pcommand`).
- The operational semantics (single step `pstep` and multiple step `pmultistep`) for passified commands describe identical behaviour to those describing the versioned commands. The differences are:
 - There is no rule for dealing with assignment (since there is no assignment command anymore).
 - As mentioned above, states are associated with the step relation instead of with a state.
 - If σ_1 and σ_2 are states, we write the single step relation between them as $\sigma_1 \xrightarrow[\text{p}]{\mu_v} \sigma_2$, where μ_v is a versioned store.

Since passification follows the SA-transformation, the passification algorithm can assume the program it needs to transform is in SA form, which makes it pretty straightforward:

$$\begin{aligned}
\text{passify}(\mathbf{assert} \ e_v) &= \mathbf{assert} \ e_v \\
\text{passify}(\mathbf{assume} \ e_v) &= \mathbf{assume} \ e_v \\
\text{passify}(\mathbf{skip}) &= \mathbf{skip} \\
\text{passify}(c_1; c_2) &= \text{passify}(c_1); \text{passify}(c_2) \\
\text{passify}(c_1 \parallel c_2) &= \text{passify}(c_1) \parallel \text{passify}(c_2) \\
\text{passify}(x := e_v) &= \mathbf{assume} \ x = e_v
\end{aligned}$$

Note how assignments are converted into **assume** statements. This explains why we first need to transform the code to SA-form. Consider the following example:

$$x := 1; x := 2; \mathbf{assert} \ x = 1$$

It is clear this program should fail. If we were to apply to passification algorithm directly, we would get

$$\mathbf{assume} \ x = 1; \mathbf{assume} \ x = 2; \mathbf{assert} \ x = 1$$

This program does *not* fail: execution will get stuck on either the first or second assume, never reach the assertion, and thus never fail. Applying the SA-transformation yields

$$x_1 := 1; x_2 := 2; \mathbf{assert} \ x_2 = 1$$

This will clearly fail, and everything is still okay. Passifying gives

$$\mathbf{assume} \ x_1 = 1; \mathbf{assume} \ x_2 = 2; \mathbf{assert} \ x_2 = 1$$

$$\begin{array}{c}
\frac{e_v(\mu_v) = \mathbf{true}}{\langle \mathbf{assert} \ e_v \rangle_p \xrightarrow{\mu_v}_p \langle \mathbf{skip} \rangle_p} \\
\\
\frac{e_v(\mu_v) \neq \mathbf{true}}{\langle \mathbf{assert} \ e_v \rangle_p \xrightarrow{\mu_v}_p \mathbf{failure}_p} \\
\\
\frac{e_v(\mu_v) = \mathbf{true}}{\langle \mathbf{assume} \ e_v \rangle_p \xrightarrow{\mu_v}_p \langle \mathbf{skip} \rangle_p} \\
\\
\frac{\langle c_{p,1} \rangle_p \xrightarrow{\mu_v}_p \langle c'_{p,1} \rangle_p}{\langle c_{p,1}; c_{p,2} \rangle_p \xrightarrow{\mu_v}_p \langle c'_{p,1}; c_{p,2} \rangle_p} \\
\\
\frac{\langle c_{p,1} \rangle_p \xrightarrow{\mu}_p \mathbf{failure}_p}{\langle c_{p,1}; c_{p,2} \rangle_p \xrightarrow{\mu_v}_p \mathbf{failure}_p} \\
\\
\frac{}{\langle \mathbf{skip}; c_p \rangle_p \xrightarrow{\mu_v}_p \langle c_p \rangle_p} \\
\\
\frac{}{\langle c_{p,1} \parallel c_{p,2} \rangle_p \xrightarrow{\mu_v}_p \langle c_{p,1} \rangle_p} \\
\\
\frac{}{\langle c_{p,1} \parallel c_{p,2} \rangle_p \xrightarrow{\mu_v}_p \langle c_{p,2} \rangle_p}
\end{array}$$

Figure 23: Passified operational semantics

```

1 Inductive step : state -> state -> Prop :=
2 | stepAssertT : forall e mu,
3   e mu = T ->
4   step (ip (cAssert e) mu) (ip cSkip mu)
5 | stepAssertF : forall e mu,
6   e mu <> T ->
7   step (ip (cAssert e) mu) (failure mu)
8 | stepAssume : forall e mu,
9   e mu = T ->
10  step (ip (cAssume e) mu) (ip cSkip mu)
11 | stepSeq : forall c1 c1' c2 mu mu',
12  step (ip c1 mu) (ip c1' mu') ->
13  step (ip (cSequence c1 c2) mu)
14      (ip (cSequence c1' c2) mu')
15 | stepSeqSkip : forall c2 mu,
16  step (ip (cSequence cSkip c2) mu) (ip c2 mu)
17 | stepSeqFail : forall c1 c2 mu mu',
18  step (ip c1 mu) (failure mu') ->
19  step (ip (cSequence c1 c2) mu) (failure mu')
20 | stepAssign : forall x e mu,
21  step (ip (cAssign x e) mu)
22      (ip cSkip (update_store mu x (e mu)))
23 | stepChoiceL : forall c1 c2 mu,
24  step (ip (cChoice c1 c2) mu)
25      (ip c1 mu)
26 | stepChoiceR : forall c1 c2 mu,
27  step (ip (cChoice c1 c2) mu)
28      (ip c2 mu).

```

Figure 24: Passified operational semantics in Coq

```

1 Parameters
2   (value : Set)
3   (T     : value)
4   (F     : value).
5 Axiom T_neq_F : T <> F.
6 Definition assume_from_assign x e :=
7   pcAssume (fun vmu => if decidable_eq_value (vmu x)
8                                     (e vmu)
9                                     then T
10                                    else F).
11 Fixpoint passify (c : vcommand) : pcommand :=
12   match c with
13   | vcAssert e      => pcAssert e
14   | vcAssume e      => pcAssume e
15   | vcSkip          => pcSkip
16   | vcSequence c1 c2 => pcSequence (passify c1)
17                                     (passify c2)
18   | vcChoice c1 c2  => pcChoice (passify c1)
19                                     (passify c2)
20   | vcAssign x e    => assume_from_assign x e
21   end.

```

Figure 25: Passification algorithm in Coq

This program will fail in a store μ_v where $\mu_v(x_1) = 1$ and $\mu_v(x_2) = 2$, which is exactly what we need.

An **assume** command requires an expression as operand, meaning we need to express our condition $x = e_v$ as a function from versioned stores to values: (**assume_from_assign**)

$$\lambda \mu_v. \text{if } \mu_v(x) = e_v(\mu_v) \text{ then true else false}$$

4.2 Failure property

In Section 3.4, we described how the SA-transformation is the first in a series of steps, passification being the second. From the passified program, we generate verification conditions (VCs). Our goal is that the validity of the verification conditions implies that the original program does not fail:

“VCs are valid”
 implies
 “original program does not fail”

We can expand this as follows⁸:

“VCs are valid”
 implies
 “passification does not fail”
 implies
 “SA-form does not fail”
 implies
 “original program does not fail”

⁸To be read as the first implies the second, the second implies the third, ...

```

1 Definition stores_veq (vmu : vstore)
2   (v : vmap)
3   (vmu' : vstore) :=
4   forall x n, n <= v x -> vmu (x, n) = vmu' (x, n).

```

Figure 26: Synchronized stores up to a certain version map in Coq

Turning this around gives

“original program fails”
 implies
 “SA-form fails”
 implies
 “passification fails”
 implies
 “VCs are not valid”

We have shown that the first implication holds in Section 3.4. In this section we will deal with the second one. We formulate this more formally:

$$\langle c_{sa}, \mu_v \rangle_v \longrightarrow^* \text{failure}_v \mu'_v \Rightarrow \langle \text{passify}(c_{sa}) \rangle_p \xrightarrow{\mu'_v}^* \text{failure}_p$$

4.3 Theorems and proofs

Definition 7. (`stores_veq`) *We say two versioned stores μ_v and μ'_v are synchronized up to a version map ν iff*

$$\mu_v \stackrel{\leq \nu}{\approx} \mu'_v \quad \equiv \quad \forall x, n \bullet n \leq \nu(x) \Rightarrow \mu_v(x_n) = \mu'_v(x_n)$$

Visually, if we represent versioned stores μ_v and μ'_v as grids with identifiers spread out horizontally and version numbers vertically, the version map ν represents a line under which the store bindings must be equal:

$$\mu_v \stackrel{\leq \nu}{\approx} \mu'_v \quad \nu(a) = 2, \nu(b) = 1, \nu(c) = 2, \nu(d) = 0$$

	μ_v		μ'_v
4	8 5 2 3		1 5 8 9
3	4 2 0 1		4 4 7 0
2	<u>7 6 6 7</u>		<u>7 3 6 0</u>
1	<u>3 6 2 9</u>		<u>3 6 2 8</u>
0	<u>8 1 3 7</u>		<u>8 1 3 7</u>
	<u>a b c d ...</u>		<u>a b c d ...</u>

Lemma 9. (`vexpr_stores_veq`) *For any expression e , version map ν and versioned stores μ_v and μ'_v , let e_v be the versioned equivalent of e with respect to ν , then if*

$$\mu_v \stackrel{\leq \nu}{\approx} \mu'_v$$

then evaluating e_v in μ_v and μ'_v yield the same value.

```

1 Definition vmap_le (v v' : vmap) :=
2   forall x, v x <= v' x.

```

Figure 27: Order relation on version maps in Coq

Lemma 10. (`vmultistep_seq_skip`) *If*

$$\langle c_1; c_2, \mu_v \rangle_v \longrightarrow_v^* \langle \mathbf{skip}, \mu'_v \rangle_v$$

then there exists a versioned store μ''_v for which

$$\begin{aligned} \langle c_1, \mu_v \rangle_v &\longrightarrow_v^* \langle \mathbf{skip}, \mu''_v \rangle_v \\ \langle c_2, \mu''_v \rangle_v &\longrightarrow_v^* \langle \mathbf{skip}, \mu'_v \rangle_v \end{aligned}$$

Lemma 11. (`single_assignment_monotonic_store`) *Given a program c_{sa} being the SA-form of a regular program c with respect to a version map ν , if*

$$\langle c_{sa}, \mu_v \rangle_v \longrightarrow_v^* \langle \mathbf{skip}, \mu'_v \rangle_v$$

then

$$\mu_v \stackrel{\leq \nu}{\sim} \mu'_v$$

Definition 8. (`vmap_le`) *We define a partial order on version maps as follows*

$$\nu \preceq \nu' \quad \equiv \quad \forall x \bullet \nu(x) \leq \nu'(x)$$

Lemma 12. (`sa_transformation_monotonic_vmap`) *For any (regular) command c and version map ν , given $(c_{sa}, \nu') = \text{SA}_\nu(c)$, then*

$$\nu \preceq \nu'$$

Lemma 13. (`stores_veq_vmap_le`) *Given*

$$\mu_v \stackrel{\leq \nu}{\sim} \mu'_v$$

and

$$\nu \preceq \nu'$$

then

$$\mu_v \stackrel{\leq \nu'}{\sim} \mu'_v$$

Intuitively, if just considering a single identifier, this lemma states that if the store values match up until version 5, they will also match up to version 4, 3, 2, 1 or 0. This of course can then be generalized to each identifier.

Lemma 14. (`stores_veq_trans`)

$$\mu_v \stackrel{\leq \nu}{\sim} \mu'_v \stackrel{\leq \nu}{\sim} \mu''_v$$

implies

$$\mu_v \stackrel{\leq \nu}{\sim} \mu''_v$$

Lemma 15. (`stores_veq_sync_vcommand`) *For any set of identifiers I , version maps ν_1 and ν_2 and versioned stores μ_v and μ'_v , if*

$$\nu_1 \preceq \nu_2$$

and

$$\langle \text{sync}(I, \nu_1, \nu_2), \mu_v \rangle_v \longrightarrow_v^* \langle \mathbf{skip}, \mu'_v \rangle_v$$

then

$$\mu_v \stackrel{\leq \nu_1}{\sim} \mu'_v$$

Lemma 16. (`vmultistep_pmultistep_sync_vcommand`) For any set of identifiers I , version maps ν and ν' , and versioned stores $\mu_\nu, \mu'_\nu, \mu''_\nu$, given

$$\begin{aligned} \delta &= \text{sync}(I, \nu, \nu') \\ \nu &\preceq \nu' \\ \langle \delta, \mu_\nu \rangle_\nu &\longrightarrow_\nu^* \langle \mathbf{skip}, \mu'_\nu \rangle_\nu \\ \mu'_\nu &\stackrel{\leq \nu'}{\sim} \mu''_\nu \end{aligned}$$

then

$$\langle \text{passify}(\delta) \rangle_\text{p} \xrightarrow{\mu''_\nu}_\text{p}^* \langle \mathbf{skip} \rangle_\text{p}$$

Lemma 17. (`vmap_le_join_l`) For any version maps ν_1 and ν_2 ,

$$\nu_1 \preceq \text{join}(\nu_1, \nu_2)$$

Theorem 3. (`vmultistep_pmultistep_skip`) For any version map ν , (regular) command c , and versioned stores $\mu_\nu, \mu''_\nu, \mu_\text{p}$, let $(c_{\text{sa}}, \nu') = \text{SA}_\nu(c)$, then if

$$\langle c_{\text{sa}}, \mu_\nu \rangle_\nu \longrightarrow_\nu^* \langle \mathbf{skip}, \mu''_\nu \rangle_\nu$$

and

$$\mu''_\nu \stackrel{\leq \nu'}{\sim} \mu_\text{p}$$

then

$$\langle \text{passify}(c_{\text{sa}}) \rangle_\text{p} \xrightarrow{\mu_\text{p}}_\text{p}^* \langle \mathbf{skip} \rangle_\text{p}$$

Proof. By structural induction on c :

- $c = \mathbf{assert} \ e$: the SA-translation is

$$c_{\text{sa}} = \mathbf{assert} \ e_\nu$$

There is only one execution path (remember it must lead to **skip**, not fail) to deal with:

$$\langle \mathbf{assert} \ e_\nu, \mu_\nu \rangle_\nu \longrightarrow_\nu \langle \mathbf{skip}, \mu_\nu \rangle_\nu$$

where evaluating e_ν in μ_ν yields **true**. We need to show that

$$\langle \mathbf{assert} \ e_\nu \rangle_\text{p} \xrightarrow{\mu_\text{p}}_\text{p} \langle \mathbf{skip} \rangle_\text{p}$$

This is the case if e_ν evaluated in μ_p results in **true**. This is indeed so by Lemma 9.

- $c = \mathbf{assume} \ e$: similar to the $c = \mathbf{assert} \ e$ case.
- $c = \mathbf{skip}$: trivial.
- $c = x := e$. The SA-translation yields

$$c_{\text{sa}} = x_{\nu(x)+1} := e_\nu$$

and

$$\nu' = \nu[x \mapsto \nu(x) + 1]$$

where e_ν is e 's versioned equivalent with respect to version map ν . The only execution path is

$$\langle x_{\nu(x)+1} := e_\nu, \mu_\nu \rangle_\nu \longrightarrow_\nu \langle \mathbf{skip}, \mu_\nu[x_{\nu(x)+1} \mapsto e_\nu(\mu_\nu)] \rangle_\nu$$

Given that

$$\mu_\nu[x_{\nu(x)+1} \mapsto e_\nu(\mu_\nu)] \stackrel{\leq \nu'}{\sim} \mu_\nu \quad (5)$$

we need to show that

$$\langle \mathbf{assume} \ x_{\nu(x)+1} = e_\nu \rangle_{\mathbb{P}} \xrightarrow{\mu_\mathbb{P}} \langle \mathbf{skip} \rangle_{\mathbb{P}}$$

This is the case if

$$\mu_\mathbb{P}(x_{\nu(x)+1}) = e_\nu(\mu_\mathbb{P})$$

From (5) follows that

$$\mu_\mathbb{P}(x_\nu(x) + 1) = e_\nu(\mu_\nu)$$

Our new goal is to prove that

$$e_\nu(\mu_\nu) = e_\nu(\mu_\mathbb{P})$$

This follows from the fact that e_ν only refers to “old” variables, i.e. variables for which the bindings in μ_ν and $\mu_\mathbb{P}$ coincide.

- $c = c_1; c_2$: from the SA-translation we have

$$\begin{aligned} (c_{1,\text{sa}}, \nu') &= \text{SA}_\nu(c_1) \\ (c_{2,\text{sa}}, \nu'') &= \text{SA}_{\nu'}(c_2) \\ c_{\text{sa}} &= c_{1,\text{sa}}; c_{2,\text{sa}} \end{aligned}$$

From Lemma 10 we know that there exists a versioned store μ_ν so that

$$\begin{aligned} \langle c_1, \mu_\nu \rangle_\nu &\longrightarrow^* \langle \mathbf{skip}, \mu'_\nu \rangle_\nu \\ \langle c_2, \mu'_\nu \rangle_\nu &\longrightarrow^* \langle \mathbf{skip}, \mu''_\nu \rangle_\nu \end{aligned}$$

Applying Lemma 11 leads to

$$\mu_\nu \stackrel{\leq \nu}{\sim} \mu'_\nu \stackrel{\leq \nu'}{\sim} \mu''_\nu$$

Lemma 12 gives

$$\nu \preceq \nu' \preceq \nu''$$

Before being able to invoke the induction hypothesis, we need to show that

$$\mu'_\nu \stackrel{\leq \nu'}{\sim} \mu_\mathbb{P}$$

From the premises we know

$$\mu''_\nu \stackrel{\leq \nu''}{\sim} \mu_\mathbb{P}$$

Lemma 13 turns this into (since $\nu' \preceq \nu''$)

$$\mu''_\nu \stackrel{\leq \nu'}{\sim} \mu_\mathbb{P}$$

And because $\mu'_\nu \stackrel{\leq \nu'}{\sim} \mu''_\nu$, Lemma 14 gives us

$$\mu'_\nu \stackrel{\leq \nu'}{\sim} \mu_\mathbb{P}$$

which is what we need to apply the induction hypothesis. We now get

$$\langle \text{passify}(c_{1,\text{sa}}) \rangle_{\mathbb{P}} \xrightarrow{\mu_\mathbb{P}}^* \langle \mathbf{skip} \rangle_{\mathbb{P}}$$

Next, we need to show that

$$\langle \text{passify}(c_{2,\text{sa}}) \rangle_{\mathbb{P}} \xrightarrow{\mu_\mathbb{P}}^* \langle \mathbf{skip} \rangle_{\mathbb{P}} \quad (6)$$

so that we can chain both together in one execution path

$$\langle \text{passify}(c_{1,\text{sa}}; c_{2,\text{sa}}) \rangle_{\text{P}} \xrightarrow{\mu_{\text{P}}^*} \langle \mathbf{skip} \rangle_{\text{P}}$$

which is our ultimate goal. To prove (6), we need to apply the induction hypothesis again. In order to be able to apply it, we need

$$\mu_{\text{V}}'' \stackrel{\leq \nu''}{\sim} \mu_{\text{P}}$$

which we know is true from the premises.

- $c = c_1 \parallel c_2$: the SA-transformation is

$$\begin{aligned} (c_{1,\text{sa}}, \nu_1) &= \text{SA}_{\nu}(c_1) \\ (c_{2,\text{sa}}, \nu_2) &= \text{SA}_{\nu}(c_2) \\ c_{\text{sa}} &= c_{1,\text{sa}}; \delta_1 \parallel c_{2,\text{sa}}; \delta_2 \end{aligned}$$

We know it reaches **skip**, thus either

$$\langle c_{1,\text{sa}}; \delta_1, \mu_{\text{V}} \rangle_{\text{V}} \longrightarrow_{\text{V}}^* \langle \mathbf{skip}, \mu_{\text{V}}'' \rangle_{\text{V}} \quad (7)$$

or

$$\langle c_{2,\text{sa}}; \delta_2, \mu_{\text{V}} \rangle_{\text{V}} \longrightarrow_{\text{V}}^* \langle \mathbf{skip}, \mu_{\text{V}}'' \rangle_{\text{V}}$$

We only deal with the former case, the latter being nearly identical. We need to show that

$$\langle \text{passify}(c_{1,\text{sa}}; \delta_1) \rangle_{\text{P}} \xrightarrow{\mu_{\text{P}}^*} \langle \mathbf{skip} \rangle_{\text{P}}$$

which is equivalent with

$$\langle \text{passify}(c_{1,\text{sa}}); \text{passify}(\delta_1) \rangle_{\text{P}} \xrightarrow{\mu_{\text{P}}^*} \langle \mathbf{skip} \rangle_{\text{P}} \quad (8)$$

Applying Lemma 10 to (7) tells us there exists a μ_{V}' such that

$$\langle c_{1,\text{sa}}, \mu_{\text{V}} \rangle_{\text{V}} \longrightarrow_{\text{V}}^* \langle \mathbf{skip}, \mu_{\text{V}}' \rangle_{\text{V}} \quad (9)$$

$$\langle \delta_1, \mu_{\text{V}}' \rangle_{\text{V}} \longrightarrow_{\text{V}}^* \langle \mathbf{skip}, \mu_{\text{V}}'' \rangle_{\text{V}} \quad (10)$$

We split up the goal (8) in the following subgoals

$$\langle \text{passify}(c_{1,\text{sa}}) \rangle_{\text{P}} \xrightarrow{\mu_{\text{P}}^*} \langle \mathbf{skip} \rangle_{\text{P}} \quad (11)$$

$$\langle \text{passify}(\delta_1) \rangle_{\text{P}} \xrightarrow{\mu_{\text{P}}^*} \langle \mathbf{skip} \rangle_{\text{P}} \quad (12)$$

We focus on (11). We need to apply the induction hypothesis, for which we first need to show that

$$\mu_{\text{V}}' \stackrel{\leq \nu_1}{\sim} \mu_{\text{P}}$$

We know from the premises that

$$\mu_{\text{V}}'' \stackrel{\leq \text{join}(\nu_1, \nu_2)}{\sim} \mu_{\text{P}} \quad (13)$$

Lemma 17 states

$$\nu_1 \preceq \text{join}(\nu_1, \nu_2) \quad (14)$$

Using this with Lemma 15 and (10) yields

$$\mu_{\text{V}}' \stackrel{\leq \nu_1}{\sim} \mu_{\text{V}}'' \quad (15)$$

Applying Lemma 13 on (13) and (14) leads to

$$\mu_v'' \stackrel{\leq \nu_1}{\approx} \mu_p$$

which, with Lemma 14 and (15), states

$$\mu_v' \stackrel{\leq \nu_1}{\approx} \mu_p$$

which is what we needed to prove to use the induction hypothesis, which yields our first subgoal (11). We now deal with the second subgoal (12). This is mostly a matter of applying Lemma 16 with the help of (14). □

Lemma 18. (`vmultistep_seq_fail`) *If*

$$\langle c_1; c_2, \mu_v \rangle_v \longrightarrow_v^* \text{failure}_v \mu_v''$$

then, either

$$\langle c_1, \mu_v \rangle_v \longrightarrow_v^* \text{failure}_v \mu_v''$$

or there exists a versioned store μ_v' such that

$$\begin{aligned} \langle c_1, \mu_v \rangle_v &\longrightarrow_v^* \langle \text{skip}, \mu_v' \rangle_v \\ \langle c_2, \mu_v' \rangle_v &\longrightarrow_v^* \text{failure}_v \mu_v'' \end{aligned}$$

Lemma 19. (`single_assignment_monotonic_store_fail`) *With c_{sa} be the SA-form of a (regular) program c with respect to a version map ν , if*

$$\langle c_{\text{sa}}, \mu_v \rangle_v \longrightarrow_v^* \text{failure}_v \mu_v'$$

then

$$\mu_v \stackrel{\leq \nu}{\approx} \mu_v'$$

Lemma 20. (`sync_vcommand_does_not_fail`) *For any set of identifiers I , version maps ν and ν' , and versioned stores μ_v and μ_v' ,*

$$\neg \langle \text{sync}(I, \nu, \nu'), \mu_v \rangle_v \longrightarrow_v^* \text{failure}_v \mu_v'$$

Theorem 4. (`vmultistep_multistep_fail`) *For any (regular) program c , version map ν and versioned stores μ_v and μ_v'' , let c_{sa} be the SA-transformation of c with respect to ν , then, if*

$$\langle c_{\text{sa}}, \mu_v \rangle_v \longrightarrow_v^* \text{failure}_v \mu_v''$$

then

$$\langle \text{passify}(c_{\text{sa}}) \rangle_p \xrightarrow{\mu_v''}^* \text{failure}_p$$

Proof. By structural induction on c :

- $c = \text{assert } e$: the SA-translation is

$$c_{\text{sa}} = \text{assert } e_v$$

The passified version is

$$c_p = \text{assert } e_v$$

The only execution path for

$$\langle \text{assert } e_v, \mu_v \rangle_v \longrightarrow_v^* \text{failure}_v \mu_v''$$

is

$$\langle \mathbf{assert} \ e_v, \mu_v \rangle_v \longrightarrow_v \mathbf{failure}_v \ \mu_v$$

Thus,

$$\mu_v = \mu_v''$$

and

$$e_v(\mu_v) \neq \mathbf{true}$$

We need to show that

$$\langle \mathbf{assert} \ e_v \rangle_p \xrightarrow[\text{p}]{\mu_v} \mathbf{failure}_p$$

This is clearly the case, as the same expression e_v is evaluated in the same versioned store μ_v and thus will yield the same value which we know is not equal to \mathbf{true} .

- $c = \mathbf{assume} \ e$: there is no path leading to failure.
- $c = \mathbf{skip}$: there is no path leading to failure.
- $c = x := e$: there is no path leading to failure.
- $c = c_1; c_2$: the SA-transformation gives us

$$\begin{aligned} (c_{1,\text{sa}}, \nu') &= \text{SA}_{\nu'}(c_1) \\ (c_{2,\text{sa}}, \nu'') &= \text{SA}_{\nu''}(c_2) \end{aligned}$$

Lemma 18 distinguishes two cases:

- $c_{1,\text{sa}}$ fails:

$$\langle c_{1,\text{sa}}, \mu_v \rangle_v \longrightarrow_v^* \mathbf{failure}_v \ \mu_v''$$

We directly apply the induction hypothesis.

- $c_{1,\text{sa}}$ skips, $c_{2,\text{sa}}$ fails, thus there exists a versioned store μ_v' such that

$$\langle c_{1,\text{sa}}, \mu_v \rangle_v \longrightarrow_v^* \langle \mathbf{skip}, \mu_v' \rangle_v \tag{16}$$

$$\langle c_{2,\text{sa}}, \mu_v' \rangle_v \longrightarrow_v^* \mathbf{failure}_v \ \mu_v'' \tag{17}$$

From the induction hypothesis we get

$$\langle \text{passify}(c_{2,\text{sa}}) \rangle_p \xrightarrow[\text{p}]{\mu_v''} \mathbf{failure}_p$$

Thus, we only need to prove that

$$\langle \text{passify}(c_{1,\text{sa}}) \rangle_p \xrightarrow[\text{p}]{\mu_v'} \mathbf{failure}_p$$

which follows from Theorem 3 if we can prove that

$$\mu_v' \stackrel{\leq \nu'}{\sim} \mu_v''$$

Applying Lemma 19 on (17) gives us this.

- $c = c_1 \parallel c_2$: the SA-transformation yields

$$c_{\text{sa}} = c_{1,\text{sa}}; \delta_1 \parallel c_{2,\text{sa}}; \delta_2$$

We know that

$$\langle c_{1,\text{sa}}; \delta_1 \parallel c_{2,\text{sa}}; \delta_2, \mu_v \rangle_v \longrightarrow_v^* \mathbf{failure}_v \ \mu_v''$$

This means that either

$$\langle c_{1,sa}; \delta_1, \mu_v \rangle_v \longrightarrow_v^* \text{failure}_v \mu''_v \quad (18)$$

or

$$\langle c_{2,sa}; \delta_2, \mu_v \rangle_v \longrightarrow_v^* \text{failure}_v \mu''_v \quad (19)$$

We only deal with the former case, the latter being similar. Using Lemma 18 with (18) we get that either

$$\langle c_{1,sa}, \mu_v \rangle_v \longrightarrow_v^* \text{failure}_v \mu''_v \quad (20)$$

or there exists some μ'_v such that

$$\langle \delta_1, \mu'_v \rangle_v \longrightarrow_v^* \text{failure}_v \mu''_v \quad (21)$$

From Lemma 20 we know that (21) cannot occur. Thus, we need to prove that, given (20),

$$\langle \text{passify}(c_{1,sa}; \delta_1 \parallel c_{2,sa}; \delta_2) \rangle_p \xrightarrow{\mu''_v}^* \text{failure}_p$$

which is implied by

$$\langle \text{passify}(c_{1,sa}; \delta_1) \parallel \text{passified}(c_{2,sa}; \delta_2) \rangle_p \xrightarrow{\mu''_v}^* \text{failure}_p$$

which is implied by

$$\langle \text{passify}(c_{1,sa}; \delta_1) \rangle_p \xrightarrow{\mu''_v}^* \text{failure}_p$$

which is implied by

$$\langle \text{passify}(c_{1,sa}); \text{passify}(\delta_1) \rangle_p \xrightarrow{\mu''_v}^* \text{failure}_p$$

which is implied by

$$\langle \text{passify}(c_{1,sa}) \rangle_p \xrightarrow{\mu''_v}^* \text{failure}_p$$

which is implied by the induction hypothesis.

□

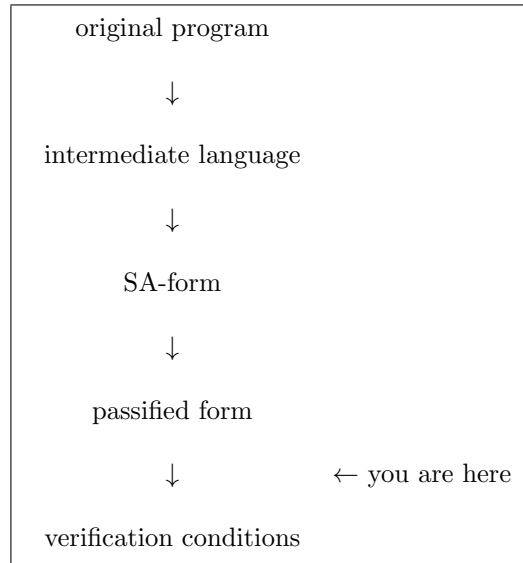


Figure 28: Steps in the verification algorithm

5 Weakest preconditions

Generating the verification conditions is the last step in the transformation chain. We set up two goals for the verification condition generation:

- they must be sound, i.e. their validity must imply program correctness;
- they must be efficient, i.e. their size must remain polynomial with respect to the original program size.

For more information on weakest preconditions, we refer the reader to [6, 4, 5, 7].

5.1 Stuck states

A stuck state is a state for which no reduction rule applies: execution cannot proceed. We distinguish three kinds of stuck states:

- Failure states: once execution reaches a failure state, there is no way out of it.
- Skip states: this is a state whose command component is **skip**.
- Nested assume states: a state where no reduction applies because an assumption expression does not evaluate to **true**.

5.2 Conservative and liberal

We can generate weakest preconditions for a certain program c with respect to a certain condition Q . Since this happens after SA-transformation and passification, we are only interested in generating weakest preconditions for passified programs. We distinguish two kinds of weakest preconditions, each making different guarantees:

- The *weakest conservative preconditions* of a program c with respect to a condition Q guarantee that their validity implies that, regardless of which (versioned) store μ_v is used⁹,

⁹We are using execution of passified programs, so the store is never altered and thus remain unchanged during the entire execution.

```

1 Fixpoint wp (vmu : vstore) (c : pcommand) (Q : Prop) :=
2   match c with
3     | pcAssert e      => e vmu = T /\ Q
4     | pcAssume e      => e vmu = T -> Q
5     | pcChoice c1 c2 => wp vmu c1 Q /\ wp vmu c2 Q
6     | pcSequence c1 c2 => wp vmu c1 (wp vmu c2 Q)
7     | pcSkip          => Q
8   end.

```

Figure 29: Weakest conservative preconditions in Coq

```

1 Fixpoint wlp (vmu : vstore) (c : pcommand) (Q : Prop) :=
2   match c with
3     | pcAssert e      => e vmu = T -> Q
4     | pcAssume e      => e vmu = T -> Q
5     | pcChoice c1 c2 => wlp vmu c1 Q /\ wlp vmu c2 Q
6     | pcSequence c1 c2 => wlp vmu c1 (wlp vmu c2 Q)
7     | pcSkip          => Q
8   end.

```

Figure 30: Weakest liberal preconditions in Coq

execution of c will never encounter failure. Also, if the program ends up in a skip state, the condition Q will be satisfied in the store μ_v .

- The *weakest liberal preconditions* of a program c with respect to a condition Q only guarantee that their validity implies that if execution in a store μ_v ends up in a skip state, the condition Q will hold in μ .

We will prove these facts in a later section. First, we define how these conditions are generated.

Definition 9 (Weakest conservative preconditions). (**wp**) *The weakest conservative preconditions of a passified program c with respect to a condition Q , written $\mathbf{wp}(c, Q)$ are defined as follows*

$$\begin{aligned}
\mathbf{wp}(\mathbf{assert} \ e, Q) &= e \wedge Q \\
\mathbf{wp}(\mathbf{assume} \ e, Q) &= e \Rightarrow Q \\
\mathbf{wp}(\mathbf{skip}, Q) &= Q \\
\mathbf{wp}(c_1; c_2, Q) &= \mathbf{wp}(c_1, \mathbf{wp}(c_2, Q)) \\
\mathbf{wp}(c_1 \parallel c_2, Q) &= \mathbf{wp}(c_1, Q) \wedge \mathbf{wp}(c_2, Q)
\end{aligned}$$

Definition 10 (Weakest liberal preconditions). (**wlp**) *The weakest liberal preconditions of a passified program c with respect to a condition Q , written $\mathbf{wlp}(c, Q)$ are defined as follows*

$$\begin{aligned}
\mathbf{wlp}(\mathbf{assert} \ e, Q) &= e \Rightarrow Q \\
\mathbf{wlp}(\mathbf{assume} \ e, Q) &= e \Rightarrow Q \\
\mathbf{wlp}(\mathbf{skip}, Q) &= Q \\
\mathbf{wlp}(c_1; c_2, Q) &= \mathbf{wlp}(c_1, \mathbf{wlp}(c_2, Q)) \\
\mathbf{wlp}(c_1 \parallel c_2, Q) &= \mathbf{wlp}(c_1, Q) \wedge \mathbf{wlp}(c_2, Q)
\end{aligned}$$

Since our goal is to prevent failure, the reader may wonder why we would need to weakest liberal preconditions. The answer to this question will come in Section 5.4.

5.3 Soundness of the weakest preconditions

Lemma 21. (`pstep_wp_prevents_failure`) For any passified command c , versioned store μ_v and proposition Q ,

$$\mu_v \models \mathbf{wp}(c, Q) \quad \Rightarrow \quad \neg \langle c \rangle_{\mathbf{p}} \xrightarrow{\mu_v}_{\mathbf{p}} \mathbf{failure}_{\mathbf{p}}$$

Lemma 22. (`pstep_wp_preservation`) For any passified commands c and c' , versioned store μ_v and proposition Q ,

$$\mu_v \models \mathbf{wp}(c, Q) \quad \Rightarrow \quad \langle c \rangle_{\mathbf{p}} \xrightarrow{\mu_v}_{\mathbf{p}} \langle c' \rangle_{\mathbf{p}} \quad \Rightarrow \quad \mu_v \models \mathbf{wp}(c', Q)$$

Lemma 23. (`pmultistep_wp_preservation`) For any passified commands c and c' , versioned store μ_v and proposition Q ,

$$\mu_v \models \mathbf{wp}(c, Q) \quad \Rightarrow \quad \langle c \rangle_{\mathbf{p}} \xrightarrow{\mu_v}_{\mathbf{p}}^* \langle c' \rangle_{\mathbf{p}} \quad \Rightarrow \quad \mu_v \models \mathbf{wp}(c', Q)$$

Theorem 5. (`pmultistep_wp_prevents_failure`) For any versioned store μ_v , passified command c and proposition Q , if

$$\mu_v \models \mathbf{wp}(c, Q) \quad \Rightarrow \quad \neg \langle c \rangle_{\mathbf{p}} \xrightarrow{\mu_v}_{\mathbf{p}}^* \mathbf{failure}_{\mathbf{p}}$$

Proof. By combining Lemma 21 and Lemma 23. □

Theorem 6. For any versioned store μ_v , passified command c and proposition Q , $i^{\mathbf{A}0}$

$$\mu_v \models \mathbf{wp}(c, Q) \quad \Rightarrow \quad \langle c \rangle_{\mathbf{p}} \xrightarrow{\mu_v}_{\mathbf{p}}^* \langle \mathbf{skip} \rangle_{\mathbf{p}} \quad \Rightarrow \quad Q$$

Proof. We append an assertion command to c :

$$c' = c; \mathbf{assert} \ Q$$

From Theorem 5 we know that execution of c' will not fail, which means that $\mathbf{assert} \ Q$ did not cause failure. Hence, Q must be true. □

5.4 Efficient weakest preconditions

If we take a closer look at the weakest conservative preconditions, we notice they grow exponentially because of the way the choice commands are handled. In this section, we show how the weakest preconditions can be rewritten so that they only grow polynomially. See [6, 4] for more information.

We know the weakest conservative preconditions $\mathbf{wp}(c, Q)$ guarantee two things:

- no failures will occur during execution of c ;
- skip-states end up in state satisfying Q .

The weakest liberal preconditions $\mathbf{wlp}(c, Q)$ guarantee only that

- skip-states end up in state satisfying Q .

We can split the weakest conservative preconditions' guarantees up as follows:

$$\mathbf{wp}(c, Q) \iff \underbrace{\mathbf{wp}(c, \mathbf{true})}_{\text{no failure}} \wedge \underbrace{\mathbf{wlp}(c, Q)}_{Q \text{ true at skip}}$$

¹⁰This theorem has not been proven in Coq, but it is not relied upon: we only provide this theorem for completeness.

Unfortunately, the weakest liberal preconditions suffer the same problem as the weakest conservative preconditions: they blow up exponentially, meaning they also need a rewrite. We can reformulate their guarantee as “either execution does not reach skip, or Q is true”.

$$\mathbf{wlp}(c, Q) \iff \underbrace{\mathbf{wlp}(c, \mathbf{false})}_{\text{no skip}} \vee Q$$

Combining these two rewrites yields

$$\mathbf{wp}(c, Q) \iff \mathbf{wp}(c, \mathbf{true}) \wedge (\mathbf{wlp}(c, \mathbf{false}) \vee Q)$$

This rewrite is only valid for passified programs, hence the need for the transformations. We now set out to prove these equivalences formally.

Lemma 24. (*monotonic_wlp*) *If*

$$Q \Rightarrow R$$

and

$$\mu_v \models \mathbf{wlp}(c, Q)$$

then

$$\mu_v \models \mathbf{wlp}(c, R)$$

Lemma 25. (*Q_impl_wlpQ*)

$$\mu_v \models Q \Rightarrow \mu_v \models \mathbf{wlp}(c, Q)$$

Theorem 7. (*wlp_rewrite*)

$$\mu_v \models \mathbf{wlp}(c, Q) \iff \mu_v \models \mathbf{wlp}(c, \mathbf{false}) \vee Q$$

Proof. We first focus on the left-to-right implication:

$$\mu_v \models \mathbf{wlp}(c, Q) \Rightarrow \mu_v \models \mathbf{wlp}(c, \mathbf{false}) \vee Q$$

By structural induction on c :

- $c = \mathbf{assert} e_v$: we need to prove that

$$\mu_v \models \mathbf{wlp}(\mathbf{assert} e_v, Q) \Rightarrow \mu_v \models \mathbf{wlp}(\mathbf{assert} e_v, \mathbf{false}) \vee Q$$

which becomes

$$\mu_v \models (e_v \Rightarrow Q) \Rightarrow \mu_v \models (e_v \Rightarrow \mathbf{false}) \vee Q$$

Either e_v is true, in which case Q is implied, or, e_v is false, in which case $e_v \Rightarrow \mathbf{false}$ is true.

- $c = \mathbf{assume} e_v$: equivalent to the previous case.
- $c = \mathbf{skip}$: we need to prove that

$$\mu_v \models \mathbf{wlp}(\mathbf{skip}, Q) \Rightarrow \mu_v \models \mathbf{wlp}(\mathbf{skip}, \mathbf{false}) \vee Q$$

which is equivalent with

$$\mu_v \models Q \Rightarrow \mu_v \models \mathbf{false} \vee Q$$

which is clearly true.

- $c = c_1; c_2$: we need to prove that

$$\mu_v \models \mathbf{wlp}(c_1; c_2, Q) \quad \Rightarrow \quad \mu_v \models (\mathbf{wlp}(c_1; c_2, \mathbf{false}) \vee Q)$$

which unfolds to

$$\mu_v \models \mathbf{wlp}(c_1, \mathbf{wlp}(c_2, Q)) \quad \Rightarrow \quad \mu_v \models (\mathbf{wlp}(c_1, \mathbf{wlp}(c_2, \mathbf{false})) \vee Q)$$

From the induction theorem, taking $Q = \mathbf{wlp}(c_2, Q)$, we know

$$\mu_v \models (\mathbf{wlp}(c_1, \mathbf{wlp}(c_2, Q))) \quad \Rightarrow \quad \mu_v \models (\mathbf{wlp}(c_1, \mathbf{false}) \vee \mathbf{wlp}(c_2, Q))$$

Our goal becomes

$$\mu_v \models (\mathbf{wlp}(c_1, \mathbf{false}) \vee \mathbf{wlp}(c_2, Q)) \quad \Rightarrow \quad \mu_v \models (\mathbf{wlp}(c_1, \mathbf{wlp}(c_2, \mathbf{false})) \vee Q)$$

We split this up into two subgoals:

- Subgoal 1:

$$\mu_v \models \mathbf{wlp}(c_1, \mathbf{false}) \quad \Rightarrow \quad \mu_v \models (\mathbf{wlp}(c_1, \mathbf{wlp}(c_2, \mathbf{false})) \vee Q)$$

We focus on the disjunction's left operand. We know that

$$\mathbf{false} \Rightarrow \mathbf{wlp}(c_2, \mathbf{false})$$

Lemma 24 then gives us

$$\mu_v \models \mathbf{wlp}(c_1, \mathbf{false}) \quad \Rightarrow \quad \mu_v \models \mathbf{wlp}(c_1, \mathbf{wlp}(c_2, \mathbf{false}))$$

- Subgoal 2:

$$\mu_v \models \mathbf{wlp}(c_2, Q) \quad \Rightarrow \quad \mu_v \models (\mathbf{wlp}(c_1, \mathbf{wlp}(c_2, \mathbf{false})) \vee Q)$$

From the induction hypothesis:

$$\mu_v \models \mathbf{wlp}(c_2, Q) \quad \Rightarrow \quad \mu_v \models \mathbf{wlp}(c_2, \mathbf{false}) \vee Q$$

Our new goal becomes

$$\mu_v \models (\mathbf{wlp}(c_2, \mathbf{false}) \vee Q) \quad \Rightarrow \quad \mu_v \models (\mathbf{wlp}(c_1, \mathbf{wlp}(c_2, \mathbf{false})) \vee Q)$$

Again, we split up in two subgoals:

- * Subgoal 2a:

$$\mu_v \models \mathbf{wlp}(c_2, \mathbf{false}) \quad \Rightarrow \quad \mu_v \models (\mathbf{wlp}(c_1, \mathbf{wlp}(c_2, \mathbf{false})) \vee Q)$$

which is implied by

$$\mu_v \models \mathbf{wlp}(c_2, \mathbf{false}) \quad \Rightarrow \quad \mu_v \models \mathbf{wlp}(c_1, \mathbf{wlp}(c_2, \mathbf{false}))$$

which is given by Lemma 25.

- * Subgoal 2b:

$$\mu_v \models Q \quad \Rightarrow \quad \mu_v \models (\mathbf{wlp}(c_1, \mathbf{wlp}(c_2, \mathbf{false})) \vee Q)$$

Trivially true.

- $c = c_1 \parallel c_2$: we need to prove that

$$\mu_v \models \mathbf{wlp}(c_1 \parallel c_2, Q) \Rightarrow \mu_v \models (\mathbf{wlp}(c_1 \parallel c_2, \mathbf{false}) \vee Q)$$

which unfolds to

$$\mu_v \models (\mathbf{wlp}(c_1, Q) \wedge \mathbf{wlp}(c_2, Q)) \Rightarrow \mu_v \models ((\mathbf{wlp}(c_1, \mathbf{false}) \wedge \mathbf{wlp}(c_2, \mathbf{false})) \vee Q)$$

From the induction hypothesis we know

$$\mu_v \models \mathbf{wlp}(c_1, Q) \Rightarrow \mu_v \models (\mathbf{wlp}(c_1, \mathbf{false}) \vee Q)$$

$$\mu_v \models \mathbf{wlp}(c_2, Q) \Rightarrow \mu_v \models (\mathbf{wlp}(c_2, \mathbf{false}) \vee Q)$$

which transforms our goal to

$$\begin{aligned} \mu_v \models ((\mathbf{wlp}(c_1, \mathbf{false}) \vee Q) \wedge (\mathbf{wlp}(c_2, \mathbf{false}) \vee Q)) \\ \Downarrow \\ \mu_v \models ((\mathbf{wlp}(c_1, \mathbf{false}) \wedge \mathbf{wlp}(c_2, \mathbf{false})) \vee Q) \end{aligned}$$

We can split up

$$\mu_v \models ((\mathbf{wlp}(c_1, \mathbf{false}) \vee Q) \wedge (\mathbf{wlp}(c_2, \mathbf{false}) \vee Q))$$

in four cases, each of which is trivially true.

We now focus on the right-to-left implication:

$$\mu_v \models (\mathbf{wlp}(c, \mathbf{false}) \vee Q) \Rightarrow \mu_v \models \mathbf{wlp}(c, Q)$$

By structural induction on c :

- $c = \mathbf{assert} e_v$: we need to prove

$$\mu_v \models (\mathbf{wlp}(\mathbf{assert} e_v, \mathbf{false}) \vee Q) \Rightarrow \mu_v \models \mathbf{wlp}(\mathbf{assert} e_v, Q)$$

We split this up in two subgoals:

- Subgoal 1:

$$\mu_v \models \mathbf{wlp}(\mathbf{assert} e_v, \mathbf{false}) \Rightarrow \mu_v \models \mathbf{wlp}(\mathbf{assert} e_v, Q)$$

Combining

$$\mathbf{false} \Rightarrow Q$$

with Lemma 24 leads directly to the goal.

- Subgoal 2:

$$\mu_v \models Q \Rightarrow \mu_v \models \mathbf{wlp}(\mathbf{assert} e_v, Q)$$

Follows directly from Lemma 25.

- $c = \mathbf{assume} e_v$: equivalent to the previous case.
- $c = \mathbf{skip}$: we need to show that

$$\mu_v \models (\mathbf{wlp}(\mathbf{skip}, \mathbf{false}) \vee Q) \Rightarrow \mu_v \models \mathbf{wlp}(\mathbf{skip}, Q)$$

which unfolds to

$$\mu_v \models (\mathbf{false} \vee Q) \Rightarrow \mu_v \models Q$$

which is trivially true.

- $c = c_1; c_2$: we need to prove that

$$\mu_v \models (\mathbf{wlp}(c_1; c_2, \mathbf{false}) \vee Q) \Rightarrow \mu_v \models \mathbf{wlp}(c_1; c_2, Q)$$

We distinguish two subgoals:

- Subgoal 1:

$$\mu_v \models \mathbf{wlp}(c_1; c_2, \mathbf{false}) \Rightarrow \mu_v \models \mathbf{wlp}(c_1; c_2, Q)$$

By applying Lemma 24 with

$$\mathbf{false} \Rightarrow Q$$

- Subgoal 2:

$$\mu_v \models Q \Rightarrow \mu_v \models \mathbf{wlp}(c_1; c_2, Q)$$

By applying Lemma 25.

- $c = c_1 \parallel c_2$: we need to prove that

$$\mu_v \models (\mathbf{wlp}(c_1 \parallel c_2, \mathbf{false}) \vee Q) \Rightarrow \mu_v \models \mathbf{wlp}(c_1 \parallel c_2, Q)$$

We distinguish two subgoals:

- Subgoal 1:

$$\mu_v \models \mathbf{wlp}(c_1 \parallel c_2, \mathbf{false}) \Rightarrow \mu_v \models \mathbf{wlp}(c_1 \parallel c_2, Q)$$

By applying Lemma 24 with

$$\mathbf{false} \Rightarrow Q$$

- Subgoal 2:

$$\mu_v \models Q \Rightarrow \mu_v \models \mathbf{wlp}(c_1 \parallel c_2, Q)$$

By applying Lemma 25.

□

Lemma 26. (*wp_true*)

$$\mu_v \models \mathbf{wp}(c, Q) \Rightarrow \mu_v \models \mathbf{wp}(c, \mathbf{true})$$

Lemma 27. (*wp_impl_wlp*)

$$\mu_v \models \mathbf{wp}(c, Q) \Rightarrow \mu_v \models \mathbf{wlp}(c, Q)$$

Lemma 28. (*conjunctive_wp*)

$$\mu_v \models \mathbf{wp}(c, Q) \wedge \mathbf{wp}(c, R) \iff \mu_v \models \mathbf{wp}(c, Q \wedge R)$$

Lemma 29. (*monotonic_wp*) *If*

$$Q \Rightarrow R$$

and

$$\mu_v \models \mathbf{wp}(c, Q)$$

then

$$\mu_v \models \mathbf{wp}(c, R)$$

Theorem 8. (*wp_rewrite*)

$$\mu_v \models \mathbf{wp}(c, Q) \iff \mu_v \models (\mathbf{wp}(c, \mathbf{true}) \wedge \mathbf{wlp}(c, Q))$$

Proof. We first deal with the left-to-right implication:

$$\mu_v \models \mathbf{wp}(c, Q) \Rightarrow \mu_v \models (\mathbf{wp}(c, \mathbf{true}) \wedge \mathbf{wlp}(c, Q))$$

This follows directly from Lemma 26 and Lemma 27.

We now deal with the right-to-left implication.

$$\mu_v \models (\mathbf{wp}(c, \mathbf{true}) \wedge \mathbf{wlp}(c, Q)) \Rightarrow \mu_v \models \mathbf{wp}(c, Q)$$

By structural induction on c .

- $c = \mathbf{assert} e_v$: we need to show that

$$\mu_v \models (\mathbf{wp}(\mathbf{assert} e_v, \mathbf{true}) \wedge \mathbf{wlp}(\mathbf{assert} e_v, Q)) \Rightarrow \mu_v \models \mathbf{wp}(\mathbf{assert} e_v, Q)$$

which unfolds to

$$\mu_v \models (e_v \wedge \mathbf{true} \wedge (e_v \Rightarrow Q)) \Rightarrow \mu_v \models (e_v \wedge Q)$$

which is clearly true.

- $c = \mathbf{assume} e_v$

$$\mu_v \models (\mathbf{wp}(\mathbf{assume} e_v, \mathbf{true}) \wedge \mathbf{wlp}(\mathbf{assume} e_v, Q)) \Rightarrow \mu_v \models \mathbf{wp}(\mathbf{assume} e_v, Q)$$

which is equivalent with

$$\mu_v \models ((e_v \Rightarrow \mathbf{true}) \wedge (e_v \Rightarrow Q)) \Rightarrow \mu_v \models \mathbf{wp}(e_v \Rightarrow Q)$$

Trivial.

- $c = \mathbf{skip}$

$$\mu_v \models (\mathbf{wp}(\mathbf{skip}, \mathbf{true}) \wedge \mathbf{wlp}(\mathbf{skip}, Q)) \Rightarrow \mu_v \models \mathbf{wp}(\mathbf{skip}, Q)$$

which is implied by

$$\mu_v \models (\mathbf{true} \wedge Q) \Rightarrow \mu_v \models Q$$

Trivial.

- $c = c_1; c_2$

$$\mu_v \models (\mathbf{wp}(c_1; c_2, \mathbf{true}) \wedge \mathbf{wlp}(c_1; c_2, Q)) \Rightarrow \mu_v \models \mathbf{wp}(c_1; c_2, Q)$$

which unfolds to

$$\mu_v \models (\mathbf{wp}(c_1, \mathbf{wp}(c_2, \mathbf{true})) \wedge \mathbf{wlp}(c_1, \mathbf{wlp}(c_2, Q))) \Rightarrow \mu_v \models \mathbf{wp}(c_1, \mathbf{wp}(c_2, Q))$$

Thus, assuming

$$\mu_v \models \mathbf{wp}(c_1, \mathbf{wp}(c_2, \mathbf{true})) \tag{22}$$

$$\mu_v \models \mathbf{wlp}(c_1, \mathbf{wlp}(c_2, Q)) \tag{23}$$

we need to prove

$$\mu_v \models \mathbf{wp}(c_1, \mathbf{wp}(c_2, Q))$$

From Lemma 26 and (22) we know

$$\mu_v \models \mathbf{wp}(c_1, \mathbf{true})$$

which we can use with the induction hypothesis and (23) to get

$$\mu_v \models \mathbf{wp}(c_1, \mathbf{wlp}(c_2, Q)) \tag{24}$$

Using Lemma 28, (22) and (24) yields

$$\mu_v \models \mathbf{wp}(c_1, \mathbf{wp}(c_2, \mathbf{true})) \wedge \mathbf{wlp}(c_2, Q)$$

From the induction hypothesis we know

$$\mu_v \models \mathbf{wp}(c_2, \mathbf{true}) \wedge \mathbf{wlp}(c_2, Q) \quad \Rightarrow \quad \mu_v \models \mathbf{wp}(c_2, Q)$$

Using this with Lemma 29 ends this part of the proof.

- $c = c_1 \parallel c_2$

$$\mu_v \models (\mathbf{wp}(c_1 \parallel c_2, \mathbf{true}) \wedge \mathbf{wlp}(c_1 \parallel c_2, Q)) \quad \Rightarrow \quad \mu_v \models \mathbf{wp}(c_1 \parallel c_2, Q)$$

This expands into

$$\begin{aligned} \mu_v \models (\mathbf{wp}(c_1, \mathbf{true}) \wedge \mathbf{wp}(c_2, \mathbf{true}) \wedge \mathbf{wlp}(c_1, Q) \wedge \mathbf{wlp}(c_2, Q)) \\ \Downarrow \\ \mu_v \models (\mathbf{wp}(c_1, Q) \wedge \mathbf{wp}(c_2, Q)) \end{aligned}$$

We can prove this by applying the induction hypothesis:

$$\mu_v \models (\mathbf{wp}(c_1, \mathbf{true}) \wedge \mathbf{wlp}(c_1, Q)) \quad \Rightarrow \quad \mu_v \models \mathbf{wp}(c_1, Q)$$

$$\mu_v \models (\mathbf{wp}(c_2, \mathbf{true}) \wedge \mathbf{wlp}(c_2, Q)) \quad \Rightarrow \quad \mu_v \models \mathbf{wp}(c_2, Q)$$

Making the conjunction of both ends the proof. □

We are now ready to define the efficient equivalents of the conservative and liberal weakest preconditions.

Definition 11 (Efficient weakest conservative preconditions). (**efficient_wp**) *The efficient weakest conservative preconditions of a passified program c with respect to a condition Q , written $\mathbf{wp}_e(c, Q)$ are defined as follows*

$$\begin{aligned} \mathbf{wp}_e(\mathbf{assert } e, Q) &= e \wedge Q \\ \mathbf{wp}_e(\mathbf{assume } e, Q) &= e \Rightarrow Q \\ \mathbf{wp}_e(\mathbf{skip}, Q) &= Q \\ \mathbf{wp}_e(c_1; c_2, Q) &= \mathbf{wp}_e(c_1, \mathbf{wp}_e(c_2, Q)) \\ \mathbf{wp}_e(c_1 \parallel c_2, Q) &= \mathbf{wp}_e(c_1, \mathbf{true}) \wedge \mathbf{wp}_e(c_2, \mathbf{true}) \wedge \mathbf{wlp}_e(c_1 \parallel c_2, Q) \end{aligned}$$

Definition 12 (Efficient weakest liberal preconditions). (**efficient_wlp**) *The efficient weakest liberal preconditions of a passified program c with respect to a condition Q , written $\mathbf{wlp}_e(c, Q)$ are defined as follows*

$$\begin{aligned} \mathbf{wlp}_e(\mathbf{assert } e, Q) &= e \Rightarrow Q \\ \mathbf{wlp}_e(\mathbf{assume } e, Q) &= e \Rightarrow Q \\ \mathbf{wlp}_e(\mathbf{skip}, Q) &= Q \\ \mathbf{wlp}_e(c_1; c_2, Q) &= \mathbf{wlp}_e(c_1, \mathbf{wlp}_e(c_2, Q)) \\ \mathbf{wlp}_e(c_1 \parallel c_2, Q) &= (\mathbf{wlp}_e(c_1, \mathbf{false}) \wedge \mathbf{wlp}_e(c_2, \mathbf{false})) \vee Q \end{aligned}$$

Theorem 9. (**efficient_wlp_equivalence**) *The weakest liberal preconditions and efficient weakest liberal preconditions are equivalent.*

$$\mu_v \models \mathbf{wlp}(c, Q) \quad \Longleftrightarrow \quad \mu_v \models \mathbf{wlp}_e(c, Q)$$

```

1  Fixpoint efficient_wlp (vmu : vstore)
2      (c : pcommand)
3      (Q : Prop) : Prop :=
4      match c with
5      | pcAssert e      => e vmu = T -> Q
6      | pcAssume e      => e vmu = T -> Q
7      | pcSequence c1 c2 =>
8          efficient_wlp vmu c1 (efficient_wlp vmu c2 Q)
9      | pcSkip          => Q
10     | pcChoice c1 c2  =>
11         (efficient_wlp vmu c1 False /\
12          efficient_wlp vmu c2 False) \/ Q
13     end.

14 Fixpoint efficient_wp (vmu : vstore)
15     (c : pcommand)
16     (Q : Prop) : Prop :=
17     match c with
18     | pcAssert e      => e vmu = T /\ Q
19     | pcAssume e      => e vmu = T -> Q
20     | pcSequence c1 c2 =>
21         efficient_wp vmu c1 (efficient_wp vmu c2 Q)
22     | pcSkip          => Q
23     | pcChoice c1 c2  =>
24         efficient_wp vmu c1 True /\
25         efficient_wp vmu c2 True /\
26         efficient_wlp vmu (pcChoice c1 c2) Q
27     end.

```

Figure 31: Efficient weakest preconditions in Coq

Proof. For the left-to-right arrow: by structural induction on c . We only consider the nontrivial cases.

- $c = c_1; c_2$: we need to prove

$$\mu_v \models \mathbf{wlp}(c_1, \mathbf{wlp}(c_2, Q)) \Rightarrow \mu_v \models \mathbf{wlp}_e(c_1, \mathbf{wlp}_e(c_2, Q))$$

From the induction hypothesis we know

$$\mu_v \models \mathbf{wlp}(c_2, Q) \Rightarrow \mu_v \models \mathbf{wlp}_e(c_2, Q)$$

Combining this with Lemma 24 yields

$$\mu_v \models \mathbf{wlp}(c_1, \mathbf{wlp}_e(c_2, Q))$$

The induction hypothesis transforms this to

$$\mu_v \models \mathbf{wlp}_e(c_1, \mathbf{wlp}_e(c_2, Q))$$

- $c = c_1 \parallel c_2$: we need to prove

$$\mu_v \models \mathbf{wlp}(c_1 \parallel c_2, Q) \Rightarrow \mu_v \models \mathbf{wlp}_e(c_1 \parallel c_2, Q)$$

which unfolds to

$$\mu_v \models (\mathbf{wlp}(c_1, Q) \wedge \mathbf{wlp}(c_2, Q)) \Rightarrow \mu_v \models ((\mathbf{wlp}_e(c_1, \mathbf{false}) \wedge \mathbf{wlp}_e(c_2, \mathbf{false})) \vee Q)$$

With the help from Theorem 7 we get

$$\mu_v \models \mathbf{wlp}(c_1, \mathbf{false}) \vee Q$$

$$\mu_v \models \mathbf{wlp}(c_2, \mathbf{false}) \vee Q$$

Using these with the induction hypothesis gets us

$$\mu_v \models \mathbf{wlp}_e(c_1, \mathbf{false}) \vee Q$$

$$\mu_v \models \mathbf{wlp}_e(c_2, \mathbf{false}) \vee Q$$

which clearly implies the goal.

For the right-to-left arrow: by structural induction on c , skipping the trivial cases we have

- $c = c_1; c_2$: we need to prove

$$\mu_v \models \mathbf{wlp}_e(c_1, \mathbf{wlp}_e(c_2, Q)) \Rightarrow \mu_v \models \mathbf{wlp}(c_1, \mathbf{wlp}(c_2, Q))$$

From the induction hypothesis we first get

$$\mu_v \models \mathbf{wlp}(c_1, \mathbf{wlp}_e(c_2, Q))$$

and also

$$\mu_v \models \mathbf{wlp}_e(c_2, Q) \Rightarrow \mu_v \models \mathbf{wlp}(c_2, Q)$$

Combining both using Lemma 24 yields the goal.

- $c = c_1 \parallel c_2$: we need to prove

$$\mu_v \models \mathbf{wlp}_e(c_1 \parallel c_2, Q) \Rightarrow \mu_v \models \mathbf{wlp}(c_1 \parallel c_2, Q)$$

Thus, by unfolding, we can assume that

$$\mu_v \models ((\mathbf{wlp}_e(c_1, \mathbf{false}) \wedge \mathbf{wlp}_e(c_2, \mathbf{false})) \vee Q) \quad (25)$$

and we need to show that

$$\mu_v \models (\mathbf{wlp}(c_1, Q) \wedge \mathbf{wlp}(c_2, Q))$$

From Theorem 7 we know

$$\mu_v \models (\mathbf{wlp}(c_1, \mathbf{false}) \vee Q) \Rightarrow \mu_v \models \mathbf{wlp}(c_1, Q) \quad (26)$$

$$\mu_v \models (\mathbf{wlp}(c_2, \mathbf{false}) \vee Q) \Rightarrow \mu_v \models \mathbf{wlp}(c_2, Q) \quad (27)$$

From the induction hypothesis

$$\mu_v \models \mathbf{wlp}_e(c_1, \mathbf{false}) \Rightarrow \mu_v \models \mathbf{wlp}(c_1, \mathbf{false}) \quad (28)$$

$$\mu_v \models \mathbf{wlp}_e(c_2, \mathbf{false}) \Rightarrow \mu_v \models \mathbf{wlp}(c_2, \mathbf{false}) \quad (29)$$

From (25) we distinguish two subcases:

- Subgoal 1:

$$\mu_v \models (\mathbf{wlp}_e(c_1, \mathbf{false}) \wedge \mathbf{wlp}_e(c_2, \mathbf{false})) \Rightarrow \mu_v \models (\mathbf{wlp}(c_1, Q) \wedge \mathbf{wlp}(c_2, Q))$$

Using (28) and (29) followed by (26) and (27) finishes this part of the proof.

- Subgoal 2:

$$\mu_v \models Q \Rightarrow \mu_v \models (\mathbf{wlp}(c_1, Q) \wedge \mathbf{wlp}(c_2, Q))$$

which we can easily prove by applying (26) and (27).

□

Theorem 10. (efficient_wp_equivalence) *The weakest conservative preconditions and efficient weakest conservative preconditions are equivalent.*

$$\mu_v \models \mathbf{wp}(c, Q) \iff \mu_v \models \mathbf{wp}_e(c, Q)$$

Proof. Left to right implication: by structural induction on c , we skip the trivial cases.

- $c = c_1; c_2$: we need to show that

$$\mu_v \models \mathbf{wp}(c_1; c_2, Q) \Rightarrow \mu_v \models \mathbf{wp}_e(c_1; c_2, Q)$$

which unfolds to

$$\mu_v \models \mathbf{wp}(c_1, \mathbf{wp}(c_2, Q)) \Rightarrow \mu_v \models \mathbf{wp}_e(c_1, \mathbf{wp}_e(c_2, Q))$$

From the induction hypothesis:

$$\mu_v \models \mathbf{wp}(c_2, Q) \Rightarrow \mu_v \models \mathbf{wp}_e(c_2, Q)$$

With the help of lemma Lemma 29 our new goal becomes:

$$\mu_v \models \mathbf{wp}(c_1, \mathbf{wp}_e(c_2, Q)) \Rightarrow \mu_v \models \mathbf{wp}_e(c_1, \mathbf{wp}_e(c_2, Q))$$

which is proven by applying the induction hypothesis.

- $c = c_1 \parallel c_2$: we need to prove that

$$\mu_v \models \mathbf{wp}(c_1 \parallel c_2, Q) \quad \Rightarrow \quad \mu_v \models \mathbf{wp}_e(c_1 \parallel c_2, Q)$$

which unfolds to

$$\begin{aligned} \mu_v \models (\mathbf{wp}(c_1, Q) \wedge \mathbf{wp}(c_2, Q)) \\ \Downarrow \\ \mu_v \models (\mathbf{wp}_e(c_1, \mathbf{true}) \wedge \mathbf{wp}_e(c_2, \mathbf{true}) \wedge ((\mathbf{wlp}_e(c_1, \mathbf{false}) \wedge \mathbf{wlp}_e(c_2, \mathbf{false})) \vee Q)) \end{aligned}$$

From Theorem 8

$$\mu_v \models \mathbf{wp}(c_1, Q) \quad \Rightarrow \quad \mu_v \models (\mathbf{wp}(c_1, \mathbf{true}) \wedge \mathbf{wlp}(c_1, Q)) \quad (30)$$

$$\mu_v \models \mathbf{wp}(c_2, Q) \quad \Rightarrow \quad \mu_v \models (\mathbf{wp}(c_2, \mathbf{true}) \wedge \mathbf{wlp}(c_2, Q)) \quad (31)$$

Our goal with extended hypotheses becomes

$$\begin{aligned} \mu_v \models (\mathbf{wp}(c_1, Q) \wedge \mathbf{wp}(c_2, Q) \wedge \mathbf{wp}(c_1, \mathbf{true}) \wedge \mathbf{wlp}(c_1, Q) \wedge \mathbf{wp}(c_2, \mathbf{true}) \wedge \mathbf{wlp}(c_2, Q)) \\ \Downarrow \\ \mu_v \models (\mathbf{wp}_e(c_1, \mathbf{true}) \wedge \mathbf{wp}_e(c_2, \mathbf{true}) \wedge ((\mathbf{wlp}_e(c_1, \mathbf{false}) \wedge \mathbf{wlp}_e(c_2, \mathbf{false})) \vee Q)) \end{aligned}$$

We split up the goal in three subgoals (dropping unneeded hypotheses for clarity)

- Subgoal 1:

$$\begin{aligned} \mu_v \models \mathbf{wp}(c_1, \mathbf{true}) \\ \Downarrow \\ \mu_v \models \mathbf{wp}_e(c_1, \mathbf{true}) \end{aligned}$$

Follows directly from the induction hypothesis.

- Subgoal 2:

$$\begin{aligned} \mu_v \models \mathbf{wp}(c_2, \mathbf{true}) \\ \Downarrow \\ \mu_v \models \mathbf{wp}_e(c_2, \mathbf{true}) \end{aligned}$$

Follows directly from the induction hypothesis.

- Subgoal 3:

$$\begin{aligned} \mu_v \models (\mathbf{wlp}(c_1, Q) \wedge \mathbf{wlp}(c_2, Q)) \\ \Downarrow \\ \mu_v \models ((\mathbf{wlp}_e(c_1, \mathbf{false}) \wedge \mathbf{wlp}_e(c_2, \mathbf{false})) \vee Q) \end{aligned}$$

From Theorem 7 we know

$$\mu_v \models \mathbf{wlp}(c_1, Q) \quad \Rightarrow \quad \mu_v \models (\mathbf{wlp}(c_1, \mathbf{false}) \vee Q)$$

$$\mu_v \models \mathbf{wlp}(c_2, Q) \quad \Rightarrow \quad \mu_v \models (\mathbf{wlp}(c_2, \mathbf{false}) \vee Q)$$

Thus, our new proof state becomes

$$\begin{aligned} \mu_v \models ((\mathbf{wlp}(c_1, \mathbf{false}) \vee Q) \wedge (\mathbf{wlp}(c_2, \mathbf{false}) \vee Q)) \\ \Downarrow \\ \mu_v \models ((\mathbf{wlp}_e(c_1, \mathbf{false}) \wedge \mathbf{wlp}_e(c_2, \mathbf{false})) \vee Q) \end{aligned}$$

From Theorem 9 we know

$$\mu_v \models \mathbf{wlp}(c_1, \mathbf{false}) \quad \Rightarrow \quad \mu_v \models \mathbf{wlp}_e(c_1, \mathbf{false})$$

$$\mu_v \models \mathbf{wlp}(c_2, \mathbf{false}) \quad \Rightarrow \quad \mu_v \models \mathbf{wlp}_e(c_2, \mathbf{false})$$

which transforms the proof state into

$$\begin{aligned} \mu_v \models ((\mathbf{wlp}_e(c_1, \mathbf{false}) \vee Q) \wedge (\mathbf{wlp}_e(c_2, \mathbf{false}) \vee Q)) \\ \Downarrow \\ \mu_v \models ((\mathbf{wlp}_e(c_1, \mathbf{false}) \wedge \mathbf{wlp}_e(c_2, \mathbf{false})) \vee Q) \end{aligned}$$

which is clearly true.

Right to left implication:

$$\mu_v \models \mathbf{wp}_e(c, Q) \Rightarrow \mu_v \models \mathbf{wp}(c, Q)$$

By structural induction on c , skipping the trivial cases:

- $c = c_1; c_2$: after unfolding, we need to prove that

$$\mu_v \models \mathbf{wp}_e(c_1, \mathbf{wp}_e(c_2, Q)) \Rightarrow \mu_v \models \mathbf{wp}(c_1, \mathbf{wp}(c_2, Q))$$

From the induction hypothesis we get

$$\mu_v \models \mathbf{wp}_e(c_2, Q) \Rightarrow \mu_v \models \mathbf{wp}(c_2, Q)$$

$$\mu_v \models \mathbf{wp}_e(c_1, \mathbf{wp}_e(c_2, Q)) \Rightarrow \mu_v \models \mathbf{wp}(c_1, \mathbf{wp}_e(c_2, Q))$$

Using Lemma 29 we are able to prove the goal.

- $c = c_1 \parallel c_2$: a combination of applying Theorem 7, Theorem 8 and Theorem 9.

□

5.5 Soundness

We are finally able to put all the pieces of the puzzle together to prove the soundness of the verification condition generation algorithm we use. We have proven the following theorems:

- Theorem 2: For any command c , stores μ and μ' , versioned store μ_v and version map ν , let $(c_{sa}, \nu') = \text{SA}_\nu(c, \nu)$, then, if

$$\langle c, \mu \rangle \longrightarrow^* \text{failure}_v \mu'$$

and

$$\mu \sim^\nu \mu_v$$

then there exists some μ'_v for which

$$\langle c', \mu_v \rangle_v \longrightarrow_v^* \text{failure}_v \mu'_v$$

- Theorem 4: For any (regular) program c , version map ν and versioned stores μ_v and μ'_v , let c_{sa} be the SA-transformation of c with respect to ν , then, if

$$\langle c_{sa}, \mu_v \rangle_v \longrightarrow_v^* \text{failure}_v \mu'_v$$

then

$$\langle \text{passify}(c_{sa}) \rangle_p \xrightarrow{\mu'_v}^* \text{failure}_p$$

- Theorem 5: For any versioned store μ_v , passified command c and proposition Q , if

$$\mu_v \models \mathbf{wp}(c, Q) \Rightarrow \neg \langle c \rangle_p \xrightarrow{\mu_v}^* \text{failure}_p$$

- Theorem 10: The weakest conservative preconditions and efficient weakest conservative preconditions are equivalent.

$$\mu_v \models \mathbf{wp}(c, Q) \iff \mu_v \models \mathbf{wp}_e(c, Q)$$

We now chain them together into one main soundness theorem.

Definition 13. (`init_vmap`) We define the initial version map ν_0 as

$$\lambda id.0$$

Lemma 30. (`versioned_store_exists`) *For any store μ and version map ν there exists a versioned store μ_ν so that*

$$\mu \sim^\nu \mu_\nu$$

Theorem 11. (`soundness_efficient_wp`) *For any (regular) program c , let c_{sa} be its single-assignment form with respect to ν_0 and let c_p be c_{sa} 's passification, then, if for all versioned stores μ_ν ,*

$$\mu_\nu \models \mathbf{wp}_e(c_p, \mathbf{true})$$

then, for all stores μ there does not exist a store μ' so that

$$\langle c, \mu \rangle \longrightarrow^* \text{failure } \mu'$$

Proof. We show that if

$$\langle c, \mu \rangle \longrightarrow^* \text{failure } \mu'$$

we encounter a contradiction. We know from Lemma 30 that there exists a μ_ν so that

$$\mu \sim^{\nu_0} \mu_\nu$$

From Theorem 2 we know that there exists a versioned store μ'_ν so that

$$\langle c_{\text{sa}}, \mu_\nu \rangle_\nu \longrightarrow^*_{\nu} \text{failure}_\nu \mu'_\nu$$

From Theorem 4 we get

$$\langle c_p \rangle_p \xrightarrow{\mu'_\nu}_{p}^* \text{failure}_p \quad (32)$$

Theorem 10 yields

$$\mu'_\nu \models \mathbf{wp}(c_p, \mathbf{true})$$

From Theorem 5 we know that

$$\neg \langle c_p \rangle_p \xrightarrow{\mu'_\nu}_{p}^* \text{failure}_p$$

which contradicts (32). □

5.6 Size

After having proved the weakest preconditions sound, we now proceed to show that they vary polynomially in relation to the original program size. For this, we need to define metrics which allow us to express the size of commands and formulae. We define them on all three kinds of commands (regular, versioned and passified), see Figure 32.

Definition 14 (Size of commands). (`command_metric` `vcommand_metric` `pcommand_metric`) *We define a metric $|c|$ on commands as follows:*

$$\begin{aligned} |\mathbf{assert } e| &= 2 \\ |\mathbf{assume } e| &= 2 \\ |x := e| &= 2 \\ |\mathbf{skip}| &= 1 \\ |c_1; c_2| &= 1 + |c_1| + |c_2| \\ |c_1 \parallel c_2| &= 1 + |c_1| + |c_2| \end{aligned}$$

Measuring the size of propositions in Coq is not possible, which is why we defined our own inductive type `formula` and expressed the weakest preconditions in terms of these (see Figure 33, `wp'` and `wlp'`).

```

1  Fixpoint command_metric (c : command) : nat :=
2  match c with
3  | cAssert _      => 2
4  | cAssume _      => 2
5  | cAssign _ _    => 2
6  | cSequence x y => S
7      (command_metric x + command_metric y)
8  | cSkip          => 1
9  | cChoice x y    =>
10     S (command_metric x + command_metric y)
11 end.

12 Fixpoint vcommand_metric (c : vcommand) : nat :=
13 match c with
14 | vcAssert _      => 2
15 | vcAssume _      => 2
16 | vcAssign _ _    => 2
17 | vcSequence x y =>
18     S (vcommand_metric x + vcommand_metric y)
19 | vcSkip          => 1
20 | vcChoice x y    =>
21     S (vcommand_metric x + vcommand_metric y)
22 end.

23 Fixpoint pcommand_metric (c : pcommand) : nat :=
24 match c with
25 | pcAssert _      => 2
26 | pcAssume _      => 2
27 | pcSequence x y =>
28     S (pcommand_metric x + pcommand_metric y)
29 | pcSkip          => 1
30 | pcChoice x y    =>
31     S (pcommand_metric x + pcommand_metric y)
32 end.

```

Figure 32: Formulas as data

```

1  Inductive formula : Set :=
2  | fConjunction : formula -> formula -> formula
3  | fDisjunction : formula -> formula -> formula
4  | fImplication : formula -> formula -> formula
5  | fAtom       : formula.

6  Fixpoint formula_metric (f : formula) : nat :=
7    match f with
8      | fConjunction x y =>
9        S (formula_metric x + formula_metric y)
10     | fDisjunction x y =>
11       S (formula_metric x + formula_metric y)
12     | fImplication x y =>
13       S (formula_metric x + formula_metric y)
14     | fAtom           => 1
15   end.

16  Fixpoint wlp' (c : pcommand)
17    (Q : formula)
18    {struct c} : formula :=
19    match c with
20      | pcAssert _      => fImplication fAtom Q
21      | pcAssume _      => fImplication fAtom Q
22      | pcSequence c1 c2 => wlp' c1 (wlp' c2 Q)
23      | pcSkip          => fAtom
24      | pcChoice c1 c2  =>
25        fDisjunction (fConjunction (wlp' c1 fAtom)
26                      (wlp' c2 fAtom))
27                      Q
28    end.

29  Fixpoint wp' (c : pcommand)
30    (Q : formula)
31    {struct c} : formula :=
32    match c with
33      | pcAssert _      => fConjunction fAtom Q
34      | pcAssume _      => fImplication fAtom Q
35      | pcSequence c1 c2 => wp' c1 (wp' c2 Q)
36      | pcSkip          => fAtom
37      | pcChoice c1 c2  =>
38        fConjunction (fConjunction (wp' c1 fAtom)
39                      (wp' c2 fAtom))
40                      (wlp' c Q)
41    end.

```

Figure 33: Formulas as data

Definition 15 (Size of logical formulae). (`formula_metric`) We define a metric $|P|$ on logical formulae as follows:

$$\begin{aligned} |P \wedge Q| &= 1 + |P| + |Q| \\ |P \vee Q| &= 1 + |P| + |Q| \\ |P \Rightarrow Q| &= 1 + |P| + |Q| \\ |atomic| &= 1 \end{aligned}$$

We restrict ourselves to the subset of logical operators needed to express the verification conditions. Perhaps a less arbitrary way would have been to use a set of operators with which all possible formulae can be expressed. We did not choose this approach because the translation of the verification conditions into a new form would have to be verified manually, resulting in an additional weak spot in the Coq script.

Lemma 31. (`sync_vcommand_size`) The size of a synchronization command¹¹ $\delta = \text{sync}(I, \nu, \nu')$ varies linearly in the size of the identifier set I .

$$|\text{sync}(I, \nu, \nu')| = O(|I|)$$

Lemma 32. (`quadratic_sa_transformation`) The SA-transformation¹² varies quadratically in input program size.

$$|\text{SA}_\nu(c)| = O(|c|^2)$$

Lemma 33. (`passify_maintains_size`) Passification maintains program size.

$$|\text{passify}(c)| = |c|$$

Lemma 34. (`linear_wlp'`) The weakest liberal preconditions¹³ vary linearly with respect to the passified input program.

$$|\text{wlp}_e(c, Q)| = O(|c| + |Q|)$$

Lemma 35. (`quadratic_wp'`) The weakest conservative preconditions¹⁴ vary quadratically with respect to the passified input program.

$$|\text{wp}_e(c, Q)| = O(|c|^2 + |Q|)$$

Theorem 12. (`polynomial_wps`) The weakest conservative preconditions vary biquadratically with respect to the input program.

$$|\text{wp}_e(\text{passify}(\text{SA}_\nu(c)), Q)| = O(|c|^4 + |Q|)$$

Proof. Chaining together Lemma 32, Lemma 33 and Lemma 35. □

¹¹See Definition 4.

¹²See Definition 2.

¹³See Definition 12.

¹⁴See Definition 11.

6 The Coq script

The Coq script [3] can be seen as an inverted pyramid: starting with a small base (the axioms), we grow a large number of lemmas and theorems on them towards a specific goal. Coq [2] guarantees (assuming Coq is sound) that the building process is flawless, but cannot make any promises about the foundations of our construction. It is up to us to manually check these.

In this section, we identify which parts make up of the “Achilles heel” of the Coq script. In order to improve trust in this part of the script, we also proved a number of additional theorems which show that the axioms do indeed behave as intended. These extra theorems are not needed for the script’s main theorems (i.e. the soundness and size of the verification conditions).

6.1 The operational semantics

The validity of the proof depends heavily on the correctness of the operational semantics we defined. If there is an error in the program behaviour they describe, the whole Coq proof script is discredited. We will now discuss all definitions and axioms related to the operational semantics and give the rationale behind them.

The intermediate language supports variables which are referred to using identifiers. The Coq type for these identifiers is `id` (see Figure 34, line 1) and is defined as `nat`. We could have kept the fully abstract, but in order to implement the SA-transformation algorithm, we needed to work with sets of identifiers, for which we make use of `FSetLists` from Coq’s standard library. This implementation of sets requires the element type to be an ordered type (`OrderedType`), making `nat` the obvious choice (see `Identifier_OT` in the Coq proof script).

The versioned identifier type `vid` is trivial (see Figure 34, line 2): it is a pair whose first component is a regular identifier (`id`) and second component is the version number as a `nat`. For readers wondering about the `%type` annotation: it tells Coq to interpret the part between parenthesis in the context `type`, so that `*` is interpreted as the Cartesian product between two types, instead of the product between natural numbers.

Expressions (occurring in `assert`, `assume` and assignment commands) evaluate to values, for which we need to define a new type `value`. We also define two special values `T` and `F`, standing for true and false¹⁵. We also state two properties about the `value` type:

- Figure 34, line 7, `decidable_eq_value` states that the operation of finding out whether two values are equal or not is decidable, i.e. there exists an algorithm which checks for equality and always terminates.
- Figure 34, line 8, `T_neq_F` states that `T` and `F` are not equal, thus defining `value` as being populated by at least two different values.

Next, we define the following concepts (lines 9–13 in Figure 34):

- Stores (`store`) are total functions from `id` to `value`.
- Versioned stores (`vstore`) are total functions from `vid` to `value`.
- Version maps (`vmap`) are total functions from `id` to `nat`.
- Expressions (`expr`) are total functions from `store` to `value`.
- Versioned expressions (`vexpr`) are total functions from `vstore` to `value`.

The three types of commands (regular, versioned and passified) can be found on lines 14–33 in Figure 34. These definitions do not hide any surprises, so we won’t discuss these any further. The metrics on the three types of commands (lines 1–30 in Figure 35) are fairly important, as defining them incorrectly (e.g. too “small”, such as 0 for any command) would defeat the purpose

¹⁵Technically, `F` stands for “a value different from `T`” to prevent `value` to be a singleton type, something we rely upon in the definition of `assume_from_assign`.

```

1 Definition id := nat.
2 Definition vid := (id * nat)%type.
3 Parameters
4   (value : Set)
5   (T      : value)
6   (F      : value).
7 Axiom decidable_eq_value : decidable_eq value.
8 Axiom T_neq_F : T <> F.
9 Definition store := id -> value.
10 Definition vstore := vid -> value.
11 Definition vmap := id -> nat.
12 Definition expr := store -> value.
13 Definition vexpr := vstore -> value.
14 Inductive command : Set :=
15 | cAssert   : expr -> command
16 | cAssume   : expr -> command
17 | cAssign   : id -> expr -> command
18 | cSequence : command -> command -> command
19 | cSkip     : command
20 | cChoice   : command -> command -> command.
21 Inductive vcommand : Set :=
22 | vcAssert   : vexpr -> vcommand
23 | vcAssume   : vexpr -> vcommand
24 | vcAssign   : vid -> vexpr -> vcommand
25 | vcSequence : vcommand -> vcommand -> vcommand
26 | vcSkip     : vcommand
27 | vcChoice   : vcommand -> vcommand -> vcommand.
28 Inductive pcommand : Set :=
29 | pcAssert   : vexpr -> pcommand
30 | pcAssume   : vexpr -> pcommand
31 | pcSequence : pcommand -> pcommand -> pcommand
32 | pcSkip     : pcommand
33 | pcChoice   : pcommand -> pcommand -> pcommand.

```

Figure 34: Parts of the Coq script that need manual verification

```

1  Fixpoint command_metric (c : command) : nat :=
2    match c with
3      | cAssert _      => 2
4      | cAssume _      => 2
5      | cAssign _ _    => 2
6      | cSequence x y => S (command_metric x + command_metric y)
7      | cSkip          => 1
8      | cChoice x y   => S (command_metric x + command_metric y)
9    end.

10 Fixpoint vcommand_metric (c : vcommand) : nat :=
11   match c with
12     | vcAssert _      => 2
13     | vcAssume _      => 2
14     | vcAssign _ _    => 2
15     | vcSequence x y =>
16       S (vcommand_metric x + vcommand_metric y)
17     | vcSkip          => 1
18     | vcChoice x y   =>
19       S (vcommand_metric x + vcommand_metric y)
20   end.

21 Fixpoint pcommand_metric (c : pcommand) : nat :=
22   match c with
23     | pcAssert _      => 2
24     | pcAssume _      => 2
25     | pcSequence x y =>
26       S (pcommand_metric x + pcommand_metric y)
27     | pcSkip          => 1
28     | pcChoice x y   =>
29       S (pcommand_metric x + pcommand_metric y)
30   end.

31 Definition decidable_eq_id : decidable_eq id.
32 Definition decidable_eq_vid : decidable_eq vid.
33 Definition rebind (A B : Set) (eq_dec : decidable_eq A)
34   (f : A -> B) (x : A) (y : B) :=
35   fun a => if eq_dec a x then y else f a.

36 Definition update_store (mu : store) (x : id) (v : value) :=
37   rebind decidable_eq_id mu x v.

38 Definition update_vstore (mu : vstore)(x : vid)(v : value) :=
39   rebind decidable_eq_vid mu x v.

40 Definition equivalent_functions (A B : Type) f g :=
41   forall (x : A), f x = g x.

42 Axiom expression_evaluation : forall (e : expr) mu mu',
43   equivalent_functions mu mu' -> e mu = e mu'.

```

Figure 35: Parts of the Coq script that need manual verification

of having an upper bound on the size of the weakest preconditions with respect to the size of the program to be verified.

The next few definitions set up the necessary foundations to define assignment: assignment consists of a single binding in the store. For this, we first need to be able to decide whether two `id` (or two `vids`) are equal or not: this is taken care of by `decidable_eq_id` and `decidable_eq_vid`. Lines 33–35 in Figure 35 define `rebind`, a polymorphic function which takes a function and replaces one single binding. We use it to define `update_store` and `update_vstore`, which will be useful when specifying the operational semantics.

Lines 40–43 state that evaluation of an expression in “equivalent” stores yields the same value, where equivalent stores are stores which bind all identifiers to the same values. This is, in a way, a kind of “localized” functional extensionality. We decided to keep the number of axioms to a minimum and as “small” as possible, hence our restriction to expression evaluation¹⁶.

The state definitions (Figure 36, lines 1–9) require no explanation. Next are the different operational semantics `step` (Figure 36, lines 10–37), `vstep` (Figure 37) and `pstep` (Figure 38, lines 1–24). These are required to faithfully model the rules shown in Figure 43, Figure 45 and Figure 47, respectively. The three multistep relations `multistep`, `vmultistep` and `pmultistep` are trivial to check.

Many theorems have been proven to show that the behaviour described by the operational semantics has certain properties we expect it to have, among which

- the step rules always decrease the size of a command, guaranteeing termination;
- the different stuck states;
- the correspondence between the different step rules;
- ...

After all these definitions needed to formalize the operational semantics in Coq, we only need to check the main soundness theorem (Figure 38, lines 44–48), to ascertain that what we prove is actually what we need to prove.

6.2 Size of the weakest preconditions

Finally, we also need to manually verify the definition of `formula` and `formula_metric`, the reformulation of the weakest preconditions `wp'` and `wlp'` in terms of `formula` and the theorem `polynomial_wps`. We had to define our own type for logical formulae so that we could define a metric on them (as far as we know, it is impossible to define a metric on `Prop` values in Coq). It is important that `wp'` and `wlp'` exhibit exactly the same structure as their `Prop`-counterparts `efficient_wp` and `efficient_wlp`, respectively, otherwise measuring their size becomes an exercise in futility, and theorem `polynomial_wps` would lose all meaning.

¹⁶Ironically, the axiom of functional extensionality is stated in the Coq proof (see `functional_extensionality_dep`), but we added in at a later stage in order to prove a few extra theorems which are not necessary for proving the main theorems.

```

1 Inductive state : Set :=
2 | ip      : command -> store -> state
3 | failure : store -> state.

4 Inductive vstate : Set :=
5 | vip      : vcommand -> vstore -> vstate
6 | vfailure : vstore -> vstate.

7 Inductive pstate : Set :=
8 | pip      : pcommand -> pstate
9 | pfailure : pstate.

10 Inductive step : state -> state -> Prop :=
11 | stepAssertT : forall e mu,
12   e mu = T ->
13   step (ip (cAssert e) mu) (ip cSkip mu)
14 | stepAssertF : forall e mu,
15   e mu <> T ->
16   step (ip (cAssert e) mu) (failure mu)
17 | stepAssume : forall e mu,
18   e mu = T ->
19   step (ip (cAssume e) mu) (ip cSkip mu)
20 | stepSeq : forall c1 c1' c2 mu mu',
21   step (ip c1 mu) (ip c1' mu') ->
22   step (ip (cSequence c1 c2) mu)
23     (ip (cSequence c1' c2) mu')
24 | stepSeqSkip : forall c2 mu,
25   step (ip (cSequence cSkip c2) mu) (ip c2 mu)
26 | stepSeqFail : forall c1 c2 mu mu',
27   step (ip c1 mu) (failure mu') ->
28   step (ip (cSequence c1 c2) mu) (failure mu')
29 | stepAssign : forall x e mu,
30   step (ip (cAssign x e) mu)
31     (ip cSkip (update_store mu x (e mu)))
32 | stepChoiceL : forall c1 c2 mu,
33   step (ip (cChoice c1 c2) mu)
34     (ip c1 mu)
35 | stepChoiceR : forall c1 c2 mu,
36   step (ip (cChoice c1 c2) mu)
37     (ip c2 mu).

```

Figure 36: Parts of the Coq script that need manual verification

```

1 Inductive vstep : vstate -> vstate -> Prop :=
2 | vstepAssertT : forall e vmu,
3   e vmu = T ->
4   vstep (vip (vcAssert e) vmu) (vip vcSkip vmu)
5 | vstepAssertF : forall e vmu,
6   e vmu <> T ->
7   vstep (vip (vcAssert e) vmu) (vfailure vmu)
8 | vstepAssume  : forall e vmu, e vmu = T ->
9   vstep (vip (vcAssume e) vmu) (vip vcSkip vmu)
10 | vstepSeq     : forall c1 c1' c2 vmu vmu',
11   vstep (vip c1 vmu) (vip c1' vmu') ->
12   vstep (vip (vcSequence c1 c2) vmu)
13     (vip (vcSequence c1' c2) vmu')
14 | vstepSeqSkip : forall c2 vmu,
15   vstep (vip (vcSequence vcSkip c2) vmu) (vip c2 vmu)
16 | vstepSeqFail : forall c1 c2 vmu vmu',
17   vstep (vip c1 vmu) (vfailure vmu') ->
18   vstep (vip (vcSequence c1 c2) vmu) (vfailure vmu')
19 | vstepAssign  : forall x e vmu,
20   vstep (vip (vcAssign x e) vmu)
21     (vip vcSkip (update_vstore vmu x (e vmu)))
22 | vstepChoiceL : forall c1 c2 vmu,
23   vstep (vip (vcChoice c1 c2) vmu) (vip c1 vmu)
24 | vstepChoiceR : forall c1 c2 vmu,
25   vstep (vip (vcChoice c1 c2) vmu) (vip c2 vmu).

```

Figure 37: Parts of the Coq script that need manual verification

```

1 Inductive pstep (vmu : vstore) : pstate -> pstate -> Prop :=
2 | pstepAssertT : forall e,
3   e vmu = T ->
4   pstep vmu (pip (pcAssert e)) (pip pcSkip)
5 | pstepAssertF : forall e,
6   e vmu <> T ->
7   pstep vmu (pip (pcAssert e)) pfailure
8 | pstepAssume : forall e,
9   e vmu = T ->
10  pstep vmu (pip (pcAssume e)) (pip pcSkip)
11 | pstepSeq : forall c1 c1' c2,
12  pstep vmu (pip c1) (pip c1') ->
13  pstep vmu (pip (pcSequence c1 c2))
14            (pip (pcSequence c1' c2))
15 | pstepSeqSkip : forall c2,
16  pstep vmu (pip (pcSequence pcSkip c2))
17            (pip c2)
18 | pstepSeqFail : forall c1 c2,
19  pstep vmu (pip c1) pfailure ->
20  pstep vmu (pip (pcSequence c1 c2)) pfailure
21 | pstepChoiceL : forall c1 c2,
22  pstep vmu (pip (pcChoice c1 c2)) (pip c1)
23 | pstepChoiceR : forall c1 c2,
24  pstep vmu (pip (pcChoice c1 c2)) (pip c2).

25 Inductive multistep : state -> state -> Prop :=
26 | multiReflexivity : forall s, multistep s s
27 | multiStep : forall s1 s2 s3,
28   step s1 s2 ->
29   multistep s2 s3 ->
30   multistep s1 s3.

31 Inductive vmultistep : vstate -> vstate -> Prop :=
32 | vmultiReflexivity : forall s, vmultistep s s
33 | vmultiStep : forall s1 s2 s3,
34   vstep s1 s2 ->
35   vmultistep s2 s3 ->
36   vmultistep s1 s3.

37 Inductive pmultistep (vmu : vstore) :
38   pstate -> pstate -> Prop :=
39 | pmultiReflexivity : forall s, pmultistep vmu s s
40 | pmultiStep : forall s1 s2 s3,
41   pstep vmu s1 s2 ->
42   pmultistep vmu s2 s3 ->
43   pmultistep vmu s1 s3.

44 Theorem soundness_efficient_wp :
45   forall c,
46     (forall vmu, efficient_wp vmu (passified c) True) ->
47     forall mu, ~ exists mu', multistep (ip c mu)
48                                     (failure mu').

```

Figure 38: Parts of the Coq script that need manual verification

```

1 Inductive formula : Set :=
2 | fConjunction : formula -> formula -> formula
3 | fDisjunction : formula -> formula -> formula
4 | fImplication : formula -> formula -> formula
5 | fAtom       : formula.

6 Fixpoint formula_metric (f : formula) : nat :=
7   match f with
8     | fConjunction x y =>
9       S (formula_metric x + formula_metric y)
10    | fDisjunction x y =>
11      S (formula_metric x + formula_metric y)
12    | fImplication x y =>
13      S (formula_metric x + formula_metric y)
14    | fAtom           => 1
15  end.

16 Fixpoint wlp' (c : pcommand) (Q : formula) : formula :=
17   match c with
18     | pcAssert _      => fImplication fAtom Q
19     | pcAssume _      => fImplication fAtom Q
20     | pcSequence c1 c2 => wlp' c1 (wlp' c2 Q)
21     | pcSkip          => fAtom
22     | pcChoice c1 c2  =>
23       fDisjunction (fConjunction (wlp' c1 fAtom)
24                        (wlp' c2 fAtom))
25                     Q
26  end.

27 Fixpoint wp' (c : pcommand) (Q : formula) : formula :=
28   match c with
29     | pcAssert _      => fConjunction fAtom Q
30     | pcAssume _      => fImplication fAtom Q
31     | pcSequence c1 c2 => wp' c1 (wp' c2 Q)
32     | pcSkip          => fAtom
33     | pcChoice c1 c2  =>
34       fConjunction (fConjunction (wp' c1 fAtom)
35                        (wp' c2 fAtom))
36                     (wlp' c Q)
37  end.

38 Theorem polynomial_wps :
39   exists N4, exists N3, exists N2, exists N1, forall c Q,
40   let cp := passified c in
41   let x := command_metric c in
42   let wp := wp' cp Q in
43     formula_metric wp <= N4 * x * x * x * x +
44                          N3 * x * x * x +
45                          N2 * x * x +
46                          N1 * x +
47     formula_metric Q.

```

Figure 39: Parts of the Coq script that need manual verification

A Appendix

$$c ::= \begin{array}{l} \mathbf{assert} \ e \\ \mathbf{assume} \ e \\ \mathbf{skip} \\ x := e \\ c_1 ; c_2 \\ c_1 \parallel c_2 \end{array}$$

Figure 40: Regular commands

$$c_v ::= \begin{array}{l} \mathbf{assert} \ e_v \\ \mathbf{assume} \ e_v \\ \mathbf{skip} \\ x_i := e_v \\ c_{v,1} ; c_{v,2} \\ c_{v,1} \parallel c_{v,2} \end{array}$$

Figure 41: Versioned commands

$$c_p ::= \begin{array}{l} \mathbf{assert} \ e_v \\ \mathbf{assume} \ e_v \\ \mathbf{skip} \\ c_{p,1} ; c_{p,2} \\ c_{p,1} \parallel c_{p,2} \end{array}$$

Figure 42: Passified commands

$$\frac{e(\mu) = \mathbf{true}}{\langle \mathbf{assert} \ e, \mu \rangle \longrightarrow \langle \mathbf{skip}, \mu \rangle} \text{ ASSERTTRUE}$$

$$\frac{e(\mu) \neq \mathbf{true}}{\langle \mathbf{assert} \ e, \mu \rangle \longrightarrow \text{failure } \mu} \text{ ASSERTFALSE}$$

$$\frac{e(\mu) = \mathbf{true}}{\langle \mathbf{assume} \ e, \mu \rangle \longrightarrow \langle \mathbf{skip}, \mu \rangle} \text{ ASSUME}$$

$$\frac{\langle c_1, \mu \rangle \longrightarrow \langle c'_1, \mu' \rangle}{\langle c_1; c_2, \mu \rangle \longrightarrow \langle c'_1; c_2, \mu' \rangle} \text{ SEQUENCE}$$

$$\frac{\langle c_1, \mu \rangle \longrightarrow \text{failure } \mu'}{\langle c_1; c_2, \mu \rangle \longrightarrow \text{failure } \mu'} \text{ SEQUENCEFAIL}$$

$$\frac{}{\langle \mathbf{skip}; c, \mu \rangle \longrightarrow \langle c, \mu \rangle} \text{ SEQUENCESKIP}$$

$$\frac{}{\langle x := e, \mu \rangle \longrightarrow \langle \mathbf{skip}, \mu[x \mapsto e(\mu)] \rangle} \text{ ASSIGN}$$

$$\frac{}{\langle c_1 \parallel c_2, \mu \rangle \longrightarrow \langle c_1, \mu \rangle} \text{ CHOICELEFT}$$

$$\frac{}{\langle c_1 \parallel c_2, \mu \rangle \longrightarrow \langle c_2, \mu \rangle} \text{ CHOICERIGHT}$$

Figure 43: Single step regular operational semantics

$$\frac{}{\langle c, \mu \rangle \longrightarrow^* \langle c, \mu \rangle}$$

$$\frac{\langle c, \mu \rangle \longrightarrow \langle c', \mu' \rangle \longrightarrow^* \langle c'', \mu'' \rangle}{\langle c, \mu \rangle \longrightarrow^* \langle c'', \mu \rangle}$$

Figure 44: Multiple step regular operational semantics

$$\frac{e_v(\mu_v) = \mathbf{true}}{\langle \mathbf{assert} \ e_v, \mu_v \rangle_v \longrightarrow_v \langle \mathbf{skip}, \mu_v \rangle_v} \text{ V-ASSERTTRUE}$$

$$\frac{e_v(\mu_v) \neq \mathbf{true}}{\langle \mathbf{assert} \ e_v, \mu_v \rangle_v \longrightarrow_v \mathbf{failure}_v \ \mu_v} \text{ V-ASSERTFALSE}$$

$$\frac{e_v(\mu_v) = \mathbf{true}}{\langle \mathbf{assume} \ e_v, \mu_v \rangle_v \longrightarrow_v \langle \mathbf{skip}, \mu_v \rangle_v} \text{ V-ASSUME}$$

$$\frac{\langle c_{v,1}, \mu_v \rangle_v \longrightarrow_v \langle c'_{v,1}, \mu'_v \rangle_v}{\langle c_{v,1}; c_{v,2}, \mu_v \rangle_v \longrightarrow_v \langle c'_{v,1}; c_{v,2}, \mu'_v \rangle_v} \text{ V-SEQUENCE}$$

$$\frac{\langle c_{v,1}, \mu_v \rangle_v \longrightarrow_v \mathbf{failure}_v \ \mu'_v}{\langle c_{v,1}; c_{v,2}, \mu_v \rangle_v \longrightarrow_v \mathbf{failure}_v \ \mu'_v} \text{ V-SEQUENCEFAIL}$$

$$\frac{}{\langle \mathbf{skip}; c, \mu_v \rangle_v \longrightarrow_v \langle c, \mu_v \rangle_v} \text{ V-SEQUENCESKIP}$$

$$\frac{}{\langle x_i := e_v, \mu_v \rangle_v \longrightarrow_v \langle \mathbf{skip}, \mu_v[x_i \mapsto e_v(\mu)] \rangle_v} \text{ V-ASSIGN}$$

$$\frac{}{\langle c_{v,1} \parallel c_{v,2}, \mu_v \rangle_v \longrightarrow_v \langle c_{v,1}, \mu_v \rangle_v} \text{ V-CHOICELEFT}$$

$$\frac{}{\langle c_{v,1} \parallel c_{v,2}, \mu_v \rangle_v \longrightarrow_v \langle c_{v,2}, \mu_v \rangle_v} \text{ V-CHOICERIGHT}$$

Figure 45: Single step versioned operational semantics

$$\frac{}{\langle c_v, \mu \rangle_v \longrightarrow_v^* \langle c_v, \mu \rangle_v} \text{ V*}-\text{REFLEXIVITY}$$

$$\frac{\langle c_v, \mu \rangle_v \longrightarrow_v \langle c'_v, \mu' \rangle_v \longrightarrow_v^* \langle c''_v, \mu'' \rangle_v}{\langle c_v, \mu \rangle_v \longrightarrow_v^* \langle c''_v, \mu'' \rangle_v} \text{ V*}-\text{STEP}$$

Figure 46: Multiple step versioned operational semantics

$$\begin{array}{c}
\frac{e_v(\mu_v) = \mathbf{true}}{\langle \mathbf{assert} \ e_v \rangle_p \xrightarrow{\mu_v}_p \langle \mathbf{skip} \rangle_p} \\
\\
\frac{e_v(\mu_v) \neq \mathbf{true}}{\langle \mathbf{assert} \ e_v \rangle_p \xrightarrow{\mu_v}_p \mathbf{failure}_p} \\
\\
\frac{e_v(\mu_v) = \mathbf{true}}{\langle \mathbf{assume} \ e_v \rangle_p \xrightarrow{\mu_v}_p \langle \mathbf{skip} \rangle_p} \\
\\
\frac{\langle c_{p,1} \rangle_p \xrightarrow{\mu_v}_p \langle c'_{p,1} \rangle_p}{\langle c_{p,1}; c_{p,2} \rangle_p \xrightarrow{\mu_v}_p \langle c'_{p,1}; c_{p,2} \rangle_p} \\
\\
\frac{\langle c_{p,1} \rangle_p \xrightarrow{\mu}_p \mathbf{failure}_p}{\langle c_{p,1}; c_{p,2} \rangle_p \xrightarrow{\mu_v}_p \mathbf{failure}_p} \\
\\
\frac{}{\langle \mathbf{skip}; c_p \rangle_p \xrightarrow{\mu_v}_p \langle c_p \rangle_p} \\
\\
\frac{}{\langle c_{p,1} \parallel c_{p,2} \rangle_p \xrightarrow{\mu_v}_p \langle c_{p,1} \rangle_p} \\
\\
\frac{}{\langle c_{p,1} \parallel c_{p,2} \rangle_p \xrightarrow{\mu_v}_p \langle c_{p,2} \rangle_p}
\end{array}$$

Figure 47: Single step passified operational semantics

$$\begin{array}{c}
\frac{}{\langle c_p \rangle_p \xrightarrow{\mu}_p^* \langle c_p \rangle_p} \\
\\
\frac{\langle c_p \rangle_p \xrightarrow{\mu}_p \langle c'_p \rangle_p \xrightarrow{\mu}_p^* \langle c''_p \rangle_p}{\langle c_p \rangle_p \xrightarrow{\mu}_p^* \langle c''_p \rangle_p}
\end{array}$$

Figure 48: Multiple step passified operational semantics

References

- [1] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO 2005, volume 4111 of LNCS*, pages 364–387. Springer, 2006.
- [2] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Springer, 2004.
- [3] Coq script. <http://www.cs.kuleuven.be/~frederic/papers/efficient/passification.v>.
- [4] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *POPL*, pages 193–205, 2001.
- [5] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM*, 12(10):576–580, 1969.
- [6] K. Rustan M. Leino. Efficient weakest preconditions. *Inf. Process. Lett.*, 93(6):281–288, 2005.
- [7] K. Rustan M. Leino and Wolfram Schulte. A verifying compiler for a multi-threaded object-oriented language. Marktoberdorf lecture notes, 2007. In Manfred Broy, Johannes Grünbauer, Tony Hoare (eds.). *Software System Reliability and Security*. IOS Press, 2007.
- [8] Frédéric Vogels, Bart Jacobs, and Frank Piessens. A machine checked soundness proof for an intermediate verification language. In Mogens Nielsen, Antonín Kucera, Peter Bro Miltersen, Catuscia Palamidessi, Petr Tuma, and Frank D. Valencia, editors, *35th International Conference on Current Trends in Theory and Practice of Computer Science (Sofsem 2009)*, volume 5404 of *Lecture Notes in Computer Science*, pages 570–581. Springer, 2009.
- [9] Frédéric Vogels, Bart Jacobs, and Frank Piessens. A machine-checked soundness proof for an efficient verification condition generator. In *SAC 2010: Proceedings of the 2010 ACM symposium on Applied Computing*, pages 2517–2522. ACM, 2010.

Contents

1	Library E:\sandbox\efficient\coq\passification	2
1.1	Custom Tactics	2
1.2	General definitions	13
1.2.1	Decidability	13
1.2.2	Equivalent functions	14
1.3	Paper specific	16
1.3.1	Language definitions and theorems	16
1.3.2	Single Assignment	69
1.3.3	Passification	110
1.3.4	Weakest Preconditions Soundness	131
1.3.5	Weakest Preconditions Size	145

Chapter 1

Library

E:\sandbox\efficient\coq\passification

IMPORTANT: This proof script was developed with Coq 8.1. It will NOT work with Coq 8.2 (due to differences in the standard library).

The script contains four Coq axioms (search for "Axiom"). One is borrowed from Coq 8.2's standard library, namely *functional_extensionality_dep*. The others are directly related to the paper and are trivial (but necessary). All proofs are complete, none has been ended with "Admitted" (which would tell Coq to just accept them as axioms).

The validity of the entire Coq script depends on a small core of definitions. An error here could make the entire proof worthless. We identify these "Achilles heel spots" with a clear message. Fortunately, many of these "fragile definitions" are trivial.

```
Require Import Arith.
Require Import Omega.
Require Import FSets.
Require Import Max.
Require Import List.
Require Import Relations.
Require Import Setoid.
Require Import ProofIrrelevance.
Set Printing Width 200.
```

1.1 Custom Tactics

Tactic which introduces a new identifier *id*, which is equal to *term*.

```
=====
Goal
```

```
introduce x (3 + 5)
```

```
x : nat
H : x = 3 + 5
=====
Goal
```

Generally useful to simplify expressions by substituting entire subexpressions with a single identifier (using `rewrite`), or to apply the `induction` tactic which sometimes tends to throw some information away.

```
Ltac introduce_eq id term :=
  ( set (id := term);
    assert (id = term);
    [ trivial | clearbody id ] ).
```

New tactic notation: instead of `introduce_eq x t` we can write `introduce new identifier x for t`. Tactic Notation "introduce" "new" "identifier" *ident(x)* "for" *constr(t)* := `introduce_eq x t`.

Same as `introduce_eq`, but also performs a rewrite in *H*.

```
x : nat
y : nat
z : nat
H : x = (3 + y) * z
=====
Goal
```

```
introduce_eq_in a (3 + y) H.
```

```
x : nat
y : nat
z : nat
a : nat
H0 : a = 3 + y
H : x = a * z
=====
Goal
```

```
Ltac introduce_eq_in id term H :=
  ( let H' := fresh in
    set (id := term);
```

```

    assert ( $H' : id = term$ );
    [ trivial | clearbody id ];
    rewrite ←  $H'$  in  $H$  ).

```

Adds a new hypothesis, whose proof is given by *term*.

```

a : nat
b : nat
c : nat
H : a < b
H0 : b < c
=====
Goal

```

```
state_fresh (lt_trans _ _ _ H H0).
```

```

a : nat
b : nat
c : nat
H : a < b
H0 : b < c
H1 : a < c
=====
Goal

```

```

Ltac state_fresh term :=
  let id := fresh in
    (assert ( $id := term$ )).

```

Same as *state_fresh*, but instead of using a fresh id, it uses *H*.

```

a : nat
b : nat
c : nat
H : a < b
H0 : b < c
=====
Goal

```

```
state_as Foo (lt_trans _ _ _ H H0).
```

```

a : nat
b : nat
c : nat
H : a < b

```

```

H0 : b < c
Foo : a < c
=====
Goal

```

```

Ltac state_as H term :=
  (assert (H := term)).

```

Introduces a new tactic notation: *state_fresh term* can now be written *state term*. Tactic Notation "state" *constr(x)* := *state_fresh x*.

Introduces a new tactic notation: *state_as H term* can now be written *state term as H*. Tactic Notation "state" *constr(x)* "as" *ident(H)* := *state_as H x*.

Applies *x* to *H*.

```

a : nat
b : nat
c : nat
H : a < b
H0 : b < c
=====
Goal

```

```

state lt_trans.

```

```

a : nat
b : nat
c : nat
H : a < b
H0 : b < c
H1 : forall n m p : nat, n < m -> m < p -> n < p
=====
Goal

```

```

specify_single H1 a.

```

```

a : nat
b : nat
c : nat
H : a < b
H0 : b < c
H1 : forall m p : nat, a < m -> m < p -> a < p

```

```

=====
Goal

specify_single H1 b.

a : nat
b : nat
c : nat
H : a < b
H0 : b < c
H1 : forall p : nat, a < b -> b < p -> a < p
=====

```

```

Goal

specify_single H1 c.

a : nat
b : nat
c : nat
H : a < b
H0 : b < c
H1 : a < b -> b < c -> a < c
=====

```

```

Goal

specify_single H1 H.

a : nat
b : nat
c : nat
H : a < b
H0 : b < c
H1 : b < c -> a < c
=====

```

```

Goal

specify_single H1 H0.

a : nat
b : nat
c : nat
H : a < b

```

H0 : b < c

H1 : a < c

=====

Goal

Useful in keeping hypotheses "up-to-date" with the proof developments. Also simplifies things a bit when the terms applied are large. Ltac *specify_single* H x :=

```
(let H' := fresh in
  state (H x) as H';
  clear H;
  rename H' into H).
```

I have not found how to work with lists in Ltac, so I just defined a series of tactic notation which apply *specify_single* a certain number of times. Tactic Notation "specify" hyp(H) constr(x1) :=

```
specify_single H x1.
```

Tactic Notation "specify" hyp(H) constr(x1) constr(x2) :=

```
specify_single H x1;
specify_single H x2.
```

Tactic Notation "specify" hyp(H) constr(x1) constr(x2) constr(x3) :=

```
specify_single H x1;
specify_single H x2;
specify_single H x3.
```

Tactic Notation "specify" hyp(H) constr(x1) constr(x2) constr(x3) constr(x4) :=

```
specify_single H x1;
specify_single H x2;
specify_single H x3;
specify_single H x4.
```

Tactic Notation "specify" hyp(H) constr(x1) constr(x2) constr(x3) constr(x4) constr(x5) :=

```
specify_single H x1;
specify_single H x2;
specify_single H x3;
```

specify_single H x4;
specify_single H x5.

Tactic Notation "specify" *hyp(H) constr(x1)*
constr(x2)
constr(x3)
constr(x4)
constr(x5)
constr(x6) :=

specify_single H x1;
specify_single H x2;
specify_single H x3;
specify_single H x4;
specify_single H x5;
specify_single H x6.

Tactic Notation "specify" *hyp(H) constr(x1)*
constr(x2)
constr(x3)
constr(x4)
constr(x5)
constr(x6)
constr(x7) :=

specify_single H x1;
specify_single H x2;
specify_single H x3;
specify_single H x4;
specify_single H x5;
specify_single H x6;
specify_single H x7.

Tactic Notation "specify" *hyp(H) constr(x1)*
constr(x2)
constr(x3)
constr(x4)
constr(x5)
constr(x6)
constr(x7)
constr(x8) :=

specify_single H x1;
specify_single H x2;
specify_single H x3;
specify_single H x4;
specify_single H x5;
specify_single H x6;

specify_single *H* *x7*;
specify_single *H* *x8*.

Tactic Notation "specify" *hyp*(*H*) *constr*(*x1*)
constr(*x2*)
constr(*x3*)
constr(*x4*)
constr(*x5*)
constr(*x6*)
constr(*x7*)
constr(*x8*)
constr(*x9*) :=

specify_single *H* *x1*;
specify_single *H* *x2*;
specify_single *H* *x3*;
specify_single *H* *x4*;
specify_single *H* *x5*;
specify_single *H* *x6*;
specify_single *H* *x7*;
specify_single *H* *x8*;
specify_single *H* *x9*.

Tactic Notation "specify" *hyp*(*H*) *constr*(*x1*)
constr(*x2*)
constr(*x3*)
constr(*x4*)
constr(*x5*)
constr(*x6*)
constr(*x7*)
constr(*x8*)
constr(*x9*)
constr(*x10*) :=

specify_single *H* *x1*;
specify_single *H* *x2*;
specify_single *H* *x3*;
specify_single *H* *x4*;
specify_single *H* *x5*;
specify_single *H* *x6*;
specify_single *H* *x7*;
specify_single *H* *x8*;
specify_single *H* *x9*;
specify_single *H* *x10*.

Fancy `injection` tactic, which also introduces the equalities produced by `injection` as hypotheses and performs substitutions.

```

a : nat
b : nat
c : nat
H : S a = S b
H0 : a = c

```

```

=====
b = c

```

strip H.

```

b : nat
c : nat
H : c = b

```

```

=====
b = c

```

Ltac *strip h* :=
(injection *h*; clear *h*; intros; subst).

Breaks open a pair *p* and names the components *x* and *y*. Used to call a function returning two values as pair and binding both values to identifiers in one step. Ltac *introduce_pair p x y* :=

```

(let z := fresh in introduce_eq z p;
  destruct z as [ x y ]).

```

New tactic notation: *introduce_pair p x y* can be written as *introduce pair p as x y*.
Tactic Notation "introduce" "pair" *constr(p)* "as" *ident(x) ident(y)* :=
(*introduce_pair p x y*).

Deals with an existential in a hypothesis.

```

H : exists n : nat, forall k : nat, n > k
=====
False

```

elim_ex H x.

```

x : nat
H : forall k : nat, x > k
=====
False

```

Ltac *elim_ex H n* :=
let *H'* := fresh in elim *H*;
 intros *n H'*;

```
clear H;
rename H' into H.
```

New tactic notation: *elim-ex H x* can be written *eliminate existential x in H*. We also define notations to get rid of multiple existentials in one step. Tactic Notation "eliminate"

```
"existential" ident(x) "in" hyp(H) :=
  elim-ex H x.
```

```
Tactic Notation "eliminate" "existentials" ident(x) ident(y) "in" hyp(H) :=
  elim-ex H x; elim-ex H y.
```

```
Tactic Notation "eliminate" "existentials" ident(x) ident(y) ident(z) "in" hyp(H) :=
  elim-ex H x; elim-ex H y; elim-ex H z.
```

```
Tactic Notation "eliminate" "existential" ident(x) "in" hyp(H) "as" ident(H') :=
  elim-ex H x; rename H into H'.
```

```
Tactic Notation "eliminate" "existentials" ident(x) ident(y) "in" hyp(H) "as" ident(H')
:=
  elim-ex H x; elim-ex H y; rename H into H'.
```

Tactic Notation

```
"eliminate" "existentials" ident(x) ident(y) ident(z) "in" hyp(H) "as" ident(H') :=
  elim-ex H x; elim-ex H y; elim-ex H z; rename H into H'.
```

Used where an inequality needs to be proved. Can refine both left and right bound.
Refining the left bound:

$$\begin{array}{ccc} & & X \leq Y \\ X \leq Y & & Y \leq Z \\ \text{-----} & \implies & \text{-----} \quad /\backslash \quad \text{-----} \\ X \leq Z & & Y \leq Z \quad \quad X \leq Z \end{array}$$

where the right goal is proved automatically using the omega tactic.
Refining the right bound:

$$\begin{array}{ccc} & & X \leq Y \\ Y \leq Z & & Y \leq Z \\ \text{-----} & \implies & \text{-----} \quad /\backslash \quad \text{-----} \\ X \leq Z & & X \leq Y \quad \quad X \leq Z \end{array}$$

where the right goal is proved automatically using the omega tactic.

```
Ltac refine_le H :=
```

```
  match goal with
  | [ H : le ?X ?Y ⊢ le ?X ?Z ] ⇒ cut (Y ≤ Z); [ intros; omega | idtac ]
  | [ H : le ?Y ?Z ⊢ le ?X ?Z ] ⇒ cut (X ≤ Y); [ intros; omega | idtac ]
  end.
```

Tactic to perform an algebraic manipulation. *algebraic_rewrite x y* first needs to prove that *x* and *y* are equivalent, which it tries to do automatically (if this fails, a message is

printed and the proof is left to the user). Next, it also rewrites x as y in all hypotheses and the goal.

```
a : nat
b : nat
=====
(a + b) * (a + b) >= a * a
```

`algebraic_rewrite ((a + b) * (a + b)) (a * a + 2 * a * b + b * b).`

```
a : nat
b : nat
=====
a * a + 2 * a * b + b * b >= a * a
```

```
Ltac algebraic_rewrite x y :=
  let H := fresh in assert (H : x = y);
    [ try ring;
      idtac "Failed to automatically prove rewrite (not an error)"
    | rewrite H in × ⊢ ×;
      clear H ].
```

New tactic notation: `algebraic_rewrite x y` can be written as `algebraically rewrite x as y`.
 Tactic Notation "algebraically" "rewrite" *constr(x)* "as" *constr(y)* :=
`algebraic_rewrite x y`.

All-in-one tactic to solve nat-related goals. `Ltac solveq :=`
`simpl in × ⊢ ×; solve [auto with arith | ring | omega].`

Tactic which helps with proving inequalities. Used to bring the lower bound closer to the upper bound.

```
a : nat
b : nat
=====
a - b <= a + 5
```

`refine_le_left a.`

```
a : nat
b : nat
=====
a <= a + 5
```

`Ltac refine_le_left Y :=`

```

match goal with
| ⊢ le ?X ?Z ⇒ let H := fresh in
                assert (H : X ≤ Y);
                [ try solveq; idtac "Failed to refine automatically (not an error)"
                | refine_le H; clear H ]
end.

```

Tactic which helps with proving inequalities. Used to bring the upper bound closer to the lower bound.

```

a : nat
b : nat
=====
a - b <= a + 5

```

`refine_le_right a.`

```

a : nat
b : nat
=====
a - b <= a

```

```

Ltac refine_le_right Y :=
  match goal with
  | ⊢ le ?X ?Z ⇒ let H := fresh in
                  assert (H : Y ≤ Z);
                  [ try solveq; idtac "Failed to refine automatically (not an error)"
                  | refine_le H; clear H ]
  end.

```

New tactic notation: `refine_le_left x` can be written `refine left bound with x`. **Tactic Notation** "refine" "left" "bound" "with" `constr(x)` := `refine_le_left x`.

New tactic notation: `refine_le_right x` can be written `refine right bound with x`. **Tactic Notation** "refine" "right" "bound" "with" `constr(x)` := `refine_le_right x`.

1.2 General definitions

1.2.1 Decidability

We define decidability as a Set, so that we can use it in algorithms. **Definition** *decidable* $(P : \text{Prop}) := \{P\} + \{\sim P\}$.

decidable_eq A means that we can decide whether two values of type *A* are equal or not.
Definition *decidable_eq* (*A* : **Set**) :=
 $\forall x y : A, \text{decidable } (x = y).$

1.2.2 Equivalent functions

We say two functions *f* and *g* with domain *A* are equivalent when $f x = g x$, for all *x* in *A*.
Function equivalence is an equivalence relation, as will be proved in later theorems.

Needs manual checking **Definition** *function* (*A B* : **Type**) :=
 $A \rightarrow B.$

Needs manual checking **Definition** *equivalent_functions* (*A B* : **Type**) (*f g* : *function*
A B) :=
 $\forall (x : A), f x = g x.$

Implicit Arguments *equivalent_functions* [*A B*].

We show a function is equivalent with itself (reflexivity). **Theorem** *equivalent_functions_refl*
: $\forall A B f,$
(*@equivalent_functions A B f f*).

Proof.

`compute; trivial.`

Qed.

We show that if *f* is equivalent with *g* and *g* is equivalent with *h*, then *f* is equivalent
with *h* (transitivity). **Theorem** *equivalent_functions_trans* : $\forall A B f g h,$
(*@equivalent_functions A B f g*) \rightarrow
(*@equivalent_functions A B g h*) \rightarrow
(*@equivalent_functions A B f h*).

Proof.

`compute; intros.`

`specify H x; specify H0 x; rewrite H; trivial.`

Qed.

We show that if *f* is equivalent with *g*, then *g* is equivalent with *f* (symmetricity).
Theorem *equivalent_functions_symm* : $\forall A B f g,$
(*@equivalent_functions A B f g*) \rightarrow
(*@equivalent_functions A B g f*).

Proof.

`compute; intros.`

`specify H x; symmetry; trivial.`

Qed.

Implicit Arguments *equivalent_functions_refl* [*A B*].

Implicit Arguments *equivalent_functions_trans* [*A B*].

Implicit Arguments *equivalent_functions_trans* [*A B*].

We put these theorems in the hint database so that we can use the `auto` tactic to easily prove goals that rely on these properties. `Hint Resolve equivalent_functions_refl`.

`Hint Resolve equivalent_functions_symm`.

Add *Relation function equivalent_functions*
 reflexivity proved by equivalent_functions_refl
 symmetry proved by equivalent_functions_symm
 transitivity proved by equivalent_functions_trans
as *equivalent_functions_rel*.

Functional extensionality. Taken from Coq 8.2's standard library. `Axiom functional_extensionality_dep`
: $\forall A (B : A \rightarrow \text{Type}),$
 $\forall (f g : \forall x : A, B x),$
 $(\forall x, f x = g x) \rightarrow f = g.$

`Lemma functional_extensionality A B (f g : A \rightarrow B) :`
 $(\forall x, f x = g x) \rightarrow f = g.$

`Proof.`

`intros.`
 `apply functional_extensionality_dep.`
 `trivial.`

`Qed.`

`Lemma equal_f : $\forall (A B : \text{Type}) (f g : A \rightarrow B),$`
 $f = g \rightarrow \forall x, f x = g x.$

`Proof.`

`intros.`
 `subst.`
 `trivial.`

`Qed.`

`Lemma eta_expansion_dep A (B : A \rightarrow Type) (f : $\forall x : A, B x) :$`
 $f = \text{fun } x \Rightarrow f x.$

`Proof.`

`intros.`
 `apply functional_extensionality_dep.`
 `trivial.`

`Qed.`

`Lemma eta_expansion A B (f : A \rightarrow B) : $f = \text{fun } x \Rightarrow f x.$`

`Proof.`

`intros.`
 `apply eta_expansion_dep.`

`Qed.`

1.3 Paper specific

1.3.1 Language definitions and theorems

We define identifiers as natural numbers (*nat*). We don't rely on this internal representation (we make *id* opaque a bit later) except for the *id*-set definitions, for which we need an ordered type; for this reason, *nat* seemed like a natural choice.

Needs manual checking `Definition id := nat.`

A vid (versioned id) is an id with a version number. The `%type` suffix tells Coq it has to evaluate (*id* × *nat*) in the type scope, otherwise it will interpret × as *nat*-multiplication.

Needs manual checking `Definition vid := (id × nat)%type.`

We define *id* as an ordered type (will be used later to define identifier-sets). `Module Identifier_OT <: OrderedType.`

`Definition t := id.`

`Definition eq (x y : t) := x = y.`

`Definition lt (x y : t) := x < y.`

`Theorem eq_refl : ∀ x, eq x x.`

`Proof.`

`unfold eq; auto.`

`Qed.`

`Theorem eq_sym : ∀ x y : t, eq x y → eq y x.`

`unfold eq; auto.`

`Qed.`

`Theorem eq_trans : ∀ x y z, eq x y → eq y z → eq x z.`

`Proof.`

`unfold eq; intros; subst; trivial.`

`Qed.`

`Definition lt_trans := lt_trans.`

`Theorem lt_not_eq : ∀ x y, lt x y → ¬ eq x y.`

`Proof.`

`unfold lt; unfold eq; intros; induction x; destruct y.`

`inversion H.`

`discriminate.`

`discriminate.`

`red; intros.`

`strip H0; elim (lt_irrefl (S y)); trivial.`

`Qed.`

`Definition compare : ∀ x y, Compare lt eq x y.`

`Proof.`

```

  unfold lt; unfold eq; intros; destruct (gt_eq_gt_dec x y).
  destruct s.
  red in g; apply LT; trivial.
  apply EQ; trivial.
  red in g; apply GT; trivial.

```

Defined.

End *Identifier_OT*.

We define sets of identifiers `Module IdSet := FSetList.Make(Identifier_OT)`.

As promised earlier, we make `id` opaque, making sure we don't make use of its internal implementation. *Opaque id*.

Generate a few extra theorems for `IdSets`. `Module IdSetProperties := Properties (IdSet)`.

If \cdot , then `and` \cdot . `Lemma union_subset : $\forall x y z$,
IdSet.Subset (IdSet.union x y) z \rightarrow
IdSet.Subset x z \wedge IdSet.Subset y z.`

Proof.

```

  unfold IdSet.Subset in  $\times \vdash \times$ ; intros.
  split; intros.
  apply (H - (@IdSet.union_2 x y a H0)).
  apply (H - (@IdSet.union_3 x y a H0)).

```

Qed.

Decidability theorems stated as a definition to make it transparent (needed for the single assignment algorithm to be "fully evaluateable").

This theorem states that we can decide whether two *ids* are equal or not.

Needs manual checking `Definition decidable_eq_id : decidable_eq id`.

```

  unfold decidable_eq; unfold decidable; intros.
  apply (eq_nat_dec x y).

```

Defined.

States that we can decide whether two *vids* are equal or not.

Needs manual checking `Definition decidable_eq_vid : decidable_eq vid`.

```

  unfold decidable_eq; unfold decidable; unfold vid; intros.
  destruct x; destruct y.
  rename i0 into j; rename n0 into m.
  destruct (decidable_eq_id i j);
  destruct (eq_nat_dec n m);
  try (left; subst; trivial; fail);
  right; red; intros; injection H; intros; contradiction.

```

Defined.

We define the set of values as well as two elements: `T` (true) and "some other value" `F` which is not equal to true.

Needs manual checking **Parameters**

(*value* : **Set**)

(*T* : *value*)

(*F* : *value*).

We state we can check whether two *values* are equal.

Needs manual checking **Axiom** *decidable_eq_value* : *decidable_eq value*.

We have defined the existence of two *values*, *T* and *F*. We define them to be unequal.

Needs manual checking **Axiom** *T_neq_F* : *T* \neq *F*.

A *store* is a total mapping from *ids* to *values*.

Needs manual checking **Definition** *store* := *id* \rightarrow *value*.

A *vstore* is a total mapping from *ids* to *values*.

Needs manual checking **Definition** *vstore* := *vid* \rightarrow *value*.

A *vmap* (short for version map) maps non-versioned identifiers (*vids*) to a version number (*nat*).

Needs manual checking **Definition** *vmap* := *id* \rightarrow *nat*.

An expression is a total mapping from *stores* to *values*.

Needs manual checking **Definition** *expr* := *store* \rightarrow *value*.

Definition *vexpr* := *vstore* \rightarrow *value*.

The original set of commands.

Needs manual checking **Inductive** *command* : **Set** :=

| *cAssert* : *expr* \rightarrow *command*
| *cAssume* : *expr* \rightarrow *command*
| *cAssign* : *id* \rightarrow *expr* \rightarrow *command*
| *cSequence* : *command* \rightarrow *command* \rightarrow *command*
| *cSkip* : *command*
| *cChoice* : *command* \rightarrow *command* \rightarrow *command*.

The versioned set of commands. Will be produced by the *transform_sa* algorithm.

Needs manual checking **Inductive** *vcommand* : **Set** :=

| *vcAssert* : *vexpr* \rightarrow *vcommand*
| *vcAssume* : *vexpr* \rightarrow *vcommand*
| *vcAssign* : *vid* \rightarrow *vexpr* \rightarrow *vcommand*
| *vcSequence* : *vcommand* \rightarrow *vcommand* \rightarrow *vcommand*
| *vcSkip* : *vcommand*
| *vcChoice* : *vcommand* \rightarrow *vcommand* \rightarrow *vcommand*.

The versioned set of commands, without assignment Will be produced by the *passify* algorithm.

Needs manual checking **Inductive** *pcommand* : **Set** :=

| *pcAssert* : *vexpr* \rightarrow *pcommand*
| *pcAssume* : *vexpr* \rightarrow *pcommand*
| *pcSequence* : *pcommand* \rightarrow *pcommand* \rightarrow *pcommand*

| *pcSkip* : *pcommand*
 | *pcChoice* : *pcommand* → *pcommand* → *pcommand*.

Metric on commands.

Needs manual checking **Fixpoint** *command_metric* (*c* : *command*) : *nat* :=

```
match c with
| cAssert _ ⇒ 2
| cAssume _ ⇒ 2
| cAssign _ _ ⇒ 2
| cSequence x y ⇒ S (command_metric x + command_metric y)
| cSkip ⇒ 1
| cChoice x y ⇒ S (command_metric x + command_metric y)
```

end.

Metric on vcommands.

Needs manual checking **Fixpoint** *vcommand_metric* (*c* : *vcommand*) : *nat* :=

```
match c with
| vcAssert _ ⇒ 2
| vcAssume _ ⇒ 2
| vcAssign _ _ ⇒ 2
| vcSequence x y ⇒ S (vcommand_metric x + vcommand_metric y)
| vcSkip ⇒ 1
| vcChoice x y ⇒ S (vcommand_metric x + vcommand_metric y)
```

end.

Metric on pcommands.

Needs manual checking **Fixpoint** *pcommand_metric* (*c* : *pcommand*) : *nat* :=

```
match c with
| pcAssert _ ⇒ 2
| pcAssume _ ⇒ 2
| pcSequence x y ⇒ S (pcommand_metric x + pcommand_metric y)
| pcSkip ⇒ 1
| pcChoice x y ⇒ S (pcommand_metric x + pcommand_metric y)
```

end.

Create a versioned expression of a command. E.g. the expression "x == y + 1" will be transformed to "x_5 == y_3 + 1" under the version map { x -> 5, y -> 3 }. **Definition**

```
version_expr (e : expr) (v : vmap) : vexpr :=
fun (vmu : vstore) ⇒ e (fun x ⇒ vmu (x, v x)).
```

Modifies a single mapping of a total function.

```
(rebind f x y) x = y
(rebind f x y) x' = f x' with x <> x'
```

Needs manual checking **Definition** *rebind* (*A B* : **Set**)
 (*eq_dec* : *decidable_eq* *A*)

```

      (f : A → B)
      (x : A)
      (y : B) :=
  fun a => if eq_dec a x then y else f a.

```

Implicit Arguments *rebind* [*A B*].

Increment the version of an identifier.

E.g.

```

  inc { x -> 5, y -> 3, ... } x = { x -> 6, y -> 3, ... }

```

Definition *inc* (*v* : *vmap*) (*x* : *id*) :=
rebind decidable_eq_id v x (S (v x)).

Updates a store binding.

Needs manual checking **Definition** *update_store* (*mu* : *store*) (*x* : *id*) (*v* : *value*) :=
rebind decidable_eq_id mu x v.

Update a versioned store binding.

Needs manual checking **Definition** *update_vstore* (*mu* : *vstore*) (*x* : *vid*) (*v* : *value*)
:=
rebind decidable_eq_vid mu x v.

Checks if a regular store and a versioned store are equivalent under a certain version map.

E.g. the regular store { *x* -> 12, *y* -> 44 } is equivalent with the versioned store { *x*₀ -> -5, *x*₁ -> 2, *x*₂ -> 12, *y*₀ -> 44 } under the version map { *x* -> 2, *y* -> 0 } **Definition** *store_sync_vstore* (*mu* : *store*) (*v* : *vmap*) (*vmu* : *vstore*) :=
equivalent_functions mu (fun x => vmu (x, v x)).

Two expression evaluate to the same value under equivalent stores.

Needs manual checking **Axiom** *expression_evaluation* : $\forall (e : \text{expr}) \mu \mu',$
equivalent_functions mu mu' $\rightarrow e \mu = e \mu'$.

Theorem *sync_stores* : $\forall \mu v \text{vmu} (e : \text{expr}),$
store_sync_vstore mu v vmu $\rightarrow e \mu = ((\text{version_expr } e \ v) \text{vmu}).$

Proof.

```

  intros.
  unfold version_expr.
  unfold store_sync_vstore in H.
  apply expression_evaluation.
  trivial.

```

Qed.

States for the original command language.

Needs manual checking **Inductive** *state* : **Set** :=
| *ip* : *command* → *store* → *state*
| *failure* : *store* → *state*.

States for the single assignment phase.

Needs manual checking **Inductive** *vstate* : **Set** :=

| *vip* : *vcommand* → *vstore* → *vstate*
| *vfailure* : *vstore* → *vstate*.

States for the passification phase.

Needs manual checking **Inductive** *pstate* : **Set** :=

| *pip* : *pcommand* → *pstate*
| *pfailure* : *pstate*.

Metric on states **Definition** *state_metric* (*s* : *state*) : *nat* :=

match *s* **with**
| *ip* *c* _ ⇒ *command_metric* *c*
| *failure* _ ⇒ *O*

end.

Metric on vstates **Definition** *vstate_metric* (*s* : *vstate*) : *nat* :=

match *s* **with**
| *vip* *c* _ ⇒ *vcommand_metric* *c*
| *vfailure* _ ⇒ *O*

end.

Metric on pstates **Definition** *pstate_metric* (*s* : *pstate*) : *nat* :=

match *s* **with**
| *pip* *c* ⇒ *pcommand_metric* *c*
| *pfailure* ⇒ *O*

end.

Regular single step operational semantics.

Needs manual checking **Inductive** *step* : *state* → *state* → **Prop** :=

| *stepAssertT* : ∀ *e mu*,
 e mu = *T* →
 step (*ip* (*cAssert* *e*) *mu*) (*ip* *cSkip* *mu*)

| *stepAssertF* : ∀ *e mu*,
 e mu ≠ *T* →
 step (*ip* (*cAssert* *e*) *mu*) (*failure* *mu*)

| *stepAssume* : ∀ *e mu*,
 e mu = *T* →
 step (*ip* (*cAssume* *e*) *mu*) (*ip* *cSkip* *mu*)

| *stepSeq* : ∀ *c1 c1' c2 mu mu'*,
 step (*ip* *c1* *mu*) (*ip* *c1'* *mu'*) →
 step (*ip* (*cSequence* *c1* *c2*) *mu*)
 (*ip* (*cSequence* *c1'* *c2*) *mu'*)

| *stepSeqSkip* : $\forall c2 \ mu,$
 $step (ip (cSequence cSkip c2) mu) (ip c2 mu)$

| *stepSeqFail* : $\forall c1 \ c2 \ mu \ mu',$
 $step (ip c1 mu) (failure mu') \rightarrow$
 $step (ip (cSequence c1 c2) mu) (failure mu')$

| *stepAssign* : $\forall x \ e \ mu,$
 $step (ip (cAssign x e) mu)$
 $(ip cSkip (update_store mu x (e mu)))$

| *stepChoiceL* : $\forall c1 \ c2 \ mu,$
 $step (ip (cChoice c1 c2) mu)$
 $(ip c1 mu)$

| *stepChoiceR* : $\forall c1 \ c2 \ mu,$
 $step (ip (cChoice c1 c2) mu)$
 $(ip c2 mu).$

Versioned single step operational semantics.

Needs manual checking **Inductive** *vstep* : *vstate* \rightarrow *vstate* \rightarrow **Prop** :=

| *vstepAssertT* : $\forall e \ vmu,$
 $e \ vmu = T \rightarrow$
 $vstep (vip (vcAssert e) vmu) (vip vcSkip vmu)$

| *vstepAssertF* : $\forall e \ vmu,$
 $e \ vmu \neq T \rightarrow$
 $vstep (vip (vcAssert e) vmu) (vfailure vmu)$

| *vstepAssume* : $\forall e \ vmu, e \ vmu = T \rightarrow$
 $vstep (vip (vcAssume e) vmu) (vip vcSkip vmu)$

| *vstepSeq* : $\forall c1 \ c1' \ c2 \ vmu \ vmu',$
 $vstep (vip c1 vmu) (vip c1' vmu') \rightarrow$
 $vstep (vip (vcSequence c1 c2) vmu)$
 $(vip (vcSequence c1' c2) vmu')$

| *vstepSeqSkip* : $\forall c2 \ vmu,$
 $vstep (vip (vcSequence vcSkip c2) vmu) (vip c2 vmu)$

| *vstepSeqFail* : $\forall c1 \ c2 \ vmu \ vmu',$
 $vstep (vip c1 vmu) (vfailure vmu') \rightarrow$

$$\begin{aligned}
& \text{vstep } (\text{vip } (\text{vcSequence } c1 \ c2) \ \text{vmu}) \ (\text{vfailure } \text{vmu}') \\
| \text{vstepAssign} & : \forall x \ e \ \text{vmu}, \\
& \text{vstep } (\text{vip } (\text{vcAssign } x \ e) \ \text{vmu}) \\
& \quad (\text{vip } \text{vcSkip } (\text{update_vstore } \text{vmu } x \ (e \ \text{vmu}))) \\
| \text{vstepChoiceL} & : \forall c1 \ c2 \ \text{vmu}, \\
& \text{vstep } (\text{vip } (\text{vcChoice } c1 \ c2) \ \text{vmu}) \ (\text{vip } c1 \ \text{vmu}) \\
| \text{vstepChoiceR} & : \forall c1 \ c2 \ \text{vmu}, \\
& \text{vstep } (\text{vip } (\text{vcChoice } c1 \ c2) \ \text{vmu}) \ (\text{vip } c2 \ \text{vmu}). \\
& \text{Versioned stateless single step operational semantics.} \\
& \text{Needs manual checking } \mathbf{Inductive} \ \text{pstep } (\text{vmu} : \text{vstore}) : \text{pstate} \rightarrow \text{pstate} \rightarrow \mathbf{Prop} := \\
| \text{pstepAssertT} & : \forall e, \\
& e \ \text{vmu} = T \rightarrow \\
& \text{pstep } \text{vmu} \ (\text{pip } (\text{pcAssert } e)) \ (\text{pip } \text{pcSkip}) \\
| \text{pstepAssertF} & : \forall e, \\
& e \ \text{vmu} \neq T \rightarrow \\
& \text{pstep } \text{vmu} \ (\text{pip } (\text{pcAssert } e)) \ \text{pfailure} \\
| \text{pstepAssume} & : \forall e, \\
& e \ \text{vmu} = T \rightarrow \\
& \text{pstep } \text{vmu} \ (\text{pip } (\text{pcAssume } e)) \ (\text{pip } \text{pcSkip}) \\
| \text{pstepSeq} & : \forall c1 \ c1' \ c2, \\
& \text{pstep } \text{vmu} \ (\text{pip } c1) \ (\text{pip } c1') \rightarrow \\
& \text{pstep } \text{vmu} \ (\text{pip } (\text{pcSequence } c1 \ c2)) \ (\text{pip } (\text{pcSequence } c1' \ c2)) \\
| \text{pstepSeqSkip} & : \forall c2, \\
& \text{pstep } \text{vmu} \ (\text{pip } (\text{pcSequence } \text{pcSkip } c2)) \\
& \quad (\text{pip } c2) \\
| \text{pstepSeqFail} & : \forall c1 \ c2, \\
& \text{pstep } \text{vmu} \ (\text{pip } c1) \ \text{pfailure} \rightarrow \\
& \text{pstep } \text{vmu} \ (\text{pip } (\text{pcSequence } c1 \ c2)) \ \text{pfailure} \\
| \text{pstepChoiceL} & : \forall c1 \ c2, \\
& \text{pstep } \text{vmu} \ (\text{pip } (\text{pcChoice } c1 \ c2)) \ (\text{pip } c1) \\
| \text{pstepChoiceR} & : \forall c1 \ c2, \\
& \text{pstep } \text{vmu} \ (\text{pip } (\text{pcChoice } c1 \ c2)) \ (\text{pip } c2).
\end{aligned}$$

Regular multistep.

Needs manual checking `Inductive multistep : state → state → Prop :=`

| `multiReflexivity` : $\forall s, \text{multistep } s \ s$

| `multiStep` : $\forall s1 \ s2 \ s3,$

`step` $s1 \ s2 \rightarrow$

`multistep` $s2 \ s3 \rightarrow$

`multistep` $s1 \ s3$.

Versioned multistep.

Needs manual checking `Inductive vmultistep : vstate → vstate → Prop :=`

| `vmultiReflexivity` : $\forall s, \text{vmultistep } s \ s$

| `vmultiStep` : $\forall s1 \ s2 \ s3,$

`vstep` $s1 \ s2 \rightarrow$

`vmultistep` $s2 \ s3 \rightarrow$

`vmultistep` $s1 \ s3$.

Versioned stateless multistep.

Needs manual checking `Inductive pmultistep (vmu : vstore) : pstate → pstate → Prop`

`:=`

| `pmultiReflexivity` : $\forall s, \text{pmultistep } \text{vmu } s \ s$

| `pmultiStep` : $\forall s1 \ s2 \ s3,$

`pstep` $\text{vmu } s1 \ s2 \rightarrow$

`pmultistep` $\text{vmu } s2 \ s3 \rightarrow$

`pmultistep` $\text{vmu } s1 \ s3$.

Helper tactic to perform induction on `step/multistep/vstep/vmultistep`. Using the `induction` tactic sometimes throws important information away.

For example, if we don't use `introduce_step_states`, we get

```
c : command
c' : command
mu : store
mu' : store
H : multistep (ip c mu) (ip c' mu')
=====
command_metric c' <= command_metric c
```

`induction H.`

`subgoal 1 is:`

```
c : command
c' : command
mu : store
mu' : store
```

```

s : state
=====
command_metric c' <= command_metric c

```

subgoal 2 is:

```

c : command
c' : command
mu : store
mu' : store
s1 : state
s2 : state
s3 : state
H : step s1 s2
H0 : multistep s2 s3
IHmultistep : command_metric c' <= command_metric c
=====
command_metric c' <= command_metric c

```

As you can see, there is no way of solving subgoal 1: the hypotheses do not provide any kind of information about either c or c' . We can use *introduce_step_states* to preserve this information:

```

c : command
c' : command
mu : store
mu' : store
H : multistep (ip c mu) (ip c' mu')
=====
command_metric c' <= command_metric c

```

`introduce_step_states H s s'.`

```

c : command
c' : command
mu : store
mu' : store
s : state
H0 : s = ip c mu
s' : state
H1 : s' = ip c' mu'
H : multistep s s'
=====

```

```

    command_metric c' <= command_metric c

revert c c' mu mu' H0 H1.

s : state
s' : state
H : multistep s s'
=====
forall (c c' : command) (mu mu' : store),
  s = ip c mu -> s' = ip c' mu' -> command_metric c' <= command_metric c

subgoal 1 is:

c : command
c' : command
mu : store
mu' : store
H1 : ip c mu = ip c' mu'
=====
  command_metric c' <= command_metric c

subgoal 2 is:

s2 : state
c : command
c' : command
mu : store
mu' : store
H0 : multistep s2 (ip c' mu')
IHmultistep : forall (c c'0 : command) (mu mu'0 : store),
  s2 = ip c mu -> ip c' mu' = ip c'0 mu'0 -> command_metric c'0 <= command_metric c
H : step (ip c mu) s2
=====
  command_metric c' <= command_metric c

Ltac introduce_step_states H s s' :=
match goal with
| [ H : step ?X ?Y ⊢ _ ] =>
  introduce_eq_in s X H; introduce_eq_in s' Y H
| [ H : multistep ?X ?Y ⊢ _ ] =>
  introduce_eq_in s X H; introduce_eq_in s' Y H
| [ H : vstep ?X ?Y ⊢ _ ] =>
  introduce_eq_in s X H; introduce_eq_in s' Y H

```

```

| [ H : vmultistep ?X ?Y ⊢ _ ] ⇒
  introduce_eq_in s X H; introduce_eq_in s' Y H
| _ ⇒ idtac "No step hypothesis found"
end.

```

Same as *introduce_step_states*, but adapted to *pstep* and *pmultistep*. Ltac *introduce_step_states_p* *H vmu s s'* :=

```

match goal with
| [ H : pstep ?S ?X ?Y ⊢ _ ] ⇒
  introduce_eq_in vmu S H;
  introduce_eq_in s X H;
  introduce_eq_in s' Y H
| [ H : pmultistep ?S ?X ?Y ⊢ _ ] ⇒
  introduce_eq_in vmu S H;
  introduce_eq_in s X H;
  introduce_eq_in s' Y H
| _ ⇒ idtac "No step hypothesis found"
end.

```

New tactic notation: *introduce_step_states H s s'* can be written *introduce states s s' in H*. Tactic Notation "introduce" "states" *ident(s) ident(s')* "in" *hyp(H)* := *introduce_step_states H s s'*.

New tactic notation: *introduce_step_states_p H vmu s s'* can be written *introduce states vmu s s' in H*. Tactic Notation "introduce" "states" *ident(vmu) ident(s) ident(s')* "in" *hyp(H)* := *introduce_step_states_p H vmu s s'*.

Splits up a goal *multistep s1 s2* to *step s1 Y* and *step Y s2*, and similarly for *vmultistep* and *pmultistep*. This tactic just makes it unnecessary to copy-paste *s1* and *s2* when applying the *multiStep/vmultistep/pmultistep* theorem. Ltac *split_step_in_goal Y* :=

```

match goal with
| ⊢ multistep ?X ?Z ⇒ apply ( multiStep X Y Z)
| ⊢ vmultistep ?X ?Z ⇒ apply ( vmultiStep X Y Z)
| ⊢ pmultistep ?S ?X ?Z ⇒ apply ( pmultiStep S X Y Z)
| _ ⇒ fail
end.

```

New tactic notation: *split_step_in_goal X* can be written *step X*. Tactic Notation "step" *constr(X)* := *split_step_in_goal X*.

Builds up a (finite) set of identifiers which are assigned to in the given command *Fixpoint targets (c : command) : IdSet.t* :=

```

match c with
| cAssert _ ⇒ IdSet.empty
| cAssume _ ⇒ IdSet.empty
| cAssign x _ ⇒ IdSet.singleton x

```

```

| cSequence c1 c2 ⇒ IdSet.union (targets c1) (targets c2)
| cChoice c1 c2 ⇒ IdSet.union (targets c1) (targets c2)
| cSkip ⇒ IdSet.empty

```

end.

When needing to prove $multistep (ip (cChoice c1 c2) mu)$, takes the left path and thus changes the goal to $multistep (ip c1 mu)$. Similarly for $vmultistep$ and $pmultistep$. **Ltac** *step_choice_left* :=

```

match goal with
| ⊢ multistep ( ip ( cChoice ?C1 ?C2 ) ?MU ) ?S ⇒ step ( ip C1 MU); [ constructor | idtac
|
| ⊢ vmultistep ( vip ( vcChoice ?C1 ?C2 ) ?VMU ) ?S ⇒ step ( vip C1 VMU); [ constructor
| idtac ]
| ⊢ pmultistep ?VMU ( pip ( pcChoice ?C1 ?C2 ) ) ?S ⇒ step ( pip C1 ); [ constructor | idtac
|

```

end.

When needing to prove $multistep (ip (cChoice c1 c2) mu)$, takes the right path and thus changes the goal to $multistep (ip c2 mu)$. Similarly for $vmultistep$ and $pmultistep$. **Ltac** *step_choice_right* :=

```

match goal with
| ⊢ multistep ( ip ( cChoice ?C1 ?C2 ) ?MU ) ?S ⇒
    step ( ip C2 MU); [ constructor | idtac ]
| ⊢ vmultistep ( vip ( vcChoice ?C1 ?C2 ) ?VMU ) ?S ⇒
    step ( vip C2 VMU); [ constructor | idtac ]
| ⊢ pmultistep ?VMU ( pip ( pcChoice ?C1 ?C2 ) ) ?S ⇒
    step ( pip C2 ); [ constructor | idtac ]

```

end.

New tactic notation: *step_choice_left* can be written *step choice left*. **Tactic Notation** "step" "choice" "left" := *step_choice_left*.

New tactic notation: *step_choice_right* can be written *step choice right*. **Tactic Notation** "step" "choice" "right" := *step_choice_right*.

A few theorems about the steps and multisteps relations. **Lemma** *multistep_ip_ip* : $\forall s c' mu'$,

$multistep s (ip c' mu') \rightarrow \exists c, \exists mu, s = ip c mu.$

Proof.

```

intros.
destruct s.
∃ c; ∃ s; trivial.
inversion H; subst.
inversion H0.

```

Qed.

Lemma *vmultistep_vip_vip* : $\forall s c' vmu'$,

$vmultistep\ s\ (vip\ c'\ vmu') \rightarrow \exists\ c, \exists\ vmu, s = vip\ c\ vmu.$

Proof.

```
intros.
destruct s.
 $\exists\ v; \exists\ v0; \text{trivial}.$ 
inversion H; subst.
inversion H0.
```

Qed.

Lemma *pmultistep_pip_pip* : $\forall\ s\ c'\ vmu,$
 $pmultistep\ vmu\ s\ (pip\ c') \rightarrow \exists\ c, s = pip\ c.$

Proof.

```
intros.
destruct s.
 $\exists\ p; \text{trivial}.$ 
inversion H; subst.
inversion H0.
```

Qed.

If a sequence *c1; c2* goes to skip, both *c1* and *c2* go to skip separately. Theorem *multistep_seq_skip* :

```
 $\forall\ c1\ c2\ mu\ mu'',$   
 $multistep\ (ip\ (cSequence\ c1\ c2)\ mu)\ (ip\ cSkip\ mu'') \rightarrow$   
 $\exists\ mu', multistep\ (ip\ c1\ mu)\ (ip\ cSkip\ mu') \wedge multistep\ (ip\ c2\ mu')\ (ip\ cSkip\ mu'').$ 
```

Proof.

```
intros.
introduce states s1 s2 in H.
revert c1 c2 mu mu'' H0 H1.
induction H; intros; subst.
discriminate.
elim (multistep_ip_ip _ _ _ H0); intros; subst.
elim H1; intros; subst; clear H1.
rename x into c0; rename x0 into mu0.
inversion H; subst.
elim (IHmultistep _ _ _ (refl_equal _) (refl_equal _)); clear IHmultistep; intros.
destruct H1.
 $\exists\ x.$ 
split.
apply (multiStep _ _ _ H2 H1).
trivial.
 $\exists\ mu0; \text{split}.$ 
constructor.
trivial.
```

Qed.

If a sequence $c1; c2$ goes to skip, both $c1$ and $c2$ go to skip separately. **Theorem**
 $vmultistep_seq_skip :$

$\forall c1\ c2\ vmu\ vmu'',$
 $vmultistep\ (vip\ (vcSequence\ c1\ c2)\ vmu)\ (vip\ vcSkip\ vmu'') \rightarrow$
 $\exists\ vmu',\ vmultistep\ (vip\ c1\ vmu)\ (vip\ vcSkip\ vmu') \wedge\ vmultistep\ (vip\ c2\ vmu')\ (vip$
 $vcSkip\ vmu'').$

Proof.

`intros.`
`introduce\ states\ s1\ s2\ in\ H.`
`revert\ c1\ c2\ vmu\ vmu''\ H0\ H1.`
`induction\ H;\ intros;\ subst.`
`discriminate.`
`elim\ (vmultistep_vip_vip\ _ _ _ H0);\ intros;\ subst.`
`elim\ H1;\ intros;\ subst;\ clear\ H1.`
`rename\ x\ into\ c0;\ rename\ x0\ into\ mu0.`
`inversion\ H;\ subst.`
`elim\ (IHvmultistep\ _ _ _ _ (refl_equal\ _)\ (refl_equal\ _));\ clear\ IHvmultistep;\ intros.`
`destruct\ H1.`
 `$\exists\ x.$`
`split.`
`apply\ (vmultiStep\ _ _ _ H2\ H1).`
`trivial.`
 `$\exists\ mu0;\ split.$`
`constructor.`
`trivial.`

Qed.

Lemma to turn a proof of $multistep\ s1\ s2$ into one of $step\ s1\ s2$. **Lemma**
 $multistep_step_to_multi : \forall\ s1\ s2,\ step\ s1\ s2 \rightarrow multistep\ s1\ s2.$

Proof.

`intros.`
`apply\ (multiStep\ s1\ s2\ s2);\ [trivial | constructor].`

Qed.

Lemma to turn a proof of $vmultistep\ s1\ s2$ into one of $vstep\ s1\ s2$. **Lemma**
 $vmultistep_step_to_multi : \forall\ s1\ s2,\ vstep\ s1\ s2 \rightarrow vmultistep\ s1\ s2.$

Proof.

`intros.`
`apply\ (vmultiStep\ s1\ s2\ s2);\ [trivial | constructor].`

Qed.

Lemma to turn a proof of $pmultistep\ vmu\ s1\ s2$ into one of $pstep\ vmu\ s1\ s2$. **Lemma**
 $pmultistep_step_to_multi : \forall\ vmu\ s1\ s2,\ pstep\ vmu\ s1\ s2 \rightarrow pmultistep\ vmu\ s1\ s2.$

Proof.

`intros.`

apply (pmultiStep vmu s1 s2 s2); [trivial | constructor].
 Qed.

If a sequence c1; c2 fails, either c1 fails, or c1 skips and c2 fails. **Theorem multi-step-seq-fail :**

$\forall c1\ c2\ mu\ mu'',$
 $multistep\ (ip\ (cSequence\ c1\ c2)\ mu)\ (failure\ mu'') \rightarrow$
 $multistep\ (ip\ c1\ mu)\ (failure\ mu'') \vee (\exists\ mu',\ multistep\ (ip\ c1\ mu)\ (ip\ cSkip\ mu') \wedge$
 $multistep\ (ip\ c2\ mu')\ (failure\ mu'')).$

Proof with subst.

intros.
 introduce states s1 s2 in H.
 revert c1 c2 mu mu'' H0 H1.
 induction H; intros...
 discriminate.
 inversion H...
 destruct (IHmultistep - - - (refl_equal -) (refl_equal -)).
 left.
 step (ip c1' mu'); trivial.
 elim H1; intros; clear IHmultistep H1.
 destruct H2.
 right.
 $\exists x$; split.
 step (ip c1' mu'); trivial.
 trivial.
 right; $\exists mu$; split.
 constructor.
 trivial.
 inversion H0...
 left; apply multistep_step_to_multi.
 trivial.
 inversion H1.

Qed.

If a sequence c1; c2 fails, either c1 fails, or c1 skips and c2 fails. **Theorem vmulti-step-seq-fail :**

$\forall c1\ c2\ vmu\ vmu'',$
 $vmultistep\ (vip\ (vcSequence\ c1\ c2)\ vmu)\ (vfailure\ vmu'') \rightarrow$
 $vmultistep\ (vip\ c1\ vmu)\ (vfailure\ vmu'') \vee (\exists\ vmu',\ vmultistep\ (vip\ c1\ vmu)\ (vip\ vcSkip\ vmu')$
 $\wedge\ vmultistep\ (vip\ c2\ vmu')\ (vfailure\ vmu'')).$

Proof with subst.

intros.
 introduce states s1 s2 in H.

```

revert c1 c2 vmu vmu" H0 H1.
induction H; intros...

discriminate.

inversion H...
destruct (IHvmultistep _ _ _ (refl_equal _) (refl_equal _)); clear IHvmultistep.
left; step (vip c1' vmu'); trivial.
elim H1; intros; clear H1.
destruct H2.
right;  $\exists x$ .
split.
step (vip c1' vmu'); trivial.
trivial.
right;  $\exists vmu$ ; split.
constructor.
trivial.
inversion H0...
left.
apply vmultistep_step_to_multi.
trivial.
inversion H1.

```

Qed.

If a choice $c1\ c2$ skips, then $c1$ skips or $c2$ skips. **Theorem *multistep_choice_skip*** :

$$\forall c1\ c2\ mu\ mu',$$

$$multistep\ (ip\ (cChoice\ c1\ c2)\ mu)\ (ip\ cSkip\ mu') \rightarrow multistep\ (ip\ c1\ mu)\ (ip\ cSkip\ mu') \vee multistep\ (ip\ c2\ mu)\ (ip\ cSkip\ mu').$$

Proof with subst.

```

intros.
inversion H...
inversion H0; subst; solve [ left; trivial | right; trivial ].

```

Qed.

If a choice $c1\ c2$ fails, then either $c1$ or $c2$ fails. **Theorem *multistep_choice_fail*** :

$$\forall c1\ c2\ mu\ mu',$$

$$multistep\ (ip\ (cChoice\ c1\ c2)\ mu)\ (failure\ mu') \rightarrow multistep\ (ip\ c1\ mu)\ (failure\ mu') \vee multistep\ (ip\ c2\ mu)\ (failure\ mu').$$

Proof with subst.

```

intros.
inversion H...
inversion H0; subst; solve [ left; trivial | right; trivial ].

```

Qed.

If a choice $c1\ c2$ skips, then $c1$ skips or $c2$ skips. **Theorem *vmultistep_choice_skip*** :

$$\forall c1\ c2\ vmu\ vmu',$$

$vmultistep (vip (vcChoice c1 c2) vmu) (vip vcSkip vmu') \rightarrow vmultistep (vip c1 vmu) (vip vcSkip vmu') \vee vmultistep (vip c2 vmu) (vip vcSkip vmu')$.

Proof with subst.

```
intros.
inversion H...
inversion H0; subst; solve [ left; trivial | right; trivial ].
```

Qed.

If a choice $c1 c2$ fails, then either $c1$ or $c2$ fails. Theorem *vmultistep_choice_fail* :

$\forall c1 c2 vmu vmu'$,

$vmultistep (vip (vcChoice c1 c2) vmu) (vfailure vmu') \rightarrow vmultistep (vip c1 vmu) (vfailure vmu') \vee vmultistep (vip c2 vmu) (vfailure vmu')$.

Proof with subst.

```
intros.
inversion H...
inversion H0; subst; solve [ left; trivial | right; trivial ].
```

Qed.

Transitivity of multistep Theorem *multistep_trans* : $\forall s1 s2 s3, multistep s1 s2 \rightarrow multistep s2 s3 \rightarrow multistep s1 s3$.

Proof.

```
intros.
revert s3 H0.
induction H; intros.
trivial.
rename s0 into s4.
eapply multiStep.
apply H.
apply IHmultistep.
trivial.
```

Qed.

Transitivity of vmultistep Theorem *vmultistep_trans* : $\forall s1 s2 s3, vmultistep s1 s2 \rightarrow vmultistep s2 s3 \rightarrow vmultistep s1 s3$.

Proof.

```
intros.
revert s3 H0.
induction H; intros.
trivial.
rename s0 into s4.
eapply vmultiStep.
apply H.
apply IHvmultistep.
trivial.
```

Qed.

Transitivity of `pmultistep` `Theorem pmultistep_trans` : $\forall vmu\ s1\ s2\ s3, pmultistep\ vmu\ s1\ s2 \rightarrow pmultistep\ vmu\ s2\ s3 \rightarrow pmultistep\ vmu\ s1\ s3$.

`Proof.`

```
intros.  
revert s3 H0.  
induction H; intros.  
trivial.  
rename s0 into s4.  
eapply pmultiStep.  
apply H.  
apply IHpmultistep.  
trivial.
```

`Qed.`

`commands` are at least 1 big. `Lemma command_metric_min_size` : $\forall c, 1 \leq command_metric\ c$.

`Proof.`

```
induction c; simpl; auto with arith.
```

`Qed.`

`Hint Resolve command_metric_min_size.`

`vcommands` are at least 1 big. `Lemma vcommand_metric_min_size` : $\forall c, 1 \leq vcommand_metric\ c$.

`Proof.`

```
induction c; simpl; auto with arith.
```

`Qed.`

`Hint Resolve vcommand_metric_min_size.`

`pcommands` are at least 1 big. `Lemma pcommand_metric_min_size` : $\forall c, 1 \leq pcommand_metric\ c$.

`Proof.`

```
induction c; simpl; auto with arith.
```

`Qed.`

`Hint Resolve pcommand_metric_min_size.`

Shows that the `step` relation reduces the command's size, i.e. evaluation of a program is guaranteed to end. `Theorem step_monotonic_commands` : $\forall c\ c'\ mu\ mu', step\ (ip\ c\ mu)\ (ip\ c'\ mu') \rightarrow command_metric\ c' < command_metric\ c$.

`Proof with subst.`

```
induction c; intros; simpl.  
  
inversion H...  
simpl; auto with arith.  
  
inversion H...  
simpl; auto with arith.
```

```

inversion H...
simpl; auto with arith.
clear IHc2.
inversion H...
specify IHc1 c1' mu mu' H1.
simpl.
omega.
simpl.
omega.

inversion H.

inversion H; subst; omega.
Qed.

```

Shows that the *vstep* relation reduces the command's size, i.e. evaluation of a program is guaranteed to end. **Theorem** *vstep_monotonic_commands* : $\forall c c' vmu vmu', vstep (vip c vmu) (vip c' vmu') \rightarrow vcommand_metric c' < vcommand_metric c$.

Proof with subst.

```

induction c; intros; simpl.

inversion H...
simpl; auto with arith.

inversion H...
simpl; auto with arith.

inversion H...
simpl; auto with arith.

clear IHc2.
inversion H...
specify IHc1 c1' vmu vmu' H1.
simpl.
omega.
simpl.
omega.

inversion H.

inversion H; subst; omega.
Qed.

```

Shows that the *pstep* relation reduces the command's size, i.e. evaluation of a program is guaranteed to end. **Theorem** *pstep_monotonic_commands* : $\forall c c' vmu, pstep vmu (pip c) (pip c') \rightarrow pcommand_metric c' < pcommand_metric c$.

Proof with subst.

```

induction c; intros; simpl.

inversion H...

```

```

simpl; auto with arith.
inversion H...
simpl; auto with arith.
clear IHc2.
inversion H...
specify IHc1 c1' vmu H1.
simpl.
omega.
simpl.
omega.

inversion H.

inversion H; subst; omega.
Qed.

Theorem step_monotonic_states :  $\forall s s', \text{step } s s' \rightarrow \text{state\_metric } s' < \text{state\_metric } s$ .
Proof.
  intros.
  destruct s; destruct s'.
  apply (step_monotonic_commands _ _ _ H).
  simpl; auto with arith.
  inversion H.
  inversion H.
Qed.

Theorem vstep_monotonic_states :  $\forall s s', \text{vstep } s s' \rightarrow \text{vstate\_metric } s' < \text{vstate\_metric } s$ .
Proof.
  intros.
  destruct s; destruct s'.
  apply (vstep_monotonic_commands _ _ _ H).
  simpl; auto with arith.
  inversion H.
  inversion H.
Qed.

Theorem pstep_monotonic_states :  $\forall s s' \text{ vmu}, \text{pstep } \text{vmu } s s' \rightarrow \text{pstate\_metric } s' < \text{pstate\_metric } s$ .
Proof.
  intros.
  destruct s; destruct s'.
  apply (pstep_monotonic_commands _ _ H).
  simpl; auto with arith.
  inversion H.
  inversion H.
Qed.

```

Theorem *multistep_monotonic_commands* : $\forall c c' \mu \mu', \text{multistep } (ip \ c \ \mu) \ (ip \ c' \ \mu') \rightarrow \text{command_metric } c' \leq \text{command_metric } c.$

Proof.

```

intros.
introduce states s s' in H.
revert c c' mu mu' H0 H1.
induction H; intros; subst.
strip H1; omega.
destruct s2.
rename mu' into mu"; rename s into mu'.
rename c' into c"; rename c0 into c'.
state (IHmultistep _ _ _ (refl_equal _) (refl_equal _)); clear IHmultistep.
state (step_monotonic_commands _ _ _ H).
omega.
inversion H0...
inversion H1.

```

Qed.

Theorem *vmultistep_monotonic_commands* : $\forall c c' \text{vmu} \text{vmu}', \text{vmultistep } (vip \ c \ \text{vmu}) \ (vip \ c' \ \text{vmu}') \rightarrow \text{vcommand_metric } c' \leq \text{vcommand_metric } c.$

Proof.

```

intros.
introduce states s s' in H.
revert c c' vmu vmu' H0 H1.
induction H; intros; subst.
strip H1; omega.
destruct s2.
rename vmu' into vmu"; rename v0 into vmu'.
rename c' into c"; rename v into c'.
state (IHvmultistep _ _ _ (refl_equal _) (refl_equal _)); clear IHvmultistep.
state (vstep_monotonic_commands _ _ _ H).
omega.
inversion H0...
inversion H1.

```

Qed.

Theorem *pmultistep_monotonic_commands* : $\forall c c' \text{vmu}, \text{pmultistep } \text{vmu} \ (pip \ c) \ (pip \ c') \rightarrow \text{pcommand_metric } c' \leq \text{pcommand_metric } c.$

Proof.

```

intros.
introduce states vmu' s s' in H; subst vmu'.
revert c c' H1 H2.
induction H; intros; subst.
strip H2; omega.

```

```

destruct s2.
rename c' into c"; rename p into c'.
state (IHpmultistep - - (refl_equal -) (refl_equal -)); clear IHpmultistep.
state (pstep_monotonic_commands - - - H).
omega.
inversion H0...
inversion H1.

```

Qed.

Theorem *multistep_monotonic_states* : $\forall s s', \text{multistep } s s' \rightarrow \text{state_metric } s' \leq \text{state_metric } s$.

Proof with subst.

```

intros.
destruct s; destruct s'.
simpl; apply (multistep_monotonic_commands - - - H).
simpl; auto with arith.
inversion H...
inversion H0.
simpl; auto with arith.

```

Qed.

Theorem *vmultistep_monotonic_states* : $\forall s s', \text{vmultistep } s s' \rightarrow \text{vstate_metric } s' \leq \text{vstate_metric } s$.

Proof with subst.

```

intros.
destruct s; destruct s'.
simpl; apply (vmultistep_monotonic_commands - - - H).
simpl; auto with arith.
inversion H...
inversion H0.
simpl; auto with arith.

```

Qed.

Theorem *pmultistep_monotonic_states* : $\forall s s' \text{ vmu}, \text{pmultistep } \text{vmu } s s' \rightarrow \text{pstate_metric } s' \leq \text{pstate_metric } s$.

Proof with subst.

```

intros.
destruct s; destruct s'.
simpl; apply (pmultistep_monotonic_commands - - - H).
simpl; auto with arith.
inversion H...
inversion H0.
simpl; auto with arith.

```

Qed.

Theorem *step_asymmetric* : $\forall s s', \text{step } s s' \rightarrow \neg \text{step } s' s$.

Proof.

```
red; intros.  
state (step_monotonic_states - - H).  
state (step_monotonic_states - - H0).  
omega.
```

Qed.

Theorem *vstep_asymmetric* : $\forall s s', \text{vstep } s s' \rightarrow \neg \text{vstep } s' s$.

Proof.

```
red; intros.  
state (vstep_monotonic_states - - H).  
state (vstep_monotonic_states - - H0).  
omega.
```

Qed.

Theorem *pstep_asymmetric* : $\forall \text{vmu } s s', \text{pstep } \text{vmu } s s' \rightarrow \neg \text{pstep } \text{vmu } s' s$.

Proof.

```
red; intros.  
state (pstep_monotonic_states - - - H).  
state (pstep_monotonic_states - - - H0).  
omega.
```

Qed.

Theorem *multistep_antisymmetric* : $\forall s s', \text{multistep } s s' \rightarrow \text{multistep } s' s \rightarrow s = s'$.

Proof.

```
intros.  
destruct H.  
trivial.  
destruct H0.  
trivial.  
assert False; [ idtac | contradiction ].  
state (step_monotonic_states - - H); clear H.  
state (step_monotonic_states - - H0); clear H0.  
state (multistep_monotonic_states - - H1); clear H1.  
state (multistep_monotonic_states - - H2); clear H2.  
omega.
```

Qed.

Theorem *vmultistep_antisymmetric* : $\forall s s', \text{vmultistep } s s' \rightarrow \text{vmultistep } s' s \rightarrow s = s'$.

Proof.

```
intros.  
destruct H.  
trivial.  
destruct H0.
```

```

trivial.
assert False; [ idtac | contradiction ].
state (vstep_monotonic_states - - H); clear H.
state (vstep_monotonic_states - - H0); clear H0.
state (vmultistep_monotonic_states - - H1); clear H1.
state (vmultistep_monotonic_states - - H2); clear H2.
omega.

```

Qed.

Theorem *pmultistep_antisymmetric* : $\forall s s' \text{ vmu}, \text{pmultistep vmu } s s' \rightarrow \text{pmultistep vmu } s' s \rightarrow s = s'$.

Proof.

```

intros.
destruct H.
trivial.
destruct H0.
trivial.
assert False; [ idtac | contradiction ].
state (pstep_monotonic_states - - - H); clear H.
state (pstep_monotonic_states - - - H0); clear H0.
state (pmultistep_monotonic_states - - - H1); clear H1.
state (pmultistep_monotonic_states - - - H2); clear H2.
omega.

```

Qed.

Converts an *id* to a *vid*, using a version map *v*. Definition *id_to_vid* (*id* : *id*) (*v* : *vmap*) : *vid* := (*id*, *v id*).

Converts a *vstore* to a *store*, using a version map *v*. Definition *vstore_to_store* (*vmu* : *vstore*) (*v* : *vmap*) : *store* := fun *id* => *vmu* (*id_to_vid id v*).

Converts an *expr* to a *vexpr*, using a version map *v*. Definition *expr_to_vexpr* (*e* : *expr*) (*v* : *vmap*) : *vexpr* := fun *vmu* => *e* (*vstore_to_store vmu v*).

Converts a *command* to a *vcommand*, using a version map *v*. Fixpoint *command_to_vcommand* (*c* : *command*) (*v* : *vmap*) : *vcommand* := match *c* with | *cAssert e* => *vcAssert (expr_to_vexpr e v)* | *cAssume e* => *vcAssume (expr_to_vexpr e v)* | *cAssign x e* => *vcAssign (id_to_vid x v) (expr_to_vexpr e v)* | *cSequence c1 c2* => *vcSequence (command_to_vcommand c1 v) (command_to_vcommand c2 v)* | *cSkip* => *vcSkip*

```

    | cChoice c1 c2 => vcChoice (command_to_vcommand c1 v) (command_to_vcommand
c2 v)
end.

```

Theorem *vstore_to_store_surjective* : $\forall v \mu, \exists \nu \mu, \mu = \text{vstore_to_store } \nu v$.

Proof.

```

intros.
unfold vstore_to_store.
unfold id_to_vid.
∃ (fun x : vid => let (i, _) := x in mu i).
rewrite ← (eta_expansion id value mu).
trivial.

```

Qed.

Theorem *expr_to_vexpr_injective* : $\forall e e' v, \text{expr_to_vexpr } e v = \text{expr_to_vexpr } e' v \rightarrow e = e'$.

Proof.

```

intros.
unfold expr_to_vexpr in H.
state (equal_f _ _ _ H).
clear H; simpl in H0.
assert (∀ mu, e mu = e' mu).
intros.
elim (vstore_to_store_surjective v mu); intros.
specify H0 x.
rewrite ← H in H0.
trivial.
apply (functional_extensionality _ _ _ H).

```

Qed.

Theorem *command_to_vcommand_injective* : $\forall c c' v, \text{command_to_vcommand } c v = \text{command_to_vcommand } c' v \rightarrow c = c'$.

Proof.

```

induction c; simpl; intros; destruct c'; simpl in H; try ( solve [ discriminate ] ).
strip H.
f_equal.
apply (expr_to_vexpr_injective e e0 v).
trivial.

f_equal; strip H.
apply (expr_to_vexpr_injective e e0 v).
trivial.

strip H.
clear H0; f_equal.
apply (expr_to_vexpr_injective e e0 v).

```

trivial.

strip H.

specify IHc1 c'1 v H0.

specify IHc2 c'2 v H.

subst; trivial.

trivial.

strip H.

rewrite (IHc1 c'1 v H0).

rewrite (IHc2 c'2 v H).

trivial.

Qed.

Shows that *step* and *vstep* behave the same when not ending up in failure. **Theorem**
step_sim_vstep : $\forall c c' mu mu' vmu v,$

let *vc* := *command_to_vcommand c v* in

let *vc'* := *command_to_vcommand c' v* in

step (ip c mu) (ip c' mu') → store_sync_vstore mu v vmu → ∃ vmu', vstep (vip vc vmu) (vip vc' vmu') ∧ store_sync_vstore mu' v vmu'.

Proof with subst.

intros; subst *vc*; subst *vc'*.

revert c' mu mu' vmu v H H0.

induction *c*; simpl; intros.

inversion H...

$\exists vmu.$

split.

apply *vstepAssertT*.

compute.

state (expression_evaluation e mu' (fun id0 ⇒ vmu (id0, v id0))).

compute in *H1*.

rewrite ← *H1*.

trivial.

revert H0; clear; intros.

compute in $\times \vdash \times$.

intros; *specify H0 x*; trivial.

trivial.

inversion H...

$\exists vmu$; split.

apply *vstepAssume*.

compute.

state (expression_evaluation e mu' (fun id0 ⇒ vmu (id0, v id0))).

compute in *H1*.

```

rewrite ← H1.
trivial.
revert H0; clear; intros; compute in × ⊢ ×.
intros; apply (H0 x).
trivial.

inversion H...
∃ (update_vstore vmu (id_to_vid i v) ((expr_to_vexpr e v) vmu)).
split.
apply vstepAssign.
revert H0; clear; intros.
unfold store_sync_vstore.
unfold equivalent_functions.
intros.
cbv delta [ update_store update_vstore rebind ].
simpl.
destruct (decidable_eq_id x i).
destruct (eq_nat_dec (v x) (v i)).
unfold expr_to_vexpr.
apply (expression_evaluation e mu (vstore_to_store vmu v)).
trivial.
subst.
elim n.
trivial.
destruct (eq_nat_dec (v x) (v i)).
trivial.
trivial.

clear IHc2.
inversion H...
clear H.
specify IHc1 c1' mu mu' vmu v H2 H0.
eliminate existential vmu' in IHc1.
∃ vmu'.
destruct IHc1.
split.
simpl.
apply vstepSeq.
trivial.
trivial.
∃ vmu.
split.
apply vstepSeqSkip.
trivial.

```

```

inversion H.
clear IHc1 IHc2.
inversion H...
∃ vmu.
split.
apply vstepChoiceL.
trivial.
∃ vmu.
split.
apply vstepChoiceR.
trivial.

```

Qed.

Shows that *step* and *vstep* behave the same when ending up in failure. Theorem
 $step_sim_vstep_fail : \forall c \mu \mu' vmu v,$
`let $vc := command_to_vcommand\ c\ v$ in`
 $step\ (ip\ c\ \mu)\ (failure\ \mu') \rightarrow store_sync_vstore\ \mu\ v\ vmu \rightarrow \exists\ vmu',\ vstep\ (vip\ vc\ vmu)\ (vfailure\ vmu') \wedge store_sync_vstore\ \mu' \ v\ vmu'.$

Proof with subst.

```

intros.
subst vc.
revert mu mu' vmu v H H0; induction c; simpl; intros.

inversion H...
∃ vmu; split.
apply vstepAssertF.
red; intros.
elim H2; clear H2.
clear H.
compute in H1.
rewrite (expression_evaluation e mu' (fun id : nat => vmu (id, v id))).
exact H1.
revert H0; clear; intros.
compute in × ⊢ ×.
exact H0.
trivial.

inversion H.
inversion H.

clear IHc2.
inversion H...
clear H.
specify IHc1 mu mu' vmu v H2 H0.
eliminate existential vmu' in IHc1.

```

```

destruct IHc1.
∃ vmu'; split.
apply vstepSeqFail.
trivial.
trivial.

inversion H.

inversion H.
Qed.

Shows that multistep and vmultistep behave similarly when not ending up in failure.
Theorem multistep_sim_vmultistep : ∀ c c' mu mu' vmu v,
  let vc := command_to_vcommand c v in
  let vc' := command_to_vcommand c' v in
  multistep (ip c mu) (ip c' mu') → store_sync_vstore mu v vmu → ∃ vmu', vmultistep
  (vip vc vmu) (vip vc' vmu') ∧ store_sync_vstore mu' v vmu'.
Proof with subst.
  intros; subst vc; subst vc'.
  introduce states s1 s2 in H.
  revert c c' mu mu' vmu v H0 H1 H2.
  induction H; intros...

  strip H2.
  ∃ vmu.
  split.
  constructor.
  trivial.

  state multistep_ip_ip.
  specify H2 s2 c' mu' H0.
  rename c' into c3.
  rename mu' into mu3.
  eliminate existentials c2 mu2 in H2.
  subst.
  specify IHmultistep c2 c3 mu2 mu3.
  state step_sim_vstep.
  specify H2 c c2 mu mu2 vmu v.
  simpl in H2.
  specify H2 H H1.
  eliminate existential vmu2 in H2.
  destruct H2.
  specify IHmultistep vmu2 v H3.
  state (IHmultistep (refl_equal _) (refl_equal _)) as H4; clear IHmultistep.
  eliminate existential vmu3 in H4.
  destruct H4.

```

$\exists v\mu 3$; split.
step (*vip* (*command_to_vcommand* *c2 v*) *v\mu 2*); trivial.
trivial.

Qed.

Shows that *multistep* can't hop from one failure state to another one with a different store. Lemma *multistep_fail_fail_equal_stores* : $\forall \mu \mu'$, *multistep* (*failure* μ) (*failure* μ') $\rightarrow \mu = \mu'$.

Proof with subst.

intros.
introduce states *s s'* in *H*.
revert $\mu \mu'$ *H0 H1*.
destruct *H*; intros...
strip *H1*.
trivial.
inversion *H*...

Qed.

Shows that *vmultistep* can't hop from one failure state to another one with a different store. Lemma *vmultistep_fail_fail_equal_stores* : $\forall v\mu v\mu'$, *vmultistep* (*vfailure* $v\mu$) (*vfailure* $v\mu'$) $\rightarrow v\mu = v\mu'$.

Proof with subst.

intros.
introduce states *s s'* in *H*.
revert $v\mu v\mu'$ *H0 H1*.
destruct *H*; intros...
strip *H1*.
trivial.
inversion *H*...

Qed.

Shows that *multistep* and *vmultistep* behave similarly when ending up in failure. Theorem *multistep_sim_vmultistep_fail* : $\forall c \mu \mu' v\mu v$,

let *vc* := *command_to_vcommand* *c v* in
multistep (*ip* *c* μ) (*failure* μ') \rightarrow *store_sync_vstore* $\mu v v\mu \rightarrow \exists v\mu'$, *vmultistep* (*vip* *vc v\mu*) (*vfailure* $v\mu'$) \wedge *store_sync_vstore* $\mu' v v\mu'$.

Proof.

intros.
subst *vc*.
introduce states *s s'* in *H*.
revert *c* $\mu \mu' v\mu$ *H0 H1 H2*.
induction *H*; intros; subst.
discriminate.

```

destruct s2.
rename c0 into c'; rename mu' into mu"; rename s into mu'.
specify IHmultistep c' mu' mu".
state step_sim_vstep.
simpl in H2; specify H2 c c' mu mu' vmu v H H1.
eliminate existential vmu' in H2.
destruct H2.
state (IHmultistep vmu' H3 (refl_equal _) (refl_equal _)); clear IHmultistep.
eliminate existential vmu" in H4.
destruct H4.
∃ vmu".
split.
step (vip (command_to_vcommand c' v) vmu').
trivial.
trivial.
trivial.

assert (s = mu').
revert H0; clear; intros.
state multistep_fail_fail_equal_stores.
specify H s mu' H0.
subst; trivial.
subst.
state step_sim_vstep_fail.
simpl in H2.
specify H2 c mu mu' vmu v H H1.
eliminate existential vmu' in H2.
destruct H2.
∃ vmu'.
split.
apply vmultistep_step_to_multi.
trivial.
trivial.

```

Qed.

If we start of in a *vcommand* which is the result of the translation of a *command*, the *step* relation will lead us only to *vcommands* which themselves are also translations of some *command*.

In other words, the *command_to_vcommand* function is invertible, with some strings attached. Lemma *vcommand_to_command* : $\forall c \text{ } v c' \text{ } v m u \text{ } v m u' \text{ } v, \text{ } v \text{step} (vip (command_to_vcommand c v) vmu) (vip v c' vmu') \rightarrow \exists c', v c' = command_to_vcommand c' v$.

Proof.

```

intros.
introduce states s s' in H.

```

```

revert c vc' vmu vmu' H0 H1.
induction H; intros; try ( solve [ discriminate ] ); strip H0; strip H1.
∃ cSkip.
simpl.
trivial.
∃ cSkip.
simpl.
trivial.

rename IHstep into IH.
rename c1 into vc1; rename c1' into vc1'; rename c2 into vc2.
rename vmu0 into vmu; rename vmu'0 into vmu'.
destruct c; try ( solve [ discriminate ] ).
simpl in H2.
strip H2.
specify IH c1.
specify IH vc1'.
specify IH vmu vmu'.
state (IH (refl_equal _) (refl_equal _)); clear IH.
eliminate existential c1' in H0.
∃ (cSequence c1' c2).
simpl.
rewrite H0.
trivial.

destruct c; try ( solve [ discriminate ] ).
simpl in H0.
strip H0.
destruct c1; try ( solve [ discriminate ] ).
clear H0.
∃ c2.
trivial.
∃ cSkip.
simpl.
trivial.

destruct c; try ( solve [ discriminate ] ).
simpl in H0.
strip H0.
∃ c1.
trivial.

destruct c; try ( solve [ discriminate ] ).
simpl in H0.
strip H0.

```

```

  ∃ c3.
  trivial.
Qed.

  vstep and step behave similarly when not leading to a failure state. Theorem vstep_sim_step
: ∀ v c c' mu vmu vmu',
  let vc := command_to_vcommand c v in
  let vc' := command_to_vcommand c' v in
  store_sync_vstore mu v vmu → vstep (vip vc vmu) (vip vc' vmu') → ∃ mu', step (ip c
mu) (ip c' mu') ∧ store_sync_vstore mu' v vmu'.
Proof with subst.
  intros.
  subst vc; subst vc'.
  revert c' mu vmu vmu' H H0.
  induction c; simpl; intros.

  inversion H0...
  assert (c' = cSkip).
  revert H4; clear; intros.
  destruct c'; simpl in H4; solve [ discriminate | trivial ].
  subst c'.
  ∃ mu.
  split.
  constructor.
  revert H H2; clear; intros.
  unfold expr_to_vexpr in H2.
  unfold vstore_to_store in H2.
  state expression_evaluation.
  specify H0 e mu (fun id : id ⇒ vmu' (id_to_vid id v)).
  rewrite H0.
  trivial.
  revert H; clear; intros.
  compute in × ⊢ ×.
  trivial.
  trivial.

  inversion H0...
  ∃ mu.
  split; [ idtac | solve [ trivial ] ].
  assert (c' = cSkip).
  revert H4; clear; intros.
  destruct c'; simpl in H4; solve [ discriminate | trivial ].
  subst c'.
  constructor.
  revert H H2; clear; intros.

```

```

compute in  $\times \vdash \times$ .
rewrite (expression_evaluation e mu (fun id : nat  $\Rightarrow$  vmu' (id, v id))).
trivial.
compute.
trivial.

 $\exists$  (update_store mu i (e mu)).
inversion H0...
assert (c' = cSkip).
revert H5; clear; intros.
destruct c'; simpl in H5; solve [discriminate | trivial].
subst c'.
split.
constructor.
revert H; clear; intros.
unfold store_sync_vstore in  $\times \vdash \times$ .
unfold equivalent_functions in  $\times \vdash \times$ .
intros.
unfold update_store; unfold update_vstore.
unfold rebind.
destruct (decidable_eq_id x i); destruct (decidable_eq_vid (x, v x) (id_to_vid i v)).
unfold id_to_vid in e1.
strip e1.
clear H0 H1.
unfold expr_to_vexpr.
apply (expression_evaluation e mu (vstore_to_store vmu v)).
compute.
trivial.
unfold id_to_vid in n.
subst.
elim n; trivial.
unfold id_to_vid in e0.
strip e0.
elim n; trivial.
clear n n0.
trivial.

inversion H0...
clear IHc2.
rename IHc1 into IH.
rename c1' into vc1'.
state vcommand_to_command.
specify H1 c1 vc1' vmu vmu' v.
specify H1 H2.

```

```

eliminate existential c1' in H1.
specify IH c1' mu vmu vmu' H.
rewrite H1 in H2.
specify IH H2.
eliminate existential mu' in IH.
destruct IH.
∃ mu'; split; [ idtac | trivial ].
destruct c'; try ( solve [ discriminate ] ).
simpl in H5.
strip H5.
revert H5 H6 H3; clear; intros.
state command_to_vcommand_injective.
state (H _ _ H5).
state (H _ _ H6); clear H.
subst.
constructor.
trivial.

destruct c1; simpl in H2; try ( solve [ discriminate ] ).
∃ mu.
split; [ idtac | trivial ].
revert H5; clear; intros.
rewrite (command_to_vcommand_injective _ _ H5).
constructor.

inversion H0.

clear IHc1 IHc2.
inversion H0...
rewrite (command_to_vcommand_injective _ _ H5).
∃ mu; split; [ idtac | trivial ].
constructor.
rewrite (command_to_vcommand_injective _ _ H5).
∃ mu; split; [ idtac | trivial ].
constructor.

```

Qed.

vstep and *step* behave similarly when leading to a failure state. **Theorem** *vstep_sim_step_fail*
 $: \forall v c mu vmu vmu',$

```

let vc := command_to_vcommand c v in
store_sync_vstore mu v vmu → vstep (vip vc vmu) (vfailure vmu') → ∃ mu', step (ip
c mu) (failure mu') ∧ store_sync_vstore mu' v vmu'.

```

Proof with subst.

```

intros.
subst vc.

```

```

introduce states s s' in H0.
revert c mu vmu vmu' H H1 H2.
induction H0; intros; try ( solve [ discriminate ] ).

strip H2.
rename vmu0 into vmu.
destruct c; simpl in H1; try ( solve [ discriminate ] ).
strip H1.
∃ mu.
split; [ idtac | trivial ].
constructor.
red; intros.
elim H; intros; clear H.
rename e0 into e.
compute in × ⊢ ×.
state (expression_evaluation e (fun id0 : nat ⇒ vmu (id0, v id0)) mu).
compute in H.
rewrite ← H in H1.
trivial.
intros.
symmetry.
trivial.

strip H2.
destruct c; simpl in H1; try ( solve [ discriminate ] ).
strip H1.
rename vmu0 into vmu; rename vmu'0 into vmu'.
rename IHvstep into IH.
rename c3 into c1; rename c4 into c2.
specify IH c1 mu vmu vmu' H.
state (IH (refl_equal _) (refl_equal _)); clear IH.
eliminate existential mu' in H1.
destruct H1.
∃ mu'.
split; [ idtac | trivial ].
constructor.
trivial.
Qed.

```

vmultistep and *multistep* behave similarly when not ending in failure. **Theorem *vmultistep_sim_multistep*** : $\forall c c' mu vmu vmu' v$,
let $vc := \text{command_to_vcommand } c \ v$ **in**
let $vc' := \text{command_to_vcommand } c' \ v$ **in**
 $\text{store_sync_vstore } mu \ v \ vmu \rightarrow \text{vmultistep } (vip \ vc \ vmu) \ (vip \ vc' \ vmu') \rightarrow \exists mu'$,
 $\text{multistep } (ip \ c \ mu) \ (ip \ c' \ mu') \wedge \text{store_sync_vstore } mu' \ v \ vmu'$.

Proof with subst.

```
intros.
subst vc; subst vc'.
introduce states s s' in H0.
revert c c' mu vmu vmu' H H1 H2.
induction H0; intros...

strip H2.
∃ mu.
split; [ idtac | trivial ].
rewrite (command_to_vcommand_injective c c' v).
constructor.
trivial.

rename IHmultistep into IH.
rename c' into c"; rename vmu' into vmu".
state vmultistep_vip_vip.
specify H2 s2 (command_to_vcommand c" v) vmu" H0.
eliminate existentials vc' vmu' in H2.
subst s2.
state (vcommand_to_command _ _ _ _ H).
eliminate existential c' in H2.
subst vc'.
specify IH c' c".
state vstep_sim_step.
specify H2 v c c' mu vmu vmu'.
simpl in H2.
specify H2 H1 H.
eliminate existential mu' in H2.
destruct H2.
specify IH mu' vmu' vmu" H3.
state (IH (refl_equal _) (refl_equal _)); clear IH.
eliminate existential mu" in H4.
destruct H4.
∃ mu".
split; [ idtac | trivial ].
step (ip c' mu').
trivial.
trivial.
```

Qed.

vmultistep and *multistep* behave similarly when ending in failure. **Theorem** *vmultistep_sim_multistep_fail* : $\forall c \mu vmu vmu' v$,

```
let vc := command_to_vcommand c v in
store_sync_vstore mu v vmu → vmultistep (vip vc vmu) (vfailure vmu') → ∃ mu',
```

multistep (ip c mu) (failure mu') \wedge store_sync_vstore mu' v vmu'.

Proof with subst.

```
intros.
subst vc.
introduce states s s' in H0.
revert c mu vmu vmu' H H1 H2.
induction H0; intros...

discriminate.

rename IHvmultistep into IH.
destruct s2.
rename v0 into vc'.
rename vmu' into vmu''.
rename v1 into vmu'.
rename c into c1.
state (vcommand_to_command _ _ _ v H).
eliminate existential c1' in H2.
subst vc'.
state vstep_sim_step.
specify H2 v c1 c1' mu vmu vmu'.
simpl in H2.
specify H2 H1 H.
eliminate existential mu' in H2.
destruct H2.
specify IH c1' mu' vmu' vmu'' H3.
state (IH (refl_equal _) (refl_equal _)); clear IH.
eliminate existential mu'' in H4.
destruct H4.
 $\exists$  mu''.
split; [ idtac | trivial ].
step (ip c1' mu').
trivial.
trivial.

clear IH.
rename vmu' into vmu''.
rename v0 into vmu'.
state (vmultistep_fail_fail_equal_stores _ _ H0).
subst vmu'.
state vstep_sim_step_fail.
specify H2 v c mu vmu vmu''.
simpl in H2.
specify H2 H1 H.
eliminate existential mu'' in H2.
```

```

destruct H2.
∃ mu".
split; [ idtac | trivial ].
apply multistep_step_to_multi.
trivial.
Qed.

```

Our goal is to verify that *vstep* and *pstep* behave similarly. In order to do this, we need to translate *vcommands* to *pcommands* (this translation must not be confused with the passification step later on). The problem is that *vcommands* can contain assignments, while *pcommands* cannot. Thus, we can only compare assignment-less programs. This function determines whether a *vcommand* contains no assignments. `Fixpoint no_assignments (c : vcommand) : Prop :=`

```

match c with
| vcAssert _ => True
| vcAssume _ => True
| vcAssign _ _ => False
| vcSequence c1 c2 => no_assignments c1 ∧ no_assignments c2
| vcSkip => True
| vcChoice c1 c2 => no_assignments c1 ∧ no_assignments c2
end.

```

Using *vstep*, if a program does not contain assignments, the store remains unchanged. **Theorem** `no_assignments_preserves_store : ∀ c c' mu mu', no_assignments c → vstep (vip c mu) (vip c' mu') → mu = mu'`.

Proof with subst.

```

induction c; intros; try ( solve [ inversion H0; subst; trivial ] ).
simpl in H.
contradiction.
inversion H0...
simpl in H.
destruct H.
exact (IHc1 c1' mu mu' H H2).
trivial.

```

Qed.

Using *vstep*, if a program does not contain assignments, the store remains unchanged. **Theorem** `no_assignments_preserves_store_fail : ∀ c mu mu', no_assignments c → vstep (vip c mu) (vfailure mu') → mu = mu'`.

Proof with subst.

```

induction c; intros; try ( solve [ inversion H0 ] ).
inversion H0...
trivial.
inversion H0...
simpl in H.

```

```

destruct H.
exact (IHc1 mu mu' H H2).
Qed.

```

vstep reduces commands without assignments to commands which also do not contain assignments. In other words, the *vstep* reduction rules do not introduce assignments. **Theorem** *vstep-preserves-no-assignments* : $\forall c c' mu mu', no_assignments\ c \rightarrow vstep\ (vip\ c\ mu)\ (vip\ c'\ mu') \rightarrow no_assignments\ c'$.

Proof with subst.

```

intros.
introduce states s s' in H0.
revert c c' mu mu' H H1 H2.
induction H0; intros...

strip H1; strip H2.
trivial.

discriminate.

strip H1; strip H2.
trivial.

strip H1; strip H2.
simpl.
simpl in H.
destruct H.
split; [ idtac | trivial ].
apply (IHvstep c1 c1' mu mu' H (refl_equal -) (refl_equal -)).

strip H1; strip H2.
simpl in H.
destruct H; trivial.

discriminate.

strip H1; strip H2.
simpl in H.
contradiction.

strip H1; strip H2.
simpl in H; destruct H; trivial.

strip H1; strip H2.
simpl in H; destruct H; trivial.

```

Qed.

Transform a *vcommand* into an equivalent *pcommand*, on the condition that the *vcommand* does not contain assignments. **Definition** *vcommand_to_pcommand* (*vc* : *vcommand*) (*H* : *no_assignments vc*) : *pcommand*.
induction *vc*; intros.

```

exact (pcAssert v).
exact (pcAssume v).
simpl in H; contradiction.
simpl in H; destruct H.
exact (pcSequence (IHvc1 H) (IHvc2 H0)).
exact pcSkip.
simpl in H; destruct H.
exact (pcChoice (IHvc1 H) (IHvc2 H0)).
Defined.

```

vstep behaves similarly to *pstep* for assignmentless programs. **Theorem *vstep_sim_pstep***

$$: \forall vc\ vc'\ H\ vmu\ vmu'\ (H0 : vstep\ (vip\ vc\ vmu)\ (vip\ vc'\ vmu')),$$
 let $pc := vcommand_to_pcommand\ vc\ H$ in
 let $pc' := vcommand_to_pcommand\ vc'\ (vstep_preserves_no_assignments\ _ _ _ _ H\ H0)$
 in

$$pstep\ vmu\ (pip\ pc)\ (pip\ pc').$$

Proof with subst.

```

induction vc; intros; subst pc; subst pc'.
inversion H0...
simpl.
constructor.
trivial.
inversion H0...
simpl.
constructor.
trivial.
simpl in H.
contradiction.
clear IHvc2; rename IHvc1 into IH.
inversion H0...
rename c1' into vc1'.
simpl in H.
elim H; intros.
specify IH vc1' H1 vmu vmu' H2.
simpl in IH.
simpl.
case_eq H.
intros.
case_eq (vstep_preserves_no_assignments (vcSequence vc1 vc2) (vcSequence vc1' vc2)
vmu vmu' (conj n n0) H0).
intros.
state pstepSeq.

```

```

    specify H6 vmu (vcommand_to_pcommand vc1 n) (vcommand_to_pcommand vc1' n1)
(vcommand_to_pcommand vc2 n2).
    state (proof_irrelevance - (vstep_preserves_no_assignments vc1 vc1' vmu vmu' H1 H2)
n1).
    state (proof_irrelevance - H1 n).
    rewrite H7 in IH.
    rewrite H8 in IH.
    specify H6 IH; clear IH.
    state (proof_irrelevance - n0 n2).
    subst n0.
    exact H6.
    simpl.
    case_eq H.
    intros.
    state (proof_irrelevance - n0 (vstep_preserves_no_assignments (vcSequence vcSkip vc')
vc' vmu' vmu' (conj n n0) H0)).
    rewrite ← H2.
    constructor.

    inversion H0.

    clear IHvc1 IHvc2.
    inversion H0...
    simpl.
    case_eq H; intros.
    rewrite ← (proof_irrelevance - n (vstep_preserves_no_assignments (vcChoice vc' vc2)
vc' vmu' vmu' (conj n n0) H0)).
    constructor.
    simpl.
    case_eq H; intros.
    rewrite ← (proof_irrelevance - n0 (vstep_preserves_no_assignments (vcChoice vc1 vc')
vc' vmu' vmu' (conj n n0) H0)).
    constructor.
Qed.

```

vstep behaves similarly to *pstep* for assignmentless programs. **Theorem *vstep_sim_pstep_fail***
 $: \forall vc H vmu vmu' (H0 : vstep (vip\ vc\ vmu) (vfailure\ vmu'))$,

```

    let pc := vcommand_to_pcommand vc H in
    pstep vmu (pip pc) pfailure.

```

Proof with subst.

```

    induction vc; intros; subst pc; try ( solve [ inversion H0 ] ).
    simpl.
    inversion H0...
    constructor.

```

```

trivial.
inversion H0...
clear IHvc2; rename IHvc1 into IH.
simpl in H.
elim H.
intros.
specify IH H1 vmu vmu' H2.
simpl in IH.
simpl.
case_eq H.
intros.
constructor.
rewrite (proof_irrelevance - n H1).
trivial.
Qed.

```

If one starts with an assignmentless command, one will always end up in an assignmentless command using *vmultistep*. **Theorem** *vmultistep_preserves_no_assignments* : $\forall c c' vmu vmu', no_assignments\ c \rightarrow vmultistep\ (vip\ c\ vmu)\ (vip\ c'\ vmu') \rightarrow no_assignments\ c'$.
Proof with *subst*.

```

intros.
introduce states s s' in H0.
revert c c' vmu vmu' H H1 H2.
induction H0; intros...

strip H2.
trivial.

rename c' into c".
rename vmu' into vmu".
state vmultistep_vip_vip.
state (H2 - - - H0).
clear H2.
eliminate existentials c' vmu' in H3.
subst s2.
state vstep_preserves_no_assignments.
specify H2 c c' vmu vmu' H1 H.
apply (IHvmultistep - - - H2 (refl_equal -) (refl_equal -)).
Qed.

```

Translates a *pcommand* to an equivalent *vcommand*. **Fixpoint** *pcommand_to_vcommand* (*pc* : *pcommand*) : *vcommand* :=
match *pc* **with**
| *pcAssert* *e* \Rightarrow *vcAssert* *e*
| *pcAssume* *e* \Rightarrow *vcAssume* *e*

```

| pcSkip  $\Rightarrow$  vcSkip
| pcSequence c1 c2  $\Rightarrow$  vcSequence (pcommand_to_vcommand c1) (pcommand_to_vcommand
c2)
| pcChoice c1 c2  $\Rightarrow$  vcChoice (pcommand_to_vcommand c1) (pcommand_to_vcommand
c2)
end.

```

Shows that *pstep* behaves similarly to *vstep*. **Theorem** *pstep_sim_vstep* : $\forall pc pc' vmu$,
let *vc* := *pcommand_to_vcommand* *pc* **in**
let *vc'* := *pcommand_to_vcommand* *pc'* **in**
pstep *vmu* (*pip* *pc*) (*pip* *pc'*) \rightarrow *vstep* (*vip* *vc* *vmu*) (*vip* *vc'* *vmu*).

Proof with subst.

```

induction pc; intros; subst vc; subst vc'.
inversion H...
simpl.
constructor.
trivial.
inversion H...
simpl.
constructor.
trivial.
inversion H...
rename c1' into pc1'.
simpl.
clear IHpc2; rename IHpc1 into IH.
simpl in IH.
specify IH pc1' vmu H1.
constructor.
trivial.
simpl.
constructor.
inversion H.
inversion H...
simpl.
constructor.
simpl.
constructor.

```

Qed.

Shows that *pstep* behaves similarly to *vstep*. **Theorem** *pstep_sim_vstep_fail* : $\forall pc vmu$,
let *vc* := *pcommand_to_vcommand* *pc* **in**
pstep *vmu* (*pip* *pc*) *pfailure* \rightarrow *vstep* (*vip* *vc* *vmu*) (*vpfailure* *vmu*).

Proof with subst.

```

induction pc; intros; subst vc; simpl; try ( solve [ inversion H ] ).
inversion H...
constructor.
trivial.

inversion H...
constructor.
clear IHpc2; rename IHpc1 into IH.
specify IH vmu.
simpl in IH.
specify IH H1.
trivial.
Qed.

Shows that pmultistep behaves similarly to vmultistep. Theorem pmultistep_sim_vmultistep
:  $\forall pc\ pc'\ vmu,$ 
  let vc := pcommand_to_vcommand pc in
  let vc' := pcommand_to_vcommand pc' in
    pmultistep vmu (pip pc) (pip pc')  $\rightarrow$  vmultistep (vip vc vmu) (vip vc' vmu).

```

Proof.

```

intros.
subst vc; subst vc'.
introduce states vmu0 s s' in H.
subst vmu0.
revert pc pc' H1 H2.
induction H; intros; subst.

strip H2.
constructor.

rename pc' into pc''.
state (pmultistep_pip_pip - - - H0).
eliminate existential pc' in H1.
subst s2.
state (IHpmultistep - - (refl_equal -) (refl_equal -)); clear IHpmultistep.
state pstep_sim_vstep.
specify H2 pc pc' vmu.
simpl in H2.
specify H2 H.
step (vip (pcommand_to_vcommand pc') vmu).
trivial.
trivial.

```

Qed.

Shows that *pmultistep* behaves similarly to *vmultistep*. **Theorem *pmultistep_sim_vmultistep_fail***
: $\forall pc\ vmu,$

```

let vc := pcommand_to_vcommand pc in
  pmultistep vmu (pip pc) pfailure → vmultistep (vip vc vmu) (vfailure vmu).

```

Proof.

```

intros.
subst vc.
introduce states vmu0 s s' in H.
subst vmu0.
revert pc H1 H2.
induction H; intros; subst.

discriminate.

destruct s2.
rename p into pc'.
state (IHpmultistep _ (refl_equal _) (refl_equal _)); clear IHpmultistep.
state (pstep_sim_vstep - - H).
step (vip (pcommand_to_vcommand pc') vmu).
trivial.
trivial.

revert H; clear; intros.
apply vmultistep_step_to_multi.
apply (pstep_sim_vstep_fail - - H).

```

Qed.

We wish to show that execution only gets stuck on a) a "failing" assume, b) a failure state, c) skip. Of these, only the former is hard (relatively to b and c) to recognize, as the assume may be hidden deep in a tree structure.

nested_assume describes commands where execution will need to deal with an assume-command first. *nested_assume e c* states that *c* is a command where *cAssert e* is the next command to be executed. **Inductive** *nested_assume* (*e* : *expr*) : *command* → **Prop** :=
| *naAssume* : *nested_assume e (cAssume e)*
| *naSequence* : ∀ *c1 c2*, *nested_assume e c1* → *nested_assume e (cSequence c1 c2)*.

Defines the general form of a stuck state. Our intention is to prove that no reduction rule applies on a command for which *stuck_state* is true, and conversely, that if a state *s* cannot be reduced any further, *stuck_state* must be true. **Inductive** *stuck_state* : *state* → **Prop** :=

```

| ssAssume : ∀ c (e : expr) mu, e mu ≠ T → nested_assume e c → stuck_state (ip c mu)
| ssSkip : ∀ mu, stuck_state (ip cSkip mu)
| ssFail : ∀ mu, stuck_state (failure mu).

```

A state *s* is *decidable_reducible* if either there exists a state *s'* for which *step s s'* is true (i.e. it is reducible), or, *s* is a stuck state. **Definition** *decidable_reducible* (*s* : *state*) := { *s' : state* | *step s s'* } + { *stuck_state s* }.

States that we can determine in finite time whether a command *c* is the skip command or not. **Lemma** *decidable_skip* : ∀ *c*, { *c = cSkip* } + { *c ≠ cSkip* }.

Proof.

```
clear; intros.  
destruct c; solve [ left; trivial | right; intros; subst; discriminate ].  
Qed.
```

States that every state s is *decidable_reducible*, i.e. that either it there exists a state s' for which *state* $s s'$ is true, or that *stuck_state* s holds. **Theorem** *decidable_reducible_states* : $\forall s, \text{decidable_reducible } s$.

Proof with subst.

```
unfold decidable_reducible.  
destruct s.  
induction c.  
  
left; destruct (decidable_eq_value (e s) T).  
   $\exists (ip \text{ cSkip } s); \text{constructor}; \text{trivial}.$   
   $\exists (\text{failure } s); \text{constructor}; \text{trivial}.$   
  
destruct (decidable_eq_value (e s) T).  
left;  $\exists (ip \text{ cSkip } s); \text{constructor}; \text{trivial}.$   
right; apply (ssAssume (cAssume e) e s); trivial.  
  constructor.  
  
left;  $\exists (ip \text{ cSkip } (\text{update\_store } s \ i \ (e \ s))); \text{constructor}.$   
  
clear IHc2; destruct IHc1.  
elim s0; intros; clear s0.  
destruct x.  
left;  $\exists (ip \text{ (cSequence } c \ c2) \ s0); \text{constructor}; \text{trivial}.$   
left;  $\exists (\text{failure } s0); \text{constructor}; \text{trivial}.$   
destruct (decidable_skip c1).  
subst.  
left;  $\exists (ip \text{ } c2 \ s); \text{constructor}.$   
right.  
inversion s0...  
inversion H2...  
apply (ssAssume (cSequence (cAssume e) c2) e s); trivial.  
  constructor; trivial.  
apply (ssAssume (cSequence (cSequence c0 c3) c2) e s).  
trivial.  
  constructor.  
trivial.  
elim n; trivial.  
right; constructor.  
left;  $\exists (ip \text{ } c1 \ s); \text{constructor}.$   
right; constructor.
```

Qed.

We show the soundness of *stuck_state*, i.e. that if *stuck_state* claims a state *s* is false, that there indeed is no applicable reduction step. **Theorem** *soundness_stuck_state* : $\forall s, stuck_state\ s \rightarrow \neg \exists s', step\ s\ s'$.

Proof with subst.

```
destruct s; red; intros.
rename s into mu.
elim H0; intros; clear H0.
rename x into s.
revert mu H s H1.
induction c; intros.

inversion H...
inversion H4.

destruct (decidable_eq_value (e mu) T).
inversion H...
inversion H4...
contradiction.
inversion H1...
contradiction.

inversion H...
inversion H4.

clear IHc2.
destruct (decidable_skip c1).
subst.
inversion H...
inversion H4...
inversion H2.
destruct (decidable_reducible_states (ip c1 mu)).
elim s0; intros; clear s0.
assert (stuck_state (ip c1 mu)).
revert H; clear; intros.
inversion H...
apply (ssAssume c1 e mu).
trivial.
inversion H3...
trivial.
specify IHc1 mu H0; clear H0.
apply (IHc1 - p).
specify IHc1 mu s0.
inversion H1...
apply (IHc1 - H5).
```

```

elim n; trivial.
apply (IHc1 - H5).
inversion H1.
inversion H...
inversion H4.
elim H0; intros; clear H0.
inversion H1.

```

Qed.

We show the completeness of *stuck_state*: if there is no reduction step starting from *s*, *s* is indeed recognized by *stuck_state* as being stuck. **Theorem** *completeness_stuck_state* : $\forall s, (\neg \exists s', \text{step } s \ s') \rightarrow \text{stuck_state } s$.

Proof.

```

intros.
destruct (decidable_reducible_states s).
elim s0; intros; clear s0.
elim H.
 $\exists x$ .
trivial.
trivial.

```

Qed.

apply_multistep_transitivity T splits a goal *multistep S S'* into two subgoals *multistep S T* and *multistep T S'*. Works similarly for *vmultistep* and *pmultistep*.

```

s : state
s' : state
s'' : state
=====
multistep s s''

```

apply_multistep_transitivity s'.

subgoal 1 is:

```

s : state
s' : state
s'' : state
=====
multistep s s'

```

subgoal 2 is:

```

s : state
s' : state
s'' : state
=====
multistep s' s''

```

```

Ltac apply_multistep_transitivity t :=
  match goal with
  | ⊢ multistep ?X ?Y ⇒ first [ apply (multistep_trans X t Y) | idtac "Failed to apply
transitivity" ]
  | ⊢ vmultistep ?X ?Y ⇒ first [ apply (vmultistep_trans X t Y) | idtac "Failed to apply
transitivity" ]
  | ⊢ pmultistep ?S ?X ?Y ⇒ first [ apply (pmultistep_trans S X t Y) | idtac "Failed to
apply transitivity" ]
  end.

```

New tactic notation: `apply_multistep_transitivity t` can be written as `step transitivity with t`. Tactic Notation "step" "transitivity" "with" *constr(t) := apply_multistep_transitivity t*.

Lifts $c1 \text{ -}^* \text{>} c1'$ into a sequence: $c1; c2 \text{ -}^* \text{>} c1'; c2$ **Theorem** *multistep_lift_seq* : $\forall c c' c2 \mu \mu'$,
 $\text{multistep } (ip \ c \ \mu) \ (ip \ c' \ \mu') \rightarrow$
 $\text{multistep } (ip \ (cSequence \ c \ c2) \ \mu) \ (ip \ (cSequence \ c' \ c2) \ \mu')$.

Proof.

```

intros.
introduce states s1 s2 in H.
revert c c' c2 mu mu' H0 H1.
induction H; intros; subst.
strip H1.
constructor.
elim (multistep_ip_ip _ _ _ H0); intros.
elim H1; intros; subst; clear H1.
state (IHmultistep x - c2 x0 mu' (refl_equal _) (refl_equal _)); clear IHmultistep.
refine (multiStep _ _ _ _ H1).
constructor.
trivial.

```

Qed.

Lifts $c1 \text{ -}^* \text{>} c1'$ into a sequence: $c1; c2 \text{ -}^* \text{>} c1'; c2$ **Theorem** *vmultistep_lift_seq* :
 $\forall c c' c2 \nu \mu \nu'$,
 $\text{vmultistep } (\nu ip \ c \ \nu \mu) \ (\nu ip \ c' \ \nu \mu') \rightarrow$
 $\text{vmultistep } (\nu ip \ (vcSequence \ c \ c2) \ \nu \mu) \ (\nu ip \ (vcSequence \ c' \ c2) \ \nu \mu')$.

Proof.

```

intros.

```

```

introduce states s1 s2 in H.
revert c c' c2 vmu vmu' H0 H1.
induction H; intros; subst.
strip H1.
constructor.
elim (vmultistep-vip-vip - - - H0); intros.
elim H1; intros; subst; clear H1.
state (IHvmultistep x - c2 x0 vmu' (refl_equal -) (refl_equal -)); clear IHvmultistep.
refine (vmultiStep - - - - H1).
constructor.
trivial.

```

Qed.

Lifts $c1 \text{ -*> } c1'$ into a sequence: $c1; c2 \text{ -*> } c1'$; $c2$ **Theorem pmultistep_lift_seq** : $\forall c c' c2 vmu,$
 $pmultistep vmu (pip c) (pip c') \rightarrow$
 $pmultistep vmu (pip (pcSequence c c2)) (pip (pcSequence c' c2)).$

Proof.

```

intros.
introduce states vmu' s1 s2 in H.
subst vmu'.
revert c c' c2 H1 H2.
induction H; intros; subst.
strip H2.
constructor.
rename c' into c''.
destruct s2.
rename p into c'.
state (IHpmultistep - - c2 (refl_equal -) (refl_equal -)); clear IHpmultistep.
step (pip (pcSequence c' c2)).
constructor.
trivial.
trivial.
inversion H0; subst.
inversion H1.

```

Qed.

Lifts $c \text{ -*> fail}$ into a sequence: $c; c' \text{ -*> fail}$ **Lemma multistep_lift_seq_fail** : $\forall c c' mu mu',$
 $multistep (ip c mu) (failure mu') \rightarrow$
 $multistep (ip (cSequence c c') mu) (failure mu').$

Proof with subst.

```

intros.
introduce states s1 s2 in H.

```

```

revert c c' mu mu' H0 H1.
induction H; intros...
discriminate.
destruct s2.
rename c0 into c2.
rename mu' into mu"; rename s into mu'.
step (ip (cSequence c2 c') mu').
constructor.
trivial.
apply (IHmultistep c2 c' - - (refl_equal -) (refl_equal -)); clear IHmultistep.
inversion H0...
step (failure mu').
apply stepSeqFail.
trivial.
constructor.
inversion H1.

```

Qed.

Lifts $c \text{ -*> fail}$ into a sequence: $c; c' \text{ -*> fail}$ Lemma *vmultistep_lift_seq_fail* : $\forall c c' vmu vmu', vmultistep (vip c vmu) (vfailure vmu') \rightarrow vmultistep (vip (vcSequence c c') vmu) (vfailure vmu')$.

Proof with subst.

```

intros.
introduce states s1 s2 in H.
revert c c' vmu vmu' H0 H1.
induction H; intros...
discriminate.
destruct s2.
rename v into c2.
rename vmu' into vmu"; rename v0 into mu'.
step (vip (vcSequence c2 c') mu').
constructor.
trivial.
apply (IHvmultistep c2 c' - - (refl_equal -) (refl_equal -)); clear IHvmultistep.
inversion H0...
step (vfailure vmu').
apply vstepSeqFail.
trivial.
constructor.
inversion H1.

```

Qed.

Lifts $c \text{ -*> fail}$ into a sequence: $c; c' \text{ -*> fail}$ Lemma *pmultistep_lift_seq_fail* : $\forall c1 c2 vmu, pmultistep vmu (pip c1) pfailure \rightarrow pmultistep vmu (pip (pcSequence c1 c2))$

pfailure.

Proof with subst.

```
intros.
introduce states vmu' s1 s2 in H.
subst vmu'.
revert c1 c2 H1 H2.
induction H; intros...
discriminate.
destruct s2.
rename p into c1'.
step (pip (pcSequence c1' c2)).
constructor.
trivial.
apply (IHpmultistep _ c2 (refl_equal _) (refl_equal _)); clear IHpmultistep.
step pfailure.
constructor.
trivial.
constructor.
```

Qed.

1.3.2 Single Assignment

delta_id $A f g d$ expresses that for all $ids x$, $f x = g x$, except possibly for the ids contained in the set d , where f and g are total functions with domain id and range A . **Definition** $delta_id (A : Set) (f g : id \rightarrow A) (d : IdSet.t) := \forall x : id, f x = g x \vee IdSet.In x d$.

A few theorems about *delta_id*.

Lemma *delta_id_x_x* : $\forall A x d, delta_id A x x d$.

Proof.

```
unfold delta_id; intros.
left; trivial.
```

Qed.

Hint Resolve *delta_id_x_x*.

Lemma *delta_id_unionl* : $\forall A f g d1 d2, delta_id A f g d1 \rightarrow delta_id A f g (IdSet.union d1 d2)$.

Proof.

```
unfold delta_id; intros.
destruct (H x); clear H.
left; trivial.
right.
apply (@IdSet.union_2 d1 d2 x H0).
```

Qed.

Lemma *delta_id_unionr* : $\forall A f g d1 d2, \text{delta_id } A f g d2 \rightarrow \text{delta_id } A f g (\text{IdSet.union } d1 d2)$.

Proof.

```

  unfold delta_id.
  intros.
  destruct (H x); clear H.
  left; trivial.
  right.
  apply (@IdSet.union_3 d1 d2 x H0).

```

Qed.

Lemma *delta_id_combine* : $\forall A f g h d1 d2, \text{delta_id } A f g d1 \rightarrow \text{delta_id } A g h d2 \rightarrow \text{delta_id } A f h (\text{IdSet.union } d1 d2)$.

Proof.

```

  unfold delta_id; intros.
  destruct (H x); destruct (H0 x); try (right; solve [ apply IdSet.union_2; trivial |
  apply IdSet.union_3; trivial ]).
  left; rewrite H1; rewrite H2; trivial.

```

Qed.

Lemma *delta_id_extend* : $\forall A f g d1 d2, \text{delta_id } A f g d1 \rightarrow \text{IdSet.Subset } d1 d2 \rightarrow \text{delta_id } A f g d2$.

Proof.

```

  unfold delta_id; unfold IdSet.Subset; intros.
  destruct (H x); clear H.
  left; trivial.
  right; apply (H0 - H1).

```

Qed.

We specialize *delta_id* for *stores*. Definition *store_delta* := *delta_id value*.

We specialize *delta_id* for *vmaps*. Definition *vmap_delta* := *delta_id nat*.

Definition *vmap_delta_combine* := *delta_id_combine nat*.

Theorem *store_delta_mu_mu* : $\forall (\mu : \text{store}) (d : \text{IdSet.t}), \text{store_delta } \mu \mu d$.

Proof.

```

  unfold store_delta.
  apply (delta_id_x_x value).

```

Qed.

Theorem *vmap_delta_v_v* : $\forall (v : \text{vmap}) (d : \text{IdSet.t}), \text{vmap_delta } v v d$.

Proof.

```

  unfold vmap_delta.
  apply (delta_id_x_x nat).

```

Qed.

Theorem *vmap_delta_extend* : $\forall v1 v2 d1 d2, \text{vmap_delta } v1 v2 d1 \rightarrow \text{IdSet.Subset } d1 d2 \rightarrow \text{vmap_delta } v1 v2 d2$.

Proof.

```
unfold vmap_delta.
apply delta_id_extend.
```

Qed.

Hint Resolve *store_delta_mu_mu*.

Hint Resolve *vmap_delta_v_v*.

Definition *store_delta_unionl* := *delta_id_unionl* value.

Definition *vmap_delta_unionl* := *delta_id_unionl* nat.

Definition *store_delta_unionr* := *delta_id_unionr* value.

Definition *vmap_delta_unionr* := *delta_id_unionr* nat.

We prove that the *step* relation keeps the store unchanged, except for the command's targets (the set of identifiers appearing on the left side of assignments). **Theorem** *step_store_delta* : $\forall c c' \mu \mu' d, \text{IdSet.Subset } (\text{targets } c) d \rightarrow \text{step } (ip\ c\ \mu) (ip\ c'\ \mu') \rightarrow \text{store_delta } \mu\ \mu' d$.

Proof.

```
intros.
introduce states s s' in H0.
revert c c' d mu mu' H H1 H2.
induction H0; subst; intros; try (first | discriminate; fail | strip H1; strip H2;
auto ).
refine (IHstep c1 c1' d mu0 mu'0 _ (refl_equal _) (refl_equal _)); clear IHstep.
simpl in H.
destruct (union_subset _ _ _ H); trivial.

unfold store_delta; unfold delta_id; intros.
unfold update_store.
unfold rebind.
unfold IdSet.Subset in H.
destruct (decidable_eq_id x0 x).
subst.
right.
simpl in H.
apply (H x).
apply IdSet.singleton_2.
unfold IdSet.E.eq.
unfold Identifier_OT.eq.
trivial.
left; trivial.
```

Qed.

The targets set of a command will not grow through the *step* relation. **Theorem** *step_targets_subset* : $\forall c c' \mu \mu', \text{step } (ip\ c\ \mu) (ip\ c'\ \mu') \rightarrow \text{IdSet.Subset } (\text{targets } c') (\text{targets } c)$.

Proof.

```
intros.
introduce states s s' in H.
revert c c' mu mu' H0 H1.
induction H; intros; subst; first [ discriminate; fail | strip H0; strip H1; simpl ].
apply IdSetProperties.subset_empty.
apply IdSetProperties.subset_empty.
state (IHstep - - - (refl_equal _) (refl_equal _)); clear IHstep.
apply IdSetProperties.union_subset_4; trivial.
apply IdSetProperties.union_subset_2.
apply IdSetProperties.subset_empty.
apply IdSetProperties.union_subset_1.
apply IdSetProperties.union_subset_2.
```

Qed.

Given the stores mu , mu' and mu'' , and $mu\ x = mu'\ x$ for all x except for those in some set d , and $mu'\ x = mu''\ x$ for all x except for those in that same set d , then $mu\ x = mu''\ x$ for all x except for those in d . **Theorem** *store_delta_trans* : $\forall\ mu\ mu'\ mu''\ d$, *store_delta* $mu\ mu'\ d \rightarrow store_delta\ mu'\ mu''\ d \rightarrow store_delta\ mu\ mu''\ d$.

Proof.

```
unfold store_delta; unfold delta_id; intros.
destruct (H x); destruct (H0 x); try (right; trivial; fail).
left; rewrite H1; rewrite H2; trivial.
```

Qed.

step_store_delta adapted to the multistep relation. **Theorem** *multistep_store_delta* : $\forall\ c\ c''\ mu\ mu''\ d$, *IdSet.Subset* (*targets* c) $d \rightarrow multistep\ (ip\ c\ mu)\ (ip\ c''\ mu'') \rightarrow store_delta\ mu\ mu''\ d$.

Proof.

```
intros.
introduce states s s' in H0.
revert c c'' d mu mu'' H H1 H2; induction H0; intros; subst.
strip H2.
auto.
elim (multistep_ip_ip - - - H0); intros.
elim H2; intros; clear H2; subst.
rename x into c'; rename x0 into mu'.
assert (IdSet.Subset (targets c') d).
state (step_targets_subset - - - H).
apply (IdSetProperties.subset_trans H2 H1).
state (IHmultistep c' c'' d mu' mu'' H2 (refl_equal _) (refl_equal _)); clear IHmultistep.
state (step_store_delta - - - - H1 H).
apply (store_delta_trans - - - - H4 H3).
```

Qed.

Checks if two versioned stores under a certain vmap are equivalent. **Definition** *vs-tore-sync_vstore* (*mu* : *vstore*) (*v v'* : *vmap*) (*mu'* : *vstore*) :=
 $\forall x : id, mu(x, v\ x) = mu'(x, v'\ x).$

Generates a command which copies across versions:

```
copy_vcmd x n m == x_m := x_n
```

Definition *copy_vcmd* (*x* : *id*) (*n m* : *nat*) :=
vcAssign (*x*, *m*) (**fun** (*vmu* : *vstore*) $\Rightarrow vmu(x, n)$).

Shows that *copy_vcmd_works* as expected: execution of *copy_vcmd x n m* in a store *vmu* will lead to a new store *vmu'* where *vmu'* (*x*, *m*) = *vmu* (*x*, *n*) (i.e. *x_m* is now equal to *x_n*), while all other store bindings remain unchanged. **Theorem** *copy_vcmd_works* :

$$\forall x\ n\ m\ vmu, \exists vmu',$$

$$vstep(vip(copy_vcmd\ x\ n\ m)\ vmu)\ (vip\ vcSkip\ vmu') \wedge$$

$$vmu'(x, m) = vmu(x, n) \wedge$$

$$\forall y\ k, (x \neq y \vee k \neq m) \rightarrow vmu(y, k) = vmu'(y, k).$$

Proof.

```
unfold copy_vcmd; intros.
 $\exists$  (update_vstore vmu (x, m) (vmu (x, n))).
split.
apply (vstepAssign (x, m) (fun vmu0 : vstore  $\Rightarrow$  vmu0 (x, n)) vmu).
split.
unfold update_vstore.
unfold rebind.
destruct (decidable_eq_vid (x, m) (x, m)).
trivial.
elim n0; trivial.
intros.
unfold update_vstore.
unfold rebind.
destruct (decidable_eq_vid (y, k) (x, m)).
strip e.
destruct H; elim H; trivial.
trivial.
```

Qed.

Right fold on lists of *ids*. **Fixpoint** *foldr* (*B* : **Set**) (*f* : *id* \rightarrow *B* \rightarrow *B*) (*s* : *list id*) : *B* \rightarrow *B* :=

```
fun i  $\Rightarrow$  match s with
  | nil  $\Rightarrow$  i
  | x :: l  $\Rightarrow$  f x (foldr B f l i)
end.
```

Right fold on finite *id*-sets **Definition** *fset_foldr* (*B* : **Set**) (*f* : *id* \rightarrow *B* \rightarrow *B*) (*s* : *IdSet.t*) (*init* : *B*) : *B*.

```

intros.
unfold IdSet.t in s.
destruct s.
unfold IdSet.Raw.t in this.
apply (foldr B f this init).
Defined.

```

If the *id* x is in need of a synchronisation (i.e. the version map v and v' assign differing versions to x) this function generates an assignment command. c is a "continuation" so as to make it possible to create a chain of multiple assignments.

Thus, if $v\ x = v'\ x$, then $insert_copy_vcmd\ v\ v'\ x\ c$ doesn't need to produce a synchronizing assignment command and just returns c . Conversely, if $v\ x \neq v'\ x$, the sequence $copy_vcmd\ x\ (v\ x)\ (v'\ x); c$ will be returned. **Definition** $insert_copy_vcmd\ v\ v'\ x\ c :=$
 if $decidable_eq_id\ (v\ x)\ (v'\ x)$
 then c
 else $(vcSequence\ (copy_vcmd\ x\ (v\ x)\ (v'\ x))\ c)$.

Given two version maps v and v' and a set of identifiers, $sync_vcommand$ generates a command which perform a "store synchronization" from v to v' .

Example:

```

mu = { x_4 -> 8, y_3 -> 14, z_5 -> 2, ... }
v  = { x -> 4, y -> 3, z -> 5, ... }
v' = { x -> 7, y -> 4, z -> 5, ... }

```

x and y are assigned different versions by v and v' (4 vs 7, 3 vs 4), meaning they need synchronization. $sync_vcommand$ thus generates

```

x_7 := x_4;
y_4 := y_3;
skip

```

Definition $sync_vcommand\ (ids : IdSet.t)\ (v\ v' : vmap) : vcommand :=$
 $(fset_foldr\ vcommand\ (insert_copy_vcmd\ v\ v')\ ids\ vcSkip)$.

Shows that synchronization commands evaluate to skip. **Theorem** $sync_vcommand_goes_to_skip$
 $: \forall\ ids\ v\ v'\ vmu, \exists\ vmu', vmultistep\ (vip\ (sync_vcommand\ ids\ v\ v')\ vmu)\ (vip\ vcSkip\ vmu')$.

Proof with subst.

```

destruct ids.
induction this; intros.

compute.
 $\exists\ vmu; constructor$ .

rename a into x.
rename sorted into x_xs.
inversion x_xs...
rename H1 into xs.

```

```

simpl.
unfold insert_copy_vcmd at 1.
destruct (decidable_eq_id (v x) (v' x)).
elim (IHthis xs v v' vmu); intros.
rename x0 into vmu';  $\exists$  vmu'.
trivial.

evar (vmu' : vstore).
elim (IHthis xs v v' vmu'); clear IHthis; intros.
rename x0 into vmu''.
 $\exists$  vmu''.
unfold copy_vcmd.
step (vip (vcSequence vcSkip (foldr vcommand (insert_copy_vcmd v v') this vcSkip))
vmu').
constructor.
instantiate (1 := update_vstore vmu (x, v' x) ((fun vmu0 : vstore  $\Rightarrow$  vmu0 (x, v x))
vmu)) in (Value of vmu').
subst vmu'.
constructor.
step (vip (foldr vcommand (insert_copy_vcmd v v') this vcSkip) vmu').
constructor.
trivial.

```

Qed.

Shows execution of a synchronization commands always ends up in the same state.

Theorem *sync_vcommand_determinism* : \forall *ids v v' vmu vmu1' vmu2'*,

```

let c := sync_vcommand ids v v' in
  vmultistep (vip c vmu) (vip vcSkip vmu1')  $\rightarrow$  vmultistep (vip c vmu) (vip vcSkip vmu2')
 $\rightarrow$  vmu1' = vmu2'.

```

Proof with subst.

```

destruct ids.
induction this; intros.

simpl in c.
subst c.
inversion H...
inversion H0...
trivial.
inversion H1.
inversion H1.

rename a into x; rename sorted into x_xs; inversion x_xs...
rename H3 into xs.
simpl in c.
unfold insert_copy_vcmd in c.

```

```

destruct (decidable_eq_id (v x) (v' x)).
specify IHthis xs v v' vmu vmu1' vmu2'.
simpl in IHthis.
subst c.
specify IHthis H H0.
trivial.

subst c.
inversion H...
inversion H1...
inversion H8...
clear H H1 H8.
inversion H2...
inversion H...
inversion H8.
clear H2 H.
inversion H0...
inversion H...
inversion H8...
clear H0 H H8.
inversion H2...
inversion H...
inversion H8.
clear H2 H.
specify IHthis xs v v'.
specify IHthis (update_vstore vmu (x, v' x) (vmu (x, v x))).
specify IHthis vmu1' vmu2'; simpl in IHthis.
specify IHthis H1 H0.
trivial.
inversion H8.
inversion H2...
inversion H3.
inversion H1...
inversion H3.
inversion H2...
inversion H3.

```

Qed.

Lemma *sync_vcommand_empty_delta* :

$$\forall \text{ids } v \ v' \ \text{vmu } \text{vmu}',$$

$$\text{equivalent_functions } v \ v' \rightarrow$$

$$\text{vmultistep } (\text{vip } (\text{sync_vcommand } \text{ids } v \ v') \ \text{vmu}) \ (\text{vip } \text{vcSkip } \text{vmu}') \rightarrow$$

$$\text{vstore_sync_vstore } \text{vmu } v \ v' \ \text{vmu}'.$$

Proof with subst.

```

destruct ids.
induction this; intros.

simpl in H0.
inversion H0...
unfold vstore_sync_vstore.
intros.
unfold equivalent_functions in H.
specify H x.
rewrite H.
trivial.
inversion H1.

rename a into x; rename sorted into x_xs; inversion x_xs...
rename H3 into xs.
specify IHthis xs.
simpl in H0.
unfold insert_copy_vcmd in H0.
destruct (decidable_eq_id (v x) (v' x)).
specify IHthis v v' vmu vmu' H H0.
trivial.
elim n.
unfold equivalent_functions in H.
apply (H x).
Qed.

```

A localized version of *vstore_sync_vstore*: it need only apply for the *ids* contained in *D*. **Definition** *vstore_sync_vstore_local* (*mu* : *vstore*) (*v v'* : *vmap*) (*mu'* : *vstore*) (*D* : *IdSet.t*) :=

$$\forall x : id, IdSet.In x D \rightarrow mu (x, v x) = mu' (x, v' x).$$

Store bindings for all versions of identifiers not in *D* remain untouched. **Lemma** *sync_vcommand_pres* :

$$\forall D a v v' vmu vmu' n, \\ \neg IdSet.In a D \rightarrow vmultistep (vip (sync_vcommand D v v') vmu) (vip vcSkip vmu') \\ \rightarrow vmu (a, n) = vmu' (a, n).$$

Proof with subst.

```

destruct D.
induction this; intros.

simpl in H0.
inversion H0...
trivial.
inversion H1.

rename a into x; rename sorted into x_xs; inversion x_xs...
rename H3 into xs.

```

```

rename a0 into y.
assert ( $\neg$  IdSet.In y (IdSet.Build_slist xs)).
revert H; clear; intros.
red; intros; elim H; clear H.
unfold IdSet.In in  $\times \vdash \times$ .
simpl.
apply InA_cons_tl.
simpl in H0.
trivial.
assert ( $x \neq y$ ).
red; intros.
elim H.
unfold IdSet.In.
simpl.
subst.
constructor.
auto.
simpl in H0.
unfold insert_copy_vcmd in H0.
destruct (decidable_eq_id (v x) (v' x)).
specify IHthis xs y v v' vmu vmu' n H1 H0.
trivial.
inversion H0...
inversion H3...
inversion H10...
clear H0 H3 H10.
inversion H5...
inversion H0...
inversion H10.
clear H5 H0.
specify IHthis xs y v v' (update_vstore vmu (x, v' x) (vmu (x, v x))) vmu' n H1 H3.
clear H3.
unfold update_vstore in IHthis.
unfold rebind in IHthis.
destruct (decidable_eq_vid (y, n) (x, v' x)).
strip e.
elim H2; trivial.
trivial.
inversion H3...
inversion H6.
inversion H5...
inversion H6.

```

Qed.

Opaque decidable_eq_vid.

Auxiliary for `sync_vcommand_works`. Lemma *sync_vcommand_works_aux* :

$\forall D v v' vmu vmu'$,

$vmultistep (vip (sync_vcommand D v v') vmu) (vip vSkip vmu') \rightarrow vstore_sync_vstore_local\ vmu\ v\ v'\ vmu'\ D.$

Proof with `subst`.

`destruct D.`

`induction this; intros.`

`unfold vstore_sync_vstore_local.`

`intros.`

`inversion H0.`

`rename a into x; rename sorted into x_xs; inversion x_xs...`

`rename H2 into xs; specify IHthis xs v v'.`

`simpl in H.`

`unfold insert_copy_vcnd in H.`

`destruct (decidable_eq_id (v x) (v' x)).`

`specify IHthis vmu vmu' H.`

`unfold vstore_sync_vstore_local in $\times \vdash \times$.`

`intro y; intros.`

`specify IHthis y.`

`destruct (IdSetProperties.In_dec y (IdSet.Build_slist xs)).`

`apply (IHthis i).`

`clear IHthis.`

`assert (x = y).`

`revert H0 n; clear; intros.`

`inversion H0...`

`compute in H1.`

`symmetry; trivial.`

`contradiction.`

`state sync_vcommand_preservation.`

`specify H2 (IdSet.Build_slist xs).`

`state (H2 - - - - (v y) n H).`

`clear H2.`

`compute in $\times \vdash \times$.`

`subst.`

`rewrite H4.`

`rewrite e.`

`trivial.`

`inversion H...`

`inversion H0...`

```

inversion H7...
clear H H0 H7.
inversion H1...
inversion H...
inversion H7.
clear H1 H.
state (IHthis _ _ H0); clear IHthis.
unfold vstore_sync_vstore_local in  $\times \vdash \times$ .
intro y; intros; specify H y.
destruct (IdSetProperties.In_dec y (IdSet.Build_slist xs)).
specify H i.
unfold update_vstore in H.
unfold rebind in H.
compute in x.
destruct (decidable_eq_vid (y, v y) (x, v' x)).
clear H0.
injection e; intros.
symmetry in H2.
subst.
contradiction.
trivial.

clear H.
assert (x = y).
revert H1 n0; clear; intros.
inversion H1...
compute in H0; symmetry in H0; trivial.
contradiction.
subst.
state sync_vcommand_preservation.
specify H (IdSet.Build_slist xs) y v v'.
state (H _ _ (v y) n0 H0).
state (H _ _ (v' y) n0 H0).
clear H H0 n0.
unfold update_vstore in  $\times \vdash$ .
unfold rebind in  $\times \vdash$ .
compute in  $\times \vdash \times$ .
destruct (decidable_eq_vid (y, v y) (y, v' y)).
strip e.
rewrite H in  $\times \vdash \times$ .
trivial.
destruct (decidable_eq_vid (y, v' y) (y, v' y)).
trivial.

```

```

elim n1.
trivial.
inversion H0...
inversion H2.
inversion H1...
inversion H2.

```

Qed.

Given two *vmaps* v and v' whose mappings are the same except for the identifiers contained in ids , then stepping through *sync_vcommand* starting with store mu results in a new store mu' which is synchronized with mu with respect to v and v' . **Theorem *sync_vcommand_works*** :

```

 $\forall ids\ v\ v'\ vmu\ vmu',$ 
   $vmap\_delta\ v\ v'\ ids \rightarrow vmultistep\ (vip\ (sync\_vcommand\ ids\ v\ v')\ vmu)\ (vip\ vcSkip\ vmu') \rightarrow vstore\_sync\_vstore\ vmu\ v\ v'\ vmu'.$ 

```

Proof with subst.

```

intros.
state sync_vcommand_works_aux.
specify H1 ids v v' vmu vmu' H0.
unfold vstore_sync_vstore.
intros.
destruct (IdSetProperties.In_dec x ids).
unfold vstore_sync_vstore_local in H1.
specify H1 x i.
trivial.
unfold vmap_delta in H.
specify H x.
destruct H.
state sync_vcommand_preservation.
specify H2 ids x v v' vmu vmu' (v x) n H0.
rewrite H in  $\times \vdash \times$ .
trivial.
contradiction.

```

Qed.

A kind of transitivity. **Theorem *combine_vmaps*** : $\forall mu\ vmu\ vmu'\ v\ v', store_sync_vstore\ mu\ v\ vmu \rightarrow vstore_sync_vstore\ vmu\ v\ v'\ vmu' \rightarrow store_sync_vstore\ mu\ v'\ vmu'$.

Proof.

```

compute; intros.
specify H x.
specify H0 x.
rewrite H; rewrite H0.
trivial.

```

Qed.

Transparent decidable_eq_vid.

Joins two *vmaps* together by taking the maximum version of each *id*. Definition *join*
 $(v1\ v2 : vmap) :=$

`fun x => max (v1 x) (v2 x).`

Transforms the *command* *c* to an equivalent *vcommand*. Fixpoint *transform_sa* (*c* :
command) (*v* : *vmap*) : (*vcommand* × *vmap*) :=

```

match c with
| cAssert e => (vcAssert (version_expr e v), v)
| cAssume e => (vcAssume (version_expr e v), v)
| cAssign x e => (vcAssign (x, S (v x)) (version_expr e v), inc v x)
| cSequence c1 c2 => let (c1', v') := transform_sa c1 v in
                      let (c2', v'') := transform_sa c2 v' in
                      (vcSequence c1' c2', v'')
| cSkip => (vcSkip, v)
| cChoice c1 c2 => let (c1', v1') := transform_sa c1 v in
                    let (c2', v2') := transform_sa c2 v in
                    let t1 := targets c1 in
                    let t2 := targets c2 in
                    let v' := join v1' v2' in
                    let t := IdSet.union t1 t2 in
                    let d1 := sync_vcommand t v1' v' in
                    let d2 := sync_vcommand t v2' v' in
                    (vcChoice (vcSequence c1' d1) (vcSequence c2' d2), v')

```

`end.`

Theorem *sync_vcommand_size* : $\forall D\ v\ v'$,

`let c := sync_vcommand D v v' in`

`vcommand_metric c ≤ 3 × IdSet.cardinal D + 1.`

Proof with `subst`.

`destruct D.`

`induction this.`

`intros.`

`subst c; simpl.`

`trivial.`

`rename a into x.`

`rename this into xs.`

`rename sorted into sorted_x_xs.`

`inversion sorted_x_xs...`

`rename H1 into sorted_xs.`

`intros.`

`unfold IdSet.cardinal.`

`simpl IdSet.Raw.cardinal.`

```

specify IHthis sorted_xs v v'.
cbv zeta in IHthis.
subst c.
simpl sync_vcommand.
unfold insert_copy_vcmd.
destruct (decidable_eq_id (v x) (v' x)).
change (foldr vcommand
  (fun (x0 : id) (c : vcommand) =>
    if decidable_eq_id (v x0) (v' x0)
    then c
    else vcSequence (copy_vcmd x0
      (v x0)
      (v' x0))
    c)
  xs
  vcSkip)
with
(sync_vcommand (IdSet.Build_slist sorted_xs) v v').
unfold IdSet.cardinal in IHthis.
introduce new identifier N for
(vcommand_metric (sync_vcommand (IdSet.Build_slist sorted_xs) v v')).
rewrite ← H in IHthis.
algebraically rewrite
(3 × S (IdSet.Raw.cardinal xs) + 1)
as
(3 + 3 × IdSet.Raw.cardinal xs + 1).
change (IdSet.Raw.cardinal (IdSet.Build_slist sorted_xs)) with (IdSet.Raw.cardinal xs) in
IHthis.
omega.
change (foldr vcommand
  (fun (x0 : id)
    (c : vcommand) => if decidable_eq_id (v x0)
      (v' x0)
    then c
    else vcSequence (copy_vcmd x0 (v x0)
      (v' x0))
    c)
  xs
  vcSkip)
with
(sync_vcommand (IdSet.Build_slist sorted_xs) v v').
introduce new identifier c for (sync_vcommand (IdSet.Build_slist sorted_xs) v v').

```

```

rewrite ← H in IHthis.
simpl vcommand_metric.
change (IdSet.cardinal (IdSet.Build_slist sorted_xs)) with (IdSet.Raw.cardinal xs) in IHthis.
omega.

```

Qed.

Theorem *targets_cardinality_le* : $\forall c : \text{command}, \text{IdSet.cardinal} (\text{targets } c) \leq \text{command_metric } c$.

Proof.

```

induction c; simpl; auto with arith.
state (IdSetProperties.union_cardinality_le (targets c1) (targets c2)).
refine_le H; clear H.
omega.
state (IdSetProperties.union_cardinality_le (targets c1) (targets c2)).
omega.

```

Qed.

Theorem *linear_sync_vcommand* : $\forall c v v', \text{let } vc := \text{sync_vcommand} (\text{targets } c) v v' \text{ in } \text{vcommand_metric } vc \leq 3 \times \text{command_metric } c + 1$.

Proof.

```

intros.
subst vc.
state (sync_vcommand_size (targets c) v v').
cbv zeta in H.
state (targets_cardinality_le c).
omega.

```

Qed.

We show that the SA transformation is quadratic in size. **Theorem** *quadratic_sa_transformation* : $\forall c v,$

```

let (c', _) := transform_sa c v in
let x := command_metric c in
vcommand_metric c' ≤ 5 × x × x + 5 × x.

```

Proof.

```

induction c; intros;
try ( solve [ simpl; auto with arith ] ).
simpl transform_sa.
introduce pair (transform_sa c1 v) as c1' v'.
introduce pair (transform_sa c2 v') as c2' v''.
specify IHc1 v.
specify IHc2 v'.
rewrite ← H0 in IHc1.
rewrite ← H1 in IHc2.
cbv zeta in × ⊢ ×.

```

```

simpl vcommand_metric.
simpl command_metric.
algebraically rewrite
  ( $S (vcommand\_metric\ c1' + vcommand\_metric\ c2')$ )
  as
  ( $vcommand\_metric\ c1' + vcommand\_metric\ c2' + 1$ ).
refine left bound with
  ( $5 \times command\_metric\ c1 \times command\_metric\ c1 +$ 
   $5 \times command\_metric\ c1 +$ 
   $5 \times command\_metric\ c2 \times command\_metric\ c2 +$ 
   $5 \times command\_metric\ c2 + 1$ ); clear IHc1 IHc2.
introduce new identifier a for (command_metric c1).
introduce new identifier b for (command_metric c2).
clear.
algebraically rewrite
  ( $5 \times S (a + b) \times S (a + b) + 5 \times S (a + b)$ ) as
  ( $10 + 15*a + 5*a*a + 15*b + 10*a*b + 5*b*b$ ).
algebraically rewrite
  ( $5 \times a \times a + 5 \times a + 5 \times b \times b + 5 \times b + 1$ ) as
  ( $5 \times a + 5 \times b \times b + 5 \times b + 1 + 5 \times a \times a$ ).
algebraically rewrite
  ( $10 + 15 \times a + 5 \times a \times a +$ 
   $15 \times b + 10 \times a \times b + 5 \times b \times b$ )
  as
  ( $10 + 15 \times a + 15 \times b + 10 \times a \times b +$ 
   $5 \times b \times b + 5 \times a \times a$ ).
apply plus_le_compat_r.
algebraically rewrite
  ( $5 \times a + 5 \times b \times b + 5 \times b + 1$ ) as
  ( $0 + (5 \times a + 5 \times b \times b + 5 \times b + 1)$ ).
algebraically rewrite
  ( $10 + 15 \times a + 15 \times b + 10 \times a \times b + 5 \times b \times b$ ) as
  ( $9 + 10 \times a + 10 \times b + 10 \times a \times b +$ 
   $(5 \times a + 5 \times b \times b + 5 \times b + 1)$ ).
apply plus_le_compat_r.
auto with arith.

specify IHc1 v.
specify IHc2 v.
cbv zeta in  $\times \vdash \times$ .
simpl transform_sa.
introduce pair (transform_sa c1 v) as  $c1' v1'$ .
introduce pair (transform_sa c2 v) as  $c2' v2'$ .

```

```

rewrite ← H0 in IHc1.
rewrite ← H1 in IHc2.
simpl vcommand_metric.
simpl command_metric.
introduce new identifier ts for
  (IdSet.union (targets c1) (targets c2)).
state (linear_sync_vcommand
      (cChoice c1 c2)
      v1'
      (join v1' v2')).
state (linear_sync_vcommand
      (cChoice c1 c2)
      v2'
      (join v1' v2')).
cbv zeta in H2, H3.
change (foldr vcommand
        (insert_copy_vcmd v1'
                          (join v1' v2'))
        (IdSet.Raw.union (targets c1)
                          (targets c2))
        vcSkip)
with
  (sync_vcommand (IdSet.union (targets c1)
                              (targets c2))
                 v1'
                 (join v1' v2')).
change (foldr vcommand
        (insert_copy_vcmd v2'
                          (join v1' v2'))
        (IdSet.Raw.union (targets c1)
                          (targets c2))
        vcSkip) with
  (sync_vcommand (IdSet.union (targets c1)
                              (targets c2))
                 v2'
                 (join v1' v2')).

rewrite ← H.
simpl targets in H2, H3.
rewrite ← H in H2, H3.
introduce new identifier s1 for
  (vcommand_metric (sync_vcommand ts v1' (join v1' v2'))).
rewrite ← H4 in H2.

```

```

introduce new identifier s2 for
  (vcommand_metric (sync_vcommand ts v2' (join v1' v2'))).
rewrite ← H5 in H3.
simpl command_metric in H2, H3.
algebraically rewrite
  (3 × S (command_metric c1 + command_metric c2) + 1) as
  (3 × command_metric c1 + 3 × command_metric c2 + 4).
clear H4 H5.
algebraically rewrite
  (S (S (vcommand_metric c1' + s1 +
    S (vcommand_metric c2' + s2))))
  as
  (vcommand_metric c1' + s1 + vcommand_metric c2' + s2 + 3).
refine left bound with
  (5 × command_metric c1 × command_metric c1 +
  5 × command_metric c1 + s1 +
  5 × command_metric c2 × command_metric c2 +
  5 × command_metric c2 + s2 + 3).
introduce new identifier a for (command_metric c1).
introduce new identifier b for (command_metric c2).
state (command_metric_min_size c1).
state (command_metric_min_size c2).
rewrite ← H4 in H6.
rewrite ← H5 in H7.
clear IHc1 IHc2.
refine left bound with
  (5 × a × a + 5 × a + 3 × command_metric c1 +
  3 × command_metric c2 + 4 + 5 × b × b +
  5 × b + 3 × command_metric c1 +
  3 × command_metric c2 + 4 + 3).
clear H2 H3.
rewrite ← H4; rewrite ← H5.
revert H6 H7; clear; intros.
algebraically rewrite
  (5 × a × a + 5 × a + 3 × a + 3 × b + 4 +
  5 × b × b + 5 × b + 3 × a + 3 × b + 4 + 3)
  as
  (11 + 11*a + 5*a*a + 11*b + 5*b*b).
algebraically rewrite
  (5 × S (a + b) × S (a + b) + 5 × S (a + b)) as
  (10 + 15*a + 5*a*a + 15*b + 10*a*b + 5*b*b).
algebraically rewrite

```

```

    (11 + 11 × a + 5 × a × a + 11 × b + 5 × b × b) as
    (11 + 11 × a + 11 × b + (5 × a × a + 5 × b × b)).
  algebraically rewrite
    (10 + 15 × a + 5 × a × a + 15 × b +
     10 × a × b + 5 × b × b)
  as
    (10 + 15 × a + 15 × b + 10 × a × b +
     (5 × a × a + 5 × b × b)).
  apply plus_le_compat_r.
  algebraically rewrite
    (11 + 11 × a + 11 × b) as
    (1 + (10 + 11 × a + 11 × b)).
  algebraically rewrite
    (10 + 15 × a + 15 × b + 10 × a × b) as
    (4 × a + 4 × b + 10 × a × b + (10 + 11 × a + 11 × b)).
  apply plus_le_compat_r.
  assert (1 ≤ 4 × a).
  omega.
  assert (0 ≤ 4 × b + 10 × a × b).
  auto with arith.
  omega.
Qed.

```

Some tests to make sure the algorithms work as expected. Of course, this does not count as proof of correctness. `Module Tests.`

```
Import IdSet.
```

We temporarily break open the `id` abstraction, so that we can define programs, stores, etc. `id` is made opaque again at the end of the module. *Transparent id.*

```
Section tests.
```

```
Definition x1 : id := 1.
```

```
Definition x2 : id := 2.
```

```
Definition x3 : id := 3.
```

```
Definition x4 : id := 4.
```

```
Definition x5 : id := 5.
```

```
Variables e1 e2 e3 e4 : expr.
```

```
Definition c1 : command := cAssert e1.
```

```
Definition c2 : command := cAssign x1 e2.
```

```
Definition c3 : command := cSequence (cAssign x2 e3) (cAssert e3).
```

```
Definition c4 : command := cSequence (cAssign x1 e1) (cAssign x2 e2).
```

```
Definition c5 : command := cSequence (cAssign x1 e3) (cAssign x3 e4).
```

```
Definition c6 : command := cChoice c4 c5.
```

```
Definition c7 : command := cSequence (cAssign x1 e1) (cAssign x1 e2).
```

Definition *targets_c1* := *Empty*.
 Definition *targets_c2* := *singleton x1*.
 Definition *targets_c3* := *singleton x2*.
 Definition *targets_c4* := *union (singleton x1) (singleton x2)*.
 Definition *targets_c5* := *union (singleton x1) (singleton x3)*.
 Definition *targets_c6* := *union (union (singleton x1) (singleton x2)) (union (singleton x1) (singleton x3))*.

Theorem *test_targets_c1* : *Equal (targets c1) empty*.
 Proof.
 clear; simpl; unfold *Equal*; split; intros; trivial.
 Qed.

Theorem *test_targets_c2* : *Equal (targets c2) targets_c2*.
 Proof.
 clear; simpl; unfold *Equal*; split; intros; trivial.
 Qed.

Theorem *test_targets_c3* : *Equal (targets c3) targets_c3*.
 Proof.
 clear; simpl; unfold *Equal*.
 split; intros.
 destruct (*union_1 H*).
 trivial.
 inversion *H0*.
 apply (*union_2 empty H*).
 Qed.

Theorem *test_targets_c4* : *Equal (targets c4) targets_c4*.
 Proof.
 clear; simpl; unfold *Equal*; split; intros; trivial.
 Qed.

Theorem *test_targets_c5* : *Equal (targets c5) targets_c5*.
 Proof.
 clear; simpl; unfold *Equal*; split; intros; trivial.
 Qed.

Theorem *test_targets_c6* : *Equal (targets c6) targets_c6*.
 Proof.
 clear; simpl; unfold *Equal*; split; intros; trivial.
 Qed.

Theorem *test_fset_foldr_1* : $\forall n, \text{fset_foldr nat (fun x y} \Rightarrow 1) \text{ empty } n = n$.
 Proof.
 simpl; trivial.
 Qed.

Theorem *test_fset_foldr_2* : $\forall n m, \text{fset_foldr nat (fun } x y \Rightarrow m) (\text{singleton } x1) n = m$.

Proof.

 simpl; trivial.

Qed.

Theorem *test_fset_foldr_3* : $\forall n, \text{fset_foldr nat (fun } x y \Rightarrow x + y) (\text{singleton } x1) n = S n$.

Proof.

 clear; simpl; trivial.

Qed.

Theorem *test_fset_foldr_4* : $\forall n, \text{fset_foldr nat (fun } x y \Rightarrow x + y) \text{targets_c6 } n = x1 + x2 + x3 + n$.

Proof.

 clear; simpl; trivial.

Qed.

Definition *v1* : *vmap* := fun _ \Rightarrow 0.

Definition *v2* : *vmap* := fun _ \Rightarrow 5.

Definition *v3* : *vmap* := fun *x* \Rightarrow

 match *x* with

 | 1 \Rightarrow 5

 | 2 \Rightarrow 8

 | 3 \Rightarrow 1

 | 4 \Rightarrow 6

 | 5 \Rightarrow 3

 | _ \Rightarrow 0

 end.

Theorem *test_insert_copy_vcmd_1* : $\forall c, \text{insert_copy_vcmd } v1 v1 x1 c = c$.

Proof.

 clear; compute.

 trivial.

Qed.

Theorem *test_insert_copy_vcmd_2* : $\forall c, \text{insert_copy_vcmd } v1 v2 x1 c = \text{vcSequence (copy_vcmd } x1 0 5) c$.

Proof.

 clear; intros; compute; trivial.

Qed.

Theorem *test_insert_copy_vcmd_3* : $\forall c, \text{insert_copy_vcmd } v1 v3 x2 c = \text{vcSequence (copy_vcmd } x2 0 8) c$.

Proof.

 clear; intros; compute; trivial.

Qed.

Theorem *test_insert_copy_vcmd_4* : $\forall c, \text{insert_copy_vcmd } v2 \ v3 \ x4 \ c = \text{vcSequence } (\text{copy_vcmd } x4 \ 5 \ 6) \ c$.

Proof.

clear; intros; compute; trivial.

Qed.

Theorem *test_sync_vcommand_1* : $\forall v \ v', \text{sync_vcommand } \text{empty } v \ v' = \text{vcSkip}$.

Proof.

clear; compute; trivial.

Qed.

Theorem *test_sync_vcommand_2* : $\text{sync_vcommand } (\text{singleton } x1) \ v1 \ v2 = \text{vcSequence } (\text{copy_vcmd } x1 \ 0 \ 5) \ \text{vcSkip}$.

Proof.

clear; compute; trivial.

Qed.

Theorem *test_sync_vcommand_3* : $\text{sync_vcommand } \text{targets_c6 } v1 \ v3 = \text{vcSequence } (\text{copy_vcmd } x1 \ 0 \ 5) \ (\text{vcSequence } (\text{copy_vcmd } x2 \ 0 \ 8) \ (\text{vcSequence } (\text{copy_vcmd } x3 \ 0 \ 1) \ \text{vcSkip}))$.

Proof.

clear; compute; trivial.

Qed.

Theorem *test_sa_transformation_c1_c* : $\text{let } (c, v) := \text{transform_sa } c1 \ v1 \ \text{in } c = \text{vcAssert } (\text{version_expr } e1 \ v1)$.

Proof.

clear; simpl; trivial.

Qed.

Theorem *test_sa_transformation_c1_v* : $\text{let } (c, v) := \text{transform_sa } c1 \ v1 \ \text{in } \text{equivalent_functions } v1 \ v$.

Proof.

clear; simpl; auto.

Qed.

Theorem *test_sa_transformation_c2_c* : $\text{let } (c, v) := \text{transform_sa } c2 \ v1 \ \text{in } c = \text{vcAssign } (x1, 1) \ (\text{version_expr } e2 \ v1)$.

Proof.

clear; compute; trivial.

Qed.

Theorem *test_sa_transformation_c2_v* : $\text{let } (c, v) := \text{transform_sa } c2 \ v1 \ \text{in } \text{equivalent_functions } v \ (\text{fun } x \Rightarrow \text{match } x \ \text{with } 1 \Rightarrow 1 \mid _ \Rightarrow 0 \ \text{end})$.

Proof.

clear; simpl.

unfold *equivalent_functions*.

unfold *inc*.

```

unfold rebind.
intros.
destruct x.
compute.
trivial.
destruct x.
compute.
trivial.
compute.
trivial.

```

Qed.

Theorem *test_sa_transformation_c7_c* : let (*c*, *v*) := *transform_sa c7 v1* in *c* = *vcSequence* (*vcAssign* (*x1*, 1) (*version_expr e1 v1*)) (*vcAssign* (*x1*, 2) (*version_expr e2 (inc v1 x1)*)).

Proof.

```

clear.
simpl.
f_equal.

```

Qed.

Theorem *test_sa_transformation_c7_v* : let (*c*, *v*) := *transform_sa c7 v1* in *equivalent_functions v* (*fun x* => *match x with* 1 => 2 | _ => 0 *end*).

Proof.

```

clear; simpl.
unfold equivalent_functions.
unfold inc.
unfold rebind.
intros.
destruct x.
compute; trivial.
destruct x.
compute; trivial.
compute; trivial.

```

Qed.

Theorem *test_sa_transformation_c6_c* :
 let (*c6'*, *v*) := *transform_sa c6 v1* in
 let *c4'* := *vcSequence* (*vcAssign* (*x1*, 1) (*version_expr e1 v1*)) (*vcAssign* (*x2*, 1) (*version_expr e2 (inc v1 x1)*)) in
 let *d4* := *vcSequence* (*copy_vcmd x3 0 1*) *vcSkip* in
 let *c5'* := *vcSequence* (*vcAssign* (*x1*, 1) (*version_expr e3 v1*)) (*vcAssign* (*x3*, 1) (*version_expr e4 (inc v1 x1)*)) in
 let *d5* := *vcSequence* (*copy_vcmd x2 0 1*) *vcSkip* in
c6' = *vcChoice* (*vcSequence c4' d4*) (*vcSequence c5' d5*).

```

Proof.
  simpl.
  f_equal.
Qed.
End tests.

```

Opaque id.

End Tests.

Lemma *max_x_x* : $\forall x, \text{max } x \ x = x$.

Proof.

```

intros.
destruct (max_dec x x); trivial.

```

Qed.

transform_sa only updates the versions of those identifiers who are targets of c^\wedge . This fact is important as it indicates the *vmaps* differ on a finite number of bindings. If this were not the case, it would be rather difficult to generate synchronization commands. **Theorem** *transform_sa_vmap_delta* : $\forall c \ c' \ v \ v', (c', v') = \text{transform_sa } c \ v \rightarrow \text{vmap_delta } v \ v'$ (*targets c*).

Proof.

```

induction c; intros; simpl in  $\times \vdash \times$ ; try (strip H; auto).
unfold vmap_delta.
unfold delta_id.
unfold inc.
unfold rebind.
intros.
destruct (decidable_eq_id x i).
subst.
right; apply IdSet.singleton_2.
compute; trivial.
left; trivial.
introduce pair (transform_sa c1 v) as c1' v1'.
introduce pair (transform_sa c2 v1') as c2' v2'.
rewrite  $\leftarrow H1$  in H.
rewrite  $\leftarrow H2$  in H.
strip H.
specify IHc1 c1' v v1' H1.
specify IHc2 c2' v1' v2' H2.
apply (vmap_delta_combine _ _ _ _ IHc1 IHc2).
introduce pair (transform_sa c1 v) as c1' v1'.
introduce pair (transform_sa c2 v) as c2' v2'.
rewrite  $\leftarrow H1$  in H; rewrite  $\leftarrow H2$  in H.
strip H.

```

```

specify IHc1 c1' v v1' H1.
specify IHc2 c2' v v2' H2.
unfold join; unfold vmap_delta in × ⊢ ×; unfold delta_id in × ⊢ ×.
intros.
specify IHc1 x; specify IHc2 x.
destruct IHc1; destruct IHc2; try ( solve [ right; solve [ apply IdSet.union_2; trivial
| apply IdSet.union_3; trivial ] ] ).
left; rewrite ← H; rewrite ← H0; rewrite (max_x_x (v x)); trivial.
Qed.

```

If μ and ν are synchronized with respect to v , then we can update x 's binding in both μ and ν so that they are again synchronized, this time with respect to v' , which is equivalent with v except that $v' x = v x + 1$ (i.e. x 's version is incremented by one).

Example:

```

mu   = { x -> 5, y -> 3 }
nu   = { x -> 1, y -> 4 }
mu'  = { x_1 -> 5, y_4 -> 3 }

updated mu   = { x -> 9, y -> 3 }
updated nu   = { x -> 2, y -> 4 }
updated mu'  = { x_1 -> 5, x_2 -> 9, y_4 -> 3 }

```

Lemma *store_sync_vstore_assignment* :

```

∀ mu ν v e x,
  store_sync_vstore mu v ν → store_sync_vstore (update_store mu x (e mu)) (inc v
x) (update_vstore ν (x, S (v x)) (version_expr e v ν)).

```

Proof.

```

unfold store_sync_vstore; unfold inc; unfold version_expr; unfold equivalent_functions;
unfold update_store; unfold update_vstore; unfold rebind; intros.
destruct (decidable_eq_id x0 x).
subst.
destruct (decidable_eq_vid (x, S (v x)) (x, S (v x))).
apply expression_evaluation.
unfold equivalent_functions.
intros.
apply H.
elim n; trivial.
destruct (decidable_eq_vid (x0, v x0) (x, S (v x))).
strip e0.
elim n; trivial.
apply H.

```

Qed.

Theorem *vmap_delta_join_v_join* : $\forall v v1 v2 D1 D2, \text{vmap_delta } v v1 D1 \rightarrow \text{vmap_delta } v v2 D2 \rightarrow \text{vmap_delta } v (\text{join } v1 v2) (\text{IdSet.union } D1 D2)$.

Proof.

```

  unfold vmap_delta; unfold delta_id; unfold join; intros.
  specify H x; specify H0 x.
  destruct H; destruct H0; try ( solve [ right; solve [ apply IdSet.union_2; trivial |
  apply IdSet.union_3; trivial ] | ] ).
  left; rewrite ← H; rewrite ← H0.
  rewrite (max_x_x (v x)); trivial.

```

Qed.

Theorem *vmap_delta_join_v1_join* : $\forall v v1 v2 D1 D2, \text{vmap_delta } v v1 D1 \rightarrow \text{vmap_delta } v v2 D2 \rightarrow \text{vmap_delta } v1 (\text{join } v1 v2) (\text{IdSet.union } D1 D2)$.

Proof.

```

  unfold vmap_delta; unfold delta_id; unfold join; intros.
  specify H x; specify H0 x.
  destruct H; destruct H0; try ( solve [ left; trivial | right; apply IdSet.union_2;
  trivial | right; apply IdSet.union_3; trivial ] | ).
  left; rewrite ← H; rewrite ← H0; rewrite (max_x_x (v x)); trivial.

```

Qed.

Theorem *vmap_delta_join_v2_join* : $\forall v v1 v2 D1 D2, \text{vmap_delta } v v1 D1 \rightarrow \text{vmap_delta } v v2 D2 \rightarrow \text{vmap_delta } v2 (\text{join } v1 v2) (\text{IdSet.union } D1 D2)$.

Proof.

```

  unfold vmap_delta; unfold delta_id; unfold join; intros.
  specify H x; specify H0 x.
  destruct H; destruct H0; try ( solve [ left; symmetry; trivial | right; apply Id-
  Set.union_2; trivial | right; apply IdSet.union_3; trivial ] | ).
  left; rewrite ← H; rewrite ← H0; rewrite (max_x_x (v x)); trivial.

```

Qed.

Given that c' is the single assignment form of c , if c skips, so will c' (assuming the initial stores are synchronized). Also, both executions will end up in synchronized stores. Theorem *sa_transformation_skip* :

$$\forall c \mu \mu' \text{vmu } v,$$

$$\text{let } (c', v') := \text{transform_sa } c v \text{ in}$$

$$\text{multistep } (\text{ip } c \mu) (\text{ip } c\text{Skip } \mu') \rightarrow$$

$$\text{store_sync_vstore } \mu v \text{vmu} \rightarrow$$

$$\exists \text{vmu}', \text{vmultistep } (\text{vip } c' \text{vmu}) (\text{vip } c\text{Skip } \text{vmu}') \wedge$$

$$\text{store_sync_vstore } \mu' v' \text{vmu}'.$$

Proof with subst.

```

  induction c; simpl; intros.

```

```

  inversion H...

```

```

  inversion H1...

```

```

inversion H2...
rename mu' into mu.
∃ vmu.
split.
apply vmultistep_step_to_multi.
constructor.
rewrite ← H6; clear H6.
rewrite (sync_stores mu v vmu e H0).
trivial.
trivial.
inversion H3.
inversion H2...
inversion H3.

inversion H...
inversion H1...
inversion H2...
clear H H1 H2.
rename mu' into mu.
∃ vmu.
split.
apply vmultistep_step_to_multi.
constructor.
rewrite ← H6; clear H6.
rewrite (sync_stores mu v vmu e H0).
trivial.
trivial.
inversion H3.

rename i into x.
inversion H...
inversion H1...
inversion H2...
clear H1 H H2.
∃ (update_vstore vmu (x, S (v x)) ((version_expr e v) vmu)).
split.
apply vmultistep_step_to_multi.
constructor.
apply (store_sync_vstore_assignment mu vmu v e x H0).
inversion H3.

introduce pair (transform_sa c1 v) as c1' v1'.
introduce pair (transform_sa c2 v1') as c2' v2'.
intros.
elim (multistep_seq_skip - - - H); intros.

```

```

destruct H3.
rename mu' into mu"; rename x into mu'.
specify IHc1 mu mu' vmu v.
rewrite ← H0 in IHc1.
elim (IHc1 H3 H2); clear IHc1; intros.
destruct H5.
rename x into vmu'.
specify IHc2 mu' mu" vmu' v1'.
rewrite ← H1 in IHc2.
elim (IHc2 H4 H6); intros; clear IHc2.
destruct H7.
rename x into vmu".
∃ vmu".
split.
step transitivity with (vip (vcSequence vcSkip c2') vmu').
apply vmultistep_lift_seq.
trivial.
step (vip c2' vmu').
constructor.
trivial.
trivial.
∃ vmu; split.
constructor.
inversion H...
trivial.
inversion H1.
introduce pair (transform_sa c1 v) as c1' v1'.
introduce pair (transform_sa c2 v) as c2' v2'.
intros.
state transform_sa_vmap_delta as Hdelta1.
specify Hdelta1 c1 c1' v v1' H0.
state transform_sa_vmap_delta as Hdelta2.
specify Hdelta2 c2 c2' v v2' H1.
inversion H...
inversion H3...

clear IHc2.
specify IHc1 mu mu' vmu v.
rewrite ← H0 in IHc1.
elim (IHc1 H4 H2); intros; clear IHc1.
destruct H5.
rename x into vmu'.
state sync_vcommand_goes_to_skip.

```

```

specify H7 (IdSet.union (targets c1) (targets c2)).
specify H7 v1' (join v1' v2') vmu'.
elim H7; intros; clear H7.
rename x into vmu".
∃ vmu"; split.
step choice left.
step transitivity with
  (vip (vcSequence vcSkip
        (foldr vcommand
              (insert_copy_vcmd v1'
                                (join v1' v2'))
              (IdSet.Raw.union (targets c1)
                              (targets c2))
              vcSkip))
        vmu').
apply vmultistep_lift_seq.
trivial.
step (vip (foldr vcommand
              (insert_copy_vcmd v1'
                                (join v1' v2'))
              (IdSet.Raw.union (targets c1)
                              (targets c2))
              vcSkip)
        vmu').
constructor.
apply H8.
state sync_vcommand_works.
specify H7 (IdSet.union (targets c1) (targets c2)).
specify H7 v1' (join v1' v2').
specify H7 vmu' vmu".
assert (vmap_delta v1'
            (join v1' v2')
            (IdSet.union (targets c1)
                          (targets c2))).
eapply vmap_delta_join_v1_join.
apply Hdelta1.
apply Hdelta2.
specify H7 H9 H8.
apply (combine_vmaps - - - - H6 H7).
clear IHc1.
specify IHc2 mu mu' vmu v.
rewrite ← H1 in IHc2.

```

```

elim (IHc2 H4 H2); intros; clear IHc2.
destruct H5.
rename x into vmu'.
state sync_vcommand_goes_to_skip.
specify H7 (IdSet.union (targets c1) (targets c2)).
specify H7 v2' (join v1' v2') vmu'.
elim H7; intros; clear H7.
rename x into vmu".
∃ vmu"; split.
step choice right.
step transitivity with
  (vip (vcSequence vcSkip
        (foldr vcommand
              (insert_copy_vcnd v2'
                                (join v1' v2'))
              (IdSet.Raw.union (targets c1)
                              (targets c2))
              vcSkip))
        vmu')).
apply vmultistep_lift_seq.
trivial.
step (vip (foldr vcommand
                (insert_copy_vcnd v2'
                                  (join v1' v2'))
                (IdSet.Raw.union (targets c1)
                                  (targets c2))
                vcSkip)
      vmu')).
constructor.
apply H8.
state sync_vcommand_works.
specify H7 (IdSet.union (targets c1) (targets c2)).
specify H7 v2' (join v1' v2').
specify H7 vmu' vmu".
assert (vmap_delta v2'
          (join v1' v2')
          (IdSet.union (targets c1)
                      (targets c2))).
eapply vmap_delta_join_v2_join.
apply Hdelta1.
apply Hdelta2.
specify H7 H9 H8.

```

apply (combine_vmaps - - - - H6 H7).

Qed.

If a command leads to failure, so will its single assignment form, if both are starting in synchronized stores. **Theorem sa_transformation_fail :**

```
∀ c mu mu' vmu v,
  let (c', v') := transform_sa c v in
    multistep (ip c mu) (failure mu') →
    store_sync_vstore mu v vmu →
    ∃ vmu', vmultistep (vip c' vmu) (vfailure vmu').
```

Proof with subst.

induction c; simpl; intros.

∃ vmu.

inversion H...

inversion H1...

inversion H2...

inversion H3.

inversion H2...

rename mu' into mu.

clear H H1 H2.

apply vmultistep_step_to_multi; constructor.

red; intros; elim H6.

rewrite (sync_stores mu v vmu e H0).

trivial.

inversion H3.

inversion H...

inversion H1...

inversion H2...

inversion H3.

inversion H...

inversion H1...

inversion H2...

inversion H3.

introduce pair (transform_sa c1 v) as c1' v'.

introduce pair (transform_sa c2 v') as c2' v''.

intros.

destruct (multistep_seq_fail - - - H); clear H.

clear IHc2.

specify IHc1 mu mu' vmu v.

rewrite ← H0 in IHc1.

elim (IHc1 H3 H2); clear IHc1; intros.

∃ x.

```

apply vmultistep_lift_seq_fail; trivial.
clear IHc1.
elim H3; intros; clear H3.
destruct H.
rename mu' into mu''; rename x into mu'.
state sa_transformation_skip.
specify H4 c1 mu mu' vmu v.
rewrite ← H0 in H4.
elim (H4 H H2); clear H4; intros.
destruct H4.
rename x into vmu'.
specify IHc2 mu' mu'' vmu' v'.
rewrite ← H1 in IHc2.
elim (IHc2 H3 H5); clear IHc2; intros.
rename x into vmu''.
∃ vmu''.
step transitivity with (vip (vcSequence vcSkip c2') vmu').
apply vmultistep_lift_seq.
trivial.
step (vip c2' vmu').
constructor.
trivial.

inversion H...
inversion H1.

introduce pair (transform_sa c1 v) as c1' v1'.
introduce pair (transform_sa c2 v) as c2' v2'.
intros.
destruct (multistep_choice_fail - - - H).

clear IHc2.
specify IHc1 mu mu' vmu v.
rewrite ← H0 in IHc1.
elim (IHc1 H3 H2); clear IHc1; intros.
rename x into vmu'.
∃ vmu'.
step choice left.
apply vmultistep_lift_seq_fail.
trivial.

clear IHc1.
specify IHc2 mu mu' vmu v.
rewrite ← H1 in IHc2.
elim (IHc2 H3 H2); clear IHc2; intros.

```

rename x into vmu' .
 $\exists vmu'$.
 step choice right.
 apply *vmultistep_lift_seq_fail*.
 trivial.

Qed.

Inductive *assigns* ($x : vid$) : *vcommand* \rightarrow Prop :=
 | *assignsAssign* : $\forall e$, *assigns* x (*vcAssign* $x e$)
 | *assignsSequenceL* : $\forall c1 c2$, *assigns* $x c1 \rightarrow$ *assigns* x (*vcSequence* $c1 c2$)
 | *assignsSequenceR* : $\forall c1 c2$, *assigns* $x c2 \rightarrow$ *assigns* x (*vcSequence* $c1 c2$)
 | *assignsChoiceL* : $\forall c1 c2$, *assigns* $x c1 \rightarrow$ *assigns* x (*vcChoice* $c1 c2$)
 | *assignsChoiceR* : $\forall c1 c2$, *assigns* $x c2 \rightarrow$ *assigns* x (*vcChoice* $c1 c2$).

Theorem *assigns_dec* : $\forall c x$, *decidable* (*assigns* $x c$).

Proof with subst.

unfold *decidable*.
 induction c ; intros; try (*right*; red; intros; inversion H ; fail).
 destruct (*decidable_eq_vid* $v x$).
 subst.
 left.
 constructor.
right; red; intros.
 inversion H .
 symmetry in $H1$.
 contradiction.
 destruct ($IHc1 x$);
 destruct ($IHc2 x$);
 try (*left*;
 solve [apply *assignsSequenceL*; trivial
 | apply *assignsSequenceR*; trivial]).
right; red; intros.
 inversion H ; contradiction.
 destruct ($IHc1 x$);
 destruct ($IHc2 x$);
 try (*left*;
 solve [apply *assignsChoiceL*; trivial |
 apply *assignsChoiceR*; trivial]).
right; red; intros.
 inversion H ; contradiction.

Qed.

Inductive *single_assignment_vid* ($x : vid$) : *vcommand* \rightarrow Prop :=
 | *saidAssert* : $\forall e$, *single_assignment_vid* x (*vcAssert* e)
 | *saidAssume* : $\forall e$, *single_assignment_vid* x (*vcAssume* e)

```

| saidAssign :  $\forall y e, \text{single\_assignment\_vid } x \text{ (vcAssign } y e)$ 
| saidSequenceL :  $\forall c1 c2, \neg \text{assigns } x c1 \rightarrow \text{single\_assignment\_vid } x \text{ (vcSequence } c1 c2)$ 
| saidSequenceR :  $\forall c1 c2, \neg \text{assigns } x c2 \rightarrow \text{single\_assignment\_vid } x \text{ (vcSequence } c1 c2)$ 
| saidSkip :  $\text{single\_assignment\_vid } x \text{ vcSkip}$ 
| saidChoice :  $\forall c1 c2, \text{single\_assignment\_vid } x c1 \rightarrow \text{single\_assignment\_vid } x c2 \rightarrow \text{single\_assignment\_vid } x \text{ (vcChoice } c1 c2)$ .

```

Theorem *single_assignment_vid_dec* :

$\forall c x, \text{decidable (single_assignment_vid } x c)$.

Proof.

```

induction c;
  unfold decidable;
  intros;
  try ( solve [ left; constructor ] ).
destruct (assigns_dec c1 x);
destruct (assigns_dec c2 x);
try ( solve [ left; solve [ apply saidSequenceL; trivial
                        | apply saidSequenceR; trivial ] ] ).

right; red; intros.
inversion H; subst; contradiction.
destruct (IHc1 x);
destruct (IHc2 x);
try ( solve [ right; red; intros; inversion H; subst; contradiction ] ).
left; constructor; trivial.

```

Qed.

Inductive *vmap_bound* ($v v' : \text{vmap}$) : $\text{vcommand} \rightarrow \text{Prop} :=$

```

| vbAssert :  $\forall e, \text{vmap\_bound } v v' \text{ (vcAssert } e)$ 
| vbAssume :  $\forall e, \text{vmap\_bound } v v' \text{ (vcAssume } e)$ 
| vbAssign :  $\forall x n e, v x < n \rightarrow n \leq v' x \rightarrow \text{vmap\_bound } v v' \text{ (vcAssign } (x, n) e)$ 
| vbSequence :  $\forall c1 c2, \text{vmap\_bound } v v' c1 \rightarrow \text{vmap\_bound } v v' c2 \rightarrow \text{vmap\_bound } v v' \text{ (vcSequence } c1 c2)$ 
| vbSkip :  $\text{vmap\_bound } v v' \text{ vcSkip}$ 
| vbChoice :  $\forall c1 c2, \text{vmap\_bound } v v' c1 \rightarrow \text{vmap\_bound } v v' c2 \rightarrow \text{vmap\_bound } v v' \text{ (vcChoice } c1 c2)$ .

```

Definition *vmap_le* ($v v' : \text{vmap}$) := $\forall x, v x \leq v' x$.

Lemma *vmap_le_refl* : $\forall v, \text{vmap_le } v v$.

Proof.

```
compute; intros; auto.
```

Qed.

Hint Resolve *vmap_le_refl*.

Theorem *sa_transformation_monotonic_vmap* : $\forall c v, \text{let } (c', v') := \text{transform_sa } c v \text{ in } \text{vmap_le } v v'$.

Proof.

```
induction c; intros; simpl; auto.
unfold vmap_le; unfold inc; unfold rebind.
intros.
destruct (decidable_eq_id x i); subst; auto.
introduce pair (transform_sa c1 v) as c1' v'.
introduce pair (transform_sa c2 v') as c2' v''.
specify IHc1 v; specify IHc2 v'.
rewrite ← H0 in IHc1.
rewrite ← H1 in IHc2.
revert IHc1 IHc2; clear; intros.
unfold vmap_le in × ⊢ ×.
intros; specify IHc1 x; specify IHc2 x.
omega.

introduce pair (transform_sa c1 v) as c1' v1'.
introduce pair (transform_sa c2 v) as c2' v2'.
specify IHc1 v; specify IHc2 v.
rewrite ← H0 in IHc1; rewrite ← H1 in IHc2.
revert IHc1 IHc2; clear; intros.
unfold vmap_le in × ⊢ ×.
intros.
specify IHc1 x; specify IHc2 x.
unfold join.
destruct (eq_nat_dec (v1' x) (v2' x)).
rewrite e.
rewrite max_x_x.
omega.
state (le_max_l (v1' x) (v2' x)).
state (le_max_r (v1' x) (v2' x)).
omega.
```

Qed.

Lemma *vmap_bound_le_upper* : $\forall c v1 v2 v3, vmap_bound v1 v2 c \rightarrow vmap_le v2 v3 \rightarrow vmap_bound v1 v3 c$.

Proof with subst.

```
unfold vmap_le.
induction c; intros; try (solve [ constructor ] ).
rename v0 into e.
destruct v as [ x n ].
inversion H...
constructor.
trivial.
```

```

specify H0 x.
omega.
inversion H...
constructor.
apply (IHc1 - - - H3 H0).
apply (IHc2 - - - H4 H0).
inversion H...
constructor.
apply (IHc1 - - - H3 H0).
apply (IHc2 - - - H4 H0).
Qed.

Lemma vmap_bound_le_lower :  $\forall c v1 v2 v3, vmap\_le v1 v2 \rightarrow vmap\_bound v2 v3 c \rightarrow vmap\_bound v1 v3 c$ .
Proof with subst.
  unfold vmap_le.
  induction c; intros; try ( solve [ constructor ] ).
  inversion H0...
  specify H x.
  constructor; omega.
  inversion H0...
  constructor.
  apply (IHc1 - - - H H3).
  apply (IHc2 - - - H H4).
  inversion H0...
  constructor.
  apply (IHc1 - - - H H3).
  apply (IHc2 - - - H H4).
Qed.

Lemma vmap_le_join_l :  $\forall v1 v2, vmap\_le v1 (join v1 v2)$ .
Proof.
  unfold vmap_le; unfold join; intros.
  destruct (eq_nat_dec (v1 x) (v2 x)).
  rewrite e; rewrite max_x_x; auto.
  apply le_max_l.
Qed.

Lemma vmap_le_join_r :  $\forall v1 v2, vmap\_le v2 (join v1 v2)$ .
Proof.
  unfold vmap_le; unfold join; intros.
  destruct (eq_nat_dec (v1 x) (v2 x)).
  rewrite e.

```

```

rewrite max_x_x; auto.
apply le_max_r.

```

Qed.

Lemma *vmap_le_trans* : $\forall v1\ v2\ v3, \text{vmap_le } v1\ v2 \rightarrow \text{vmap_le } v2\ v3 \rightarrow \text{vmap_le } v1\ v3$.

Proof.

```

unfold vmap_le; intros.
specify H x; specify H0 x; omega.

```

Qed.

Theorem *sync_vcommand_vmap_bound_l* : $\forall D\ v1\ v2, \text{let } \textit{joined} := \text{join } v1\ v2 \text{ in } \text{vmap_bound } v1\ \textit{joined} (\text{sync_vcommand } D\ v1\ \textit{joined})$.

Proof with subst.

```

destruct D.
induction this; intros.

simpl.
constructor.

rename a into x; rename sorted into x_xs; inversion x_xs...
rename H1 into xs.
specify IHthis xs v1 v2.
simpl in IHthis.
simpl.
unfold insert_copy_vcmd at 1.
destruct (decidable_eq_id (v1 x) (joined x)).
trivial.
constructor.
unfold copy_vcmd.
constructor.
subst joined; unfold join in  $\times \vdash \times$ .
revert n; clear; intros.
destruct (max_dec (v1 x) (v2 x)).
rewrite e in n.
elim n; trivial.
state (le_max_l (v1 x) (v2 x)).
rewrite e in  $\times \vdash \times$ ; clear e.
inversion H...
contradiction.
omega.
auto.
trivial.

```

Qed.

Theorem *sync_vcommand_vmap_bound_r* : $\forall D\ v1\ v2, \text{let } \textit{joined} := \text{join } v1\ v2 \text{ in } \text{vmap_bound } v2\ \textit{joined} (\text{sync_vcommand } D\ v2\ \textit{joined})$.

Proof with subst.

```
destruct D.
induction this; intros.

simpl.
constructor.

rename a into x; rename sorted into x-xs; inversion x-xs...
rename H1 into xs.
specify IHthis xs v1 v2.
simpl in IHthis.
simpl.
unfold insert_copy_vcmd at 1.
destruct (decidable_eq_id (v2 x) (joined x)).
trivial.
constructor.
unfold copy_vcmd.
constructor.
subst joined; unfold join in  $\times \vdash \times$ .
revert n; clear; intros.
destruct (max_dec (v1 x) (v2 x)).
state (le_max_r (v1 x) (v2 x)).
rewrite e in  $\times \vdash \times$ ; clear e.
inversion H...
contradiction.
omega.
rewrite e in n.
elim n; trivial.
auto.
trivial.
```

Qed.

Theorem *sa_transformation_vmap_bound* : $\forall c v$, let $(c', v') := \text{transform_sa } c v$ in *vmap_bound* $v v' c'$.

Proof.

```
induction c; intros; try ( solve [ simpl; constructor ] ).

simpl.
constructor.
omega.
unfold inc; unfold rebind.
destruct (decidable_eq_id i i).
auto.
elim n; trivial.

simpl.
```

```

introduce pair (transform_sa c1 v) as c1' v'.
introduce pair (transform_sa c2 v') as c2' v''.
specify IHc1 v; specify IHc2 v'.
rewrite ← H0 in IHc1; rewrite ← H1 in IHc2.
state (sa_transformation_monotonic_vmap c1 v).
state (sa_transformation_monotonic_vmap c2 v').
rewrite ← H0 in H; rewrite ← H1 in H2.
constructor.
apply (vmap_bound_le_upper - - - - IHc1 H2).
apply (vmap_bound_le_lower - - - - H IHc2).

simpl.
introduce pair (transform_sa c1 v) as c1' v1'.
introduce pair (transform_sa c2 v) as c2' v2'.
specify IHc1 v; specify IHc2 v.
rewrite ← H0 in IHc1; rewrite ← H1 in IHc2.
state (sa_transformation_monotonic_vmap c1 v).
state (sa_transformation_monotonic_vmap c2 v).
rewrite ← H0 in H; rewrite ← H1 in H2.
state (vmap_le_join_l v1' v2').
state (vmap_le_join_r v1' v2').
constructor.
constructor.
apply (vmap_bound_le_upper - - - - IHc1 H3).
state vmap_bound_le_lower.
specify H5 (foldr vcommand (insert_copy_vcnd v1' (join v1' v2')) (IdSet.Raw.union (tar-
gets c1) (targets c2))) vcSkip).
specify H5 v v1' (join v1' v2').
specify H5 H.
apply H5.
state sync_vcommand_vmap_bound_l.
specify H6 (IdSet.union (targets c1) (targets c2)).
specify H6 v1' v2'.
simpl in H6.
trivial.

constructor.
apply (vmap_bound_le_upper - - - - IHc2 H4).
state vmap_bound_le_lower.
specify H5 (foldr vcommand (insert_copy_vcnd v2' (join v1' v2')) (IdSet.Raw.union (tar-
gets c1) (targets c2))) vcSkip).
specify H5 v v2' (join v1' v2').
specify H5 H2.
apply H5.

```

```

state sync_vcommand_vmap_bound_r.
specify H6 (IdSet.union (targets c1) (targets c2)).
specify H6 v1' v2'.
simpl in H6.
trivial.

```

Qed.

Theorem *assigns_vmap_bound* : $\forall c x n v v'$, *assigns* $(x, n) c \rightarrow$ *vmap_bound* $v v' c \rightarrow v x < n \wedge n \leq v' x$.

Proof with subst.

```

induction c; intros; try (solve [ inversion H ]).
destruct v.
rename i into y; rename n into xn; rename n0 into yn; rename v0 into e; rename v1 into
v.
inversion H...
inversion H0...
split; trivial.
inversion H0...
inversion H...
apply (IHc1 _ _ _ _ H2 H3).
apply (IHc2 _ _ _ _ H2 H4).
inversion H0...
inversion H...
apply (IHc1 _ _ _ _ H2 H3).
apply (IHc2 _ _ _ _ H2 H4).

```

Qed.

Theorem *sa_transformation_is_single_assignment* : $\forall c v x$, let $(c', -) :=$ *transform_sa* $c v$ in *single_assignment_vid* $x c'$.

Proof with subst.

```

induction c; intros; try ( solve [ simpl; constructor ] ).
simpl.
introduce pair (transform_sa c1 v) as c1' v'.
introduce pair (transform_sa c2 v') as c2' v''.
destruct (assigns_dec c1' x).
apply saidSequenceR.
red; intros.
state (sa_transformation_vmap_bound c1 v).
state (sa_transformation_vmap_bound c2 v').
rewrite ← H0 in H2; rewrite ← H1 in H3.
destruct x.
state (assigns_vmap_bound _ _ _ _ a H2).
state (assigns_vmap_bound _ _ _ _ H H3).

```

```

omega.
apply saidSequenceL.
trivial.

simpl.
introduce pair (transform_sa c1 v) as c1' v1'.
introduce pair (transform_sa c2 v) as c2' v2'.
specify IHc1 v x.
specify IHc2 v x.
rewrite ← H0 in IHc1; rewrite ← H1 in IHc2.
state (sa_transformation_vmap_bound c1 v).
state (sa_transformation_vmap_bound c2 v).
rewrite ← H0 in H; rewrite ← H1 in H2.
state (sync_vcommand_vmap_bound_l (IdSet.union (targets c1) (targets c2)) v1' v2').
state (sync_vcommand_vmap_bound_r (IdSet.union (targets c1) (targets c2)) v1' v2').
simpl in H3; simpl in H4.
constructor.
destruct (assigns_dec c1' x).
apply saidSequenceR.
red; intros.
destruct x.
state (assigns_vmap_bound - - - - a H).
state (assigns_vmap_bound - - - - H5 H3).
omega.
apply saidSequenceL.
trivial.

destruct (assigns_dec c2' x).
apply saidSequenceR.
red; intros.
destruct x.
state (assigns_vmap_bound - - - - a H2).
state (assigns_vmap_bound - - - - H5 H4).
omega.
apply saidSequenceL.
trivial.
Qed.

```

1.3.3 Passification

Definition *assume_from_assign* $x e :=$
 $pcAssume$ (fun $vmu \Rightarrow$ if $decidable_eq_value$ ($vmu x$) ($e vmu$)
then T
else F).

Fixpoint *passify* (*c* : *vcommand*) : *pcommand* :=

```
match c with
| vcAssert e ⇒ pcAssert e
| vcAssume e ⇒ pcAssume e
| vcSkip ⇒ pcSkip
| vcSequence c1 c2 ⇒ pcSequence (passify c1) (passify c2)
| vcChoice c1 c2 ⇒ pcChoice (passify c1) (passify c2)
| vcAssign x e ⇒ assume_from_assign x e
end.
```

Definition *stores_veq* (*vmu* : *vstore*) (*v* : *vmap*) (*vmu'* : *vstore*) := $\forall x\ n, n \leq v\ x \rightarrow vmu\ (x, n) = vmu'\ (x, n)$.

Theorem *vepr_stores_veq* : $\forall e\ v\ vmu\ vmu', stores_veq\ vmu\ v\ vmu' \rightarrow \text{let } ve := \text{version_expr } e\ v \text{ in } ve\ vmu = ve\ vmu'$.

Proof.

```
intros.
subst ve.
unfold version_expr.
unfold stores_veq in H.
apply expression_evaluation.
unfold equivalent_functions.
intros.
apply H.
auto.
```

Qed.

Theorem *stores_veq_refl* : $\forall vmu\ v, stores_veq\ vmu\ v\ vmu$.

Proof.

```
unfold stores_veq; intros; trivial.
```

Qed.

Hint Resolve *stores_veq_refl*.

Theorem *stores_veq_symm* : $\forall vmu\ vmu'\ v, stores_veq\ vmu\ v\ vmu' \rightarrow stores_veq\ vmu'\ v\ vmu$.

Proof.

```
unfold stores_veq; intros.
symmetry; apply H; trivial.
```

Qed.

Hint Resolve *stores_veq_symm*.

Theorem *stores_veq_trans* : $\forall vmu\ vmu'\ vmu''\ v, stores_veq\ vmu\ v\ vmu' \rightarrow stores_veq\ vmu'\ v\ vmu'' \rightarrow stores_veq\ vmu\ v\ vmu''$.

Proof.

```
unfold stores_veq; intros.
rewrite (H - - H1).
```

```

rewrite (H0 _ _ H1).
trivial.
Qed.

Theorem stores_veq_vmap_le :  $\forall$  vmu vmu' v v', stores_veq vmu v vmu'  $\rightarrow$  vmap_le v' v  $\rightarrow$ 
stores_veq vmu v' vmu'.
Proof.
  unfold stores_veq; intros.
  apply H.
  unfold vmap_le in H0.
  specify H0 x.
  omega.
Qed.

Lemma stores_veq_sync_vcommand :
 $\forall$  D v1 v2 vmu vmu',
  vmap_le v1 v2  $\rightarrow$  vmultistep (vip (sync_vcommand D v1 v2) vmu) (vip vcSkip vmu')
 $\rightarrow$  stores_veq vmu v1 vmu'.
Proof with subst.
  destruct D.
  induction this; unfold stores_veq; intros.

  simpl in H0.
  inversion H0...
  trivial.
  inversion H2.

  rename x into y; rename a into x; rename sorted into x_xs; inversion x_xs...
  rename H4 into xs.
  specify IHthis xs v1 v2.
  simpl in H0.
  unfold insert_copy_vcmd in H0.
  destruct (decidable_eq_id (v1 x) (v2 x)).
  specify IHthis vmu vmu' H H0.
  unfold stores_veq in IHthis.
  specify IHthis y n H1.
  trivial.
  inversion H0...
  inversion H2...
  inversion H9...
  clear H0 H2 H9.
  inversion H3...
  inversion H0...
  inversion H9.
  clear H3 H0.

```

```

specify IHthis (update_vstore vmu (x, v2 x) (vmu (x, v1 x))) vmu' H H2.
clear H2.
unfold stores_veq in IHthis.
specify IHthis y n H1.
unfold update_vstore in IHthis.
unfold rebind in IHthis.
destruct (decidable_eq_vid (y, n) (x, v2 x)).
strip e.
unfold vmap_le in H.
specify H x.
assert False.
assert (v1 x = v2 x).
omega.
contradiction.
contradiction.
trivial.
inversion H2...
inversion H4.
inversion H3...
inversion H4.
Qed.

Theorem single_assignment_monotonic_store :
  ∀ c v vmu vmu',
    let (c', v') := transform_sa c v in vmultistep (vip c' vmu) (vip vcSkip vmu') →
stores_veq vmu v vmu'.
Proof with subst.
  induction c; simpl; intros.
  inversion H...
  inversion H0...
  inversion H1...
  apply stores_veq_refl.
  inversion H2.
  inversion H1...
  inversion H2.

  inversion H...
  inversion H0...
  inversion H1...
  apply stores_veq_refl.
  inversion H2.

  inversion H...
  inversion H0...

```

```

inversion H1...
clear.
unfold stores_veq; intros.
unfold update_vstore.
unfold rebind.
destruct (decidable_eq_vid (x, n) (i, S (v i))).
strip e0.
assert False.
omega.
contradiction.
trivial.
inversion H2.

introduce pair (transform_sa c1 v) as c1' v'.
introduce pair (transform_sa c2 v') as c2' v".
intros.
elim (vmultistep_seq_skip - - - H); intros.
rename vmu' into vmu"; rename x into vmu'.
destruct H2.
specify IHc1 v.
rewrite ← H0 in IHc1.
specify IHc1 vmu vmu' H2.
specify IHc2 v' vmu' vmu".
rewrite ← H1 in IHc2.
specify IHc2 H3.

assert (stores_veq vmu' v vmu").
state (sa_transformation_monotonic_vmap c1 v).
rewrite ← H0 in H4.
apply (stores_veq_vmap_le - - - IHc2 H4).
apply (stores_veq_trans - - - IHc1 H4).

inversion H...
auto.
inversion H0.

introduce pair (transform_sa c1 v) as c1' v1'.
introduce pair (transform_sa c2 v) as c2' v2'.
intros.
destruct (vmultistep_choice_skip - - - H); clear H.
clear IHc2.
elim (vmultistep_seq_skip - - - H2); intros; clear H2.
destruct H.
rename vmu' into vmu"; rename x into vmu'.
specify IHc1 v vmu vmu'.

```

```

rewrite ← H0 in IHc1.
specify IHc1 H.
state (vmap_le_join_l v1' v2').
state (stores_veq_sync_vcommand (IdSet.union (targets c1) (targets c2))) - - - H3 H2).
state (sa_transformation_monotonic_vmap c1 v).
rewrite ← H0 in H5.
state (stores_veq_vmap_le - - - H4 H5).
apply (stores_veq_trans - - - IHc1 H6).

clear IHc1.
elim (vmultistep_seq_skip - - - H2); intros; clear H2.
destruct H.
rename vmu' into vmu"; rename x into vmu'.
specify IHc2 v vmu vmu'.
rewrite ← H1 in IHc2.
specify IHc2 H.
state (vmap_le_join_r v1' v2').
state (stores_veq_sync_vcommand (IdSet.union (targets c1) (targets c2))) - - - H3 H2).
state (sa_transformation_monotonic_vmap c2 v).
rewrite ← H1 in H5.
state (stores_veq_vmap_le - - - H4 H5).
apply (stores_veq_trans - - - IHc2 H6).
Qed.

```

Theorem *sync_vcommand_does_not_fail* :

```

∀ D v v' vmu vmu',
  ¬ vmultistep (vip (sync_vcommand D v v') vmu) (vfailure vmu').

```

Proof with subst.

```

destruct D.
induction this; red; intros.

simpl in H.
inversion H...
inversion H0.

rename a into x; rename sorted into sorted_xs; inversion sorted_xs...
rename H2 into sorted_xs.
specify IHthis sorted_xs v v'.
simpl in H.
unfold insert_copy_vcnd in H.
destruct (decidable_eq_id (v x) (v' x)).
specify IHthis vmu vmu'.
elim IHthis; clear IHthis.
apply H.
inversion H...

```

```

inversion H0...
inversion H7...
clear H H0 H7.
inversion H1...
inversion H...
inversion H7.
clear H1 H.
specify IHthis (update_vstore vmu (x, v' x) (vmu (x, v x))) vmu'.
elim IHthis; clear IHthis.
apply H0.
inversion H7.
inversion H7.

```

Qed.

Theorem *single_assignment_monotonic_store_fail* :

```

  ∀ c v vmu vmu',
    let (c', v') := transform_sa c v in vmultistep (vip c' vmu) (vfailure vmu') → stores_veq
    vmu v vmu'.

```

Proof with subst.

```

induction c; simpl; intros.

inversion H...
inversion H0...
inversion H1...
inversion H2.

inversion H1...
rename vmu' into vmu.
auto.

inversion H2.

inversion H...
inversion H0...
inversion H1...
inversion H2.

inversion H...
inversion H0...
inversion H1...
inversion H2.

introduce pair (transform_sa c1 v) as c1' v'.
introduce pair (transform_sa c2 v') as c2' v''.
intros.
destruct (vmultistep_seq_fail - - - H).
clear IHc2.

```

```

specify IHc1 v vmu vmu'.
rewrite ← H0 in IHc1.
apply (IHc1 H2).

clear IHc1.
elim H2; clear H2; intros.
destruct H2.
rename vmu' into vmu"; rename x into vmu'.
state single_assignment_monotonic_store.
specify H4 c1 v vmu vmu'.
rewrite ← H0 in H4.
specify H4 H2.
specify IHc2 v' vmu' vmu".
rewrite ← H1 in IHc2.
specify IHc2 H3.
state (sa_transformation_monotonic_vmap c1 v).
rewrite ← H0 in H5.
state (stores_veq_vmap_le vmu' vmu" v' v IHc2 H5).
apply (stores_veq_trans - - - H4 H6).

inversion H...
inversion H0.

introduce pair (transform_sa c1 v) as c1' v1'.
introduce pair (transform_sa c2 v) as c2' v2'.
intros.
destruct (vmultistep_choice_fail - - - H); clear H.

clear IHc2.
destruct (vmultistep_seq_fail - - - H2).
specify IHc1 v vmu vmu'.
rewrite ← H0 in IHc1.
apply (IHc1 H).

elim H; clear H; intros.
destruct H.
state (sync_vcommand_does_not_fail (IdSet.union (targets c1) (targets c2)) v1' (join v1'
v2') x vmu').
contradiction.

clear IHc1.
destruct (vmultistep_seq_fail - - - H2).
specify IHc2 v vmu vmu'.
rewrite ← H1 in IHc2.
apply (IHc2 H).

elim H; clear H; intros.
destruct H.

```

state (sync_vcommand_does_not_fail (IdSet.union (targets c1) (targets c2)) v2' (join v1' v2') x vmu').

contradiction.

Qed.

Lemma *sorted_list_x_lt_elts* :

$\forall (x : id) (xs : list id) (sorted_x_xs : sort (\text{fun } x \ y : id \Rightarrow x < y) (x :: xs)) (y : id),$
 $InA (\text{fun } x \ y : id \Rightarrow x = y) y \ xs \rightarrow x < y.$

Proof with *subst.*

intros.

induction H.

subst.

inversion sorted_x_xs...

inversion H2...

trivial.

rename l into xs.

rename y0 into a.

rename sorted_x_xs into sorted_x_a_xs.

inversion sorted_x_a_xs...

rename H2 into sorted_a_xs.

inversion sorted_a_xs...

rename H2 into sorted_xs.

apply IHInA.

constructor.

trivial.

destruct xs.

constructor.

constructor.

assert (x < a < i).

split.

inversion H3...

trivial.

inversion H4...

trivial.

omega.

Qed.

Lemma *sorted_list_unique_elements* :

\forall

$(x : id)$

$(xs : list id)$

$(sorted_x_xs : sort (\text{fun } x \ y : id \Rightarrow x < y) (x :: xs))$

$(H : InA (\text{fun } x \ y : id \Rightarrow x = y) x \ xs), False.$

Proof with subst.

```
intros.  
state (sorted_list_x_lt_elts x xs sorted_x_xs - H).  
omega.
```

Qed.

Theorem *vmultistep_pmultistep_sync_vcommand* :

```
∀ D v v' vmu vmu' vmu'',  
  let c := sync_vcommand D v v' in  
    vmap_le v v' →  
    vmultistep (vip c vmu) (vip vcSkip vmu') →  
    stores_veq vmu' v' vmu'' →  
    pmultistep vmu'' (pip (passify c)) (pip pcSkip).
```

Proof with subst.

```
destruct D.  
induction this.  
  
intros; simpl; constructor.  
  
intros v v' vmu vmu' vmu''; simpl; intro Hle; intros.  
rename a into x; rename sorted into x_xs; inversion x_xs...  
rename H3 into xs.  
specify IHthis xs v v'.  
simpl in H.  
  
unfold insert_copy_vcmd in H.  
unfold sync_vcommand.  
unfold fset_foldr.  
simpl.  
unfold insert_copy_vcmd.  
destruct (decidable_eq_id (v x) (v' x)).  
specify IHthis vmu vmu' vmu''.  
simpl in IHthis.  
specify IHthis Hle H H0.  
apply IHthis.  
  
inversion H...  
inversion H1...  
inversion H8...  
clear H H1 H8.  
inversion H2...  
inversion H...  
inversion H8.  
clear H2 H.  
  
specify IHthis (update_vstore vmu (x, v' x) (vmu (x, v x))) vmu' vmu''.  
simpl in IHthis.
```

```

specify IHthis Hle H1 H0.
simpl.
step (pip (pcSequence pcSkip
          (passify (foldr vcommand
                    (fun (x0 : id)
                      (c : vcommand) => if decidable_eq_id (v
x0)
                                                    (v'
x0)
                                                    then c
                                                    else vcSequence (copy_vcmd
x0 (v x0)
                                                    c)
x0))
          this
          vcSkip))))).

constructor.
unfold assume_from_assign.
constructor.
cut (vmu'' (x, v' x) = vmu'' (x, v x)); intros.
revert H; clear; intros.
compute in × ⊢ ×.
destruct (decidable_eq_value (vmu'' (x, v' x)) (vmu'' (x, v x))).
trivial.
contradiction.

rename vmu'' into vmu'''; rename vmu' into vmu''.
introduce_eq vmu' (update_vstore vmu (x, v' x) (vmu (x, v x))).
assert (vmultistep (vip (foldr vcommand
                        (fun (x : id)
                          (c : vcommand) => if decidable_eq_id (v x
x)
                                                    (v' x)
                                                    then c
                                                    else vcSequence (copy_vcmd x
x)
                                                    (v'
x))
                        this
                        vcSkip))))).
vmu')

```

```

      (vip vcSkip vmu'')).
rewrite H.
trivial.
clear H1.
assert (vmu' (x, v' x) = vmu (x, v x)).
rewrite H.
unfold update_vstore.
unfold rebind.
destruct (decidable_eq_vid (x, v' x) (x, v x)).
trivial.
elim n0; trivial.
assert (vmu' (x, v x) = vmu (x, v x)).
rewrite H.
unfold update_vstore.
unfold rebind.
destruct (decidable_eq_vid (x, v x) (x, v' x)).
strip e; contradiction.
trivial.
cut (¬ IdSet.In x (IdSet.Build_slist xs)).
intros.
state (sync_vcommand_preservation (IdSet.Build_slist xs) x v v' vmu' vmu'' (v x) H5
H2).
state (sync_vcommand_preservation (IdSet.Build_slist xs) x v v' vmu' vmu'' (v' x) H5
H2).
revert Hle n H0 H4 H H1 H3 H6 H7; clear; intros.
unfold stores_veq in H0.
state (H0 x (v' x) (le_n _)).
assert (v x ≤ v' x).
unfold vmap_le in Hle.
apply Hle.
specify H0 x (v x) H5; clear H5.
clear H Hle.
compute in × ⊢ ×.
rewrite ← H2; clear H2.
rewrite ← H0; clear H0.
rewrite ← H6; clear H6.
rewrite ← H7; clear H7.
rewrite H1.
rewrite H3.
trivial.
revert x_xs xs H4; clear; intros.

```

```

red; intros.
unfold IdSet.In in × ⊢.
unfold IdSet.E.eq in × ⊢.
unfold IdSet.E.lt in × ⊢.
unfold IdSet.Raw.elt in × ⊢.
unfold Identifier_OT.lt in × ⊢.
unfold Identifier_OT.eq in × ⊢.
unfold Identifier_OT.t in × ⊢.
simpl in H.
apply (sorted_list_unique_elements x this x_xs H).
step (pip (passify (foldr vcommand
                (fun (x0 : id)
                  (c : vcommand) ⇒ if decidable_eq_id (v x0) (v' x0)
                                     then c
                                     else vcSequence (copy_vcmd x0
                                                         (v x0)
                                                         (v'
x0))
                c)
                this
                vcSkip))).

constructor.
apply IHthis.
inversion H8.
inversion H2...
inversion H3.
Qed.

Theorem vmultistep_pmultistep_skip : ∀ c v vmu vmu' vmu'',
  let (c', v') := transform_sa c v in
    vmultistep (vip c' vmu) (vip vcSkip vmu') →
    stores_veq vmu' v' vmu'' →
    pmultistep vmu'' (pip (passify c')) (pip pcSkip).

Proof with subst.
  induction c; simpl; intros.

  inversion H...
  inversion H1...
  inversion H2...
  apply pmultistep_step_to_multi.
  constructor.
  state vexpr_stores_veq.
  specify H3 e v vmu' vmu'' H0.
  simpl in H3.

```

```

rewrite  $H6$  in  $H3$ .
symmetry in  $H3$ ; trivial.
inversion  $H3$ .
inversion  $H2\dots$ .
inversion  $H3$ .

inversion  $H\dots$ .
inversion  $H1\dots$ .
inversion  $H2\dots$ .
apply pmultistep_step_to_multi.
constructor.
state (vexpr_stores_veq  $e v vmu' vmu'' H0$ ).
simpl in  $H3$ ; rewrite  $H6$  in  $H3$ ; symmetry in  $H3$ ; trivial.
inversion  $H3$ .

inversion  $H\dots$ .
inversion  $H1\dots$ .
inversion  $H2\dots$ .
clear  $H1 H H2$ .
apply pmultistep_step_to_multi.
unfold assume_from_assign.
constructor.

assert ( $vmu'' (i, S (v i)) = version\_expr e v vmu''$ ).
unfold version_expr.
unfold stores_veq in  $H0$ .
state  $H0$ .
specify  $H0 i (S (v i))$ .
assert ( $S (v i) \leq inc v i i$ ).
unfold inc.
unfold rebind.
destruct (decidable_eq_id  $i i$ ).
constructor.
elim  $n$ ; trivial.
specify  $H0 H1$ ; clear  $H1$ .
rewrite  $\leftarrow H0$ .
clear  $H0$ .
unfold update_vstore.
unfold rebind.
destruct (decidable_eq_vid ( $i, S (v i)$ ) ( $i, S (v i)$ )).
unfold version_expr.
apply expression_evaluation.
unfold equivalent_functions.
intros.
specify  $H x (v x)$ .

```

```

assert (v x ≤ inc v i x).
unfold inc.
unfold rebind.
destruct (decidable_eq_id x i).
subst.
omega.
omega.
specify H H0.
rewrite ← H.
unfold update_vstore.
unfold rebind.
destruct (decidable_eq_vid (x, v x) (i, S (v i))).
strip e1.
assert False.
omega.
contradiction.
trivial.
elim n; trivial.
rewrite ← H.
destruct (decidable_eq_value (vmu" (i, S (v i))) (vmu" (i, S (v i)))).
trivial.
elim n; trivial.
inversion H3.

introduce pair (transform_sa c1 v) as c1' v'.
introduce pair (transform_sa c2 v') as c2' v".
intros.
elim (vmultistep_seq_skip - - - H); intros.
clear H; destruct H3.
rename vmu" into vmu"";
  rename vmu' into vmu";
  rename x into vmu'.

state (single_assignment_monotonic_store c1 v vmu vmu').
rewrite ← H0 in H4.
specify H4 H.
state (single_assignment_monotonic_store c2 v' vmu' vmu").
rewrite ← H1 in H5.
specify H5 H3.
state (sa_transformation_monotonic_vmap c1 v) as Hle1.
state (sa_transformation_monotonic_vmap c2 v') as Hle2.
rewrite ← H0 in Hle1.
rewrite ← H1 in Hle2.

simpl.

```

```

step transitivity with
  (pip (pcSequence pcSkip (passify c2'))).
apply (pmultistep_lift_seq (passify c1')
      pcSkip
      (passify c2')
      vmu''').

clear IHc2.

specify IHc1 v; rewrite ← H0 in IHc1.
specify IHc1 vmu vmu' vmu'' H.
apply IHc1.
clear IHc1.
state (stores_veq_vmap_le _ _ _ H2 Hle2).
apply (stores_veq_trans _ _ _ H5 H6).

step (pip (passify c2')).
constructor.
clear IHc1.
specify IHc2 v' vmu' vmu'' vmu'''.
rewrite ← H1 in IHc2.
specify IHc2 H3.
apply IHc2.
clear IHc2.
trivial.

constructor.

introduce pair (transform_sa c1 v) as c1' v1'.
introduce pair (transform_sa c2 v) as c2' v2'.
simpl.
intros.
inversion H...
inversion H3...

clear H3.
step (pip (pcSequence (passify c1')
                    (passify (foldr vcommand
                                (insert_copy_vcmd v1'
                                                    (join v1' v2'))
                                (IdSet.Raw.union (targets c1)
                                                  (targets c2))
                                vcSkip))))).

constructor.
clear IHc2 H.
elim (vmultistep_seq_skip _ _ _ H4); intros; clear H4.
destruct H;

```

```

    rename vmu'' into vmu''';
    rename vmu' into vmu'';
    rename x into vmu'.
step transitivity with
  (pip (pcSequence pcSkip
        (passify (foldr vcommand
                  (insert_copy_vcmd v1'
                                     (join v1' v2'))
                  (IdSet.Raw.union (targets c1)
                                   (targets c2))
                  vcSkip))))).
apply (pmultistep_lift_seq (passify c1')
      pcSkip
      (passify (foldr vcommand
                    (insert_copy_vcmd v1'
                                       (join v1' v2'))
                    (IdSet.Raw.union (targets c1)
                                      (targets c2))
                    vcSkip))
      vmu'').
specify IHc1 v; rewrite ← H0 in IHc1.
specify IHc1 vmu vmu' vmu'' H.
assert (stores_veq vmu' v1' vmu'').
clear IHc1.
state stores_veq_sync_vcommand as H5.
specify H5 (IdSet.union (targets c1)
                  (targets c2))
          v1'
          (join v1' v2')
          vmu'
          vmu''.
specify H5 (vmap_le_join_l v1' v2') H3.
state (stores_veq_vmap_le vmu''
          vmu''
          (join v1' v2')
          v1'
          H2
          (vmap_le_join_l v1' v2')).
apply (stores_veq_trans - - - H5 H4).
specify IHc1 H4; clear H4.
trivial.
step (pip (passify (foldr vcommand

```

```

      (insert_copy_vcmd v1'
        (join v1' v2'))
      (IdSet.Raw.union (targets c1)
        (targets c2))
      vcSkip))).

constructor.
clear IHc1.
state vmultistep_pmultistep_sync_vcommand.
specify H4 (IdSet.union (targets c1) (targets c2)).
specify H4 v1' (join v1' v2').
specify H4 vmu' vmu'' vmu'''.
simpl in H4.
specify H4 (vmap_le_join_l v1' v2').
specify H4 H3.
specify H4 H2.
apply H4.

clear H3.
step (pip (pcSequence (passify c2')
  (passify (foldr vcommand
    (insert_copy_vcmd v2'
      (join v1' v2'))
    (IdSet.Raw.union (targets c1)
      (targets c2))
    vcSkip)))).

constructor.
clear IHc1 H.
elim (vmultistep_seq_skip - - - H4); intros; clear H4.
destruct H;
  rename vmu'' into vmu''';
  rename vmu' into vmu'';
  rename x into vmu'.
step transitivity with
  (pip (pcSequence pcSkip
    (passify (foldr vcommand
      (insert_copy_vcmd v2'
        (join v1' v2'))
      (IdSet.Raw.union (targets c1)
        (targets c2))
      vcSkip)))).

apply (pmultistep_lift_seq (passify c2')
  pcSkip
  (passify (foldr vcommand

```

```

                                                                    (insert_copy_vcmd v2'
                                                                    (join v1' v2'))
(IdSet.Raw.union (targets c1)
                 (targets c2))
vcSkip))

                                                                    vmu'').
specify IHc2 v; rewrite ← H1 in IHc2.
specify IHc2 vmu vmu' vmu'' H.
assert (stores_veq vmu' v2' vmu'').
clear IHc2.
state stores_veq_sync_vcommand as H5.
specify H5 (IdSet.union (targets c1)
                      (targets c2))
          v2'
          (join v1' v2')
          vmu'
          vmu''.
specify H5 (vmap_le_join_r v1' v2') H3.
state (stores_veq_vmap_le vmu''
      vmu''
      (join v1' v2')
      v2'
      H2
      (vmap_le_join_r v1' v2')).
apply (stores_veq_trans - - - H5 H4).
specify IHc2 H4; clear H4.
trivial.
step (pip (passify (foldr vcommand
                      (insert_copy_vcmd v2'
                      (join v1' v2'))
                      (IdSet.Raw.union (targets c1)
                      (targets c2))
                      vcSkip))).

constructor.
clear IHc2.
state vmultistep_pmultistep_sync_vcommand.
specify H4 (IdSet.union (targets c1) (targets c2)).
specify H4 v2' (join v1' v2').
specify H4 vmu' vmu'' vmu''.
simpl in H4.
specify H4 (vmap_le_join_r v1' v2').
specify H4 H3.

```

specify H4 H2.

apply H4.

Qed.

Main theorem regarding passification: if the original program in its SA-form fails, so does its passification. **Theorem** *vmultistep_pmultistep_fail* : $\forall c v vmu vmu'$,

```
let (c', v') := transform_sa c v in
  vmultistep (vip c' vmu) (vfailure vmu') →
  pmultistep vmu' (pip (passify c')) pfailure.
```

Proof with subst.

induction c; simpl; intros.

inversion H...

inversion H0...

inversion H1...

inversion H2.

inversion H1...

apply pmultistep_step_to_multi.

constructor.

trivial.

inversion H2.

inversion H...

inversion H0...

inversion H1...

inversion H2.

inversion H...

inversion H0...

inversion H1...

inversion H2.

introduce pair (transform_sa c1 v) as c1' v'.

introduce pair (transform_sa c2 v') as c2' v''.

intros.

simpl.

destruct (vmultistep_seq_fail - - - H).

clear IHc2.

specify IHc1 v vmu vmu'.

rewrite ← H0 in IHc1.

specify IHc1 H2.

apply pmultistep_lift_seq_fail.

trivial.

elim H2; clear H2; intros.

rename vmu' into vmu''; rename x into vmu'.

destruct H2.

```

clear IHc1.
specify IHc2 v' vmu' vmu".
rewrite ← H1 in IHc2.
specify IHc2 H3.

state vmultistep_pmultistep_skip.
specify H4 c1 v vmu vmu' vmu".
rewrite ← H0 in H4.
specify H4 H2.
cut (stores_veq vmu' v' vmu").
intros.
specify H4 H5; clear H5.
step transitivity with
  (pip (pcSequence pcSkip (passify c2'))).
apply pmultistep_lift_seq.
trivial.
step (pip (passify c2')).
constructor.
trivial.
state single_assignment_monotonic_store_fail.
specify H5 c2 v' vmu' vmu".
rewrite ← H1 in H5.
apply (H5 H3).

inversion H...
inversion H0.

introduce pair (transform_sa c1 v) as c1' v1'.
introduce pair (transform_sa c2 v) as c2' v2'.
intros.
destruct (vmultistep_choice_fail _ _ _ H); clear H.

clear IHc2.
simpl.
step choice left.
destruct (vmultistep_seq_fail _ _ _ H2); clear H2.
apply pmultistep_lift_seq_fail.
specify IHc1 v vmu vmu'.
rewrite ← H0 in IHc1.
apply (IHc1 H).
elim H; clear H; intros.
destruct H.
rename vmu' into vmu"; rename x into vmu'.
state (sync_vcommand_does_not_fail (IdSet.union (targets c1)
                                                    (targets c2)))

```

```

v1'
(join v1' v2')
vmu'
vmu").

contradiction.

clear IHc1.
simpl.
step choice right.
destruct (vmultistep_seq_fail - - - H2); clear H2.
apply pmultistep_lift_seq_fail.
specify IHc2 v vmu vmu'.
rewrite ← H1 in IHc2.
apply (IHc2 H).
elim H; clear H; intros.
destruct H.
rename vmu' into vmu"; rename x into vmu'.
state (sync_vcommand_does_not_fail (IdSet.union (targets c1)
                                                    (targets c2))
      v2'
      (join v1' v2')
      vmu'
      vmu").

contradiction.
Qed.

```

1.3.4 Weakest Preconditions Soundness

```

Fixpoint wp (vmu : vstore) (c : pcommand) (Q : Prop) : Prop :=
  match c with
  | pcAssert e ⇒ e vmu = T ∧ Q
  | pcAssume e ⇒ e vmu = T → Q
  | pcChoice c1 c2 ⇒ wp vmu c1 Q ∧ wp vmu c2 Q
  | pcSequence c1 c2 ⇒ wp vmu c1 (wp vmu c2 Q)
  | pcSkip ⇒ Q
  end.

```

We show that if the weakest precondition holds, no single pstep will fail. **Theorem** *pstep_wp_prevents_failure* : $\forall (vmu : vstore) (c : pcommand) (Q : Prop), wp\ vmu\ c\ Q \rightarrow \neg pstep\ vmu\ (pip\ c)\ pfailure$.

Proof with subst.

```

intros.
red; intros.
introduce states vmu' s1 s2 in H0.

```

```

subst vmu'.
revert c Q H H2 H3.
induction H0; intros; subst; try ( solve [ discriminate ] ).
strip H2.
simpl in H0.
destruct H0.
contradiction.

strip H2; clear H3.
simpl in H.
apply (IHpstep c1 (wp vmu c2 Q) H (refl_equal -) (refl_equal -)).
Qed.

```

We prove that the weakest precondition is "preserved" along the pstep relation. **Theorem** *pstep_wp_preservation* : $\forall vmu\ c\ c'\ Q, wp\ vmu\ c\ Q \rightarrow pstep\ vmu\ (pip\ c)\ (pip\ c') \rightarrow wp\ vmu\ c'\ Q$.

Proof.

```

intros.
introduce states vmu' s1 s2 in H0; subst vmu'.
revert c c' Q H H2 H3.
induction H0; intros.

strip H2; strip H3.
simpl in  $\times \vdash \times$ .
destruct H0; trivial.

discriminate.

strip H2; strip H3.
simpl in  $\times \vdash \times$ .
apply (H0 H).

strip H2; strip H3.
simpl in  $\times \vdash \times$ .
apply (IHpstep c1 c1' (wp vmu c2 Q) H (refl_equal -) (refl_equal -)).

strip H2; strip H3.
simpl in  $\times \vdash \times$ .
trivial.

discriminate.

strip H2; strip H3.
simpl in  $\times \vdash \times$ .
destruct H; trivial.

strip H2; strip H3.
simpl in  $\times \vdash \times$ .
destruct H; trivial.

```

Qed.

Theorem *pmultistep_split_last* : $\forall \text{vmu } s1 \ s3, \text{pmultistep } \text{vmu } s1 \ s3 \rightarrow s1 = s3 \vee \exists s2, \text{pmultistep } \text{vmu } s1 \ s2 \wedge \text{pstep } \text{vmu } s2 \ s3.$

Proof.

```
intros.
induction H.
left; trivial.
destruct IHpmultistep.
subst s3.
right;  $\exists s1$ ; split; [ constructor | trivial ].
elim H1; clear H1; intros.
destruct H1.
right.
rename s3 into s4; rename x into s3.
 $\exists s3$ ; split.
step s2; trivial.
trivial.
```

Qed.

Theorem *pmultistep_append* : $\forall \text{vmu } s1 \ s2 \ s3, \text{pmultistep } \text{vmu } s1 \ s2 \rightarrow \text{pstep } \text{vmu } s2 \ s3 \rightarrow \text{pmultistep } \text{vmu } s1 \ s3.$

Proof.

```
intros.
induction H.
apply pmultistep_step_to_multi.
trivial.
rename s3 into s4; rename s0 into s3.
specify IHpmultistep H0.
step s2; trivial.
```

Qed.

Theorem *pmultistep_forward_induction_scheme_aux*

```
(P : pstate  $\rightarrow$  pstate  $\rightarrow$  Prop)
(vmu : vstore)
(Hrefl :  $\forall s, P \ s \ s$ )
(Hstep :  $\forall s1 \ s2 \ s3, \text{pmultistep } \text{vmu } s1 \ s2 \rightarrow \text{pstep } \text{vmu } s2 \ s3 \rightarrow P \ s1 \ s2 \rightarrow P \ s1 \ s3$ )
(s1 s2 s3 : pstate)
(H : P s1 s2)
(H0 : pmultistep vmu s1 s2)
(H1 : pmultistep vmu s2 s3) : P s1 s3.
```

Proof.

```
intros.
induction H1.
trivial.
```

```

rename s3 into s4.
rename s2 into s3.
rename s0 into s2.
specify Hstep s1 s2 s3 H0 H1 H.
state (pmultistep_append vmu s1 s2 s3 H0 H1).
specify IHpmultistep Hstep H3.
trivial.

```

Qed.

Theorem *pmultistep_forward_induction_scheme* :

```

∀
  (P : pstate → pstate → Prop)
  (vmu : vstore)
  (Hrefl : ∀ s, P s s)
  (Hstep : ∀ s1 s2 s3, pmultistep vmu s1 s2 →
            pstep vmu s2 s3 →
            P s1 s2 →
            P s1 s3),
  ∀ s1 s2, pmultistep vmu s1 s2 → P s1 s2.

```

Proof.

```

intros.
apply (pmultistep_forward_induction_scheme_aux
      P vmu Hrefl Hstep s1 s1 s2 (Hrefl s1) (pmultiReflexivity vmu s1) H).

```

Qed.

Theorem *pmultistep_wp_preservation* : \forall vmu c1 c2 Q,

wp vmu c1 Q \rightarrow *pmultistep* vmu (*pip* c1) (*pip* c2) \rightarrow wp vmu c2 Q.

Proof.

```

intros.
introduce states vmu' s1 s2 in H0; subst vmu'.
revert c1 c2 H H2 H3.
apply (pmultistep_forward_induction_scheme (fun s1 s2 => ∀ c1 c2 : pcommand, wp vmu
c1 Q → s1 = pip c1 → s2 = pip c2 → wp vmu c2 Q) vmu).
intros.
subst.
strip H2.
trivial.
intros.
subst.
destruct s3.
rename c2 into c3; rename p into c2.
state (H2 - - H3 (refl_equal _) (refl_equal _)); clear H2.
apply (pstep_wp_preservation - - - H4 H1).
inversion H1.

```

```

    trivial.
Qed.
Theorem pmultistep_wp_prevents_failure :  $\forall$  vmu c Q,
  wp vmu c Q  $\rightarrow$   $\neg$  pmultistep vmu (pip c) pfailure.
Proof.
  red; intros.
  introduce states vmu' s1 s2 in H0; subst vmu'.
  revert Q c H H2 H3.
  refine (pmultistep_forward_induction_scheme
    (fun s1 s2  $\Rightarrow$   $\forall$  (Q : Prop) (c : pcommand),
      wp vmu c Q  $\rightarrow$  s1 = pip c  $\rightarrow$  s2 = pfailure  $\rightarrow$  False)
    vmu _ _ s1 s2 H0).
  clear; intros.
  rewrite H0 in H1; discriminate.
  clear; intros.
  subst.
  clear H1.
  destruct s2.
  rename p into c'.
  state (pmultistep_wp_preservation _ _ _ H2 H).
  apply (pstep_wp_prevents_failure _ _ _ H1 H0).
  inversion H0.
Qed.
Fixpoint wlp (vmu : vstore) (c : pcommand) (Q : Prop) : Prop :=
  match c with
  | pcAssert e  $\Rightarrow$  e vmu = T  $\rightarrow$  Q
  | pcAssume e  $\Rightarrow$  e vmu = T  $\rightarrow$  Q
  | pcChoice c1 c2  $\Rightarrow$  wlp vmu c1 Q  $\wedge$  wlp vmu c2 Q
  | pcSequence c1 c2  $\Rightarrow$  wlp vmu c1 (wlp vmu c2 Q)
  | pcSkip  $\Rightarrow$  Q
  end.
Theorem pstep_wlp_preservation :  $\forall$  vmu c c' Q, wlp vmu c Q  $\rightarrow$  pstep vmu (pip c) (pip
c')  $\rightarrow$  wlp vmu c' Q.
Proof.
  intros.
  introduce states vmu' s1 s2 in H0; subst vmu'.
  revert Q c c' H H2 H3.
  induction H0; intros.
  strip H2; strip H3.
  simpl in  $\times$   $\vdash$   $\times$ .
  apply (H0 H).

```

discriminate.

strip H2; strip H3.

simpl in $\times \vdash \times$.

apply (*H0 H*).

strip H2; strip H3.

simpl in $\times \vdash \times$.

apply (*IHpstep - - H (refl_equal -) (refl_equal -)*).

strip H2; strip H3.

simpl in $\times \vdash \times$.

trivial.

discriminate.

strip H2; strip H3.

simpl in $\times \vdash \times$.

destruct *H*; trivial.

strip H2; strip H3.

simpl in $\times \vdash \times$.

destruct *H*; trivial.

Qed.

Theorem *monotonic_wp* : $\forall vmu\ c\ (Q\ R : Prop), (Q \rightarrow R) \rightarrow wp\ vmu\ c\ Q \rightarrow wp\ vmu\ c\ R$.

Proof.

induction *c*; intros; simpl in $\times \vdash \times$; auto.

destruct *H0*; split; [trivial | auto].

specify *IHc2 Q R H*.

apply (*IHc1 - - IHc2 H0*).

destruct *H0*; split.

apply (*IHc1 - - H H0*).

apply (*IHc2 - - H H1*).

Qed.

Theorem *monotonic_wlp* : $\forall vmu\ c\ (Q\ R : Prop), (Q \rightarrow R) \rightarrow wlp\ vmu\ c\ Q \rightarrow wlp\ vmu\ c\ R$.

Proof.

induction *c*; intros; simpl in $\times \vdash \times$; auto.

specify *IHc2 Q R H*.

apply (*IHc1 - - IHc2 H0*).

destruct *H0*; split.

apply (*IHc1 - - H H0*).

apply (*IHc2 - - H H1*).

Qed.

Theorem *wp_true* : $\forall vmu\ c\ Q, wp\ vmu\ c\ Q \rightarrow wp\ vmu\ c\ True$.

Proof.

```

    intros; apply (monotonic_wp vmu c Q True); trivial.
Qed.
Theorem conjunctive_wp :  $\forall$  vmu c Q R, wp vmu c Q  $\wedge$  wp vmu c R  $\leftrightarrow$  wp vmu c (Q  $\wedge$  R).
Proof.
  induction c; split; intros; simpl in  $\times \vdash \times$ .
  destruct H.
  destruct H; destruct H0.
  split.
  trivial.
  split; trivial.
  destruct H.
  destruct H0.
  split; split; trivial.
  destruct H.
  intros.
  specify H H1; specify H0 H1; split; trivial.
  split; intros H0; destruct (H H0); trivial.
  destruct H.
  destruct (IHc2 Q R) as [H1 _]; clear IHc2.
  destruct (IHc1 (wp vmu c2 Q) (wp vmu c2 R)) as [H2 _]; clear IHc1.
  specify H2 (conj H H0).
  apply (monotonic_wp vmu c1 - - H1 H2).
  destruct (IHc2 Q R) as [- H0]; clear IHc2.
  state (monotonic_wp vmu c1 - - H0 H); clear H H0.
  destruct (IHc1 (wp vmu c2 Q) (wp vmu c2 R)) as [- H0]; clear IHc1.
  apply (H0 H1).
  destruct H; split; trivial.
  destruct H; split; trivial.
  destruct H.
  destruct H; destruct H0.
  split.
  destruct (IHc1 Q R) as [H3 _]; apply (H3 (conj H H0)).
  destruct (IHc2 Q R) as [H3 _]; apply (H3 (conj H1 H2)).
  destruct H.
  destruct (IHc1 Q R) as [- H1].
  destruct (H1 H); clear H1 H.
  destruct (IHc2 Q R) as [- H1].
  destruct (H1 H0); clear H1 H0.
  split; split; trivial.

```

Qed.

Theorem *conjunctive_wlp* : $\forall vmu\ c\ Q\ R, wlp\ vmu\ c\ Q \wedge wlp\ vmu\ c\ R \leftrightarrow wlp\ vmu\ c\ (Q \wedge R)$.

Proof.

```
induction c; split; simpl in  $\times \vdash \times$ ; intros.
destruct H; split; auto.
split; intros H0; specify H H0; destruct H; trivial.
destruct H; split; auto.
split; intros H0; specify H H0; destruct H; trivial.
destruct H.
destruct (IHc2 Q R).
clear H2.
destruct (IHc1 (wlp vmu c2 Q) (wlp vmu c2 R)).
clear H3; specify H2 (conj H H0).
apply (monotonic_wlp vmu c1 (wlp vmu c2 Q  $\wedge$  wlp vmu c2 R) (wlp vmu c2 (Q  $\wedge$  R))).
trivial.
trivial.
destruct (IHc2 Q R).
clear H0.
state (monotonic_wlp vmu c1 - - H1 H).
clear H H1.
destruct (IHc1 (wlp vmu c2 Q) (wlp vmu c2 R)).
clear H.
specify H1 H0.
trivial.
trivial.
trivial.
destruct H.
destruct H.
destruct H0.
split.
destruct (IHc1 Q R).
apply (H3 (conj H H0)).
destruct (IHc2 Q R).
apply (H3 (conj H1 H2)).
destruct H.
destruct (IHc1 Q R).
clear H1; specify H2 H.
destruct (IHc2 Q R).
```

```

clear H1; specify H3 H0.
clear H H0; destruct H2; destruct H3.
split; split; trivial.
Qed.

Theorem Q_impl_wlpQ :  $\forall v\mu c (Q : \text{Prop}), Q \rightarrow \text{wlp } v\mu c Q$ .
Proof.
  induction c; simpl in  $\times \vdash \times$ ; intros.
  trivial.
  trivial.
  specify IHc2 Q H.
  apply (IHc1 (wlp v $\mu$  c2 Q) IHc2).
  trivial.
  specify IHc1 Q H; specify IHc2 Q H; split; trivial.
Qed.

Theorem wlp_true :  $\forall v\mu c, \text{wlp } v\mu c \text{ True}$ .
Proof.
  intros; apply (Q_impl_wlpQ v $\mu$  c True); trivial.
Qed.

Theorem wlp_rewrite :  $\forall v\mu c Q, \text{wlp } v\mu c Q \leftrightarrow \text{wlp } v\mu c \text{ False} \vee Q$ .
Proof.
  split.
  revert Q; induction c; simpl in  $\times \vdash \times$ ; intros.
  destruct (decidable_eq_value (v v $\mu$ ) T).
  right; apply (H e).
  left; intros; contradiction.
  destruct (decidable_eq_value (v v $\mu$ ) T).
  right; apply (H e).
  left; intros; contradiction.
  destruct (IHc1 (wlp v $\mu$  c2 Q) H); clear IHc1.
  left.
  apply (monotonic_wlp v $\mu$  c1 False (wlp v $\mu$  c2 False)).
  intros; contradiction.
  trivial.
  destruct (IHc2 Q H0); clear IHc2.
  left.
  apply Q_impl_wlpQ.
  trivial.
  right; trivial.
  right; trivial.
  destruct H.

```

```
destruct (IHc1 - H); destruct (IHc2 - H0); solve [ left; split; trivial | right; trivial
].
```

```
revert Q.
```

```
induction c; intros.
```

```
destruct H.
```

```
apply (monotonic_wlp vmu (pcAssert v) False Q).
```

```
intros; contradiction.
```

```
trivial.
```

```
apply Q-impl_wlpQ.
```

```
trivial.
```

```
destruct H.
```

```
apply (monotonic_wlp vmu (pcAssume v) False Q).
```

```
intros; contradiction.
```

```
trivial.
```

```
apply Q-impl_wlpQ.
```

```
trivial.
```

```
destruct H.
```

```
apply (monotonic_wlp vmu (pcSequence c1 c2) False Q).
```

```
intros; contradiction.
```

```
trivial.
```

```
apply Q-impl_wlpQ.
```

```
trivial.
```

```
destruct H.
```

```
apply (monotonic_wlp vmu pcSkip False Q).
```

```
intros; contradiction.
```

```
trivial.
```

```
apply Q-impl_wlpQ.
```

```
trivial.
```

```
destruct H.
```

```
apply (monotonic_wlp vmu (pcChoice c1 c2) False Q).
```

```
intros; contradiction.
```

```
trivial.
```

```
apply Q-impl_wlpQ.
```

```
trivial.
```

Qed.

Theorem *wp-impl-wlp* : $\forall vmu c Q, wp\ vmu\ c\ Q \rightarrow wlp\ vmu\ c\ Q$.

Proof.

```
induction c; intros; simpl in  $\times \vdash \times$ .
```

```
intros; destruct H; trivial.
```

```
intros H0; apply (H H0).
```

```

specify IHc2 Q.
specify IHc1 (wp vmu c2 Q) H.
apply (monotonic_wlp vmu c1 - - IHc2 IHc1).
trivial.

```

```

destruct H as [ H H0 ]; specify IHc1 Q H; specify IHc2 Q H0; split; trivial.
Qed.

```

Theorem *wp_rewrite* : $\forall vmu c Q, wp\ vmu\ c\ Q \leftrightarrow wp\ vmu\ c\ True \wedge wlp\ vmu\ c\ Q$.

Proof.

```

split.
intros.
split.
apply (wp_true vmu c Q H).
apply wp_impl_wlp; trivial.
intros.
destruct H.
revert Q H H0; induction c; intros; simpl in  $\times \vdash \times$ .
destruct H.
specify H0 H; split; trivial.
trivial.
state (wp_true vmu c1 (wp vmu c2 True) H).
specify IHc1 (wlp vmu c2 Q) H1 H0; clear H1.
destruct (conjunctive_wp vmu c1 (wp vmu c2 True) (wlp vmu c2 Q)) as [H1 _].
specify H1 (conj H IHc1).
assert (wp vmu c2 True  $\wedge$  wlp vmu c2 Q  $\rightarrow$  wp vmu c2 Q).
intros H2; destruct H2 as [H3 H4]; apply (IHc2 Q H3 H4).
apply (monotonic_wp vmu c1 - - H2 H1).
trivial.
destruct H; destruct H0.
split.
apply (IHc1 - H H0).
apply (IHc2 - H1 H2).

```

Qed.

```

Fixpoint efficient_wlp (vmu : vstore) (c : pcommand) (Q : Prop) : Prop :=
  match c with
  | pcAssert e  $\Rightarrow$  e vmu = T  $\rightarrow$  Q
  | pcAssume e  $\Rightarrow$  e vmu = T  $\rightarrow$  Q
  | pcSequence c1 c2  $\Rightarrow$  efficient_wlp vmu c1 (efficient_wlp vmu c2 Q)
  | pcSkip  $\Rightarrow$  Q
  | pcChoice c1 c2  $\Rightarrow$  (efficient_wlp vmu c1 False  $\wedge$  efficient_wlp vmu c2 False)  $\vee$  Q

```

end.

```
Fixpoint efficient_wp (vmu : vstore) (c : pcommand) (Q : Prop) : Prop :=
  match c with
  | pcAssert e => e vmu = T & Q
  | pcAssume e => e vmu = T -> Q
  | pcSequence c1 c2 => efficient_wp vmu c1 (efficient_wp vmu c2 Q)
  | pcSkip => Q
  | pcChoice c1 c2 => efficient_wp vmu c1 True & efficient_wp vmu c2 True & effi-
  cient_wlp vmu (pcChoice c1 c2) Q
  end.
```

Theorem *efficient_wlp_equivalence* : \forall vmu c Q, *wlp* vmu c Q \leftrightarrow *efficient_wlp* vmu c Q.

Proof.

split.

revert Q; *induction* c; *intros*; *simpl* in $\times \vdash \times$; *trivial*.

specify IHc2 Q.

state (*monotonic_wlp* vmu c1 - - IHc2 H).

apply (IHc1 - H0).

destruct H.

destruct (*wlp_rewrite* vmu c1 Q) as [H1 _].

destruct (*wlp_rewrite* vmu c2 Q) as [H2 _].

specify H1 H; *clear* H.

specify H2 H0; *clear* H0.

specify IHc1 False.

specify IHc2 False.

destruct H1; *destruct* H2; *solve* [*left*; *split*; *auto* | *right*; *trivial*].

revert Q; *induction* c; *intros*; *simpl* in $\times \vdash \times$; *trivial*.

specify IHc1 (*efficient_wlp* vmu c2 Q) H.

specify IHc2 Q.

apply (*monotonic_wlp* vmu c1 - - IHc2 IHc1).

destruct (*wlp_rewrite* vmu c1 Q) as [- H1].

destruct (*wlp_rewrite* vmu c2 Q) as [- H2].

specify IHc1 False.

specify IHc2 False.

destruct H.

destruct H; *split*; *auto*.

split; *auto*.

Qed.

Theorem *efficient_wp_equivalence* : \forall vmu c Q, *wp* vmu c Q \leftrightarrow *efficient_wp* vmu c Q.

Proof.

split.

```

revert Q; induction c; intros; simpl in × ⊢ ×; trivial.
specify IHc2 Q.
state (monotonic_wp vmu c1 - - IHc2 H).
apply (IHc1 - H0).

destruct H.
destruct (wp_rewrite vmu c1 Q) as [H1 _].
destruct (wp_rewrite vmu c2 Q) as [H2 _].
specify H1 H.
specify H2 H0.
destruct H1.
destruct H2.
split.
apply (IHc1 True H1).
split.
apply (IHc2 True H2).
clear H H0 H1 H2.
destruct (wlp_rewrite vmu c1 Q) as [H5 _].
destruct (wlp_rewrite vmu c2 Q) as [H6 _].
specify H5 H3.
specify H6 H4.
destruct (efficient_wlp_equivalence vmu c1 False) as [H7 _].
destruct (efficient_wlp_equivalence vmu c2 False) as [H8 _].
destruct H5; destruct H6; auto.

revert Q; induction c; intros; simpl in × ⊢ ×; trivial.
state (IHc1 - H); clear IHc1.
specify IHc2 Q.
apply (monotonic_wp vmu c1 - - IHc2 H0).

destruct H.
destruct H0.
specify IHc1 True H.
specify IHc2 True H0.
destruct H1.
destruct H1.
destruct (efficient_wlp_equivalence vmu c1 False) as [- H3].
destruct (efficient_wlp_equivalence vmu c2 False) as [- H4].
specify H3 H1.
specify H4 H2.
destruct (wlp_rewrite vmu c1 Q) as [- H5].
destruct (wlp_rewrite vmu c2 Q) as [- H6].
specify H5 (@or-introl - Q H3).
specify H6 (@or-introl - Q H4).

```

```

destruct (wp_rewrite vmu c1 Q) as [- H7].
destruct (wp_rewrite vmu c2 Q) as [- H8].
auto.

destruct (wlp_rewrite vmu c1 Q) as [- H3].
destruct (wlp_rewrite vmu c2 Q) as [- H4].
state (H3 (@or_intror - - H1)).
state (H4 (@or_intror - - H1)).
destruct (wp_rewrite vmu c1 Q) as [- H6].
destruct (wp_rewrite vmu c2 Q) as [- H7].
auto.

```

Qed.

Initial version map, where each identifier's version equals 0. **Definition** *init_vmap* ($x : id$) := O .

Auxiliary definition to perform SA transformation and passification in one step. **Definition** *passified* ($c : command$) :=

```

let (c', _) := transform_sa c init_vmap in
passify c'.

```

Proves that for any store μ and version map v , there is a synchronized versioned store. **Lemma** *versioned_store_exists* : $\forall \mu v, \exists vmu, store_sync_vstore \mu v vmu$.

Proof.

```

intros.
unfold store_sync_vstore.
unfold equivalent_functions.
 $\exists$  (fun vid : vid  $\Rightarrow$  let (x, _) := vid in  $\mu$  x).
intros.
trivial.

```

Qed.

Proves the soundness of the efficient (conservative) weakest preconditions: If the weakest preconditions are true, execution will not encounter failure.

Theorem *soundness_efficient_wp* :

```

 $\forall c,$ 
( $\forall vmu, efficient\_wp \ vmu \ (passified \ c) \ True$ )  $\rightarrow$ 
 $\forall \mu, \neg \exists \mu', multistep \ (ip \ c \ \mu) \ (failure \ \mu')$ .

```

Proof.

```

intros.
unfold passified in H.
introduce pair (transform_sa c init_vmap) as c' v'.
rewrite  $\leftarrow$  H1 in H.
red; intros.
eliminate existential mu' in H0.
state (sa_transformation_fail c mu mu').

```

```

elim (versioned_store_exists mu init_vmap).
intro vmu; intros.
specify H2 vmu init_vmap.
rewrite ← H1 in H2.
specify H2 H0 H3.
eliminate existential vmu' in H2.
state vmultistep_pmultistep_fail.
specify H4 c init_vmap vmu vmu'.
rewrite ← H1 in H4.
specify H4 H2.
refine (pmultistep_wp_prevents_failure
        vmu' (passify c') True _ H4).
destruct (efficient_wp_equivalence
        vmu' (passify c') True) as [- H5];
  apply H5; clear H5.
apply H.
Qed.

```

1.3.5 Weakest Preconditions Size

We can't measure the size of Props, so we define our own.

```

Needs manual checking Inductive formula : Set :=
| fConjunction : formula → formula → formula
| fDisjunction : formula → formula → formula
| fImplication : formula → formula → formula
| fAtom : formula.

```

We define a metric on formulae.

```

Needs manual checking Fixpoint formula_metric (f : formula) : nat :=
match f with
| fConjunction x y ⇒ S (formula_metric x + formula_metric y)
| fDisjunction x y ⇒ S (formula_metric x + formula_metric y)
| fImplication x y ⇒ S (formula_metric x + formula_metric y)
| fAtom ⇒ 1
end.

```

We define weakest liberal preconditions making use formula.

```

Needs manual checking Fixpoint wlp' (c : pcommand) (Q : formula) {struct c} :
formula :=
match c with
| pcAssert _ ⇒ fImplication fAtom Q
| pcAssume _ ⇒ fImplication fAtom Q
| pcSequence c1 c2 ⇒ wlp' c1 (wlp' c2 Q)
| pcSkip ⇒ fAtom

```

```
| pcChoice c1 c2 => fDisjunction (fConjunction (wlp' c1 fAtom)
                                           (wlp' c2 fAtom))
```

Q

end.

We define weakest preconditions making use formula.

Needs manual checking `Fixpoint wp' (c : pcommand) (Q : formula) {struct c} : formula :=`

```
match c with
| pcAssert _ => fConjunction fAtom Q
| pcAssume _ => fImplication fAtom Q
| pcSequence c1 c2 => wp' c1 (wp' c2 Q)
| pcSkip => fAtom
| pcChoice c1 c2 => fConjunction (fConjunction (wp' c1 fAtom)
                                           (wp' c2 fAtom))
                                           (wlp' c Q)
```

end.

Quick lemma showing that formulas are at least 1 big. `Lemma formula_metric_ge_1 : $\forall Q, 1 \leq \text{formula_metric } Q$.`

`Proof.`

`induction Q; intros; simpl; omega.`

`Qed.`

`Hint Resolve formula_metric_ge_1.`

Shows the linearity of the efficient weakest liberal preconditions with respect to the size of passified commands. `Theorem linear_wlp' : $\exists a, \forall c Q,$`
`formula_metric (wlp' c Q) $\leq a \times \text{pcommand_metric } c + \text{formula_metric } Q$.`

`Proof.`

`\exists 4; intros; revert Q; induction c; intros.`

`simpl; omega.`

`simpl; omega.`

`simpl wlp'.`

`specify IHc1 (wlp' c2 Q).`

`refine_le IHc1.`

`clear IHc1.`

`simpl pcommand_metric.`

`specify IHc2 Q.`

`refine left bound with`

```
(4  $\times$  pcommand_metric c1 +
 4  $\times$  pcommand_metric c2 +
 formula_metric Q).
```

`omega.`

```

simpl; auto.
simpl wlp'.
simpl pcommand_metric.
simpl formula_metric.
specify IHc1 fAtom.
refine left bound with
  (S (S (4 × pcommand_metric c1 + formula_metric fAtom +
        formula_metric (wlp' c2 fAtom) +
        formula_metric Q))).
clear IHc1.
specify IHc2 fAtom.
refine left bound with
  (S (S (4 × pcommand_metric c1 + formula_metric fAtom +
        4 × pcommand_metric c2 + formula_metric fAtom +
        formula_metric Q))).
clear IHc2.
algebraically rewrite
  (S (S (4 × pcommand_metric c1 +
        formula_metric fAtom +
        4 × pcommand_metric c2 +
        formula_metric fAtom +
        formula_metric Q)))
as
  (4 × pcommand_metric c1 + formula_metric fAtom +
  4 × pcommand_metric c2 + formula_metric fAtom +
  formula_metric Q + 2).
algebraically rewrite
  (4 × pcommand_metric c1 + formula_metric fAtom +
  4 × pcommand_metric c2 + formula_metric fAtom +
  formula_metric Q + 2)
as
  (4 × (pcommand_metric c1 + pcommand_metric c2) +
  formula_metric Q + 4).
simpl formula_metric.
ring.
omega.
Qed.

```

Shows the quadracity of the efficient weakest preconditions with respect to passified commands `Theorem quadratic_wlp' : ∃ a, ∃ b, ∀ c Q,`
 $formula_metric (wlp' c Q) \leq$
 $a \times pcommand_metric c \times pcommand_metric c +$
 $b \times pcommand_metric c + formula_metric Q.$

Proof.

```
state linear_wlp'.
elim H; clear H; intros N H.
∃ (S N); ∃ (S N).
intros.
revert Q; induction c; intros.

simpl; omega.

simpl; omega.

clear H.
simpl wp'.
specify IHc1 (wp' c2 Q).
refine left bound with
  (S N × pcommand_metric c1 × pcommand_metric c1 +
   S N × pcommand_metric c1 +
   formula_metric (wp' c2 Q)).
clear IHc1.
specify IHc2 Q.
refine left bound with
  (S N × pcommand_metric c1 × pcommand_metric c1 +
   S N × pcommand_metric c1 +
   S N × pcommand_metric c2 × pcommand_metric c2 +
   S N × pcommand_metric c2 +
   formula_metric Q).
clear IHc2.
simpl pcommand_metric.
introduce new identifier a for (pcommand_metric c1).
introduce new identifier b for (pcommand_metric c2).
clear.
introduce new identifier M for (S N).
clear H.
apply plus_le_compat_r.
refine right bound with (M × (a + b) × (a + b) + M × (a + b)).
algebraically rewrite (M × S (a + b)) as (M × S a + M × b).
pattern (M × (a + b)) at 2.
algebraically rewrite (M × (a + b)) as (M × a + M × b).
algebraically rewrite (M × S a) as (M + M × a).
algebraically rewrite ((M + M × a + M × b) × S (a + b) +
  (M + M × a + M × b))
as
((M + M × a + M × b) × S (a + b) +
 M +
 (M × a + M × b)).
```

```

apply plus_le_compat_r.
algebraically rewrite
  (M × (a + b) × (a + b))
as
  (M × a × a + M × b × b + 2 × M × a × b).
algebraically rewrite
  ((M + M × a + M × b) × S (a + b) + M)
as
  (M × S (a + b) + M × a × S (a + b) +
   M × b × S (a + b) + M).
refine left bound with
  (M × S (a + b) + M × a × S (a + b) + M × b × S (a + b)).
algebraically rewrite
  (M × S (a + b)) as (M + M × a + M × b).
algebraically rewrite
  (M × a × S (a + b)) as (M × a + M × a × a + M × a × b).
algebraically rewrite
  (M × b × S (a + b)) as (M × b + M × a × b + M × b × b).
algebraically rewrite
  (M + M × a + M × b +
   (M × a + M × a × a + M × a × b) +
   (M × b + M × a × b + M × b × b))
as
  (M + 2*M*a + 2*M*b + M*a*a + 2*M*a*b + M*b*b).
algebraically rewrite
  (M + 2 × M × a + 2 × M × b + M × a × a +
   2 × M × a × b + M × b × b)
as
  (M × a × a + M × b × b + 2 × M × a × b +
   M + 2 × M × a + 2 × M × b).
assert (0 ≤ M + 2 × M × a + 2 × M × b).
auto with arith.
omega.
omega.
algebraically rewrite
  (M × (a + b) × (a + b))
as
  (M × a × a + M × b × b + 2 × M × a × b).
algebraically rewrite (M × (a + b)) as (M × a + M × b).
algebraically rewrite
  (M × a × a + M × b × b +
   2 × M × a × b + (M × a + M × b))

```

```

as
  ( $M \times a \times a + M \times a + M \times b \times b +$ 
    $M \times b + 2 \times M \times a \times b$ ).
assert ( $0 \leq 2 \times M \times a \times b$ ).
auto with arith.
omega.

simpl.
auto with arith.

specify IHc1 fAtom.
specify IHc2 fAtom.
Opaque wlp'.
simpl wp'.
simpl formula_metric.
Transparent wlp'.
specify H (pcChoice c1 c2) Q.
refine left bound with
  ( $S (S (S N \times pcommand\_metric\ c1 \times pcommand\_metric\ c1 +$ 
    $S N \times pcommand\_metric\ c1 + formula\_metric\ fAtom +$ 
    $S N \times pcommand\_metric\ c2 \times pcommand\_metric\ c2 +$ 
    $S N \times pcommand\_metric\ c2 + formula\_metric\ fAtom +$ 
    $N \times pcommand\_metric\ (pcChoice\ c1\ c2) +$ 
    $formula\_metric\ Q)))$ ).
introduce new identifier M for (S N).
simpl formula_metric.
algebraically rewrite
  ( $S (S (M \times pcommand\_metric\ c1 \times pcommand\_metric\ c1 +$ 
    $M \times pcommand\_metric\ c1 + 1 +$ 
    $M \times pcommand\_metric\ c2 \times pcommand\_metric\ c2 +$ 
    $M \times pcommand\_metric\ c2 + 1 +$ 
    $N \times pcommand\_metric\ (pcChoice\ c1\ c2) +$ 
    $formula\_metric\ Q)))$ )
as
  ( $M \times pcommand\_metric\ c1 \times pcommand\_metric\ c1 +$ 
    $M \times pcommand\_metric\ c1 +$ 
    $M \times pcommand\_metric\ c2 \times pcommand\_metric\ c2 +$ 
    $M \times pcommand\_metric\ c2 +$ 
    $N \times pcommand\_metric\ (pcChoice\ c1\ c2) + 4 +$ 
    $formula\_metric\ Q$ ).
apply plus_le_compat_r.
clear IHc1 IHc2 H.

simpl pcommand_metric.
state (pcommand_metric_min_size c1).

```

```

state (pcommand_metric_min_size c2).
introduce new identifier a for (pcommand_metric c1).
introduce new identifier b for (pcommand_metric c2).
rewrite ← H2 in H.
rewrite ← H3 in H1.

subst M; clear H2 H3.
algebraically rewrite
(S N × a × a +
 S N × a +
 S N × b × b +
 S N × b +
 N × S (a + b) +
 4)
as
(S N × a × a +
 S N × a +
 S N × b × b +
 S N × b +
 4 +
 N × S (a + b)).
algebraically rewrite
(S N × S (a + b) × S (a + b) +
 S N × S (a + b))
as
(S N × S (a + b) × S (a + b) +
 S (a + b) +
 N × S (a + b)).
apply plus_le_compat_r.
algebraically rewrite
(S N × S (a + b) × S (a + b) +
 S (a + b))
as
(2 + 3 × a + a × a + 3 × b + 2 × a × b +
 b × b + N + 2 × N × a + N × a × a +
 2 × N × b + 2 × N × a × b + N × b × b).
algebraically rewrite
(S N × a × a + S N × a +
 S N × b × b + S N × b + 4)
as
(N × a × a + a + N × a +
 b × b + N × b × b + b +
 N × b + 4 + a × a).

```

algebraically rewrite

$$(2 + 3 \times a + a \times a + 3 \times b + 2 \times a \times b + b \times b + N + 2 \times N \times a + N \times a \times a + 2 \times N \times b + 2 \times N \times a \times b + N \times b \times b)$$

as

$$(2 + 3 \times a + 3 \times b + 2 \times a \times b + b \times b + N + 2 \times N \times a + N \times a \times a + 2 \times N \times b + 2 \times N \times a \times b + N \times b \times b + a \times a).$$

apply *plus_le_compat_r*.

algebraically rewrite

$$(N \times a \times a + a + N \times a + b \times b + N \times b \times b + b + N \times b + 4)$$

as

$$(N \times a \times a + a + N \times a + b \times b + b + N \times b + 4 + N \times b \times b).$$

apply *plus_le_compat_r*.

algebraically rewrite

$$(N \times a \times a + a + N \times a + b \times b + b + N \times b + 4)$$

as

$$(a + N \times a + b \times b + b + N \times b + 4 + N \times a \times a).$$

algebraically rewrite

$$(2 + 3 \times a + 3 \times b + 2 \times a \times b + b \times b + N + 2 \times N \times a + N \times a \times a + 2 \times N \times b + 2 \times N \times a \times b)$$

as

$$(2 + 3 \times a + 3 \times b + 2 \times a \times b + b \times b + N + 2 \times N \times a + 2 \times N \times b + 2 \times N \times a \times b + N \times a \times a).$$

apply *plus_le_compat_r*.

algebraically rewrite

$$(a + N \times a + b \times b + b + N \times b + 4)$$

as

$$(a + N \times a + b + N \times b + 4 + b \times b).$$

algebraically rewrite

$$(2 + 3 \times a + 3 \times b + 2 \times a \times b + b \times b + N + 2 \times N \times a + 2 \times N \times b + 2 \times N \times a \times b)$$

as

$$(2 + 3 \times a + 3 \times b + 2 \times a \times b + N + 2 \times N \times a + 2 \times N \times b + 2 \times N \times a \times b + b \times b).$$

apply *plus_le_compat_r*.

```

algebraically rewrite
  (a + N × a + b + N × b + 4)
  as
  (a + b + 4 + (N × a + N × b)).
algebraically rewrite
  (2 + 3 × a + 3 × b + 2 × a × b + N +
   2 × N × a + 2 × N × b + 2 × N × a × b)
  as
  (2 + 3 × a + 3 × b + 2 × a × b + N +
   N × a + N × b + 2 × N × a × b + (N × a + N × b)).
apply plus_le_compat_r.
algebraically rewrite (a + b + 4) as (4 + (a + b)).
algebraically rewrite
  (2 + 3 × a + 3 × b + 2 × a × b + N +
   N × a + N × b + 2 × N × a × b)
  as
  (2 + 2 × a + 2 × b + 2 × a × b + N +
   N × a + N × b + 2 × N × a × b + (a + b)).
apply plus_le_compat_r.
assert (2 ≤ 2 × a).
omega.
assert (0 ≤ 2 × b + 2 × a × b +
         N + N × a + N × b +
         2 × N × a × b).

auto with arith.
omega.
Qed.

A quick proof showing that passification results in a command which is the exact same
size as its input. Theorem passify_maintains_size : ∀ c,
  vcommand_metric c = pcommand_metric (passify c).
Proof.
  induction c; simpl; auto.
Qed.

Lemma showing that if x ≤ y, then x2 ≤ y2. Lemma monotonic_sqr : ∀ x y, x ≤
y → x × x ≤ y × y.
Proof.
  induction x; intros.
  auto with arith.
  algebraically rewrite (S x × S x) as (x × x + 2 × x + 1).
  destruct y.
  assert False.
omega.

```

contradiction.
assert ($x \leq y$).
omega.
specify IHx y H0.
algebraically rewrite ($S\ y \times S\ y$) **as** ($y \times y + 2 \times y + 1$).
omega.
Qed.

Shows that the weakest preconditions are $O(|c|^4 + |Q|)$.

Theorem *polynomial_wps* :

$\exists N4, \exists N3, \exists N2, \exists N1, \forall c\ Q,$
let $cp := \textit{passified}\ c$ **in**
let $x := \textit{command_metric}\ c$ **in**
let $wp := wp'\ cp\ Q$ **in**
 $\textit{formula_metric}\ wp \leq N4 \times x \times x \times x \times x +$
 $N3 \times x \times x \times x +$
 $N2 \times x \times x +$
 $N1 \times x +$
 $\textit{formula_metric}\ Q.$

Proof.

state quadratic_wp'.
eliminate existentials a b in H.
state passify_maintains_size.
state quadratic_sa_transformation.
 $\exists (25 \times a);$
 $\exists (50 \times a);$
 $\exists (25 \times a + 5 \times b);$
 $\exists (5 \times b).$
intros.
specify H cp Q.
subst $wp0.$
refine_le H.
clear $H.$
unfold *passified* **in** $cp.$
introduce pair (transform_sa c init_vmap) as csa v.
subst $cp.$
rewrite $\leftarrow H2.$
specify H0 csa.
rewrite $\leftarrow H0.$
specify H1 c init_vmap.
rewrite $\leftarrow H2$ **in** $H1.$
cbv zeta in H1.
assert ($x = \textit{command_metric}\ c$); [**subst**; **trivial** | *idtac*].

```

clearbody x.
rewrite ← H in H1.
apply plus_le_compat_r.
introduce new identifier y for (vcommand_metric csa).
rewrite ← H3 in × ⊢.
revert H1; clear; intros.
refine left bound with
  (a × (5 × x × x + 5 × x) × (5 × x × x + 5 × x) +
   b × (5 × x × x + 5 × x)).
introduce new identifier z for (5 × x × x + 5 × x).
rewrite ← H in H1; clear H.
cut (a × y × y ≤ a × z × z).
intros.
cut (b × y ≤ b × z).
intros.
omega.
auto with arith.
algebraically rewrite (a × y × y) as (a × (y × y)).
algebraically rewrite (a × z × z) as (a × (z × z)).
apply mult_le_compat_l.
apply monotonic_sqr.
trivial.
algebraically rewrite
  (a × (5 × x × x + 5 × x) × (5 × x × x + 5 × x) +
   b × (5 × x × x + 5 × x))
  as
  (5 × b × x +
   25 × a × x × x +
   5 × b × x × x +
   50 × a × x × x × x +
   25 × a × x × x × x × x).
algebraically rewrite
  (25 × a × x × x × x × x +
   50 × a × x × x × x +
   (25 × a + 5 × b) × x × x +
   5 × b × x)
  as
  (5 × b × x +
   25 × a × x × x +
   5 × b × x × x +
   50 × a × x × x × x +
   25 × a × x × x × x × x).

```

omega.
Qed.