Software Foundations: SECF

August 4, 2026

Contents

Chapter 1

Library SECF.Maps

1.1 Maps: Total and Partial Maps

Maps (or *dictionaries*) are ubiquitous data structures both in ordinary programming and in the theory of programming languages; we're going to need them in many places in the coming chapters.

They also make a nice case study using ideas we've seen in previous chapters, including building data structures out of higher-order functions (from *Basics* and *Poly*) and the use of reflection to streamline proofs (from *IndProp*).

We'll define two flavors of maps: *total* maps, which include a "default" element to be returned when a key being looked up doesn't exist, and *partial* maps, which instead return an **option** to indicate success or failure. Partial maps are defined in terms of total maps, using None as the default element.

1.2 The Standard Library

One small digression before we begin...

Unlike the chapters we have seen so far, this one does not `Require Import` the chapter before it (or, transitively, all the earlier chapters). Instead, in this chapter and from now on, we're going to import the definitions and theorems we need directly from Rocq's standard library. You should not notice much difference, though, because we've been careful to name our own definitions and theorems the same as their counterparts in the standard library, wherever they overlap.

```
From Stdlib Require Import Arith.
From Stdlib Require Import Bool.
From Stdlib Require Export Strings.String.
From Stdlib Require Import FunctionalExtensionality.
From Stdlib Require Import List.
Import ListNotations.
```

Documentation for the standard library can be found at https://rocq-prover.org/doc/V9.0.0/stdlib/inde

The `Search` command is a good way to look for theorems involving objects of specific types. See *Lists* for a reminder of how to use it.

If you want to find out how or where a notation is defined, the `Locate` command is useful. For example, where is the natural addition operation defined in the standard library?

```
Locate "+".
```

(There are several uses of the + notation, but only one for naturals.)

```
Print Init.Nat.add.
```

We'll see some more uses of `Locate` in the Imp chapter.

1.3 Identifiers

To define maps, we first need a type for the keys that we will use to index into our maps. In *Lists.v* we introduced a fresh type *id* for a similar purpose; here and for the rest of *Software Foundations* we will use the **string** type from Rocq's standard library.

To compare strings, we use the function eqb_refl from the String module in the standard library.

```
Check String.eqb_refl :
  ∀ x : string, (x =? x)%string = true.
```

We will often use a few basic properties of string equality... `Check String.eqb_eq :`

```
  ∀ n m : string, (n =? m)%string = true ↔ n = m.
Check String.eqb_neq :
  ∀ n m : string, (n =? m)%string = false ↔ n ≠ m.
Check String.eqb_spec :
  ∀ x y : string, reflect (x = y) (String.eqb x y).
```

1.4 Total Maps

Our main job in this chapter will be to build a definition of partial maps that is similar in behavior to the one we saw in the *Lists* chapter, plus accompanying lemmas about its behavior.

This time around, though, we're going to use *functions*, rather than lists of key-value pairs, to build maps. The advantage of this representation is that it offers a more "extensional" view of maps: two maps that respond to queries in the same way will be represented as exactly the same function, rather than just as "equivalent" list structures. This simplifies proofs that use maps.

We build up to partial maps in two steps. First, we define a type of *total maps* that return a default value when we look up a key that is not present in the map.

```
Definition total_map (A : Type) := string → A.
```

Intuitively, a total map over an element type A is just a function that can be used to look up **string**s, yielding As.

The function t_empty yields an empty total map, given a default element; this map always returns the default element when applied to any string.

```
Definition t_empty {A : Type} (v : A) : total_map A :=
  (fun _ ⇒ v).
```

More interesting is the map-updating function, which (as always) takes a map m, a key x, and a value v and returns a new map that takes x to v and takes every other key to whatever m does. The novelty here is that we achieve this effect by wrapping a new function around the old one.

```
Definition t_update {A : Type} (m : total_map A)
                    (x : string) (v : A) :=
  fun x' ⇒ if String.eqb x x' then v else m x'.
```

This definition is a nice example of higher-order programming: t_update takes a *function* m and yields a new function fun x' ⇒ ... that behaves like the desired map.

For example, we can build a map taking **string**s to **bool**s, where "foo" and "*bar*" are mapped to true and every other key is mapped to false, like this:

```
Definition examplemap :=
  t_update (t_update (t_empty false) "foo" true)
           "bar" true.
```

Next, let's introduce some notations to facilitate working with maps.

First, we use the following notation to represent an empty total map with a default value.

```
Notation "'__' '!->' v" := (t_empty v)
  (at level 100, right associativity).

Example example_empty := ( false).
```

We next introduce a symbolic notation for extending an existing map with a new binding.

```
Notation "x '!->' v ';' m" := (t_update m x v)
                                (at level 100, v constr at level 100, right associativity).
```

The examplemap above can now be defined as follows:

```
Definition examplemap' :=
  ( "bar" !-> true;
    "foo" !-> true;
    __ !-> false
  ).
```

This completes the definition of total maps. Note that we don't need to define a *find* operation on this representation of maps because it is just function application!

```
Example update_example1 : examplemap' "baz" = false.
Proof. reflexivity. Qed.

Example update_example2 : examplemap' "foo" = true.
```

```
Proof. reflexivity. Qed.

Example update_example3 : examplemap' "quux" = false.
Proof. reflexivity. Qed.

Example update_example4 : examplemap' "bar" = true.
Proof. reflexivity. Qed.
```

When we use maps in later chapters, we'll need several fundamental facts about how they behave.

Even if you don't work the following exercises, make sure you thoroughly understand the statements of the lemmas!

(Some of the proofs require the functional extensionality axiom, which was discussed in the Logic chapter.)

Exercise: 1 star, standard, optional (t_apply_empty) First, the empty map returns its default element for all keys:

```
Lemma t_apply_empty : ∀ (A : Type) (x : string) (v : A),
  (_ !-> v) x = v.
Proof.
  Admitted.
```
□

Exercise: 2 stars, standard, optional (t_update_eq) Next, if we update a map m at a key x with a new value v and then look up x in the map resulting from the update, we get back v:

```
Lemma t_update_eq : ∀ (A : Type) (m : total_map A) x v,
  (x !-> v ; m) x = v.
Proof.
  Admitted.
```
□

Exercise: 2 stars, standard, optional (t_update_neq) On the other hand, if we update a map m at a key $x1$ and then look up a *different* key $x2$ in the resulting map, we get the same result that m would have given:

```
Theorem t_update_neq : ∀ (A : Type) (m : total_map A) x1 x2 v,
  x1 ≠ x2 →
  (x1 !-> v ; m) x2 = m x2.
Proof.
  Admitted.
```
□

Exercise: 2 stars, standard, optional (t_update_shadow) If we update a map m at a key x with a value $v1$ and then update again with the same key x and another value $v2$, the resulting map behaves the same (gives the same result when applied to any key) as the simpler map obtained by performing just the second update on m:

```
Lemma t_update_shadow : ∀ (A : Type) (m : total_map A) x v1 v2,
  (x !-> v2 ; x !-> v1 ; m) = (x !-> v2 ; m).
Proof.
  Admitted.
```
□

Exercise: 2 stars, standard (t_update_same) Given strings $x1$ and $x2$, we can use the tactic `destruct (eqb_spec x1 x2)` to simultaneously perform case analysis on the result of *String.eqb x1 x2* and generate hypotheses about the equality (in the sense of $=$) of $x1$ and $x2$. With the example in chapter *IndProp* as a template, use *String.eqb_spec* to prove the following theorem, which states that if we update a map to assign key x the same value as it already has in m, then the result is equal to m:

```
Theorem t_update_same : ∀ (A : Type) (m : total_map A) x,
  (x !-> m x ; m) = m.
Proof.
  Admitted.
```
□

Exercise: 3 stars, standard, especially useful (t_update_permute) Similarly, use *String.eqb_spec* to prove one final property of the update function: If we update a map m at two distinct keys, it doesn't matter in which order we do the updates.

```
Theorem t_update_permute : ∀ (A : Type) (m : total_map A)
                             v1 v2 x1 x2,
  x2 ≠ x1 →
  (x1 !-> v1 ; x2 !-> v2 ; m)
  =
  (x2 !-> v2 ; x1 !-> v1 ; m).
Proof.
  Admitted.
```
□

1.5 Partial maps

Lastly, we define *partial maps* on top of total maps. A partial map with elements of type A is simply a total map with elements of type **option** A and default element None.

```
Definition partial_map (A : Type) := total_map (option A).
```

```
Definition empty {A : Type} : partial_map A :=
  t_empty None.

Definition update {A : Type} (m : partial_map A)
                  (x : string) (v : A) :=
  (x !-> Some v ; m).
```

We introduce a similar notation for partial maps: `Notation "x '|->' v ';' m" := (update m x v)`
`  (at level 0, x constr, v at level 200, right associativity).`

We can also hide the last case when it is empty. `Notation "x '|->' v" := (update empty x v)`
`  (at level 0, x constr, v at level 200).`

```
Definition examplepmap :=
  ("Church" |-> true ; "Turing" |-> false).
```

We now straightforwardly lift all of the basic lemmas about total maps to partial maps.

```
Lemma apply_empty : ∀ (A : Type) (x : string),
  @empty A x = None.
Proof.
  intros. unfold empty. rewrite t_apply_empty.
  reflexivity.
Qed.

Lemma update_eq : ∀ (A : Type) (m : partial_map A) x v,
  (x |-> v ; m) x = Some v.
Proof.
  intros. unfold update. rewrite t_update_eq.
  reflexivity.
Qed.
```

The update_eq lemma is used very often in proofs. Adding it to Rocq's global "hint database" allows proof-automation tactics such as `auto` to find it. `#[global] Hint Resolve update_eq : core.`

```
Theorem update_neq : ∀ (A : Type) (m : partial_map A) x1 x2 v,
  x2 ≠ x1 →
  (x2 |-> v ; m) x1 = m x1.
Proof.
  intros A m x1 x2 v H.
  unfold update. rewrite t_update_neq.
  - reflexivity.
  - apply H.
Qed.

Lemma update_shadow : ∀ (A : Type) (m : partial_map A) x v1 v2,
  (x |-> v2 ; x |-> v1 ; m) = (x |-> v2 ; m).
```

```
Proof.
  intros A m x v1 v2. unfold update. rewrite t_update_shadow.
  reflexivity.
Qed.

Theorem update_same : ∀ (A : Type) (m : partial_map A) x v,
  m x = Some v →
  (x |-> v ; m) = m.
Proof.
  intros A m x v H. unfold update. rewrite ← H.
  apply t_update_same.
Qed.

Theorem update_permute : ∀ (A : Type) (m : partial_map A)
                                    x1 x2 v1 v2,
  x2 ≠ x1 →
  (x1 |-> v1 ; x2 |-> v2 ; m) = (x2 |-> v2 ; x1 |-> v1 ; m).
Proof.
  intros A m x1 x2 v1 v2. unfold update.
  apply t_update_permute.
Qed.
```

One last thing: For partial maps, it's convenient to introduce a notion of map inclusion, stating that all the entries in one map are also present in another:

```
Definition includedin {A : Type} (m m' : partial_map A) :=
  ∀ x v, m x = Some v → m' x = Some v.
```

We can then show that map update preserves map inclusion, that is:

```
Lemma includedin_update : ∀ (A : Type) (m m' : partial_map A)
                                    (x : string) (vx : A),
  includedin m m' →
  includedin (x |-> vx ; m) (x |-> vx ; m').
Proof.
  unfold includedin.
  intros A m m' x vx H.
  intros y vy.
  destruct (eqb_spec x y) as [Hxy | Hxy].
  - rewrite Hxy.
    rewrite update_eq. rewrite update_eq. intro H1. apply H1.
  - rewrite update_neq.
    + rewrite update_neq.
      × apply H.
      × apply Hxy.
    + apply Hxy.
Qed.
```

This property is quite useful for reasoning about languages with variable binding – e.g., the Simply Typed Lambda Calculus, which we will see in *Programming Language Foundations*, where maps are used to keep track of which program variables are defined in a given scope.

Chapter 2

Library SECF.Imp

2.1 Imp: Simple Imperative Programs

In this chapter, we take a more serious look at how to use Rocq as a tool to study other things. Our case study is a *simple imperative programming language* called Imp, embodying a tiny core fragment of conventional mainstream languages such as C and Java.

Here is a familiar mathematical function written in Imp.

Z := X; Y := 1; while Z <> 0 do Y := Y * Z; Z := Z - 1 end

We concentrate here on defining the *syntax* and *semantics* of Imp; later, in *Programming Language Foundations* (*Software Foundations*, volume 2), we develop a theory of *program equivalence* and introduce *Hoare Logic*, a popular logic for reasoning about imperative programs.

```
Set Warnings "-notation-overridden".
From Stdlib Require Import Bool.
From Stdlib Require Import Init.Nat.
From Stdlib Require Import Arith.
From Stdlib Require Import EqNat. Import Nat.
From Stdlib Require Import Lia.
From Stdlib Require Import List. Import ListNotations.
From Stdlib Require Import Strings.String.
From SECF Require Import Maps.
```

2.2 Arithmetic and Boolean Expressions

We'll present Imp in three parts: first a core language of *arithmetic and boolean expressions*, then an extension of these with *variables*, and finally a language of *commands* including assignment, conditionals, sequencing, and loops.

2.2.1 Syntax

```
Module AExp.
```

These two definitions specify the *abstract syntax* of arithmetic and boolean expressions.

```
Inductive aexp : Type :=
  | ANum (n : nat)
  | APlus (a1 a2 : aexp)
  | AMinus (a1 a2 : aexp)
  | AMult (a1 a2 : aexp).

Inductive bexp : Type :=
  | BTrue
  | BFalse
  | BEq (a1 a2 : aexp)
  | BNeq (a1 a2 : aexp)
  | BLe (a1 a2 : aexp)
  | BGt (a1 a2 : aexp)
  | BNot (b : bexp)
  | BAnd (b1 b2 : bexp).
```

In this chapter, we'll mostly elide the translation from the concrete syntax that a programmer would actually write to these abstract syntax trees – the process that, for example, would translate the string "1 + 2 × 3" to the AST

APlus (ANum 1) (AMult (ANum 2) (ANum 3)).

The optional chapter *ImpParser* develops a simple lexical analyzer and parser that can perform this translation. You do not need to understand that chapter to understand this one, but if you haven't already taken a course where these techniques are covered (e.g., a course on compilers) you may want to skim it.

For comparison, here's a conventional BNF (Backus-Naur Form) grammar defining the same abstract syntax:

a := nat | a + a | a - a | a * a

b := true | false | a = a | a <> a | a <= a | a > a | ~ b | b && b

Compared to the Rocq version above...

- The BNF is more informal – for example, it gives some suggestions about the surface syntax of expressions (like the fact that the addition operation is written with an infix +) while leaving other aspects of lexical analysis and parsing (like the relative precedence of +, -, and ×, the use of parens to group subexpressions, etc.) unspecified. Some additional information – and human intelligence – would be required to turn this description into a formal definition, e.g., for implementing a compiler.

  The Rocq version consistently omits all this information and concentrates on the abstract syntax only.

- Conversely, the BNF version is lighter and easier to read. Its informality makes it flexible, a big advantage in situations like discussions at the blackboard, where conveying general ideas is more important than nailing down every detail precisely.

  Indeed, there are dozens of BNF-like notations and people switch freely among them – usually without bothering to say which kind of BNF they're using, because there is no need to: a rough-and-ready informal understanding is all that's important.

It's good to be comfortable with both sorts of notations: informal ones for communicating between humans and formal ones for carrying out implementations and proofs.

2.2.2 Evaluation

Evaluating an arithmetic expression produces a number.

```
Fixpoint aeval (a : aexp) : nat :=
  match a with
  | ANum n ⇒ n
  | APlus a1 a2 ⇒ (aeval a1) + (aeval a2)
  | AMinus a1 a2 ⇒ (aeval a1) - (aeval a2)
  | AMult a1 a2 ⇒ (aeval a1) × (aeval a2)
  end.

Example test_aeval1:
  aeval (APlus (ANum 2) (ANum 2)) = 4.
Proof. reflexivity. Qed.
```

Similarly, evaluating a boolean expression yields a boolean.

```
Fixpoint beval (b : bexp) : bool :=
  match b with
  | BTrue ⇒ true
  | BFalse ⇒ false
  | BEq a1 a2 ⇒ (aeval a1) =? (aeval a2)
  | BNeq a1 a2 ⇒ negb ((aeval a1) =? (aeval a2))
  | BLe a1 a2 ⇒ (aeval a1) <=? (aeval a2)
  | BGt a1 a2 ⇒ negb ((aeval a1) <=? (aeval a2))
  | BNot b1 ⇒ negb (beval b1)
  | BAnd b1 b2 ⇒ andb (beval b1) (beval b2)
  end.
```

2.2.3 Optimization

We haven't defined very much yet, but we can already get some mileage out of the definitions. Suppose we define a function that takes an arithmetic expression and slightly simplifies it, changing every occurrence of $0 + e$ (i.e., (APlus (ANum 0) e) into just e.

```
Fixpoint optimize_0plus (a:aexp) : aexp :=
  match a with
  | ANum n ⇒ ANum n
  | APlus (ANum 0) e2 ⇒ optimize_0plus e2
  | APlus e1 e2 ⇒ APlus (optimize_0plus e1) (optimize_0plus e2)
  | AMinus e1 e2 ⇒ AMinus (optimize_0plus e1) (optimize_0plus e2)
  | AMult e1 e2 ⇒ AMult (optimize_0plus e1) (optimize_0plus e2)
  end.
```

To gain confidence that our optimization is doing the right thing we can test it on some examples and see if the output looks OK.

```
Example test_optimize_0plus:
  optimize_0plus (APlus (ANum 2)
                        (APlus (ANum 0)
                               (APlus (ANum 0) (ANum 1))))
  = APlus (ANum 2) (ANum 1).
Proof. reflexivity. Qed.
```

But if we want to be certain the optimization is correct – that evaluating an optimized expression *always* gives the same result as the original – we should prove it!

```
Theorem optimize_0plus_sound: ∀ a,
  aeval (optimize_0plus a) = aeval a.
Proof.
  intros a. induction a.
  - reflexivity.
  - destruct a1 eqn:Ea1.
    + destruct n eqn:En.
      × simpl. apply IHa2.
      × simpl. rewrite IHa2. reflexivity.
    +
      simpl. simpl in IHa1. rewrite IHa1.
      rewrite IHa2. reflexivity.
    +
      simpl. simpl in IHa1. rewrite IHa1.
      rewrite IHa2. reflexivity.
    +
      simpl. simpl in IHa1. rewrite IHa1.
      rewrite IHa2. reflexivity.
  -
    simpl. rewrite IHa1. rewrite IHa2. reflexivity.
  -
    simpl. rewrite IHa1. rewrite IHa2. reflexivity. Qed.
```

2.3 Rocq Automation

The amount of repetition in this last proof is a little annoying. And if either the language of arithmetic expressions or the optimization being proved sound were significantly more complex, it would start to be a real problem.

So far, we've been doing all our proofs using just a small handful of Rocq's tactics and completely ignoring its powerful facilities for constructing parts of proofs automatically. This section introduces some of these facilities, and we will see more over the next several chapters. Getting used to them will take some energy – Rocq's automation is a power tool – but it will allow us to scale up our efforts to more complex definitions and more interesting properties without becoming overwhelmed by boring, repetitive, low-level details.

2.3.1 Tacticals

Tacticals is Rocq's term for tactics that take other tactics as arguments – "higher-order tactics," if you will.

The `try` Tactical

If T is a tactic, then `try` T is a tactic that is just like T except that, if T fails, `try` T *successfully* does nothing at all (rather than failing). `Theorem` silly1 : ∀ (*P* : `Prop`), *P* → *P*.

```
Proof.
  intros P HP.
  try reflexivity.    apply HP. Qed.

Theorem silly2 : ∀ ae, aeval ae = aeval ae.
Proof.
    try reflexivity. Qed.
```

There is not much reason to use `try` in completely manual proofs like these, but it is very useful for doing automated proofs in conjunction with the ; tactical, which we show next.

The ; Tactical (Simple Form)

In its most common form, the ; tactical takes two tactics as arguments. The compound tactic T;*T'* first performs T and then performs *T'* on *each subgoal* generated by T.

For example, consider the following trivial lemma:

```
Lemma foo : ∀ n, 0 <=? n = true.
Proof.
  intros.
  destruct n.
    - simpl. reflexivity.
    - simpl. reflexivity.
Qed.
```

We can simplify this proof using the ; tactical:

```
Lemma foo' : ∀ n, 0 <=? n = true.
Proof.
  intros.
  destruct n;

  simpl;

  reflexivity.
Qed.
```

Using `try` and ; together, we can get rid of the repetition in the proof that was bothering us a little while ago.

```
Theorem optimize_0plus_sound': ∀ a,
  aeval (optimize_0plus a) = aeval a.
Proof.
  intros a.
  induction a;

    try (simpl; rewrite IHa1; rewrite IHa2; reflexivity).
  - reflexivity.
  -
    destruct a1 eqn:Ea1;

      try (simpl; simpl in IHa1; rewrite IHa1;
           rewrite IHa2; reflexivity).
    + destruct n eqn:En;
      simpl; rewrite IHa2; reflexivity. Qed.
```

Rocq experts often use this "...; `try`... " idiom after a tactic like `induction` to take care of many similar cases all at once. Indeed, this practice has an analog in informal proofs. For example, here is an informal proof of the optimization theorem that matches the structure of the formal one:

Theorem: For all arithmetic expressions a,
aeval (optimize_0plus a) = aeval a.

Proof: By induction on a. Most cases follow directly from the IH. The remaining cases are as follows:

- Suppose a = ANum n for some n. We must show

  aeval (optimize_0plus (ANum n)) = aeval (ANum n).

  This is immediate from the definition of optimize_0plus.

- Suppose a = APlus $a1$ $a2$ for some $a1$ and $a2$. We must show

aeval (optimize_0plus (APlus a1 a2)) = aeval (APlus a1 a2).

Consider the possible forms of *a1*. For most of them, optimize_0plus simply calls itself recursively for the subexpressions and rebuilds a new expression of the same form as *a1*; in these cases, the result follows directly from the IH.

The interesting case is when *a1* = ANum *n* for some *n*. If *n* = 0, then

optimize_0plus (APlus a1 a2) = optimize_0plus a2

and the IH for *a2* is exactly what we need. On the other hand, if *n* = S *n'* for some *n'*, then again optimize_0plus simply calls itself recursively, and the result follows from the IH. □

However, this proof can still be improved: the first case (for *a* = ANum *n*) is very trivial – even more trivial than the cases that we said simply followed from the IH – yet we have chosen to write it out in full. It would be better and clearer to drop it and just say, at the top, "Most cases are either immediate or direct from the IH. The only interesting case is the one for APlus..." We can make the same improvement in our formal proof too. Here's how it looks:

```
Theorem optimize_0plus_sound'': ∀ a,
  aeval (optimize_0plus a) = aeval a.
Proof.
  intros a.
  induction a;

    try (simpl; rewrite IHa1; rewrite IHa2; reflexivity);

    try reflexivity.
  -
    destruct a1; try (simpl; simpl in IHa1; rewrite IHa1;
                      rewrite IHa2; reflexivity).
    + destruct n;
      simpl; rewrite IHa2; reflexivity. Qed.
```

The ; Tactical (General Form)

The ; tactical also has a more general form than the simple T;*T'* we've seen above. If T, *T1*, ..., *Tn* are tactics, then

T; *T1* | *T2* | ... | *Tn*

is a tactic that first performs T and then performs *T1* on the first subgoal generated by T, performs *T2* on the second subgoal, etc.

So T;*T'* is just special notation for the case when all of the *Ti*'s are the same tactic; i.e., T;*T'* is shorthand for:

T; *T'* | *T'* | ... | *T'*

The `repeat` Tactical

The `repeat` tactical takes another tactic and keeps applying this tactic until it fails or until it succeeds but doesn't make any progress.

Here is an example proving that 10 is in a long list using `repeat`.

```
Theorem In10 : In 10 [1;2;3;4;5;6;7;8;9;10].
Proof.
  repeat (try (left; reflexivity); right).
Qed.
```

The tactic `repeat T` never fails: if the tactic `T` doesn't apply to the original goal, then repeat *succeeds* without changing the goal at all (i.e., it repeats zero times).

```
Theorem In10' : In 10 [1;2;3;4;5;6;7;8;9;10].
Proof.
  repeat simpl.
  repeat (left; reflexivity).
  repeat (right; try (left; reflexivity)).
Qed.
```

The tactic `repeat T` does not have any upper bound on the number of times it applies `T`. If `T` is a tactic that *always* succeeds (and makes progress), then repeat `T` will loop forever.

```
Theorem repeat_loop : ∀ (m n : nat),
  m + n = n + m.
Proof.
  intros m n.
Admitted.
```

Wait – did we just write an infinite loop in Rocq?!?!

Sort of.

While evaluation in Rocq's term language, Gallina, is guaranteed to terminate, *tactic* evaluation is not. This does not affect Rocq's logical consistency, however, since the job of `repeat` and other tactics is to guide Rocq in constructing proofs; if the construction process diverges (i.e., it does not terminate), this simply means that we have failed to construct a proof at all, not that we have constructed a bad proof.

Exercise: 3 stars, standard (optimize_0plus_b_sound) Since the optimize_0plus transformation doesn't change the value of **aexp**s, we should be able to apply it to all the **aexp**s that appear in a **bexp** without changing the **bexp**'s value. Write a function that performs this transformation on **bexp**s and prove it is sound. Use the tacticals we've just seen to make the proof as short and elegant as possible.

```
Fixpoint optimize_0plus_b (b : bexp) : bexp
  . Admitted.

Example optimize_0plus_b_test1:
```

```
  optimize_0plus_b (BNot (BGt (APlus (ANum 0) (ANum 4)) (ANum 8))) =
                   (BNot (BGt (ANum 4) (ANum 8))).
Proof. Admitted.

Example optimize_0plus_b_test2:
  optimize_0plus_b (BAnd (BLe (APlus (ANum 0) (ANum 4)) (ANum 5)) BTrue) =
                   (BAnd (BLe (ANum 4) (ANum 5)) BTrue).
Proof. Admitted.

Theorem optimize_0plus_b_sound : ∀ b,
  beval (optimize_0plus_b b) = beval b.
Proof.
  Admitted.
  □
```

Exercise: 4 stars, standard, optional (optimize) *Design exercise*: The optimization implemented by our optimize_0plus function is only one of many possible optimizations on arithmetic and boolean expressions. Write a more sophisticated optimizer and prove it correct. (You will probably find it easiest to start small – add just a single, simple optimization and its correctness proof – and build up incrementally to something more interesting.)

2.3.2 Defining New Tactics

Rocq also provides facilities for "programming" in tactic scripts.

The `Ltac` idiom illustrated below gives a handy way to define "shorthand tactics" that bundle several tactics into a single command.

Ltac also includes syntactic pattern-matching on the goal and context, as well as general programming facilities.

It is useful for proof automation and there are several idioms for programming with Ltac. Because it is a language style you might not have seen before, a good reference is the textbook "Certified Programming with dependent types" *CPDT*, which is more advanced that what we will need in this course, but is considered by many the best reference for Ltac programming.

Just for future reference: Rocq provides two other ways of defining new tactics. There is a `Tactic Notation` command that allows defining new tactics with custom control over their concrete syntax. And there is also a low-level API that can be used to build tactics that directly manipulate Rocq's internal structures. We will not need either of these for present purposes.

Here's an example `Ltac` script called *invert*.

```
Ltac invert H :=
  inversion H; subst; clear H.
```

This defines a new tactic called *invert* that takes a hypothesis H as an argument and performs the sequence of commands `inversion` H; `subst`; `clear` H. This gives us quick way to do inversion on evidence and constructors, rewrite with the generated equations, and remove the redundant hypothesis at the end.

```
Lemma invert_example1: ∀ {a b c: nat}, [a ;b] = [a;c] → b = c.
  intros.
  invert H.
  reflexivity.
Qed.
```

2.3.3 The *lia* Tactic

The *lia* tactic implements a decision procedure for integer linear arithmetic, a subset of propositional logic and arithmetic.

If the goal is a universally quantified formula made out of

- numeric constants, addition (+ and S), subtraction (- and pred), and multiplication by constants (this is what makes it Presburger arithmetic),
- equality (= and ≠) and ordering (≤ and >), and
- the logical connectives ∧, ∨, ¬, and →,

then invoking *lia* will either solve the goal or fail, meaning that the goal is actually false. (If the goal is *not* of this form, *lia* will fail.)

```
Example silly_presburger_example : ∀ m n o p,
  m + n ≤ n + o ∧ o + 3 = p + 3 →
  m ≤ p.
Proof.
  intros. lia.
Qed.

Example add_comm__lia : ∀ m n,
    m + n = n + m.
Proof.
  intros. lia.
Qed.

Example add_assoc__lia : ∀ m n p,
    m + (n + p) = m + n + p.
Proof.
  intros. lia.
Qed.
```

(Note the `From` *Stdlib* `Require Import` Lia. at the top of this file, which makes *lia* available.)

2.3.4 A Few More Handy Tactics

Finally, here are some miscellaneous tactics that you may find convenient.

- `clear` *H*: Delete hypothesis *H* from the context.
- `subst` *x*: Given a variable *x*, find an assumption $x = e$ or $e = x$ in the context, replace *x* with *e* throughout the context and current goal, and clear the assumption.
- `subst`: Substitute away *all* assumptions of the form $x = e$ or $e = x$ (where *x* is a variable).
- `rename`... *into*...: Change the name of a hypothesis in the proof context. For example, if the context includes a variable named *x*, then `rename` *x into y* will change all occurrences of *x* to *y*.
- `assumption`: Try to find a hypothesis *H* in the context that exactly matches the goal; if one is found, solve the goal.
- *contradiction*: Try to find a hypothesis *H* in the context that is logically equivalent to **False**. If one is found, solve the goal.
- `constructor`: Try to find a constructor *c* (from some `Inductive` definition in the current environment) that can be applied to solve the current goal. If one is found, behave like `apply` *c*.

We'll see examples of all of these as we go along.

2.4 Evaluation as a Relation

We have presented aeval and beval as functions defined by `Fixpoint`s. Another way to think about evaluation – one that is often more flexible – is as a *relation* between expressions and their values. This perspective leads to `Inductive` definitions like the following...

```
Module AEVALR_FIRST_TRY.

Inductive aevalR : aexp → nat → Prop :=
  | E_ANum (n : nat) :
      aevalR (ANum n) n
  | E_APlus (e1 e2 : aexp) (n1 n2 : nat) :
      aevalR e1 n1 →
      aevalR e2 n2 →
      aevalR (APlus e1 e2) (n1 + n2)
  | E_AMinus (e1 e2 : aexp) (n1 n2 : nat) :
      aevalR e1 n1 →
      aevalR e2 n2 →
```

```
      aevalR (AMinus e1 e2) (n1 - n2)
  | E_AMult (e1 e2 : aexp) (n1 n2 : nat) :
      aevalR e1 n1 →
      aevalR e2 n2 →
      aevalR (AMult e1 e2) (n1 × n2).
```

```
Module HypothesisNames.
```

A small notational aside. We could also write the definition of **aevalR** as follow, with explicit names for the hypotheses in each case:

```
Inductive aevalR : aexp → nat → Prop :=
  | E_ANum (n : nat) :
      aevalR (ANum n) n
  | E_APlus (e1 e2 : aexp) (n1 n2 : nat)
      (H1 : aevalR e1 n1)
      (H2 : aevalR e2 n2) :
      aevalR (APlus e1 e2) (n1 + n2)
  | E_AMinus (e1 e2 : aexp) (n1 n2 : nat)
      (H1 : aevalR e1 n1)
      (H2 : aevalR e2 n2) :
      aevalR (AMinus e1 e2) (n1 - n2)
  | E_AMult (e1 e2 : aexp) (n1 n2 : nat)
      (H1 : aevalR e1 n1)
      (H2 : aevalR e2 n2) :
      aevalR (AMult e1 e2) (n1 × n2).
```

This style gives us more control over the names that Rocq chooses during proofs involving **aevalR**, at the cost of making the definition a little more verbose.

```
End HypothesisNames.
```

It will be convenient to have an infix notation for **aevalR**. We'll write e `==>` n to mean that arithmetic expression e evaluates to value n.

```
Notation "e '==>' n"
         := (aevalR e n)
            (at level 90, left associativity)
         : type_scope.
```

```
End aevalR_first_try.
```

As we saw in our case study of regular expressions in chapter *IndProp*, Rocq provides a way to use this notation in the definition of **aevalR** itself.

```
Reserved Notation "e '==>' n" (at level 90, left associativity).
```

```
Inductive aevalR : aexp → nat → Prop :=
  | E_ANum (n : nat) :
      (ANum n) ==> n
```

```
| E_APlus (e1 e2 : aexp) (n1 n2 : nat) :
    (e1 ==> n1) →
    (e2 ==> n2) →
    (APlus e1 e2) ==> (n1 + n2)
| E_AMinus (e1 e2 : aexp) (n1 n2 : nat) :
    (e1 ==> n1) →
    (e2 ==> n2) →
    (AMinus e1 e2) ==> (n1 - n2)
| E_AMult (e1 e2 : aexp) (n1 n2 : nat) :
    (e1 ==> n1) →
    (e2 ==> n2) →
    (AMult e1 e2) ==> (n1 × n2)

where "e '==>' n" := (aevalR e n) : type_scope.
```

2.4.1 Inference Rule Notation

In informal discussions, it is convenient to write the rules for **aevalR** and similar relations in the more readable graphical form of *inference rules*, where the premises above the line justify the conclusion below the line.

For example, the constructor E_APlus...

| E_APlus : forall (e1 e2 : aexp) (n1 n2 : nat), aevalR e1 n1 -> aevalR e2 n2 -> aevalR (APlus e1 e2) (n1 + n2)

...can be written like this as an inference rule:

e1 ==> n1 e2 ==> n2

(E_APlus) APlus e1 e2 ==> n1+n2

Formally, there is nothing deep about inference rules: they are just implications.

You can read the rule name on the right as the name of the constructor and read each of the linebreaks between the premises above the line (as well as the line itself) as →.

All the variables mentioned in the rule (*e1*, *n1*, etc.) are implicitly bound by universal quantifiers at the beginning. (Such variables are often called *metavariables* to distinguish them from the variables of the language we are defining. At the moment, our arithmetic expressions don't include variables, but we'll soon be adding them.)

The whole collection of rules is understood as being wrapped in an `Inductive` declaration. In informal prose, this is sometimes indicated by saying something like "Let **aevalR** be the smallest relation closed under the following rules...".

For example, we could define ==> as the smallest relation closed under these rules:

(E_ANum) ANum n ==> n

e1 ==> n1 e2 ==> n2

(E_APlus) APlus e1 e2 ==> n1+n2

e1 ==> n1 e2 ==> n2

(E_AMinus) AMinus e1 e2 ==> n1-n2
e1 ==> n1 e2 ==> n2

(E_AMult) AMult e1 e2 ==> n1*n2

Exercise: 1 star, standard, optional (beval_rules) Here, again, is the Rocq definition of the beval function:

Fixpoint beval (e : bexp) : bool := match e with | BTrue => true | BFalse => false | BEq a1 a2 => (aeval a1) =? (aeval a2) | BNeq a1 a2 => negb ((aeval a1) =? (aeval a2)) | BLe a1 a2 => (aeval a1) <=? (aeval a2) | BGt a1 a2 => ~((aeval a1) <=? (aeval a2)) | BNot b => negb (beval b) | BAnd b1 b2 => andb (beval b1) (beval b2) end.

Write out a corresponding definition of boolean evaluation as a relation (in inference rule notation).

```
Definition manual_grade_for_beval_rules : option (nat×string) := None.
```

□

2.4.2 Equivalence of the Definitions

It is straightforward to prove that the relational and functional definitions of evaluation agree:

```
Theorem aevalR_iff_aeval : ∀ a n,
  (a ==> n) ↔ aeval a = n.
Proof.
  split.
  -
    intros H.
    induction H; simpl.
    +
      reflexivity.
    +
      rewrite IHaevalR1. rewrite IHaevalR2. reflexivity.
    +
      rewrite IHaevalR1. rewrite IHaevalR2. reflexivity.
    +
      rewrite IHaevalR1. rewrite IHaevalR2. reflexivity.
  -
    generalize dependent n.
    induction a;
       simpl; intros; subst.
    +
```

```
    apply E_ANum.
  +
    apply E_APlus.
    × apply IHa1. reflexivity.
    × apply IHa2. reflexivity.
  +
    apply E_AMinus.
    × apply IHa1. reflexivity.
    × apply IHa2. reflexivity.
  +
    apply E_AMult.
    × apply IHa1. reflexivity.
    × apply IHa2. reflexivity.
Qed.
```

Again, we can make the proof quite a bit shorter using some tacticals.

```
Theorem aevalR_iff_aeval' : ∀ a n,
  (a ==> n) ↔ aeval a = n.
Proof.
  split.
  -
    intros H; induction H; subst; reflexivity.
  -
    generalize dependent n.
    induction a; simpl; intros; subst; constructor;
       try apply IHa1; try apply IHa2; reflexivity.
Qed.
```

Exercise: 3 stars, standard (bevalR) Write a relation **bevalR** in the same style as **aevalR**, and prove that it is equivalent to beval.

```
Reserved Notation "e '==>b' b" (at level 90, left associativity).
Inductive bevalR: bexp → bool → Prop :=

where "e '==>b' b" := (bevalR e b) : type_scope
.

Lemma bevalR_iff_beval : ∀ b bv,
  b ==>b bv ↔ beval b = bv.
Proof.
  Admitted.
  □

End AExp.
```

2.4.3 Computational vs. Relational Definitions

For the definitions of evaluation for arithmetic and boolean expressions, the choice of whether to use functional or relational definitions is mainly a matter of taste: either way works fine.

However, there are many situations where relational definitions of evaluation work much better than functional ones.

```
Module AEVALR_DIVISION.
```

For example, suppose that we wanted to extend the arithmetic operations with division:

```
Inductive aexp : Type :=
  | ANum (n : nat)
  | APlus (a1 a2 : aexp)
  | AMinus (a1 a2 : aexp)
  | AMult (a1 a2 : aexp)
  | ADiv (a1 a2 : aexp).
```

Extending the definition of aeval to handle this new operation would not be straightforward (what should we return as the result of ADiv (ANum 5) (ANum 0)?). But extending aevalR is very easy.

```
Reserved Notation "e '==>' n"
                  (at level 90, left associativity).

Inductive aevalR : aexp → nat → Prop :=
  | E_ANum (n : nat) :
      (ANum n) ==> n
  | E_APlus (a1 a2 : aexp) (n1 n2 : nat) :
      (a1 ==> n1) → (a2 ==> n2) → (APlus a1 a2) ==> (n1 + n2)
  | E_AMinus (a1 a2 : aexp) (n1 n2 : nat) :
      (a1 ==> n1) → (a2 ==> n2) → (AMinus a1 a2) ==> (n1 - n2)
  | E_AMult (a1 a2 : aexp) (n1 n2 : nat) :
      (a1 ==> n1) → (a2 ==> n2) → (AMult a1 a2) ==> (n1 × n2)
  | E_ADiv (a1 a2 : aexp) (n1 n2 n3 : nat) :
      (a1 ==> n1) → (a2 ==> n2) → (n2 > 0) →
      (mult n2 n3 = n1) → (ADiv a1 a2) ==> n3

where "a '==>' n" := (aevalR a n) : type_scope.
```

Notice that this evaluation relation corresponds to a *partial* function: There are some inputs for which it does not specify an output.

```
End AEVALR_DIVISION.
```

```
Module AEVALR_EXTENDED.
```

Or suppose that we want to extend the arithmetic operations by a nondeterministic number generator *any* that, when evaluated, may yield any number.

(Note that this is not the same as making a *probabilistic* choice among all possible numbers – we're not specifying any particular probability distribution for the results, just saying what results are *possible*.)

```
Reserved Notation "e '==>' n" (at level 90, left associativity).

Inductive aexp : Type :=
  | AAny
  | ANum (n : nat)
  | APlus (a1 a2 : aexp)
  | AMinus (a1 a2 : aexp)
  | AMult (a1 a2 : aexp).
```

Again, extending aeval would be tricky, since now evaluation is *not* a deterministic function from expressions to numbers; but extending **aevalR** is no problem...

```
Inductive aevalR : aexp → nat → Prop :=
  | E_Any (n : nat) :
      AAny ==> n
  | E_ANum (n : nat) :
      (ANum n) ==> n
  | E_APlus (a1 a2 : aexp) (n1 n2 : nat) :
      (a1 ==> n1) → (a2 ==> n2) → (APlus a1 a2) ==> (n1 + n2)
  | E_AMinus (a1 a2 : aexp) (n1 n2 : nat) :
      (a1 ==> n1) → (a2 ==> n2) → (AMinus a1 a2) ==> (n1 - n2)
  | E_AMult (a1 a2 : aexp) (n1 n2 : nat) :
      (a1 ==> n1) → (a2 ==> n2) → (AMult a1 a2) ==> (n1 × n2)

where "a '==>' n" := (aevalR a n) : type_scope.

End AEVALR_EXTENDED.
```

At this point you maybe wondering: Which of these styles should I use by default?

In the examples we've just seen, relational definitions turned out to be more useful than functional ones. For situations like these, where the thing being defined is not easy to express as a function, or indeed where it is *not* a function, there is no real choice. But what about when both styles are workable?

One point in favor of relational definitions is that they can be more elegant and easier to understand.

Another is that Rocq automatically generates nice inversion and induction principles from `Inductive` definitions.

On the other hand, functional definitions can often be more convenient:

- Functions are automatically deterministic and total; for a relational definition, we have to *prove* these properties explicitly if we need them.
- With functions we can also take advantage of Rocq's computation mechanism to simplify expressions during proofs.

Furthermore, functions can be directly "extracted" from Gallina to executable code in OCaml or Haskell.

Ultimately, the choice often comes down to either the specifics of a particular situation or simply a question of taste. Indeed, in large Rocq developments it is common to see a definition given in *both* functional and relational styles, plus a lemma stating that the two coincide, allowing further proofs to switch from one point of view to the other at will.

2.5 Expressions With Variables

Let's return to defining Imp, where the next thing we need to do is to enrich our arithmetic and boolean expressions with variables.

To keep things simple, we'll assume that all variables are global and that they only hold numbers.

2.5.1 States

Since we'll want to look variables up to find out their current values, we'll use total maps from the Maps chapter.

A *machine state* (or just *state*) represents the current values of all variables at some point in the execution of a program.

For simplicity, we assume that the state is defined for *all* variables, even though any given program is only able to mention a finite number of them. Because each variable stores a natural number, we can represent the state as a total map from strings (variable names) to **nat**, and will use 0 as default value in the store.

```
Definition state := total_map nat.
```

2.5.2 Syntax

We can add variables to the arithmetic expressions we had before simply by including one more constructor:

```
Inductive aexp : Type :=
  | ANum (n : nat)
  | AId (x : string)
  | APlus (a1 a2 : aexp)
  | AMinus (a1 a2 : aexp)
  | AMult (a1 a2 : aexp).
```

Defining a few variable names as notational shorthands will make examples easier to read:

```
Definition W : string := "W".
Definition X : string := "X".
Definition Y : string := "Y".
```

```
Definition Z : string := "Z".
```

(This convention for naming program variables (X, Y, Z) clashes a bit with our earlier use of uppercase letters for types. Since we're not using polymorphism heavily in the chapters developed to Imp, this overloading should not cause confusion.)

The definition of **bexp**s is unchanged (except that it now refers to the new **aexp**s):

```
Inductive bexp : Type :=
  | BTrue
  | BFalse
  | BEq (a1 a2 : aexp)
  | BNeq (a1 a2 : aexp)
  | BLe (a1 a2 : aexp)
  | BGt (a1 a2 : aexp)
  | BNot (b : bexp)
  | BAnd (b1 b2 : bexp).
```

2.5.3 Notations

To make Imp programs easier to read and write, we introduce some notations and implicit coercions.

You do not need to understand exactly what these declarations do.

Briefly, though:

- The `Coercion` declaration stipulates that a function (or constructor) can be implicitly used by the type system to coerce a value of the input type to a value of the output type. For instance, the coercion declaration for AId allows us to use plain strings when an **aexp** is expected; the string will implicitly be wrapped with AId.

- *Declare Custom Entry* **com** tells Rocq to create a new "custom grammar" for parsing Imp expressions and programs. The first notation declaration after this tells Rocq that anything between <{ and }> should be parsed using the Imp grammar. Again, it is not necessary to understand the details, but it is important to recognize that we are defining *new* interpretations for some familiar operators like +, -, ×, =, ≤, etc., when they occur between <{ and }>.

```
Coercion AId : string >-> aexp.
Coercion ANum : nat >-> aexp.

Declare Custom Entry com.
Declare Scope com_scope.

Notation "<{ e }>" := e
  (e custom com, format "'[hv' <{ '/ ' '[v' e ']' '/' }> ']'") : com_scope.

Notation "( x )" := x (in custom com, x at level 99).
Notation "x" := x (in custom com at level 0, x constr at level 0).
```

```
Notation "f x .. y" := (.. (f x) .. y)
                    (in custom com at level 0, only parsing,
                    f constr at level 0, x constr at level 1,
                        y constr at level 1).
Notation "x + y" := (APlus x y) (in custom com at level 50, left associativity).
Notation "x - y" := (AMinus x y) (in custom com at level 50, left associativity).
Notation "x * y" := (AMult x y) (in custom com at level 40, left associativity).
Notation "'true'" := true (at level 1).
Notation "'true'" := BTrue (in custom com at level 0).
Notation "'false'" := false (at level 1).
Notation "'false'" := BFalse (in custom com at level 0).
Notation "x <= y" := (BLe x y) (in custom com at level 70, no associativity).
Notation "x > y" := (BGt x y) (in custom com at level 70, no associativity).
Notation "x = y" := (BEq x y) (in custom com at level 70, no associativity).
Notation "x <> y" := (BNeq x y) (in custom com at level 70, no associativity).
Notation "x && y" := (BAnd x y) (in custom com at level 80, left associativity).
Notation "'~' b" := (BNot b) (in custom com at level 75, right associativity).

Open Scope com_scope.
```

We can now write 3 + (X × 2) instead of APlus 3 (AMult X 2), and true && ~(X ≤ 4) instead of BAnd true (BNot (BLe X 4)).

```
Definition example_aexp : aexp := <{ 3 + (X × 2) }>.
Definition example_bexp : bexp := <{ true && ¬(X ≤ 4) }>.
```

2.5.4 Evaluation

The arith and boolean evaluators must now be extended to handle variables in the obvious way, taking a state *st* as an extra argument:

```
Fixpoint aeval (st : state)
               (a : aexp) : nat :=
  match a with
  | ANum n ⇒ n
  | AId x ⇒ st x
  | <{a1 + a2}> ⇒ (aeval st a1) + (aeval st a2)
  | <{a1 - a2}> ⇒ (aeval st a1) - (aeval st a2)
  | <{a1 × a2}> ⇒ (aeval st a1) × (aeval st a2)
  end.

Fixpoint beval (st : state)
               (b : bexp) : bool :=
  match b with
  | <{true}> ⇒ true
  | <{false}> ⇒ false
```

```
  | <{a1 = a2}> ⇒ (aeval st a1) =? (aeval st a2)
  | <{a1 ≠ a2}> ⇒ negb ((aeval st a1) =? (aeval st a2))
  | <{a1 ≤ a2}> ⇒ (aeval st a1) <=? (aeval st a2)
  | <{a1 > a2}> ⇒ negb ((aeval st a1) <=? (aeval st a2))
  | <{¬ b1}> ⇒ negb (beval st b1)
  | <{b1 && b2}> ⇒ andb (beval st b1) (beval st b2)
  end.
```

We can use our notation for total maps in the specific case of states – i.e., we write the empty state as (__ !-> 0).

```
Definition empty_st := (__ !-> 0).
```

Also, we can add a notation for a "singleton state" with just one variable bound to a value.

```
Notation "x '!->' v" := (x !-> v ; empty_st) (at level 100, right associativity).

Example aexp1 :
    aeval (X !-> 5) <{ 3 + (X × 2) }>
  = 13.
Proof. reflexivity. Qed.
Example aexp2 :
    aeval (X !-> 5 ; Y !-> 4) <{ Z + (X × Y) }>
  = 20.
Proof. reflexivity. Qed.

Example bexp1 :
    beval (X !-> 5) <{ true && ¬(X ≤ 4) }>
  = true.
Proof. reflexivity. Qed.
```

2.6 Commands

Now we are ready to define the syntax and behavior of Imp *commands* (or *statements*).

2.6.1 Syntax

Informally, commands c are described by the following BNF grammar.

c := skip | x := a | c ; c | if b then c else c end | while b do c end

Here is the formal definition of the abstract syntax of commands:

```
Inductive com : Type :=
  | CSkip
  | CAsgn (x : string) (a : aexp)
  | CSeq (c1 c2 : com)
  | CIf (b : bexp) (c1 c2 : com)
  | CWhile (b : bexp) (c : com).
```

As we did for expressions, we can use a few `Notation` declarations to make reading and writing Imp programs more convenient.

```
Notation "'skip'" := CSkip
  (in custom com at level 0) : com_scope.
Notation "x := y" := (CAsgn x y)
  (in custom com at level 0, x constr at level 0, y at level 85, no associativity,
    format "x := y") : com_scope.
Notation "x ; y" := (CSeq x y)
  (in custom com at level 90,
    right associativity,
    format "'[v' x ; '/' y ']'") : com_scope.
Notation "'if' x 'then' y 'else' z 'end'" := (CIf x y z)
  (in custom com at level 89, x at level 99, y at level 99, z at level 99,
    format "'[v' 'if' x 'then' '/  ' y '/' 'else' '/  ' z '/' 'end' ']'") : com_scope.
Notation "'while' x 'do' y 'end'" := (CWhile x y)
  (in custom com at level 89, x at level 99, y at level 99,
    format "'[v' 'while' x 'do' '/  ' y '/' 'end' ']'") : com_scope.
```

For example, here is the factorial function again, written as a formal Rocq definition. When this command terminates, the variable Y will contain the factorial of the initial value of X.

```
Definition fact_in_coq : com :=
  <{ Z := X;
     Y := 1;
     while Z ≠ 0 do
       Y := Y × Z;
       Z := Z - 1
     end }>.

Print fact_in_coq.
```

2.6.2 Desugaring Notations

Rocq offers a rich set of features to manage the increasing complexity of the objects we work with, such as coercions and notations. However, their heavy usage can make it hard to understand what the expressions we enter actually mean. In such situations it is often instructive to "turn off" those features to get a more elementary picture of things, using the following commands:

- `Unset Printing Notations` (undo with `Set Printing Notations`)
- `Set Printing Coercions` (undo with `Unset Printing Coercions`)
- `Set Printing All` (undo with `Unset Printing All`)

These commands can also be used in the middle of a proof, to elaborate the current goal and context.

```
Unset Printing Notations.
Print fact_in_coq.
Set Printing Notations.

Print example_bexp.

Set Printing Coercions.
Print example_bexp.

Print fact_in_coq.
Unset Printing Coercions.
```

2.6.3 `Locate` Again

Finding identifiers

When used with an identifier, the `Locate` prints the full path to every value in scope with the same name. This is useful to troubleshoot problems due to variable shadowing. `Locate aexp`.

Finding notations

When faced with an unknown notation, you can use `Locate` with a string containing one of its symbols to see its possible interpretations. `Locate "&&".`

```
Locate ";".

Locate "while".
```

2.6.4 More Examples

Assignment:

```
Definition plus2 : com :=
  <{ X := X + 2 }>.

Definition XtimesYinZ : com :=
  <{ Z := X × Y }>.
```

Loops

```
Definition subtract_slowly_body : com :=
  <{ Z := Z - 1 ;
     X := X - 1 }>.

Definition subtract_slowly : com :=
```

```
  <{ while X ≠ 0 do
       subtract_slowly_body
     end }>.
Definition subtract_3_from_5_slowly : com :=
  <{ X := 3 ;
     Z := 5 ;
     subtract_slowly }>.
```

An infinite loop:

```
Definition loop : com :=
  <{ while true do
       skip
     end }>.
```

2.7 Evaluating Commands

Next we need to define what it means to evaluate an Imp command. The fact that *while* loops don't necessarily terminate makes defining an evaluation function tricky...

2.7.1 Evaluation as a Function (Failed Attempt)

Here's an attempt at defining an evaluation function for commands (with a bogus *while* case).

```
Fixpoint ceval_fun_no_while (st : state) (c : com) : state :=
  match c with
    | <{ skip }> ⇒
        st
    | <{ x := a }> ⇒
        (x !-> aeval st a ; st)
    | <{ c1 ; c2 }> ⇒
        let st' := ceval_fun_no_while st c1 in
        ceval_fun_no_while st' c2
    | <{ if b then c1 else c2 end}> ⇒
        if (beval st b)
          then ceval_fun_no_while st c1
          else ceval_fun_no_while st c2
    | <{ while b do c end }> ⇒
        st
  end.
```

In a more conventional functional programming language like OCaml or Haskell we could add the *while* case as follows:

Fixpoint ceval_fun (st : state) (c : com) : state := match c with ... | <{ while b do c end}> => if (beval st b) then ceval_fun st <{c ; while b do c end}> else st end.

Rocq doesn't accept such a definition ("Error: Cannot guess decreasing argument of fix") because the function we want to define is not guaranteed to terminate. Indeed, it *doesn't* always terminate: for example, the full version of the *ceval_fun* function applied to the loop program above would never terminate. Since Rocq aims to be not just a functional programming language but also a consistent logic, any potentially non-terminating function needs to be rejected.

Here is an example showing what would go wrong if Rocq allowed non-terminating recursive functions:

Fixpoint loop_false (n : nat) : False := loop_false n.

That is, propositions like **False** would become provable (*loop_false* 0 would be a proof of **False**), which would be a disaster for Rocq's logical consistency.

Thus, because it doesn't terminate on all inputs, *ceval_fun* cannot be written in Rocq – at least not without additional tricks and workarounds (see chapter *ImpCEvalFun* if you're curious about those).

2.7.2 Evaluation as a Relation

Here's a better way: define **ceval** as a *relation* rather than a *function* – i.e., make its result a `Prop` rather than a state, similar to what we did for **aevalR** above.

This is an important change. Besides freeing us from awkward workarounds, it gives us a ton more flexibility in the definition. For example, if we add nondeterministic features like *any* to the language, we want the definition of evaluation to be nondeterministic – i.e., not only will it not be total, it will not even be a function!

We'll use the notation *st* =[*c*]=> *st'* for the **ceval** relation: *st* =[*c*]=> *st'* means that executing program *c* in a starting state *st* results in an ending state *st'*. This can be pronounced "*c* takes state *st* to *st'*".

Operational Semantics

Here is an informal definition of evaluation, presented as inference rules for readability:

(E_Skip) st = *skip* => st
  aeval st a = n

(E_Asgn) st = *x* := *a* => (x !-> n ; st)
  st = *c1* => st' st' = *c2* => st''

(E_Seq) st = *c1*;*c2* => st''
  beval st b = true st = *c1* => st'

(E_IfTrue) st = `if` *b* `then` *c1* `else` *c2* `end` => st'
beval st b = false st = *c2* => st'

(E_IfFalse) st = `if` *b* `then` *c1* `else` *c2* `end` => st'
beval st b = true

(E_WhileFalse) st = *while* *b* `do` *c* `end` => st
beval st b = true st = *c* => st' st' = *while* *b* `do` *c* `end` => st''

(E_WhileTrue) st = *while* *b* `do` *c* `end` => st''

Here is the formal definition. Make sure you understand how it corresponds to the inference rules.

```
Reserved Notation
         "st0 '=[' c ']=>' st1"
         (at level 40, c custom com at level 99,
          st0 constr, st1 constr at next level,
          format "'[hv' st0 =[ '/ ' '[' c ']' '/' ]=> st1 ']'").

Inductive ceval : com → state → state → Prop :=
  | E_Skip : ∀ st,
      st =[ skip ]=> st
  | E_Asgn : ∀ st a n x,
      aeval st a = n →
      st =[ x := a ]=> (x !-> n ; st)
  | E_Seq : ∀ c1 c2 st st' st'',
      st =[ c1 ]=> st' →
      st' =[ c2 ]=> st'' →
      st =[ c1 ; c2 ]=> st''
  | E_IfTrue : ∀ st st' b c1 c2,
      beval st b = true →
      st =[ c1 ]=> st' →
      st =[ if b then c1 else c2 end]=> st'
  | E_IfFalse : ∀ st st' b c1 c2,
      beval st b = false →
      st =[ c2 ]=> st' →
      st =[ if b then c1 else c2 end]=> st'
  | E_WhileFalse : ∀ b st c,
      beval st b = false →
      st =[ while b do c end ]=> st
  | E_WhileTrue : ∀ st st' st'' b c,
      beval st b = true →
      st =[ c ]=> st' →
```

```
      st' =[ while b do c end ]=> st'' →
      st =[ while b do c end ]=> st''

  where "st0 =[ c ]=> st1" := (ceval c st0 st1).
```

The cost of defining evaluation as a relation instead of a function is that we now need to construct a *proof* that some program evaluates to some result state, rather than just letting Rocq's computation mechanism do it for us.

```
Example ceval_example1:
  empty_st =[
     X := 2;
     if (X ≤ 1)
       then Y := 3
       else Z := 4
     end
  ]=> (Z !-> 4 ; X !-> 2).
Proof.
  apply E_Seq with (X !-> 2).
  -
    apply E_Asgn. reflexivity.
  -
    apply E_IfFalse.
    + reflexivity.
    + apply E_Asgn. reflexivity.
Qed.
```

Exercise: 2 stars, standard (ceval_example2) `Example ceval_example2:`

```
  empty_st =[
    X := 0;
    Y := 1;
    Z := 2
  ]=> (Z !-> 2 ; Y !-> 1 ; X !-> 0).
Proof.
   Admitted.
```

□

```
Set Printing Implicit.
Check @ceval_example2.
```

Exercise: 3 stars, standard, optional (pup_to_n) Write an Imp program that sums the numbers from 1 to X (inclusive: 1 + 2 + ... + X) in the variable Y. Your program should update the state as shown in theorem pup_to_2_ceval, which you can reverse-engineer to discover the program you should write. The proof of that theorem will be somewhat lengthy.

```
Definition pup_to_n : com
  . Admitted.

Theorem pup_to_2_ceval :
  (X !-> 2) =[
    pup_to_n
  ]=> (X !-> 0 ; Y !-> 3 ; X !-> 1 ; Y !-> 2 ; Y !-> 0 ; X !-> 2).
Proof.
    Admitted.
    □
```

2.7.3 Determinism of Evaluation

Changing from a computational to a relational definition of evaluation is a good move because it frees us from the artificial requirement that evaluation should be a total function. But it also raises a question: Is the second definition of evaluation really a partial *function*? Or is it possible that, beginning from the same state *st*, we could evaluate some command *c* in different ways to reach two different output states *st'* and *st''*?

In fact, this cannot happen: **ceval** *is* a partial function:

```
Theorem ceval_deterministic: ∀ c st st1 st2,
     st =[ c ]=> st1 →
     st =[ c ]=> st2 →
     st1 = st2.
Proof.
  intros c st st1 st2 E1 E2.
  generalize dependent st2.
  induction E1; intros st2 E2; inversion E2; subst.
  - reflexivity.
  - reflexivity.
  -
    rewrite (IHE1_1 st'0 H1) in *.
    apply IHE1_2. assumption.
  -
      apply IHE1. assumption.
  -
      rewrite H in H5. discriminate.
  -
      rewrite H in H5. discriminate.
  -
      apply IHE1. assumption.
  -
    reflexivity.
  -
```

```
    rewrite H in H2. discriminate.
  -
    rewrite H in H4. discriminate.
  -
    rewrite (IHE1_1 st'0 H3) in *.
    apply IHE1_2. assumption. Qed.
```

2.8 Reasoning About Imp Programs

We'll get into more systematic and powerful techniques for reasoning about Imp programs in *Programming Language Foundations*, but we can already do a few things (albeit in a somewhat low-level way) just by working with the bare definitions. This section explores some examples.

```
Theorem plus2_spec : ∀ st n st',
  st X = n →
  st =[ plus2 ]=> st' →
  st' X = n + 2.
Proof.
  intros st n st' HX Heval.
```

Inverting *Heval* essentially forces Rocq to expand one step of the **ceval** computation – in this case revealing that *st'* must be *st* extended with the new value of X, since plus2 is an assignment.

```
  inversion Heval. subst. clear Heval. simpl.
  apply t_update_eq. Qed.
```

Exercise: 3 stars, standard, optional (XtimesYinZ_spec) State and prove a specification of XtimesYinZ.

```
Definition manual_grade_for_XtimesYinZ_spec : option (nat×string) := None.
```

□

Exercise: 3 stars, standard, especially useful (loop_never_stops)

```
Theorem loop_never_stops
: ∀ st st',
  ~(st =[ loop ]=> st').
Proof.
  intros st st' contra. unfold loop in contra.
  remember <{ while true do skip end }> as loopdef
           eqn:Heqloopdef .
```

Proceed by induction on the assumed derivation showing that *loopdef* terminates. Most of the cases are immediately contradictory and so can be solved in one step with `discriminate`.

```
  Admitted.
```

□

Exercise: 3 stars, standard (no_whiles_eqv) Consider the following function:

```
Fixpoint no_whiles (c : com) : bool :=
  match c with
  | <{ skip }> ⇒
      true
  | <{ _ := _ }> ⇒
      true
  | <{ c1 ; c2 }> ⇒
      andb (no_whiles c1) (no_whiles c2)
  | <{ if _ then ct else cf end }> ⇒
      andb (no_whiles ct) (no_whiles cf)
  | <{ while _ do _ end }> ⇒
      false
  end.
```

This predicate yields `true` just on programs that have no while loops. Using `Inductive`, write a property **no_whilesR** such that **no_whilesR** *c* is provable exactly when *c* is a program with no while loops. Then prove its equivalence with no_whiles.

```
Inductive no_whilesR: com → Prop :=

.

Theorem no_whiles_eqv:
  ∀ c, no_whiles c = true ↔ no_whilesR c.
Proof.
  Admitted.
```

□

Exercise: 4 stars, standard (no_whiles_terminating) Imp programs that don't involve while loops always terminate. State and prove a theorem *no_whiles_terminating* that says this.

Use either no_whiles or **no_whilesR**, as you prefer.

```
Definition manual_grade_for_no_whiles_terminating : option (nat×string) := None.
```

□

2.9 Additional Exercises

Exercise: 3 stars, standard (stack_compiler) Old HP Calculators, programming languages like Forth and Postscript, and abstract machines like the Java Virtual Machine all evaluate arithmetic expressions using a *stack*. For instance, the expression

(2*3)+(3*(4-2))
would be written as
2 3 * 3 4 2 - * +
and evaluated like this (where we show the program being evaluated on the right and the contents of the stack on the left):
| 2 3 * 3 4 2 - * + 2 | 3 * 3 4 2 - * + 3, 2 | * 3 4 2 - * + 6 | 3 4 2 - * + 3, 6 | 4 2 - * + 4, 3, 6 | 2 - * + 2, 4, 3, 6 | - * + 2, 3, 6 | * + 6, 6 | + 12 |

The goal of this exercise is to write a small compiler that translates **aexp**s into stack machine instructions.

The instruction set for our stack language will consist of the following instructions:

- SPush n: Push the number n on the stack.
- SLoad x: Load the identifier x from the store and push it on the stack
- SPlus: Pop the two top numbers from the stack, add them, and push the result onto the stack.
- SMinus: Similar, but subtract the first number from the second.
- SMult: Similar, but multiply.

```
Inductive sinstr : Type :=
| SPush (n : nat)
| SLoad (x : string)
| SPlus
| SMinus
| SMult.
```

Write a function to evaluate programs in the stack language. It should take as input a state, a stack represented as a list of numbers (top stack item is the head of the list), and a program represented as a list of instructions, and it should return the stack after executing the program. Test your function on the examples below.

Note that it is unspecified what to do when encountering an SPlus, SMinus, or SMult instruction if the stack contains fewer than two elements. In a sense, it is immaterial what we do, since a correct compiler will never emit such a malformed program. But for sake of later exercises, it would be best to skip the offending instruction and continue with the next one.

```
Fixpoint s_execute (st : state) (stack : list nat)
                   (prog : list sinstr)
                 : list nat
  . Admitted.

Check s_execute.

Example s_execute1 :
```

```
    s_execute empty_st []
      [SPush 5; SPush 3; SPush 1; SMinus]
  = [2; 5].
  Admitted.

Example s_execute2 :
    s_execute (X !-> 3) [3;4]
      [SPush 4; SLoad X; SMult; SPlus]
  = [15; 4].
  Admitted.
```

Next, write a function that compiles an **aexp** into a stack machine program. The effect of running the program should be the same as pushing the value of the expression on the stack.

```
Fixpoint s_compile (e : aexp) : list sinstr
  . Admitted.
```

After you've defined s_compile, prove the following to test that it works.

```
Example s_compile1 :
  s_compile <{ X - (2 × Y) }>
  = [SLoad X; SPush 2; SLoad Y; SMult; SMinus].
  Admitted.
```

□

Exercise: 3 stars, standard (execute_app) Execution can be decomposed in the following sense: executing stack program p1 ++ p2 is the same as executing p1, taking the resulting stack, and executing p2 from that stack. Prove that fact.

```
Theorem execute_app : ∀ st p1 p2 stack,
  s_execute st stack (p1 ++ p2) = s_execute st (s_execute st stack p1) p2.
Proof.
  Admitted.
```

□

Exercise: 3 stars, standard (stack_compiler_correct) Now we'll prove the correctness of the compiler implemented in the previous exercise. Begin by proving the following lemma. If it becomes difficult, consider whether your implementation of s_execute or s_compile could be simplified.

```
Lemma s_compile_correct_aux : ∀ st e stack,
  s_execute st stack (s_compile e) = aeval st e :: stack.
Proof.
  Admitted.
```

The main theorem should be a very easy corollary of that lemma.

```
Theorem s_compile_correct : ∀ (st : state) (e : aexp),
  s_execute st [] (s_compile e) = [ aeval st e ].
Proof.
    Admitted.
```

□

Exercise: 3 stars, standard, optional (short_circuit) Most modern programming languages use a "short-circuit" evaluation rule for boolean **and**: to evaluate BAnd *b1* *b2*, first evaluate *b1*. If it evaluates to false, then the entire BAnd expression evaluates to false immediately, without evaluating *b2*. Otherwise, *b2* is evaluated to determine the result of the BAnd expression.

Write an alternate version of beval that performs short-circuit evaluation of BAnd in this manner, and prove that it is equivalent to beval. (N.b. This is only true because expression evaluation in Imp is rather simple. In a bigger language where evaluating an expression might diverge, the short-circuiting BAnd would *not* be equivalent to the original, since it would make more programs terminate.)

```
Module BreakImp.
```

Exercise: 4 stars, standard, optional (break_imp) Imperative languages like C and Java often include a *break* or similar statement for interrupting the execution of loops. In this exercise we consider how to add *break* to Imp. First, we need to enrich the language of commands with an additional case.

```
Inductive com : Type :=
  | CSkip
  | CBreak
  | CAsgn (x : string) (a : aexp)
  | CSeq (c1 c2 : com)
  | CIf (b : bexp) (c1 c2 : com)
  | CWhile (b : bexp) (c : com).

Notation "'break'" := CBreak (in custom com at level 0) : com_scope.
Notation "'skip'" := CSkip
  (in custom com at level 0) : com_scope.
Notation "x := y" := (CAsgn x y)
  (in custom com at level 0, x constr at level 0, y at level 85, no associativity,
    format "x := y") : com_scope.
Notation "x ; y" := (CSeq x y)
  (in custom com at level 90,
    right associativity,
    format "'[v' x ; '/' y ']'") : com_scope.
Notation "'if' x 'then' y 'else' z 'end'" := (CIf x y z)
  (in custom com at level 89, x at level 99, y at level 99, z at level 99,
```

```
    format "'[v' 'if' x 'then' '/ ' y '/' 'else' '/ ' z '/' 'end' ']'") : com_scope.
Notation "'while' x 'do' y 'end'" := (CWhile x y)
  (in custom com at level 89, x at level 99, y at level 99,
    format "'[v' 'while' x 'do' '/ ' y '/' 'end' ']'") : com_scope.
```

Next, we need to define the behavior of *break*. Informally, whenever *break* is executed in a sequence of commands, it stops the execution of that sequence and signals that the innermost enclosing loop should terminate. (If there aren't any enclosing loops, then the whole program simply terminates.) The final state should be the same as the one in which the *break* statement was executed.

One important point is what to do when there are multiple loops enclosing a given *break*. In those cases, *break* should only terminate the *innermost* loop. Thus, after executing the following...

X := 0; Y := 1; while 0 <> Y do while true do break end; X := 1; Y := Y - 1 end

... the value of X should be 1, and not 0.

One way of expressing this behavior is to add another parameter to the evaluation relation that specifies whether evaluation of a command executes a *break* statement:

```
Inductive result : Type :=
  | SContinue
  | SBreak.

Reserved Notation
         "st0 '=[' c ']=>' st1 '/' s"
         (at level 40, c custom com at level 99,
          st0 constr, st1 constr at next level,
          format "'[hv' st0 =[ '/ ' '[' c ']' '/' ]=> st1 / s ']'").
```

Intuitively, *st* =[*c*]=> *st'* / *s* means that, if *c* is started in state *st*, then it terminates in state *st'* and either signals that the innermost surrounding loop (or the whole program) should exit immediately (*s* = SBreak) or that execution should continue normally (*s* = SContinue).

The definition of the "*st* =[*c*]=> *st'* / *s*" relation is very similar to the one we gave above for the regular evaluation relation (*st* =[*c*]=> *st'*) – we just need to handle the termination signals appropriately:

- If the command is *skip*, then the state doesn't change and execution of any enclosing loop can continue normally.
- If the command is *break*, the state stays unchanged but we signal a SBreak.
- If the command is an assignment, then we update the binding for that variable in the state accordingly and signal that execution can continue normally.
- If the command is of the form `if b then c1 else c2 end`, then the state is updated as in the original semantics of Imp, except that we also propagate the signal from the execution of whichever branch was taken.

- If the command is a sequence *c1* ; *c2*, we first execute *c1*. If this yields a SBreak, we skip the execution of *c2* and propagate the SBreak signal to the surrounding context; the resulting state is the same as the one obtained by executing *c1* alone. Otherwise, we execute *c2* on the state obtained after executing *c1*, and propagate the signal generated there.

- Finally, for a loop of the form *while b* `do` *c* `end`, the semantics is almost the same as before. The only difference is that, when *b* evaluates to true, we execute *c* and check the signal that it raises. If that signal is SContinue, then the execution proceeds as in the original semantics. Otherwise, we stop the execution of the loop, and the resulting state is the same as the one resulting from the execution of the current iteration. In either case, since *break* only terminates the innermost loop, *while* signals SContinue.

Based on the above description, complete the definition of the **ceval** relation.

```
Inductive ceval : com → state → result → state → Prop :=
  | E_Skip : ∀ st,
      st =[ CSkip ]=> st / SContinue


  where "st '=[' c ']=>' st' '/' s" := (ceval c st s st').
```

Now prove the following properties of your definition of **ceval**:

```
Theorem break_ignore : ∀ c st st' s,
     st =[ break; c ]=> st' / s →
     st = st'.
Proof.
    Admitted.

Theorem while_continue : ∀ b c st st' s,
  st =[ while b do c end ]=> st' / s →
  s = SContinue.
Proof.
    Admitted.

Theorem while_stops_on_break : ∀ b c st st',
  beval st b = true →
  st =[ c ]=> st' / SBreak →
  st =[ while b do c end ]=> st' / SContinue.
Proof.
    Admitted.

Theorem seq_continue : ∀ c1 c2 st st' st'',
  st =[ c1 ]=> st' / SContinue →
  st' =[ c2 ]=> st'' / SContinue →
  st =[ c1 ; c2 ]=> st'' / SContinue.
```

```
Proof.
    Admitted.

Theorem seq_stops_on_break : ∀ c1 c2 st st',
  st =[ c1 ]=> st' / SBreak →
  st =[ c1 ; c2 ]=> st' / SBreak.
Proof.
    Admitted.
    □
```

Exercise: 3 stars, advanced, optional (while_break_true) `Theorem while_break_true : ∀ b c st st',`

```
  st =[ while b do c end ]=> st' / SContinue →
  beval st' b = true →
  ∃ st'', st'' =[ c ]=> st' / SBreak.
Proof.
    Admitted.
    □
```

Exercise: 4 stars, advanced, optional (ceval_deterministic) `Theorem ceval_deterministic: ∀ (c:com) st st1 st2 s1 s2,`

```
      st =[ c ]=> st1 / s1 →
      st =[ c ]=> st2 / s2 →
      st1 = st2 ∧ s1 = s2.
Proof.
    Admitted.

    □ End BreakImp.
```

Exercise: 4 stars, standard, optional (add_for_loop) Add C-style `for` loops to the language of commands, update the **ceval** definition to define the semantics of `for` loops, and add cases for `for` loops as needed so that all the proofs in this file are accepted by Rocq.

A `for` loop should be parameterized by (a) a statement executed initially, (b) a test that is run on each iteration of the loop to determine whether the loop should continue, (c) a statement executed at the end of each loop iteration, and (d) a statement that makes up the body of the loop. (You don't need to worry about making up a concrete Notation for `for` loops, but feel free to play with this too if you like.)

Chapter 3

Library SECF.Equiv

3.1 Equiv: Program Equivalence

```
Set Warnings "-notation-overridden".
From SECF Require Import Maps.
From Stdlib Require Import Bool.
From Stdlib Require Import Arith.
From Stdlib Require Import Init.Nat.
From Stdlib Require Import PeanoNat. Import Nat.
From Stdlib Require Import EqNat.
From Stdlib Require Import Lia.
From Stdlib Require Import List. Import ListNotations.
From Stdlib Require Import FunctionalExtensionality.
From SECF Require Export Imp.
```

Before You Get Started:

- Create a fresh directory for this volume. Do not try to mix the files from this volume with files from *Logical Foundations* in the same directory: the result will not make you happy.

  You can either start with an empty directory and populate it with the files listed below (and others as you go along), or else download the whole PLF zip file and unzip it.

- The new directory should contain at least the following files:

  - *Imp.v* (make sure it is the one from the PLF distribution, not the one from LF: they are slightly different);
  - *Maps.v* (ditto)
  - *Equiv.v* (this file)
  - *_CoqProject*, containing the following line:

∗ Q . PLF

- If you see errors like this...

  Compiled library PLF.Maps (in file .../plf/Maps.vo) makes inconsistent assumptions over library Rocq.Init.Logic

  ... it may mean something went wrong with the above steps. Doing "*make clean*" (or manually removing everything except *.v* and *_CoqProject* files) may help.

- If you are using VSCode with the VSCoq extension, you'll then want to open a new window in VSCode, click `Open` *Folder* > *plf*, and run *make*. If you get an error like "Cannot find a physical path..." error, it may be because you didn't open plf directly (you instead opened a folder containing both lf and plf, for example).

Advice for Working on Exercises:

- Most of the Rocq proofs we ask you to do in this chapter are similar to proofs that we've provided. Before starting to work on exercises, take the time to work through our proofs (both informally and in Rocq) and make sure you understand them in detail. This will save you a lot of effort.

- The Rocq proofs we're doing now are sufficiently complicated that it is more or less impossible to complete them by random experimentation or following your nose. You need to start with an idea about why the property is true and how the proof is going to go. The best way to do this is to write out at least a sketch of an informal proof on paper – one that intuitively convinces you of the truth of the theorem – before starting to work on the formal one. Alternately, grab a friend and try to convince them that the theorem is true; then try to formalize your explanation.

- Use automation to save work! The proofs in this chapter can get pretty long if you try to write out all the cases explicitly.

3.2 Behavioral Equivalence

In an earlier chapter, we investigated the correctness of a very simple program transformation: the optimize_0plus function. The programming language we were considering was the first version of the language of arithmetic expressions – with no variables – so in that setting it was very easy to define what it means for a program transformation to be correct: it should always yield a program that evaluates to the same number as the original.

To talk about the correctness of program transformations for the full Imp language – in particular, assignment – we need to consider the role of mutable state and develop a more sophisticated notion of correctness, which we'll call *behavioral equivalence*.

For example:

- X + 2 is behaviorally equivalent to 1 + X + 1
- X - X is behaviorally equivalent to 0
- (X - 1) + 1 is *not* behaviorally equivalent to X

3.2.1 Definitions

For **aexp**s and **bexp**s with variables, the definition we want is clear: Two **aexp**s or **bexp**s are "behaviorally equivalent" if they evaluate to the same result in every state.

```
Definition aequiv (a1 a2 : aexp) : Prop :=
  ∀ (st : state),
    aeval st a1 = aeval st a2.

Definition bequiv (b1 b2 : bexp) : Prop :=
  ∀ (st : state),
    beval st b1 = beval st b2.
```

Here are some simple examples of equivalences of arithmetic and boolean expressions.

```
Theorem aequiv_example:
  aequiv
    <{ X - X }>
    <{ 0 }>.
Proof.
  intros st. simpl. lia.
Qed.

Theorem bequiv_example:
  bequiv
    <{ X - X = 0 }>
    <{ true }>.
Proof.
  intros st. unfold beval.
  rewrite aequiv_example. reflexivity.
Qed.
```

For commands, the situation is a little more subtle. We can't simply say "two commands are behaviorally equivalent if they evaluate to the same ending state whenever they are started in the same initial state," because some commands, when run in some starting states, don't terminate in any final state at all!

What we need instead is this: two commands are behaviorally equivalent if, for any given starting state, they either (1) both diverge or else (2) both terminate in the same final state. A compact way to express this is "if the first one terminates in a particular state then so does the second, and vice versa."

```
Definition cequiv (c1 c2 : com) : Prop :=
  ∀ (st st' : state),
    (st =[ c1 ]=> st') ↔ (st =[ c2 ]=> st').
```

3.2.2 Simple Examples

For examples of command equivalence, let's start by looking at a trivial equivalence involving *skip*:

```
Theorem skip_left : ∀ c,
  cequiv
    <{ skip; c }>
    c.
Proof.
  intros c st st'.
  split; intros H.
  -
    inversion H. subst.
    inversion H2. subst.
    assumption.
  -
    apply E_Seq with st.
    + apply E_Skip.
    + assumption.
Qed.
```

Exercise: 2 stars, standard (skip_right) Prove that adding a *skip* *after* a command also results in an equivalent program

```
Theorem skip_right : ∀ c,
  cequiv
    <{ c; skip }>
    c.
Proof.
  Admitted.
```

□

Similarly, here is a simple equivalence that optimizes `if` commands:

```
Theorem if_true_simple : ∀ c1 c2,
  cequiv
    <{ if true then c1 else c2 end }>
    c1.
Proof.
  intros c1 c2.
  split; intros H.
```

```
- inversion H; subst.
  + assumption.
  + discriminate.
- apply E_IfTrue.
  + reflexivity.
  + assumption. Qed.
```

Of course, no programmer would write a conditional whose condition is literally true. (At least, no human programmer – compilers and macro preprocessors do this sort of thing internally all the time!) But they might write one whose condition is *equivalent* to true:

Theorem: If b is equivalent to true, then `if b then c1 else c2 end` is equivalent to $c1$.
Proof:

- ($\rightarrow$) We must show, for all st and st', that if st =[`if b then c1 else c2 end`]=> st' then st =[$c1$]=> st'.

  Proceed by cases on the rules that could possibly have been used to show st =[`if b then c1 else c2 end`]=> st', namely E_IfTrue and E_IfFalse.

  - Suppose the final rule in the derivation of st =[`if b then c1 else c2 end`]=> st' was E_IfTrue. We then have, by the premises of E_IfTrue, that st =[$c1$]=> st'. This is exactly what we set out to prove.
  - On the other hand, suppose the final rule in the derivation of st =[`if b then c1 else c2 end`]=> st' was E_IfFalse. We then know that beval st b = false and st =[$c2$]=> st'.

    Recall that b is equivalent to true, i.e., forall st, beval st b = beval st <{true}> . In particular, this means that beval st b = true, since beval st <{true}> = true. But this is a contradiction, since E_IfFalse requires that beval st b = false. Thus, the final rule could not have been E_IfFalse.

- ($\leftarrow$) We must show, for all st and st', that if st =[$c1$]=> st' then st =[`if b then c1 else c2 end`]=> st'.

  Since b is equivalent to true, we know that beval st b = beval st <{true}> = true. Together with the assumption that st =[$c1$]=> st', we can apply E_IfTrue to derive st =[`if b then c1 else c2 end`]=> st'. □

Here is the formal version of this proof:

```
Theorem if_true: ∀ b c1 c2,
  bequiv b <{true}> →
  cequiv
    <{ if b then c1 else c2 end }>
    c1.
```

```
Proof.
  intros b c1 c2 Hb.
  split; intros H.
  -
    inversion H; subst.
    +
      assumption.
    +
      unfold bequiv in Hb. simpl in Hb.
      rewrite Hb in H5.
      discriminate.
  -
    apply E_IfTrue; try assumption.
    unfold bequiv in Hb. simpl in Hb.
    apply Hb. Qed.
```

Exercise: 2 stars, standard, especially useful (if_false) `Theorem if_false : ∀ b c1 c2,`

```
  bequiv b <{false}> →
  cequiv
    <{ if b then c1 else c2 end }>
    c2.
Proof.
  Admitted.
```

□

Exercise: 3 stars, standard (swap_if_branches) Show that we can swap the branches of an `if` if we also negate its condition.

```
Theorem swap_if_branches : ∀ b c1 c2,
  cequiv
    <{ if b then c1 else c2 end }>
    <{ if ¬ b then c2 else c1 end }>.
Proof.
  Admitted.
```

□

For *while* loops, we can give a similar pair of theorems. A loop whose guard is equivalent to false is equivalent to *skip*, while a loop whose guard is equivalent to true is equivalent to *while* true do *skip* end (or any other non-terminating program).

The first of these facts is easy.

```
Theorem while_false : ∀ b c,
  bequiv b <{false}> →
  cequiv
```

```
    <{ while b do c end }>
    <{ skip }>.
Proof.
  intros b c Hb. split; intros H.
  -
    inversion H; subst.
    +
      apply E_Skip.
    +
      rewrite Hb in H2. discriminate.
  -
    inversion H. subst.
    apply E_WhileFalse.
    apply Hb. Qed.
```

Exercise: 2 stars, advanced, optional (while_false_informal) Write an informal proof of while_false.

□

To prove the second fact, we need an auxiliary lemma stating that *while* loops whose guards are equivalent to true never terminate.

Lemma: If *b* is equivalent to true, then it cannot be the case that *st* =[*while b* do *c* end]=> *st'*.

Proof: Suppose that *st* =[*while b* do *c* end]=> *st'*. We show, by induction on a derivation of *st* =[*while b* do *c* end]=> *st'*, that this assumption leads to a contradiction. The only two cases to consider are E_WhileFalse and E_WhileTrue; the others are contradictory.

- Suppose *st* =[*while b* do *c* end]=> *st'* is proved using rule E_WhileFalse. Then by assumption beval *st b* = false. But this contradicts the assumption that *b* is equivalent to true.

- Suppose *st* =[*while b* do *c* end]=> *st'* is proved using rule E_WhileTrue. We must have:

  1. beval *st b* = true, and 2. there is some *st0* such that *st* =[*c*]=> *st0* and *st0* =[*while b* do *c* end]=> *st'*. 3. Also, we are given an induction hypothesis saying that *st0* =[*while b* do *c* end]=> *st'* leads to a contradiction,

  We obtain a contradiction by 2 and 3. □

```
Lemma while_true_nonterm : ∀ b c st st',
  bequiv b <{true}> →
  ~( st =[ while b do c end ]=> st' ).
Proof.
  intros b c st st' Hb.
```

```
  intros H.
  remember <{ while b do c end }> as cw eqn:Heqcw.
  induction H;

  inversion Heqcw; subst; clear Heqcw.
  -
    unfold bequiv in Hb.
    rewrite Hb in H. discriminate.
  -
    apply IHceval2. reflexivity. Qed.
```

Exercise: 2 stars, standard, optional (while_true_nonterm_informal) Explain what the lemma while_true_nonterm means in English.

□

Exercise: 2 stars, standard, especially useful (while_true) Prove the following theorem. *Hint*: You'll want to use while_true_nonterm here.

```
Theorem while_true : ∀ b c,
  bequiv b <{true}> →
  cequiv
    <{ while b do c end }>
    <{ while true do skip end }>.
Proof.
  Admitted.
```

□

A more interesting fact about *while* commands is that any number of copies of the body can be "unrolled" without changing meaning.

Loop unrolling is an important transformation in any real compiler, so its correctness is of more than just academic interest!

```
Theorem loop_unrolling : ∀ b c,
  cequiv
    <{ while b do c end }>
    <{ if b then c ; while b do c end else skip end }>.
Proof.
  intros b c st st'.
  split; intros Hce.
  -
    inversion Hce; subst.
    +
      apply E_IfFalse.
      × assumption.
      × apply E_Skip.
```

```
    +
      apply E_IfTrue.
      × assumption.
      × apply E_Seq with (st' := st'0).
        – assumption.
        – assumption.
  -
    inversion Hce; subst.
    +
      inversion H5; subst.
      apply E_WhileTrue with (st' := st'0).
      × assumption.
      × assumption.
      × assumption.
    +
      inversion H5; subst. apply E_WhileFalse. assumption. Qed.
```

Exercise: 2 stars, standard, optional (seq_assoc) `Theorem seq_assoc : ∀ c1 c2 c3, cequiv <{(c1;c2);c3}> <{c1;(c2;c3)}>.`

```
Proof.
  Admitted.
```

□

Proving program properties involving assignments is one place where the fact that we are treating equality on program states extensionally (e.g., `x !-> m x ; m` and `m` are equal maps) comes in handy.

```
Theorem identity_assignment : ∀ x,
  cequiv
    <{ x := x }>
    <{ skip }>.
Proof.
  intros.
  split; intro H; inversion H; subst; clear H.
  -
    rewrite t_update_same.
    apply E_Skip.
  -
    assert (Hx : st' =[ x := x ]=> (x !-> st' x ; st')).
    { apply E_Asgn. reflexivity. }
    rewrite t_update_same in Hx.
    apply Hx.
Qed.
```

Exercise: 2 stars, standard, especially useful (assign_aequiv)

```
Theorem assign_aequiv
: ∀ (X : string) (a : aexp),
  aequiv <{ X }> a →
  cequiv <{ skip }> <{ X := a }>.
Proof.
```

Admitted.

□

Exercise: 2 stars, standard, optional (equiv_classes) Given the following programs, group together those that are equivalent in Imp. Your answer should be given as a list of lists, where each sub-list represents a group of equivalent programs. For example, if you think programs (a) through (h) are all equivalent to each other, but not to (i), your answer should look like this:

[prog_a;prog_b;prog_c;prog_d;prog_e;prog_f;prog_g;prog_h] ; [prog_i]

Write down your answer below in the definition of equiv_classes.

```
Definition prog_a : com :=
  <{ while X > 0 do
       X := X + 1
     end }>.

Definition prog_b : com :=
  <{ if (X = 0) then
       X := X + 1;
       Y := 1
     else
       Y := 0
     end;
     X := X - Y;
     Y := 0 }>.

Definition prog_c : com :=
  <{ skip }> .

Definition prog_d : com :=
  <{ while X ≠ 0 do
       X := (X × Y) + 1
     end }>.

Definition prog_e : com :=
  <{ Y := 0 }>.

Definition prog_f : com :=
  <{ Y := X + 1;
     while X ≠ Y do
       Y := X + 1
     end }>.
```

```
Definition prog_g : com :=
  <{ while true do
       skip
     end }>.

Definition prog_h : com :=
  <{ while X ≠ X do
       X := X + 1
     end }>.

Definition prog_i : com :=
  <{ while X ≠ Y do
       X := Y + 1
     end }>.

Definition equiv_classes : list (list com)
  . Admitted.

Definition manual_grade_for_equiv_classes : option (nat×string) := None.
```
□

3.3 Properties of Behavioral Equivalence

We next consider some fundamental properties of program equivalence.

3.3.1 Behavioral Equivalence is an Equivalence

First, let's verify that the equivalences on *aexps*, *bexps*, and **com**s really are *equivalences* – i.e., that they are reflexive, symmetric, and transitive. The proofs are all easy.

```
Lemma refl_aequiv : ∀ (a : aexp),
  aequiv a a.
Proof.
  intros a st. reflexivity. Qed.

Lemma sym_aequiv : ∀ (a1 a2 : aexp),
  aequiv a1 a2 → aequiv a2 a1.
Proof.
  intros a1 a2 H. intros st. symmetry. apply H. Qed.

Lemma trans_aequiv : ∀ (a1 a2 a3 : aexp),
  aequiv a1 a2 → aequiv a2 a3 → aequiv a1 a3.
Proof.
  unfold aequiv. intros a1 a2 a3 H12 H23 st.
  rewrite (H12 st). rewrite (H23 st). reflexivity. Qed.

Lemma refl_bequiv : ∀ (b : bexp),
  bequiv b b.
```

```
Proof.
  unfold bequiv. intros b st. reflexivity. Qed.

Lemma sym_bequiv : ∀ (b1 b2 : bexp),
  bequiv b1 b2 → bequiv b2 b1.
Proof.
  unfold bequiv. intros b1 b2 H. intros st. symmetry. apply H. Qed.

Lemma trans_bequiv : ∀ (b1 b2 b3 : bexp),
  bequiv b1 b2 → bequiv b2 b3 → bequiv b1 b3.
Proof.
  unfold bequiv. intros b1 b2 b3 H12 H23 st.
  rewrite (H12 st). rewrite (H23 st). reflexivity. Qed.

Lemma refl_cequiv : ∀ (c : com),
  cequiv c c.
Proof.
  unfold cequiv. intros c st st'. reflexivity. Qed.

Lemma sym_cequiv : ∀ (c1 c2 : com),
  cequiv c1 c2 → cequiv c2 c1.
Proof.
  unfold cequiv. intros c1 c2 H st st'.
  rewrite H. reflexivity.
Qed.

Lemma trans_cequiv : ∀ (c1 c2 c3 : com),
  cequiv c1 c2 → cequiv c2 c3 → cequiv c1 c3.
Proof.
  unfold cequiv. intros c1 c2 c3 H12 H23 st st'.
  rewrite H12. apply H23.
Qed.
```

3.3.2 Behavioral Equivalence Is a Congruence

Less obviously, behavioral equivalence is also a *congruence*. That is, the equivalence of two subprograms implies the equivalence of the larger programs in which they are embedded:

aequiv a a'

---

cequiv (x := a) (x := a')

cequiv c1 c1' cequiv c2 c2'

---

cequiv (c1;c2) (c1';c2')

... and so on for the other forms of commands.

(Note that we are using the inference rule notation here not as part of an inductive definition, but simply to write down some valid implications in a readable format. We prove

these implications below.)

We will see a concrete example of why these congruence properties are important in the following section (in the proof of fold_constants_com_sound), but the main idea is that they allow us to replace a small part of a large program with an equivalent small part and know that the whole large programs are equivalent *without* doing an explicit proof about the parts that didn't change – i.e., the "proof burden" of a small change to a large program is proportional to the size of the change, not the program!

```
Theorem CAsgn_congruence : ∀ x a a',
  aequiv a a' →
  cequiv <{x := a}> <{x := a'}>.
Proof.
  intros x a a' Heqv st st'.
  split; intros Hceval.
  -
    inversion Hceval. subst. apply E_Asgn.
    rewrite Heqv. reflexivity.
  -
    inversion Hceval. subst. apply E_Asgn.
    rewrite Heqv. reflexivity. Qed.
```

The congruence property for loops is a little more interesting, since it requires induction.

Theorem: Equivalence is a congruence for *while* – that is, if *b* is equivalent to *b'* and *c* is equivalent to *c'*, then *while b* `do` *c* `end` is equivalent to *while b'* `do` *c'* `end`.

Proof: Suppose *b* is equivalent to *b'* and *c* is equivalent to *c'*. We must show, for every *st* and *st'*, that *st* =[*while b* `do` *c* `end`]=> *st'* iff *st* =[*while b'* `do` *c'* `end`]=> *st'*. We consider the two directions separately.

- (→) We show that *st* =[*while b* `do` *c* `end`]=> *st'* implies *st* =[*while b'* `do` *c'* `end`]=> *st'*, by induction on a derivation of *st* =[*while b* `do` *c* `end`]=> *st'*. The only nontrivial cases are when the final rule in the derivation is E_WhileFalse or E_WhileTrue.

  - E_WhileFalse: In this case, the form of the rule gives us beval *st b* = false and *st* = *st'*. But then, since *b* and *b'* are equivalent, we have beval *st b'* = false, and E_WhileFalse applies, giving us *st* =[*while b'* `do` *c'* `end`]=> *st'*, as required.
  - E_WhileTrue: The form of the rule now gives us beval *st b* = true, with *st* =[*c*]=> *st'0* and *st'0* =[*while b* `do` *c* `end`]=> *st'* for some state *st'0*, with the induction hypothesis *st'0* =[*while b'* `do` *c'* `end`]=> *st'*.

    Since *c* and *c'* are equivalent, we know that *st* =[*c'*]=> *st'0*. And since *b* and *b'* are equivalent, we have beval *st b'* = true. Now E_WhileTrue applies, giving us *st* =[*while b'* `do` *c'* `end`]=> *st'*, as required.

- (←) Similar. □

```
Theorem CWhile_congruence : ∀ b b' c c',
  bequiv b b' → cequiv c c' →
  cequiv <{ while b do c end }> <{ while b' do c' end }>.
Proof.
  assert (A: ∀ (b b' : bexp) (c c' : com) (st st' : state),
              bequiv b b' → cequiv c c' →
              st =[ while b do c end ]=> st' →
              st =[ while b' do c' end ]=> st').
  { unfold bequiv,cequiv.
    intros b b' c c' st st' Hbe Hc1e Hce.
    remember <{ while b do c end }> as cwhile
      eqn:Heqcwhile.
    induction Hce; inversion Heqcwhile; subst.
    +
      apply E_WhileFalse. rewrite ← Hbe. apply H.
    +
      apply E_WhileTrue with (st' := st').
      × rewrite ← Hbe. apply H.
      ×
        apply (Hc1e st st'). apply Hce1.
      ×
        apply IHHce2. reflexivity. }

  intros. split.
  - apply A; assumption.
  - apply A.
    + apply sym_bequiv. assumption.
    + apply sym_cequiv. assumption.
Qed.
```

Exercise: 3 stars, standard, optional (CSeq_congruence) `Theorem CSeq_congruence : ∀ c1 c1' c2 c2',`
`  cequiv c1 c1' → cequiv c2 c2' →`
`  cequiv <{ c1;c2 }> <{ c1';c2' }>.`
`Proof.`
`  Admitted.`
□

Exercise: 3 stars, standard (CIf_congruence) `Theorem CIf_congruence : ∀ b b' c1 c1' c2 c2',`
`  bequiv b b' → cequiv c1 c1' → cequiv c2 c2' →`
`  cequiv <{ if b then c1 else c2 end }>`

```
        <{ if b' then c1' else c2' end }>.
Proof.
  Admitted.
```
□

For example, here are two programs and a proof of their equivalence using these congruence theorems...

```
Example congruence_example:
  cequiv

    <{ X := 0;
       if X = 0 then Y := 0
       else Y := 42 end }>

    <{ X := 0;
       if X = 0 then Y := X - X
       else Y := 42 end }>.
Proof.
  apply CSeq_congruence.
  - apply refl_cequiv.
  - apply CIf_congruence.
    + apply refl_bequiv.
    + apply CAsgn_congruence. unfold aequiv. simpl.
      symmetry. apply sub_diag.
    + apply refl_cequiv.
Qed.
```

Exercise: 3 stars, advanced (not_congr) We've shown that the cequiv relation is both an equivalence and a congruence on commands. Can you think of a relation on commands that is an equivalence but *not* a congruence? Write down the relation (formally), together with an informal sketch of a proof that it is an equivalence and a counterexample showing it is not a congruence.

```
Definition manual_grade_for_not_congr : option (nat×string) := None.
```
□

3.4 Program Transformations

A *program transformation* is a function that takes a program as input and produces a modified program as output. Compiler optimizations such as constant folding are canonical examples, but there are many others.

A program transformation is *sound* if it preserves the behavior of the original program.

```
Definition atrans_sound (atrans : aexp → aexp) : Prop :=
```

```
  ∀ (a : aexp),
    aequiv a (atrans a).
Definition btrans_sound (btrans : bexp → bexp) : Prop :=
  ∀ (b : bexp),
    bequiv b (btrans b).
Definition ctrans_sound (ctrans : com → com) : Prop :=
  ∀ (c : com),
    cequiv c (ctrans c).
```

3.4.1 The Constant-Folding Transformation

An expression is *constant* if it contains no variable references.

Constant folding is an optimization that finds constant expressions and replaces them by their values.

```
Fixpoint fold_constants_aexp (a : aexp) : aexp :=
  match a with
  | ANum n ⇒ ANum n
  | AId x ⇒ AId x
  | <{ a1 + a2 }> ⇒
    match (fold_constants_aexp a1,
           fold_constants_aexp a2)
    with
    | (ANum n1, ANum n2) ⇒ ANum (n1 + n2)
    | (a1', a2') ⇒ <{ a1' + a2' }>
    end
  | <{ a1 - a2 }> ⇒
    match (fold_constants_aexp a1,
           fold_constants_aexp a2)
    with
    | (ANum n1, ANum n2) ⇒ ANum (n1 - n2)
    | (a1', a2') ⇒ <{ a1' - a2' }>
    end
  | <{ a1 × a2 }> ⇒
    match (fold_constants_aexp a1,
           fold_constants_aexp a2)
    with
    | (ANum n1, ANum n2) ⇒ ANum (n1 × n2)
    | (a1', a2') ⇒ <{ a1' × a2' }>
    end
  end.
Example fold_aexp_ex1 :
    fold_constants_aexp <{ (1 + 2) × X }>
```

```
  = <{ 3 × X }>.
Proof. reflexivity. Qed.
```

Note that this version of constant folding doesn't do other "obvious" things like eliminating trivial additions (e.g., rewriting 0 + X to just X).: we are focusing on a single optimization for the sake of simplicity.

It is not hard to incorporate other ways of simplifying expressions – the definitions and proofs just get longer. We'll consider some in the exercises.

```
Example fold_aexp_ex2 :
  fold_constants_aexp <{ X - ((0 × 6) + Y) }> = <{ X - (0 + Y) }>.
Proof. reflexivity. Qed.
```

Not only can we lift fold_constants_aexp to **bexp**s (in the BEq, BNeq, and BLe cases); we can also look for constant *boolean* expressions and evaluate them in-place as well.

```
Fixpoint fold_constants_bexp (b : bexp) : bexp :=
  match b with
  | <{true}> ⇒ <{true}>
  | <{false}> ⇒ <{false}>
  | <{ a1 = a2 }> ⇒
      match (fold_constants_aexp a1,
             fold_constants_aexp a2) with
      | (ANum n1, ANum n2) ⇒
          if n1 =? n2 then <{true}> else <{false}>
      | (a1', a2') ⇒
          <{ a1' = a2' }>
      end
  | <{ a1 ≠ a2 }> ⇒
      match (fold_constants_aexp a1,
             fold_constants_aexp a2) with
      | (ANum n1, ANum n2) ⇒
          if negb (n1 =? n2) then <{true}> else <{false}>
      | (a1', a2') ⇒
          <{ a1' ≠ a2' }>
      end
  | <{ a1 ≤ a2 }> ⇒
      match (fold_constants_aexp a1,
             fold_constants_aexp a2) with
      | (ANum n1, ANum n2) ⇒
          if n1 <=? n2 then <{true}> else <{false}>
      | (a1', a2') ⇒
          <{ a1' ≤ a2' }>
      end
  | <{ a1 > a2 }> ⇒
```

```
      match (fold_constants_aexp a1,
              fold_constants_aexp a2) with
      | (ANum n1, ANum n2) ⇒
          if n1 <=? n2 then <{false}> else <{true}>
      | (a1', a2') ⇒
          <{ a1' > a2' }>
      end
  | <{ ¬ b1 }> ⇒
      match (fold_constants_bexp b1) with
      | <{true}> ⇒ <{false}>
      | <{false}> ⇒ <{true}>
      | b1' ⇒ <{ ¬ b1' }>
      end
  | <{ b1 && b2 }> ⇒
      match (fold_constants_bexp b1,
              fold_constants_bexp b2) with
      | (<{true}>, <{true}>) ⇒ <{true}>
      | (<{true}>, <{false}>) ⇒ <{false}>
      | (<{false}>, <{true}>) ⇒ <{false}>
      | (<{false}>, <{false}>) ⇒ <{false}>
      | (b1', b2') ⇒ <{ b1' && b2' }>
      end
  end.

Example fold_bexp_ex1 :
  fold_constants_bexp <{ true && ¬(false && true) }>
  = <{ true }>.
Proof. reflexivity. Qed.

Example fold_bexp_ex2 :
  fold_constants_bexp <{ (X = Y) && (0 = (2 - (1 + 1))) }>
  = <{ (X = Y) && true }>.
Proof. reflexivity. Qed.
```

To fold constants in a command, we simply apply the appropriate folding functions on all embedded expressions.

```
Fixpoint fold_constants_com (c : com) : com :=
  match c with
  | <{ skip }> ⇒
      <{ skip }>
  | <{ x := a }> ⇒
      <{ x := (fold_constants_aexp a) }>
  | <{ c1 ; c2 }> ⇒
      <{ fold_constants_com c1 ; fold_constants_com c2 }>
```

```
  | <{ if b then c1 else c2 end }> ⇒
      match fold_constants_bexp b with
      | <{true}> ⇒ fold_constants_com c1
      | <{false}> ⇒ fold_constants_com c2
      | b' ⇒ <{ if b' then fold_constants_com c1
                       else fold_constants_com c2 end}>
      end
  | <{ while b do c1 end }> ⇒
      match fold_constants_bexp b with
      | <{true}> ⇒ <{ while true do skip end }>
      | <{false}> ⇒ <{ skip }>
      | b' ⇒ <{ while b' do (fold_constants_com c1) end }>
      end
  end.

Example fold_com_ex1 :
  fold_constants_com

    <{ X := 4 + 5;
       Y := X - 3;
       if (X - Y) = (2 + 4) then skip
       else Y := 0 end;
       if 0 ≤ (4 - (2 + 1)) then Y := 0
       else skip end;
       while Y = 0 do
         X := X + 1
       end }>
  =
    <{ X := 9;
       Y := X - 3;
       if (X - Y) = 6 then skip
       else Y := 0 end;
       Y := 0;
       while Y = 0 do
         X := X + 1
       end }>.
Proof. reflexivity. Qed.
```

3.4.2 Soundness of Constant Folding

Now we need to show that what we've done is correct.

Here's the proof for arithmetic expressions:

```
Theorem fold_constants_aexp_sound :
```

```
  atrans_sound fold_constants_aexp.
Proof.
  unfold atrans_sound. intros a. unfold aequiv. intros st.
  induction a; simpl;

    try reflexivity;

    try (destruct (fold_constants_aexp a1);
         destruct (fold_constants_aexp a2);
         rewrite IHa1; rewrite IHa2; reflexivity). Qed.
```

Exercise: 3 stars, standard, optional (fold_bexp_Eq_informal) Here is an informal proof of the BEq case of the soundness argument for boolean expression constant folding. Read it carefully and compare it to the formal proof that follows. Then fill in the BLe case of the formal proof (without looking at the BEq case, if possible).

Theorem: The constant folding function for booleans, fold_constants_bexp, is sound.

Proof: We must show that b is equivalent to fold_constants_bexp b, for all boolean expressions b. Proceed by induction on b. We show just the case where b has the form $a1 = a2$.

In this case, we must show
beval st <{ a1 = a2 }> = beval st (fold_constants_bexp <{ a1 = a2 }>).
There are two cases to consider:

- First, suppose fold_constants_aexp $a1$ = ANum $n1$ and fold_constants_aexp $a2$ = ANum $n2$ for some $n1$ and $n2$.

  In this case, we have

  fold_constants_bexp <{ a1 = a2 }> = if n1 =? n2 then <{true}> else <{false}>

  and

  beval st <{a1 = a2}> = (aeval st a1) =? (aeval st a2).

  By the soundness of constant folding for arithmetic expressions (Lemma fold_constants_aexp_sound), we know

  aeval st a1 = aeval st (fold_constants_aexp a1) = aeval st (ANum n1) = n1

  and

  aeval st a2 = aeval st (fold_constants_aexp a2) = aeval st (ANum n2) = n2,

  so

  beval st <{a1 = a2}> = (aeval a1) =? (aeval a2) = n1 =? n2.

  Also, it is easy to see (by considering the cases $n1 = n2$ and $n1 \neq n2$ separately) that

  beval st (if n1 =? n2 then <{true}> else <{false}>) = if n1 =? n2 then beval st <{true}> else beval st <{false}> = if n1 =? n2 then true else false = n1 =? n2.

So

beval st (<{ a1 = a2 }>) = n1 =? n2. = beval st (if n1 =? n2 then <{true}> else <{false}>),

as required.

- Otherwise, one of fold_constants_aexp *a1* and fold_constants_aexp *a2* is not a constant. In this case, we must show

  beval st <{a1 = a2}> = beval st (<{ (fold_constants_aexp a1) = (fold_constants_aexp a2) }>),

  which, by the definition of beval, is the same as showing

  (aeval st a1) =? (aeval st a2) = (aeval st (fold_constants_aexp a1)) =? (aeval st (fold_constants_aexp a2)).

  But the soundness of constant folding for arithmetic expressions (fold_constants_aexp_sound) gives us

  aeval st a1 = aeval st (fold_constants_aexp a1) aeval st a2 = aeval st (fold_constants_aexp a2),

  completing the case. □

```
Theorem fold_constants_bexp_sound:
  btrans_sound fold_constants_bexp.
Proof.
  unfold btrans_sound. intros b. unfold bequiv. intros st.
  induction b;

    try reflexivity.
  -
    simpl.
    remember (fold_constants_aexp a1) as a1' eqn:Heqa1'.
    remember (fold_constants_aexp a2) as a2' eqn:Heqa2'.
    replace (aeval st a1) with (aeval st a1') by
       (subst a1'; rewrite ← fold_constants_aexp_sound; reflexivity).
    replace (aeval st a2) with (aeval st a2') by
       (subst a2'; rewrite ← fold_constants_aexp_sound; reflexivity).
    destruct a1'; destruct a2'; try reflexivity.
      simpl. destruct (n =? n0); reflexivity.
  -
    simpl.
    remember (fold_constants_aexp a1) as a1' eqn:Heqa1'.
    remember (fold_constants_aexp a2) as a2' eqn:Heqa2'.
    replace (aeval st a1) with (aeval st a1') by
```

```
      (subst a1'; rewrite ← fold_constants_aexp_sound; reflexivity).
    replace (aeval st a2) with (aeval st a2') by
      (subst a2'; rewrite ← fold_constants_aexp_sound; reflexivity).
    destruct a1'; destruct a2'; try reflexivity.
      simpl. destruct (n =? n0); reflexivity.
  -
    admit.
  -
    admit.
  -
    simpl. remember (fold_constants_bexp b) as b' eqn:Heqb'.
    rewrite IHb.
    destruct b'; reflexivity.
  -
    simpl.
    remember (fold_constants_bexp b1) as b1' eqn:Heqb1'.
    remember (fold_constants_bexp b2) as b2' eqn:Heqb2'.
    rewrite IHb1. rewrite IHb2.
    destruct b1'; destruct b2'; reflexivity.
  Admitted.
□
```

Exercise: 3 stars, standard (fold_constants_com_sound) Complete the *while* case of the following proof.

```
Theorem fold_constants_com_sound :
  ctrans_sound fold_constants_com.
Proof.
  unfold ctrans_sound. intros c.
  induction c; simpl.
  - apply refl_cequiv.
  - apply CAsgn_congruence.
                apply fold_constants_aexp_sound.
  - apply CSeq_congruence; assumption.
  -
    assert (bequiv b (fold_constants_bexp b)). {
      apply fold_constants_bexp_sound. }
    destruct (fold_constants_bexp b) eqn:Heqb;
      try (apply CIf_congruence; assumption).
    +
      apply trans_cequiv with c1; try assumption.
      apply if_true; assumption.
    +
```

```
    apply trans_cequiv with c2; try assumption.
    apply if_false; assumption.
-
  Admitted.
□
```

3.4.3 Soundness of (0 + n) Elimination, Redux

Exercise: 4 stars, standard, optional (optimize_0plus_var) Recall the definition optimize_0plus from the Imp chapter of *Logical Foundations*:

Fixpoint optimize_0plus (a:aexp) : aexp := match a with | ANum n => ANum n | <{ 0 + a2 }> => optimize_0plus a2 | <{ a1 + a2 }> => <{ (optimize_0plus a1) + (optimize_0plus a2) }> | <{ a1 - a2 }> => <{ (optimize_0plus a1) - (optimize_0plus a2) }> | <{ a1 * a2 }> => <{ (optimize_0plus a1) * (optimize_0plus a2) }> end.

Note that this function is defined over the old version of **aexp**s, without states.

Write a new version of this function that deals with variables (by leaving them alone), plus analogous ones for **bexp**s and commands:

optimize_0plus_aexp optimize_0plus_bexp optimize_0plus_com

```
Fixpoint optimize_0plus_aexp (a : aexp) : aexp
  . Admitted.

Fixpoint optimize_0plus_bexp (b : bexp) : bexp
  . Admitted.

Fixpoint optimize_0plus_com (c : com) : com
  . Admitted.

Example test_optimize_0plus:
    optimize_0plus_com
       <{ while X ≠ 0 do X := 0 + X - 1 end }>
  = <{ while X ≠ 0 do X := X - 1 end }>.
Proof.
    Admitted.
```

Prove that these three functions are sound, as we did for *fold_constants_*×. Make sure you use the congruence lemmas in the proof of optimize_0plus_com – otherwise it will be *long*!

```
Theorem optimize_0plus_aexp_sound:
  atrans_sound optimize_0plus_aexp.
Proof.
    Admitted.

Theorem optimize_0plus_bexp_sound :
  btrans_sound optimize_0plus_bexp.
Proof.
    Admitted.
```

```
Theorem optimize_0plus_com_sound :
  ctrans_sound optimize_0plus_com.
Proof.
    Admitted.
```

Finally, let's define a compound optimizer on commands that first folds constants (using fold_constants_com) and then eliminates $0 + n$ terms (using optimize_0plus_com).

```
Definition optimizer (c : com) := optimize_0plus_com (fold_constants_com c).
```

Prove that this optimizer is sound.

```
Theorem optimizer_sound :
  ctrans_sound optimizer.
Proof.
    Admitted.
    □
```

3.5 Proving Inequivalence

Next, let's look at some programs that are *not* equivalent.

Suppose that *c1* is a command of the form
X := a1; Y := a2
and *c2* is the command
X := a1; Y := a2'
where *a2'* is formed by substituting *a1* for all occurrences of X in *a2*.

For example, *c1* and *c2* might be:
c1 = (X := 42 + 53; Y := Y + X) c2 = (X := 42 + 53; Y := Y + (42 + 53))

Clearly, this *particular* *c1* and *c2* are equivalent. Is this true in general?

We will see in a moment that it is not, but it is worthwhile to pause, now, and see if you can find a counter-example on your own.

More formally, here is the function that substitutes an arithmetic expression *u* for each occurrence of a given variable *x* in another expression *a*:

```
Fixpoint subst_aexp (x : string) (u : aexp) (a : aexp) : aexp :=
  match a with
  | ANum n ⇒
      ANum n
  | AId x' ⇒
      if String.eqb x x' then u else AId x'
  | <{ a1 + a2 }> ⇒
      <{ (subst_aexp x u a1) + (subst_aexp x u a2) }>
  | <{ a1 - a2 }> ⇒
      <{ (subst_aexp x u a1) - (subst_aexp x u a2) }>
  | <{ a1 × a2 }> ⇒
```

```
      <{ (subst_aexp x u a1) × (subst_aexp x u a2) }>
  end.

Example subst_aexp_ex :
  subst_aexp X <{42 + 53}> <{Y + X}>
  = <{ Y + (42 + 53)}>.
Proof. simpl. reflexivity. Qed.
```

And here is the property we are interested in, expressing the claim that commands *c1* and *c2* as described above are always equivalent.

```
Definition subst_equiv_property : Prop := ∀ x1 x2 a1 a2,
  cequiv <{ x1 := a1 ; x2 := a2 }>
         <{ x1 := a1 ; x2 := subst_aexp x1 a1 a2 }>.
```

Sadly, the property does *not* always hold.
Here is a counterexample:
X := X + 1; Y := X
If we perform the substitution, we get
X := X + 1; Y := X + 1
which clearly isn't equivalent.

```
Theorem subst_inequiv :
  ¬ subst_equiv_property.
Proof.
  unfold subst_equiv_property.
  intros Contra.

  remember <{ X := X + 1;
              Y := X }>
      as c1.
  remember <{ X := X + 1;
              Y := X + 1 }>
      as c2.
  assert (cequiv c1 c2) by (subst; apply Contra).
  clear Contra.

  remember (Y !-> 1 ; X !-> 1) as st1.
  remember (Y !-> 2 ; X !-> 1) as st2.
  assert (H1 : empty_st =[ c1 ]=> st1);
  assert (H2 : empty_st =[ c2 ]=> st2);
  try (subst;
       apply E_Seq with (st' := (X !-> 1));
       apply E_Asgn; reflexivity).
  clear Heqc1 Heqc2.

  apply H in H1.
  clear H.
```

```
  assert (Hcontra : st1 = st2)
    by (apply (ceval_deterministic c2 empty_st); assumption).
  clear H1 H2.
  assert (Hcontra' : st1 Y = st2 Y)
    by (rewrite Hcontra; reflexivity).
  subst. discriminate. Qed.
```

Exercise: 4 stars, standard, optional (better_subst_equiv) The equivalence we had in mind above was not complete nonsense – in fact, it was actually almost right. To make it correct, we just need to exclude the case where the variable X occurs in the right-hand side of the first assignment statement.

```
Inductive var_not_used_in_aexp (x : string) : aexp → Prop :=
  | VNUNum : ∀ n, var_not_used_in_aexp x (ANum n)
  | VNUId : ∀ y, x ≠ y → var_not_used_in_aexp x (AId y)
  | VNUPlus : ∀ a1 a2,
        var_not_used_in_aexp x a1 →
        var_not_used_in_aexp x a2 →
        var_not_used_in_aexp x (<{ a1 + a2 }>)
  | VNUMinus : ∀ a1 a2,
        var_not_used_in_aexp x a1 →
        var_not_used_in_aexp x a2 →
        var_not_used_in_aexp x (<{ a1 - a2 }>)
  | VNUMult : ∀ a1 a2,
        var_not_used_in_aexp x a1 →
        var_not_used_in_aexp x a2 →
        var_not_used_in_aexp x (<{ a1 × a2 }>).

Lemma aeval_weakening : ∀ x st a ni,
  var_not_used_in_aexp x a →
  aeval (x !-> ni ; st) a = aeval st a.
Proof.
    Admitted.
```

Using var_not_used_in_aexp, formalize and prove a correct version of subst_equiv_property.

Exercise: 3 stars, standard (inequiv_exercise) Prove that an infinite loop is not equivalent to *skip*

```
Theorem inequiv_exercise:
  ¬ cequiv <{ while true do skip end }> <{ skip }>.
Proof.
    Admitted.
```

□

3.6 Extended Exercise: Nondeterministic Imp

As we have seen (in theorem ceval_deterministic in the Imp chapter), Imp's evaluation relation is deterministic. However, *non*-determinism is an important part of the definition of many real programming languages. For example, in many imperative languages (such as C and its relatives), the order in which function arguments are evaluated is unspecified: the program fragment

x = 0; f(++x, x)

might call f with arguments (1, 0) or (1, 1), depending how the compiler chooses to order things. This can be a little confusing for programmers, but it gives compiler writers useful freedom.

In this exercise, we will extend Imp with a simple nondeterministic command and study how this change affects program equivalence. The new command has the syntax *HAVOC* X, where X is an identifier. The effect of executing *HAVOC* X is to assign an *arbitrary* number to the variable X, nondeterministically. For example, after executing the program:

HAVOC Y; Z := Y * 2

the value of Y can be any number, while the value of Z is twice that of Y (so Z is always even). Note that we are not saying anything about the *probabilities* of the outcomes – just that there are (infinitely) many different outcomes that can possibly happen after executing this nondeterministic code.

In a sense, a variable on which we do *HAVOC* roughly corresponds to an uninitialized variable in a low-level language like C. After the *HAVOC*, the variable holds a fixed but arbitrary number. Most sources of nondeterminism in language definitions are there precisely because programmers don't care which choice is made (and so it is good to leave it open to the compiler to choose whichever will run faster).

We call this new language *Himp* ("Imp extended with *HAVOC*").

```
Module HIMP.
```

To formalize Himp, we first add a clause to the definition of commands.

```
Inductive com : Type :=
  | CSkip : com
  | CAsgn : string → aexp → com
  | CSeq : com → com → com
  | CIf : bexp → com → com → com
  | CWhile : bexp → com → com
  | CHavoc : string → com.

Notation "'havoc' l" := (CHavoc l)
                          (in custom com at level 60, l constr at level 0).
Notation "'skip'" := CSkip
  (in custom com at level 0) : com_scope.
Notation "x := y" := (CAsgn x y)
  (in custom com at level 0, x constr at level 0, y at level 85, no associativity,
    format "x := y") : com_scope.
```

```
Notation "x ; y" := (CSeq x y)
  (in custom com at level 90,
    right associativity,
    format "'[v' x ; '/' y ']'") : com_scope.
Notation "'if' x 'then' y 'else' z 'end'" := (CIf x y z)
  (in custom com at level 89, x at level 99, y at level 99, z at level 99,
    format "'[v' 'if' x 'then' '/ ' y '/' 'else' '/ ' z '/' 'end' ']'") : com_scope.
Notation "'while' x 'do' y 'end'" := (CWhile x y)
  (in custom com at level 89, x at level 99, y at level 99,
    format "'[v' 'while' x 'do' '/ ' y '/' 'end' ']'") : com_scope.
```

Exercise: 2 stars, standard (himp_ceval) Now, we must extend the operational semantics. We have provided a template for the **ceval** relation below, specifying the big-step semantics. What rule(s) must be added to the definition of **ceval** to formalize the behavior of the *HAVOC* command?

```
Reserved Notation
         "st0 '=[' c ']=>' st1"
         (at level 40, c custom com at level 99,
          st0 constr, st1 constr at next level,
          format "'[hv' st0 =[ '/ ' '[' c ']' '/' ]=> st1 ']'").

Inductive ceval : com → state → state → Prop :=
  | E_Skip : ∀ st,
      st =[ skip ]=> st
  | E_Asgn : ∀ st a n x,
      aeval st a = n →
      st =[ x := a ]=> (x !-> n ; st)
  | E_Seq : ∀ c1 c2 st st' st'',
      st =[ c1 ]=> st' →
      st' =[ c2 ]=> st'' →
      st =[ c1 ; c2 ]=> st''
  | E_IfTrue : ∀ st st' b c1 c2,
      beval st b = true →
      st =[ c1 ]=> st' →
      st =[ if b then c1 else c2 end ]=> st'
  | E_IfFalse : ∀ st st' b c1 c2,
      beval st b = false →
      st =[ c2 ]=> st' →
      st =[ if b then c1 else c2 end ]=> st'
  | E_WhileFalse : ∀ b st c,
      beval st b = false →
      st =[ while b do c end ]=> st
  | E_WhileTrue : ∀ st st' st'' b c,
```

```
      beval st b = true →
      st =[ c ]=> st' →
      st' =[ while b do c end ]=> st'' →
      st =[ while b do c end ]=> st''

  where "st =[ c ]=> st'" := (ceval c st st').
```

As a sanity check, the following claims should be provable for your definition:

```
Example havoc_example1 : empty_st =[ havoc X ]=> (X !-> 0).
Proof.
  Admitted.

Example havoc_example2 :
  empty_st =[ skip; havoc Z ]=> (Z !-> 42).
Proof.
  Admitted.

Definition manual_grade_for_Check_rule_for_HAVOC : option (nat×string) := None.
```

□

Finally, we repeat the definition of command equivalence from above:

```
Definition cequiv (c1 c2 : com) : Prop := ∀ st st' : state,
  st =[ c1 ]=> st' ↔ st =[ c2 ]=> st'.
```

Let's apply this definition to prove some nondeterministic programs equivalent / inequivalent.

Exercise: 3 stars, standard (havoc_swap) Are the following two programs equivalent?

```
Definition pXY :=
  <{ havoc X ; havoc Y }>.

Definition pYX :=
  <{ havoc Y; havoc X }>.
```

If you think they are equivalent, prove it. If you think they are not, prove that.

```
Theorem pXY_cequiv_pYX :
  cequiv pXY pYX ∨ ¬cequiv pXY pYX.
Proof.
  Admitted.
```

□

Exercise: 4 stars, standard, optional (havoc_copy) Are the following two programs equivalent?

```
Definition ptwice :=
```

```
  <{ havoc X; havoc Y }>.

Definition pcopy :=
  <{ havoc X; Y := X }>.
```

If you think they are equivalent, then prove it. If you think they are not, then prove that. (Hint: You may find the `assert` tactic useful.)

```
Theorem ptwice_cequiv_pcopy :
  cequiv ptwice pcopy ∨ ¬cequiv ptwice pcopy.
Proof. Admitted.
```

□

The definition of program equivalence we are using here has some subtle consequences on programs that may loop forever. What cequiv says is that the set of possible *terminating* outcomes of two equivalent programs is the same. However, in a language with nondeterminism, like Himp, some programs always terminate, some programs always diverge, and some programs can nondeterministically terminate in some runs and diverge in others. The final part of the following exercise illustrates this phenomenon.

Exercise: 4 stars, advanced (p1_p2_term) Consider the following commands:

```
Definition p1 : com :=
  <{ while ¬ (X = 0) do
       havoc Y;
       X := X + 1
     end }>.

Definition p2 : com :=
  <{ while ¬ (X = 0) do
       skip
     end }>.
```

Intuitively, p1 and p2 have the same termination behavior: either they loop forever, or they terminate in the same state they started in. We can capture the termination behavior of p1 and p2 individually with these lemmas:

```
Lemma p1_may_diverge : ∀ st st', st X ≠ 0 →
  ¬ st =[ p1 ]=> st'.
Proof. Admitted.

Lemma p2_may_diverge : ∀ st st', st X ≠ 0 →
  ¬ st =[ p2 ]=> st'.
Proof.
  Admitted.
```

□

Exercise: 4 stars, advanced (p1_p2_equiv) Use these two lemmas to prove that p1 and p2 are actually equivalent.

```
Theorem p1_p2_equiv : cequiv p1 p2.
Proof. Admitted.
```
□

Exercise: 4 stars, advanced (p3_p4_inequiv) Prove that the following programs are *not* equivalent. (Hint: What should the value of Z be when p3 terminates? What about p4?)

```
Definition p3 : com :=
  <{ Z := 1;
     while X ≠ 0 do
       havoc X;
       havoc Z
     end }>.

Definition p4 : com :=
  <{ X := 0;
     Z := 1 }>.

Theorem p3_p4_inequiv : ¬ cequiv p3 p4.
Proof. Admitted.
```
□

Exercise: 5 stars, advanced, optional (p5_p6_equiv) Prove that the following commands are equivalent. (Hint: As mentioned above, our definition of cequiv for Himp only takes into account the sets of possible terminating configurations: two programs are equivalent if and only if the set of possible terminating states is the same for both programs when given a same starting state *st*. If p5 terminates, what should the final state be? Conversely, is it always possible to make p5 terminate?)

```
Definition p5 : com :=
  <{ while X ≠ 1 do
       havoc X
     end }>.

Definition p6 : com :=
  <{ X := 1 }>.

Theorem p5_p6_equiv : cequiv p5 p6.
Proof. Admitted.
```
□

```
End Himp.
```

3.7 Additional Exercises

Exercise: 3 stars, standard, optional (swap_noninterfering_assignments) (Hint: You may or may not – depending how you approach it – need to use functional_extensionality explicitly for this one.)

```
Theorem swap_noninterfering_assignments: ∀ l1 l2 a1 a2,
  l1 ≠ l2 →
  var_not_used_in_aexp l1 a2 →
  var_not_used_in_aexp l2 a1 →
  cequiv
    <{ l1 := a1; l2 := a2 }>
    <{ l2 := a2; l1 := a1 }>.
Proof.
  Admitted.
  □
```

Exercise: 4 stars, standard, optional (for_while_equiv) This exercise extends the optional *add_for_loop* exercise from the Imp chapter, where you were asked to extend the language of commands with C-style `for` loops. Prove that the command:

for (c1; b; c2) { c3 }

is equivalent to:

c1; while b do c3; c2 end

Exercise: 4 stars, advanced, optional (capprox) In this exercise we define an asymmetric variant of program equivalence we call *program approximation*. We say that a program *c1* *approximates* a program *c2* when, for each of the initial states for which *c1* terminates, *c2* also terminates and produces the same final state. Formally, program approximation is defined as follows:

```
Definition capprox (c1 c2 : com) : Prop := ∀ (st st' : state),
  st =[ c1 ]=> st' → st =[ c2 ]=> st'.
```

For example, the program

c1 = while X <> 1 do X := X - 1 end

approximates *c2* = X := 1, but *c2* does not approximate *c1* since *c1* does not terminate when X = 0 but *c2* does. If two programs approximate each other in both directions, then they are equivalent.

Find two programs c3 and c4 such that neither approximates the other.

```
Definition c3 : com
  . Admitted.
Definition c4 : com
  . Admitted.

Theorem c3_c4_different : ¬ capprox c3 c4 ∧ ¬ capprox c4 c3.
```

```
Proof. Admitted.
```

Find a program cmin that approximates every other program.

```
Definition cmin : com
  . Admitted.

Theorem cmin_minimal : ∀ c, capprox cmin c.
Proof. Admitted.
```

Finally, find a non-trivial property which is preserved by program approximation (when going from left to right).

```
Definition zprop (c : com) : Prop
  . Admitted.

Theorem zprop_preserving : ∀ c c',
  zprop c → capprox c c' → zprop c'.
Proof. Admitted.
```

□

Chapter 4

Library SECF.Hoare

4.1 Hoare: Hoare Logic, Part I

```
Set Warnings "-notation-overridden".
From SECF Require Import Maps.
From Stdlib Require Import Bool.
From Stdlib Require Import Arith.
From Stdlib Require Import EqNat.
From Stdlib Require Import PeanoNat. Import Nat.
From Stdlib Require Import Lia.
From SECF Require Export Imp.
```

In the final chaper of *Logical Foundations* (*Software Foundations*, volume 1), we began applying the mathematical tools developed in the first part of the course to studying the theory of a small programming language, Imp.

- We defined a type of *abstract syntax trees* for Imp, together with an *evaluation relation* (a partial function on states) that specifies the *operational semantics* of programs.

  The language we defined, though small, captures some of the key features of full-blown languages like C, C++, and Java, including the fundamental notion of mutable state and some common control structures.

- We proved a number of *metatheoretic properties* – "meta" in the sense that they are properties of the language as a whole, rather than of particular programs in the language. These included:

  - determinism of evaluation
  - equivalence of some different ways of writing down the definitions (e.g., functional and relational definitions of arithmetic expression evaluation)
  - guaranteed termination of certain classes of programs

- correctness (in the sense of preserving meaning) of a number of useful program transformations
- behavioral equivalence of programs (in the Equiv chapter).

If we stopped here, we would already have something useful: a set of tools for defining and discussing programming languages and language features that are mathematically precise, flexible, and easy to work with, applied to a set of key properties. All of these properties are things that language designers, compiler writers, and users might care about knowing. Indeed, many of them are so fundamental to our understanding of the programming languages we deal with that we might not consciously recognize them as "theorems." But properties that seem intuitively obvious can sometimes be quite subtle (sometimes also subtly wrong!).

We'll return to the theme of metatheoretic properties of whole languages later in this volume when we discuss *types* and *type soundness*. In this chapter, though, we turn to a different set of issues. Our goal in this chapter is to develop the tools to work through some simple examples of *program verification* – i.e., to use the precise definition of Imp to prove formally that particular programs satisfy particular specifications of their behavior.

We'll develop a reasoning system called *Floyd-Hoare Logic* – often shortened to just *Hoare Logic* – in which each of the syntactic constructs of Imp is equipped with a generic "proof rule" that can be used to reason compositionally about the correctness of programs involving this construct.

Hoare Logic originated in the 1960s, and it continues to be the subject of intensive research right up to the present day. It lies at the core of a multitude of tools that are being used in academia and industry to specify and verify real software systems.

Hoare Logic combines two beautiful ideas: a natural way of writing down *specifications* of programs, and a *structured proof technique* for proving that programs are correct with respect to such specifications – where by "structured" we mean that the structure of proofs directly mirrors the structure of the programs that they are about.

4.2 Assertions

An *assertion* is a logical claim about the state of a program's memory – formally, a property of states.

```
Definition Assertion := state → Prop.
```

For example,

- `fun st ⇒ st X = 3` holds for states *st* in which value of X is 3,
- `fun st ⇒ True` hold for all states, and
- `fun st ⇒ False` holds for no states.

Exercise: 1 star, standard, optional (assertions) Paraphrase the following assertions in English (or your favorite natural language).

```
Module ExAssertions.
Definition assertion1 : Assertion := fun st ⇒ st X ≤ st Y.
Definition assertion2 : Assertion :=
  fun st ⇒ st X = 3 ∨ st X ≤ st Y.
Definition assertion3 : Assertion :=
  fun st ⇒ st Z × st Z ≤ st X ∧
            ¬ (((S (st Z)) × (S (st Z))) ≤ st X).
Definition assertion4 : Assertion :=
  fun st ⇒ st Z = max (st X) (st Y).
End ExAssertions.
```

□

4.2.1 Notations for Assertions

This way of writing assertions can be a little bit heavy, for two reasons: (1) every single assertion that we ever write is going to begin with `fun st ⇒` ; and (2) this state *st* is the only one that we ever use to look up variables in assertions (we will almost never need to talk about two different memory states at the same time). For discussing examples informally, we'll adopt some simplifying conventions: we'll drop the initial `fun st ⇒`, and we'll write just X to mean *st* X. Thus, instead of writing

fun st => st X = m

we'll write just

[1].

Here the "doubly curly" braces {{ and }} delimit the scope of an assertion. We'll see more examples below.

This example also illustrates a convention that we'll use throughout the Hoare Logic chapters: in informal assertions, capital letters like X, Y, and Z are Imp variables, while lowercase letters like *x*, *y*, *m*, and *n* are ordinary Rocq variables (of type **nat**). This is why, when translating from informal to formal, we replace X with *st* X but leave *m* alone.

The convention described above can be implemented in Rocq with a little syntax magic, using coercions and annotation scopes, much as we did with the <{ **com** }> notation in Imp. This new notation automatically lifts **aexp**s, numbers, and `Prop`s into Assertions when they appear in the {{ _ }} scope, or when Rocq knows that the type of an expression is Assertion.

There is no need to understand the details of how these notation hacks work, so we hide them in the HTML version of the notes. (We barely understand some of it ourselves!) For the gory details, see the Rocq development.

```
Definition Aexp : Type := state → nat.
```

[1] `X=m`

```
Definition assert_of_Prop (P : Prop) : Assertion := fun _ ⇒ P.
Definition Aexp_of_nat (n : nat) : Aexp := fun _ ⇒ n.

Definition Aexp_of_aexp (a : aexp) : Aexp := fun st ⇒ aeval st a.

Coercion assert_of_Prop : Sortclass >-> Assertion.
Coercion Aexp_of_nat : nat >-> Aexp.
Coercion Aexp_of_aexp : aexp >-> Aexp.

Arguments assert_of_Prop /.
Arguments Aexp_of_nat /.
Arguments Aexp_of_aexp /.

Declare Custom Entry assn. Declare Scope assertion_scope.
Bind Scope assertion_scope with Assertion.
Bind Scope assertion_scope with Aexp.
Delimit Scope assertion_scope with assertion.
```

One small limitation of this approach is that we don't have an automatic way to coerce a function application that appears within an assertion to make appropriate use of the state when its arguments should be interpets as Imp arithmetic expressions. Instead, we introduce a notation `#f e1 .. en` that stands for `(fun st ⇒ f (e1 st) .. (en st)`, letting us manually mark such function calls when they're needed as part of an assertion.

```
Notation "# f x ..  y" := (fun st ⇒ (.. (f ((x:Aexp) st)) .. ((y:Aexp) st)))
                  (in custom assn at level 2,
                  f constr at level 0, x custom assn at level 1,
                  y custom assn at level 1) : assertion_scope.

Notation "P -> Q" := (fun st ⇒ (P:Assertion) st → (Q:Assertion) st) (in custom assn
at level 99, right associativity) : assertion_scope.
Notation "P <-> Q" := (fun st ⇒ (P:Assertion) st ↔ (Q:Assertion) st) (in custom assn
at level 95) : assertion_scope.

Notation "P \/ Q" := (fun st ⇒ (P:Assertion) st ∨ (Q:Assertion) st) (in custom assn
at level 85, right associativity) : assertion_scope.
Notation "P /\ Q" := (fun st ⇒ (P:Assertion) st ∧ (Q:Assertion) st) (in custom assn
at level 80, right associativity) : assertion_scope.
Notation "~ P" := (fun st ⇒ ¬ ((P:Assertion) st)) (in custom assn at level 75, right
associativity) : assertion_scope.
Notation "a = b" := (fun st ⇒ (a:Aexp) st = (b:Aexp) st) (in custom assn at level
70) : assertion_scope.
Notation "a <> b" := (fun st ⇒ (a:Aexp) st ≠ (b:Aexp) st) (in custom assn at level
70) : assertion_scope.
Notation "a <= b" := (fun st ⇒ (a:Aexp) st ≤ (b:Aexp) st) (in custom assn at level
70) : assertion_scope.
Notation "a < b" := (fun st ⇒ (a:Aexp) st < (b:Aexp) st) (in custom assn at level
70) : assertion_scope.
```

```
Notation "a >= b" := (fun st ⇒ (a:Aexp) st ≥ (b:Aexp) st) (in custom assn at level
70) : assertion_scope.
Notation "a > b" := (fun st ⇒ (a:Aexp) st > (b:Aexp) st) (in custom assn at level
70) : assertion_scope.
Notation "'True'" := True.
Notation "'True'" := (fun st ⇒ True) (in custom assn at level 0) : assertion_scope.
Notation "'False'" := False.
Notation "'False'" := (fun st ⇒ False) (in custom assn at level 0) : assertion_scope.

Notation "a + b" := (fun st ⇒ (a:Aexp) st + (b:Aexp) st) (in custom assn at level
50, left associativity) : assertion_scope.
Notation "a - b" := (fun st ⇒ (a:Aexp) st - (b:Aexp) st) (in custom assn at level
50, left associativity) : assertion_scope.
Notation "a * b" := (fun st ⇒ (a:Aexp) st × (b:Aexp) st) (in custom assn at level
40, left associativity) : assertion_scope.

Notation "( x )" := x (in custom assn at level 0, x at level 99) : assertion_scope.
```

Occasionally we need to "escape" a raw "Rocq-defined" function to express a particularly complicated assertion. We can do that using a $ prefix, as in {{ $(*raw_rocq*) }}.

For example, {{ $(fun *st* ⇒ ∀ X, *st* X = 0) }} indicates an assertion that every variable of X maps to 0 in the given state.

```
Notation "$ f" := f (in custom assn at level 0, f constr at level 0) : assertion_scope.
Notation "x" := (x%assertion) (in custom assn at level 0, x constr at level 0) :
assertion_scope.

Declare Scope hoare_spec_scope.
Open Scope hoare_spec_scope.

Notation "{{ e }}" := e (at level 2, e custom assn at level 99) : assertion_scope.
Open Scope assertion_scope.
```

4.2.2 Example Assertions

Here are some example assertions that take advantage of this new notation.

```
Module ExamplePrettyAssertions.
Definition assertion1 : Assertion := {{ X = 3 }}.
Definition assertion2 : Assertion := {{ True }}.
Definition assertion3 : Assertion := {{ False }}.
Definition assertion4 : Assertion := {{ True ∨ False }}.
Definition assertion5 : Assertion := {{ X ≤ Y }}.
Definition assertion6 : Assertion := {{ X = 3 ∨ X ≤ Y }}.
Definition assertion7 : Assertion := {{ Z = (#max X Y) }}.
Definition assertion8 : Assertion := {{ Z × Z ≤ X
                                        ∧ ¬ (((#S Z) × (#S Z)) ≤ X) }}.
Definition assertion9 : Assertion := {{ #add X Y > #max Y X }}.
```

```
End ExamplePrettyAssertions.
```

4.2.3 Assertion Implication

Given two assertions P and Q, we say that P *implies* Q, written P -» Q, if, whenever P holds in some state st, Q also holds.

```
Definition assert_implies (P Q : Assertion) : Prop :=
  ∀ st, P st → Q st.
```

Note that the notation for *assertion implication* is analogous to the "usual" Rocq implication →.

```
Notation "P -» Q" := (assert_implies P Q)
                       (at level 80) : hoare_spec_scope.
```

We'll also want the "iff" variant of implication between assertions:

```
Notation "P «-» Q" := (P -» Q ∧ Q -» P)
                        (at level 80) : hoare_spec_scope.
```

(The *hoare_spec_scope* annotation here tells Rocq that this notation is not global but is intended to be used in particular contexts.)

4.3 Hoare Triples, Informally

A *Hoare triple* is a claim about the state before and after executing a command. The standard notation is

{P} c {Q}

meaning:

- If command c begins execution in a state satisfying assertion P,
- and if c eventually terminates in some final state,
- then that final state will satisfy the assertion Q.

Assertion P is called the *precondition* of the triple, and Q is the *postcondition*.

Because single braces are already used for other things in Rocq, we'll write Hoare triples with double braces:

[2] c [3] For example,

- The Hoare triple

  [4] X := X + 1 [5]

[2] P
[3] Q
[4] X=0
[5] X=1

states that command X := X + 1 will transform a state in which X = 0 to a state in which X = 1.

- On the other hand,

  forall m, [6] X := X + 1 [7]

  is a *proposition* stating that the Hoare triple {{X = m}} X := X + 1 {{X = m + 1}} is valid for any choice of m. Note that m in the two assertions is a reference to the *Rocq* variable m, which is bound outside the Hoare triple.

Exercise: 1 star, standard, optional (triples) Paraphrase the following in English.

1) [8] c [9]
2) forall m, [10] c [11]
3) [12] c [13]
4) [14] c [15]
5) forall m, [16] c [17]
6) forall m, [18] c [19]

Exercise: 1 star, standard, optional (valid_triples) Which of the following Hoare triples are *valid* – i.e., the claimed relation between P, c, and Q is true?

1) [20] X := 5 [21]
2) [22] X := X + 1 [23]
3) [24] X := 5; Y := 0 [25]
4) [26] X := 5 [27]

---

[6] `X=m`
[7] `X=m+1`
[8] `True`
[9] `X=5`
[10] `X=m`
[11] `X=m+5)`
[12] `X<=Y`
[13] `Y<=X`
[14] `True`
[15] `False`
[16] `X=m`
[17] `Y=real_factm`
[18] `X=m`
[19] `(Z*Z)<=m/\~(((SZ)*(SZ))<=m)`
[20] `True`
[21] `X=5`
[22] `X=2`
[23] `X=3`
[24] `True`
[25] `X=5`
[26] `X=2/\X=3`
[27] `X=0`

5) [28] skip [29]
6) [30] skip [31]
7) [32] while true do skip end [33]
8) [34] while X = 0 do X := X + 1 end [35]
9) [36] while X <> 0 do X := X + 1 end [37]

4.4 Hoare Triples, Formally

We formalize valid Hoare triples in Rocq as follows:

```
Definition valid_hoare_triple
           (P : Assertion) (c : com) (Q : Assertion) : Prop :=
  ∀ st st',
     st =[ c ]=> st' →
     P st →
     Q st'.
```

Notation for Hoare triples

```
Notation "{{ P }} c {{ Q }}" :=
  (valid_hoare_triple P c Q)
    (at level 2, P custom assn at level 99, c custom com at level 99,
     Q custom assn at level 99)
    : hoare_spec_scope.
```

Exercise: 1 star, standard (hoare_post_true) Prove that if Q holds in every state, then any triple with Q as its postcondition is valid.

```
Theorem hoare_post_true : ∀ (P Q : Assertion) c,
  (∀ st, Q st) →
  {{P}} c {{Q}}.
Proof.
   Admitted.
   □
```

---

[28] True
[29] False
[30] False
[31] True
[32] True
[33] False
[34] X=0
[35] X=1
[36] X=1
[37] X=100

Exercise: 1 star, standard, optional (hoare_pre_false) Prove that if P holds in no state, then any triple with P as its precondition is valid.

```
Theorem hoare_pre_false : ∀ (P Q : Assertion) c,
  (∀ st, ¬ (P st)) →
  {{P}} c {{Q}}.
Proof.
   Admitted.
   □
```

4.5 Proof Rules

The goal of Hoare logic is to provide a *compositional* method for proving the validity of specific Hoare triples. That is, we want the structure of a program's correctness proof to mirror the structure of the program itself. To this end, in the sections below, we'll introduce a rule for reasoning about each of the different syntactic forms of commands in Imp – one for assignment, one for sequencing, one for conditionals, etc. – plus a couple of "structural" rules for gluing things together. We will then be able to prove programs correct using these proof rules, without ever unfolding the definition of valid_hoare_triple.

4.5.1 Skip

Since *skip* doesn't change the state, it preserves any assertion P:

(hoare_skip) [38] skip [39]

```
Theorem hoare_skip : ∀ P,
     {{P}} skip {{P}}.
Proof.
  intros P st st' H HP. inversion H; subst. assumption.
Qed.
```

4.5.2 Sequencing

If command *c1* takes any state where P holds to a state where Q holds, and if *c2* takes any state where Q holds to one where R holds, then doing *c1* followed by *c2* will take any state where P holds to one where R holds:

[40] c1 [41] [42] c2 [43]

---

[38] P
[39] P
[40] P
[41] Q
[42] Q
[43] R

(hoare_seq) [44] c1;c2 [45]

```
Theorem hoare_seq : ∀ P Q R c1 c2,
     {{Q}} c2 {{R}} →
     {{P}} c1 {{Q}} →
     {{P}} c1; c2 {{R}}.
Proof.
  intros P Q R c1 c2 H1 H2 st st' H12 Pre.
  inversion H12; subst.
  eauto.
Qed.
```

Note that, in the formal rule hoare_seq, the premises are given in backwards order (*c2* before *c1*). This matches the natural flow of information in many of the situations where we'll use the rule, since the natural way to construct a Hoare-logic proof is to begin at the end of the program (with the final postcondition) and push postconditions backwards through commands until we reach the beginning.

4.5.3 Assignment

The rule for assignment is the most fundamental of the Hoare logic proof rules. Here's how it works.

Consider this incomplete Hoare triple:

[46] X := Y [47]

We want to assign Y to X and finish in a state where X is 1. What could the precondition be?

One possibility is Y = 1, because if Y is already 1 then assigning it to X causes X to be 1. That leads to a valid Hoare triple:

[48] X := Y [49]

It may seem as though coming up with that precondition must have taken some clever thought. But there is a mechanical way we could have done it: if we take the postcondition X = 1 and in it replace X with Y—that is, replace the left-hand side of the assignment statement with the right-hand side—we get the precondition, Y = 1.

That same idea works in more complicated cases. For example:

[50] X := X + Y [51]

[44] P
[45] R
[46] ???
[47] X=1
[48] Y=1
[49] X=1
[50] ???
[51] X=1

If we replace the X in X = 1 with X + Y, we get X + Y = 1. That again leads to a valid Hoare triple:

[52] X := X + Y [53]

Why does this technique work? The postcondition identifies some property P that we want to hold of the variable X being assigned. In this case, P is "equals 1". To complete the triple and make it valid, we need to identify a precondition that guarantees that property will hold of X. Such a precondition must ensure that the same property holds of *whatever is being assigned to* X. So, in the example, we need "equals 1" to hold of X + Y. That's exactly what the technique guarantees.

In general, the postcondition could be some arbitrary assertion Q, and the right-hand side of the assignment could be some arbitrary arithmetic expression a:

[54] X := a [55]

The precondition would then be Q, but with any occurrences of X in it replaced by a.

Let's introduce a notation for this idea of replacing occurrences: Define Q [X |-> a] to mean "Q where a is substituted in place of X".

This yields the Hoare logic rule for assignment:

[56] X := a [57]

One way of reading this rule is: If you want statement X := a to terminate in a state that satisfies assertion Q, then it suffices to start in a state that also satisfies Q, except where a is substituted for every occurrence of X.

To many people, this rule seems "backwards" at first, because it proceeds from the postcondition to the precondition. Actually it makes good sense to go in this direction: the postcondition is often what is more important, because it characterizes what will be true after running the code.

Nonetheless, it's also possible to formulate a "forward" assignment rule. We'll do that later in some exercises.

Here are some valid instances of the assignment rule:

[58] (that is, X + 1 <= 5) X := X + 1 [59]

[60] (that is, 3 = 3) X := 3 [61]

[62]. (that is, 0 <= 3 /\ 3 <= 5) X := 3 [63]

To formalize the rule, we must first formalize the idea of "substituting an expression for an Imp variable in an assertion", which we refer to as assertion substitution, or assertion_sub.

---

[52] `X+Y=1`
[53] `X=1`
[54] `???`
[55] `Q`
[56] `Q[X|->a]`
[57] `Q`
[58] `(X<=5)[X|->X+1]`
[59] `X<=5`
[60] `(X=3)[X|->3]`
[61] `X=3`
[62] `(0<=X/\X<=5)[X|->3]`
[63] `0<=X/\X<=5`

Intuitively, given a proposition P, a variable X, and an arithmetic expression a, we want to derive another proposition P' that is just the same as P except that P' should mention a wherever P mentions X.

This operation is related to the idea of substituting Imp expressions for Imp variables that we saw in Equiv (subst_aexp and friends). The difference is that, here, P is an arbitrary Rocq assertion, so we can't directly "edit" its text.

However, we can achieve the same effect by evaluating P in an updated state, defined as follows:

```
Definition assertion_sub X (a:aexp) (P:Assertion) : Assertion :=
  fun (st : state) ⇒
    (P%_assertion) (X !-> ((a:Aexp) st); st).

Notation "P [ X |-> a ]" := (assertion_sub X a P)
                              (in custom assn at level 10, left associativity,
                               P custom assn, X global, a custom com)
                            : assertion_scope.
```

This notation allows us to write this operation as:

P X |-> a

That is, P [X |-> a] stands for an assertion – let's call it P' – that behaves just like P except that, wherever P looks up the variable X in the current state, P' instead uses the value of the expression a.

To see how this works in more detail, let's calculate what happens with a couple of examples. First, suppose P' is (X ≤ 5) [X |-> 3] – that is, more formally, P' is the Rocq expression

fun st => (fun st' => st' X <= 5) (X !-> aeval st 3 ; st),

which simplifies to

fun st => (fun st' => st' X <= 5) (X !-> 3 ; st)

and further simplifies to

fun st => ((X !-> 3 ; st) X) <= 5

and finally to

fun st => 3 <= 5.

That is, P' is the assertion that 3 is less than or equal to 5 (as expected).

For a more interesting example, suppose P' is (X ≤ 5) [X |-> X + 1]. Formally, P' is the Rocq expression

fun st => (fun st' => st' X <= 5) (X !-> aeval st (X + 1); st),

which simplifies to

fun st => (X !-> aeval st (X + 1) ; st) X <= 5

and further simplifies to

fun st => (aeval st (X + 1)) <= 5.

That is, P' is the assertion that X + 1 is at most 5.

We can demonstrate formally that we have captured intuitive meaning of "assertion substitution" by proving some example logical equivalences:

```
Module ExampleAssertionSub.
Example equivalent_assertion1 :
  {{ (X ≤ 5) [X |-> 3] }} «-» {{ 3 ≤ 5 }}.
Proof.
  split; unfold assert_implies, assertion_sub; intros st H;
  simpl in *; apply H.
Qed.

Example equivalent_assertion2 :
  {{ (X ≤ 5) [X |-> X + 1] }} «-» {{ (X + 1) ≤ 5 }}.
Proof.
  split; unfold assert_implies, assertion_sub; intros st H;
  simpl in *; apply H.
Qed.
End ExampleAssertionSub.
```

Now, using the substitution operation we've just defined, we can give the precise proof rule for assignment:

(hoare_asgn) [64] X := a [65]

We can prove formally that this rule is indeed valid.

```
Theorem hoare_asgn : ∀ Q X (a:aexp),
  {{Q [X |-> a]}} X := a {{Q}}.
Proof.
  intros Q X a st st' HE HQ.
  inversion HE. subst.
  unfold assertion_sub in HQ. simpl in HQ. assumption. Qed.
```

Here's a first formal proof of a Hoare triple using this rule.

```
Example assertion_sub_example :
  {{(X < 5) [X |-> X + 1]}}
    X := X + 1
  {{X < 5}}.
Proof.
  apply hoare_asgn. Qed.
```

Of course, we'd probably prefer to work with this simpler triple: [66] X := X + 1 [67]

We will see how to do so in the next section.

Complete these Hoare triples by providing an appropriate precondition using ∃, then prove then with `apply hoare_asgn`. If you find that tactic doesn't suffice, double check that

[64] `Q[X|->a]`
[65] `Q`
[66] `X<4`
[67] `X<5`

you have completed the triple properly.

Exercise: 2 stars, standard, optional (hoare_asgn_examples1) `Example hoare_asgn_examples1 :`

```
  ∃ P,
    {{ P }}
      X := 2 × X
    {{ X ≤ 10 }}.
Proof.
   Admitted.
```

□

Exercise: 2 stars, standard, optional (hoare_asgn_examples2) `Example hoare_asgn_examples2 :`

```
  ∃ P,
    {{ P }}
      X := 3
    {{ 0 ≤ X ∧ X ≤ 5 }}.
Proof. Admitted.
```

□

Exercise: 2 stars, standard, especially useful (hoare_asgn_wrong) The assignment rule looks backward to almost everyone the first time they see it. If it still seems puzzling to you, it may help to think a little about alternative "forward" rules. Here is a seemingly natural one:

(hoare_asgn_wrong) [68] X := a [69]

Give a counterexample showing that this rule is incorrect and use it to complete the proof below, showing that it is really a counterexample. (Hint: The rule universally quantifies over the arithmetic expression a, so your counterexample needs to exhibit an a for which the rule doesn't work.)

```
Theorem hoare_asgn_wrong : ∃ a:aexp,
  ¬ {{ True }} X := a {{ X = a }}.
Proof.
   Admitted.
```

Exercise: 3 stars, advanced, optional (hoare_asgn_fwd) By using a *parameter* m (a Rocq number) to remember the original value of X we can define a Hoare rule for assignment that does, intuitively, "work forwards" rather than backwards.

---

[68] True

[69] X=a

(hoare_asgn_fwd) [70] X := a [71]

Note that we need to write out the postcondition in "desugared" form, because it needs to talk about two different states: we use the original value of X to reconstruct the state *st'* before the assignment took place. (Also note that this rule is more complicated than hoare_asgn!)

Prove that this rule is correct.

```
Theorem hoare_asgn_fwd :
  ∀ (m:nat) (a:aexp) (P : Assertion),
  {{P ∧ X = m}}
    X := a
  {{ $(fun st ⇒ (P (X !-> m ; st)
              ∧ st X = aeval (X !-> m ; st) a)) }}.
Proof.
   Admitted.
   □
```

Exercise: 2 stars, advanced, optional (hoare_asgn_fwd_exists) Another way to define a forward rule for assignment is to existentially quantify over the previous value of the assigned variable. Prove that it is correct.

(hoare_asgn_fwd_exists) [72] X := a [73]

```
Theorem hoare_asgn_fwd_exists :
  ∀ a (P : Assertion),
  {{ P }}
    X := a
  {{ $(fun st ⇒ ∃ m, P (X !-> m ; st) ∧
                  st X = aeval (X !-> m ; st) a) }}.
Proof.
   Admitted.
   □
```

4.5.4 Consequence

Sometimes the preconditions and postconditions we get from the Hoare rules won't quite be the ones we want in the particular situation at hand – they may be logically equivalent but have a different syntactic form that fails to unify with the goal we are trying to prove,

[70] funst=>Pst/\stX=m

[71] funst=>P(X!->m;st)/\stX=aeval(X!->m;st)a

[72] funst=>Pst

[73] funst=>existsm,P(X!->m;st)/\stX=aeval(X!->m;st)a

or they actually may be logically weaker (for preconditions) or stronger (for postconditions) than what we need.

For instance,

[74] X := 3 [75],

follows directly from the assignment rule, but

[76] X := 3 [77]

does not. This triple is valid, but it is not an instance of hoare_asgn because **True** and (X = 3) [X |-> 3] are not syntactically equal assertions.

However, they are logically *equivalent*, so if one triple is valid, then the other must certainly be as well. We can capture this observation with the following rule:

[78] c [79] P - P'

---

[80] c [81]

Taking this line of thought a bit further, we can see that strengthening the precondition or weakening the postcondition of a valid triple always produces another valid triple. This observation is captured by two *Rules of Consequence*.

[82] c [83] P -» P'

(hoare_consequence_pre) [84] c [85]

[86] c [87] Q' -» Q

(hoare_consequence_post) [88] c [89]

Here are the formal versions:

```
Theorem hoare_consequence_pre : ∀ (P P' Q : Assertion) c,
  {{P'}} c {{Q}} →
  P -» P' →
  {{P}} c {{Q}}.
Proof.
```

---

[74] (X=3)[X|->3]
[75] X=3
[76] True
[77] X=3
[78] P'
[79] Q
[80] P
[81] Q
[82] P'
[83] Q
[84] P
[85] Q
[86] P
[87] Q'
[88] P
[89] Q

```
  unfold valid_hoare_triple, "-»".
  intros P P' Q c Hhoare Himp st st' Heval Hpre.
  apply Hhoare with (st := st).
  - assumption.
  - apply Himp. assumption.
Qed.

Theorem hoare_consequence_post : ∀ (P Q Q' : Assertion) c,
  {{P}} c {{Q'}} →
  Q' -» Q →
  {{P}} c {{Q}}.
Proof.
  unfold valid_hoare_triple, "-»".
  intros P Q Q' c Hhoare Himp st st' Heval Hpre.
  apply Himp.
  apply Hhoare with (st := st).
  - assumption.
  - assumption.
Qed.
```

For example, we can use the first consequence rule like this:
[90] -» [91] X := 1 [92]
Or, formally...

```
Example hoare_asgn_example1 :
  {{True}} X := 1 {{X = 1}}.
Proof.
  eapply hoare_consequence_pre.
  - apply hoare_asgn.
  - unfold "-»", assertion_sub, t_update; simpl.
    intros st _. reflexivity.
Qed.
```

We can also use it to prove the example mentioned earlier.
[93] -» [94] X := X + 1 [95]
Or, formally ...

```
Example assertion_sub_example2 :
  {{X < 4}}
    X := X + 1
```

[90] True
[91] (X=1)[X|->1]
[92] X=1
[93] X<4
[94] (X<5)[X|->X+1]
[95] X<5

```
  {{X < 5}}.
Proof.
  eapply hoare_consequence_pre.
  - apply hoare_asgn.
  - unfold "-»", assertion_sub, t_update.
    intros st H. simpl in *. lia.
Qed.
```

Finally, here is a combined rule of consequence that allows us to vary both the precondition and the postcondition.

[96] c [97] P -» P' Q' -» Q

(hoare_consequence) [98] c [99]

```
Theorem hoare_consequence : ∀ (P P' Q Q' : Assertion) c,
  {{P'}} c {{Q'}} →
  P -» P' →
  Q' -» Q →
  {{P}} c {{Q}}.
Proof.
  intros P P' Q Q' c Htriple Hpre Hpost.
  apply hoare_consequence_pre with (P' := P').
  - apply hoare_consequence_post with (Q' := Q'); assumption.
  - assumption.
Qed.
```

4.5.5 Automation

Many of the proofs we have done so far with Hoare triples can be streamlined using the automation techniques that we introduced in the *Auto* chapter of *Logical Foundations*.

Recall that the `auto` tactic can be told to `unfold` definitions as part of its proof search. Let's give it that hint for the definitions and coercions we're using:

```
Hint Unfold assert_implies assertion_sub t_update : core.
Hint Unfold valid_hoare_triple : core.
Hint Unfold assert_of_Prop Aexp_of_nat Aexp_of_aexp : core.
```

Also recall that `auto` will search for a proof involving `intros` and `apply`. By default, the theorems that it will apply include any of the local hypotheses, as well as theorems in the "core" hint database.

---

[96] P'
[97] Q'
[98] P
[99] Q

The proof of hoare_consequence_pre, repeated below, looks like an opportune place for such automation, because all it does is `unfold`, `intros`, and `apply`. (It uses `assumption`, too, but that's just application of a hypothesis.)

```
Theorem hoare_consequence_pre' : ∀ (P P' Q : Assertion) c,
  {{P'}} c {{Q}} →
  P -» P' →
  {{P}} c {{Q}}.
Proof.
  unfold valid_hoare_triple, "-»".
  intros P P' Q c Hhoare Himp st st' Heval Hpre.
  apply Hhoare with (st := st).
  - assumption.
  - apply Himp. assumption.
Qed.
```

Merely using `auto`, though, doesn't complete the proof.

```
Theorem hoare_consequence_pre'' : ∀ (P P' Q : Assertion) c,
  {{P'}} c {{Q}} →
  P -» P' →
  {{P}} c {{Q}}.
Proof.
  auto. Abort.
```

The problem is the `apply` *Hhoare* `with`... part of the proof. Rocq isn't able to figure out how to instantiate *st* without some help from us. Recall, though, that there are versions of many tactics that will use *existential variables* to make progress even when the regular versions of those tactics would get stuck.

Here, the `eapply` tactic will introduce an existential variable ?*st* as a placeholder for *st*, and *eassumption* will instantiate ?*st* with *st* when it discovers *st* in assumption *Heval*. By using `eapply` we are essentially telling Rocq, "Be patient: The missing part is going to be filled in later in the proof."

```
Theorem hoare_consequence_pre''' : ∀ (P P' Q : Assertion) c,
  {{P'}} c {{Q}} →
  P -» P' →
  {{P}} c {{Q}}.
Proof.
  unfold valid_hoare_triple, "-»".
  intros P P' Q c Hhoare Himp st st' Heval Hpre.
  eapply Hhoare.
  - eassumption.
  - apply Himp. assumption.
Qed.
```

The `eauto` tactic will use `eapply` as part of its proof search. So, the entire proof can

actually be done in just one line.

```
Theorem hoare_consequence_pre'''' : ∀ (P P' Q : Assertion) c,
  {{P'}} c {{Q}} →
  P -» P' →
  {{P}} c {{Q}}.
Proof.
  eauto.
Qed.
```

Of course, it's hard to predict that `eauto` suffices here without having gone through the original proof of hoare_consequence_pre to see the tactics it used. But now that we know `eauto` worked there, it's a good bet that it will also work for hoare_consequence_post.

```
Theorem hoare_consequence_post' : ∀ (P Q Q' : Assertion) c,
  {{P}} c {{Q'}} →
  Q' -» Q →
  {{P}} c {{Q}}.
Proof.
  eauto.
Qed.
```

We can also use `eapply` to streamline a proof (hoare_asgn_example1), that we did earlier as an example of using the consequence rule:

```
Example hoare_asgn_example1' :
  {{True}} X := 1 {{X = 1}}.
Proof.
  eapply hoare_consequence_pre.    - apply hoare_asgn.
  - unfold "-»", assertion_sub, t_update.
    intros st _. simpl. reflexivity.
Qed.
```

The final bullet of that proof also looks like a candidate for automation.

```
Example hoare_asgn_example1'' :
  {{True}} X := 1 {{X = 1}}.
Proof.
  eapply hoare_consequence_pre.
  - apply hoare_asgn.
  - auto.
Qed.
```

Now we have quite a nice proof script: it simply identifies the Hoare rules that need to be used and leaves the remaining low-level details up to Rocq to figure out.

By now it might be apparent that the *entire* proof could be automated if we added hoare_consequence_pre and hoare_asgn to the hint database. We won't do that in this chapter, so that we can get a better understanding of when and how the Hoare rules are

used. In the next chapter, Hoare2, we'll dive deeper into automating entire proofs of Hoare triples.

The other example of using consequence that we did earlier, *hoare_asgn_example2*, requires a little more work to automate. We can streamline the first line with `eapply`, but we can't just use `auto` for the final bullet, since it needs *lia*.

```
Example assertion_sub_example2' :
  {{X < 4}}
    X := X + 1
  {{X < 5}}.
Proof.
  eapply hoare_consequence_pre.
  - apply hoare_asgn.
  - auto.      unfold "->>", assertion_sub, t_update.
    intros st H. simpl in *. lia.
Qed.
```

Let's introduce our own tactic to handle both that bullet and the bullet from example 1:

```
Ltac assertion_auto :=
  try auto;
  try (unfold "->>", assertion_sub, t_update;
       intros; simpl in *; lia).

Example assertion_sub_example2'' :
  {{X < 4}}
    X := X + 1
  {{X < 5}}.
Proof.
  eapply hoare_consequence_pre.
  - apply hoare_asgn.
  - assertion_auto.
Qed.

Example hoare_asgn_example1''':
  {{True}} X := 1 {{X = 1}}.
Proof.
  eapply hoare_consequence_pre.
  - apply hoare_asgn.
  - assertion_auto.
Qed.
```

Again, we have quite a nice proof script. All the low-level details of proofs about assertions have been taken care of automatically. Of course, *assertion_auto* isn't able to prove everything we could possibly want to know about assertions – there's no magic here! But it's pretty good.

Exercise: 2 stars, standard (hoare_asgn_examples_2) Prove these triples. Try to make your proof scripts nicely automated by following the examples above.

```
Example assertion_sub_ex1' :
  {{ X ≤ 5 }}
    X := 2 × X
  {{ X ≤ 10 }}.
Proof.
   Admitted.

Example assertion_sub_ex2' :
  {{ 0 ≤ 3 ∧ 3 ≤ 5 }}
    X := 3
  {{ 0 ≤ X ∧ X ≤ 5 }}.
Proof.
   Admitted.
```

□

4.5.6 Sequencing + Assignment

Here's an example of a program involving both sequencing and assignment. Note the use of hoare_seq in conjunction with hoare_consequence_pre and the `eapply` tactic.

```
Example hoare_asgn_example3 : ∀ (a:aexp) (n:nat),
  {{a = n}}
    X := a;
    skip
  {{X = n}}.
Proof.
  intros a n. eapply hoare_seq.
  -
    apply hoare_skip.
  -
    eapply hoare_consequence_pre.
    + apply hoare_asgn.
    + assertion_auto.
Qed.
```

Informally, a nice way of displaying a proof using the sequencing rule is as a "decorated program" where the intermediate assertion Q is written between *c1* and *c2*:
[100] X := a [101]; <— decoration for Q skip [102] We'll come back to the idea of decorated programs in much more detail in the next chapter.

[100] a=n
[101] X=n
[102] X=n

Exercise: 2 stars, standard, especially useful (hoare_asgn_example4) Translate this "decorated program" into a formal proof:

[103] -» [104] X := 1 [105] -» [106]; Y := 2 [107]

Note the use of "-»" decorations, each marking a use of hoare_consequence_pre.

We've started you off by providing a use of hoare_seq that explicitly identifies X = 1 as the intermediate assertion.

```
Example hoare_asgn_example4 :
  {{ True }}
    X := 1;
    Y := 2
  {{ X = 1 ∧ Y = 2 }}.
Proof.
  eapply hoare_seq with (Q := {{ X = 1 }}).
   Admitted.
```

□

Exercise: 3 stars, standard (swap_exercise) Write an Imp program c that swaps the values of X and Y and show that it satisfies the following specification:

[108] c [109]

Your proof should not need to use `unfold valid_hoare_triple`.

Hints:

- Remember that Imp commands need to be enclosed in <{...}> brackets.
- Remember that the assignment rule works best when it's applied "back to front," from the postcondition to the precondition. So your proof will want to start at the end and work back to the beginning of your program.
- Remember that `eapply` is your friend.)

```
Definition swap_program : com
  . Admitted.

Theorem swap_exercise :
  {{X ≤ Y}}
    swap_program
  {{Y ≤ X}}.
```

[103] `True`
[104] `1=1`
[105] `X=1`
[106] `X=1/\2=2`
[107] `X=1/\Y=2`
[108] `X<=Y`
[109] `Y<=X`

```
Proof.
   Admitted.
```
□

Exercise: 4 stars, advanced (invalid_triple) Show that

[110] X := 3; Y := a [111]

is not a valid Hoare triple for some choices of a and n.

Conceptual hint: Invent a particular a and n for which the triple in invalid, then use those to complete the proof.

Technical hint: Hypothesis H below begins $\forall\ a\ n$, ... You'll want to instantiate that with the particular a and n you've invented. You can do that with `assert` and `apply`, but you may remember (from *Tactics.v* in Logical Foundations) that Rocq offers an even easier tactic: `specialize`. If you write

specialize H with (a := your_a) (n := your_n)

the hypothesis will be instantiated on *your_a* and *your_n*.

Having chosen your a and n, proceed as follows:

- Use the (assumed) validity of the given hoare triple to derive a state *st'* in which Y has some value *y1*
- Use the evaluation rules (E_Seq and E_Asgn) to show that Y has a *different* value *y2* in the same final state *st'*
- Since *y1* and *y2* are both equal to *st'* Y, they are equal to each other. But we chose them to be different, so this is a contradiction, which finishes the proof.

```
Theorem invalid_triple : ¬ ∀ (a : aexp) (n : nat),
    {{ a = n }}
      X := 3; Y := a
    {{ Y = n }}.
Proof.
  unfold valid_hoare_triple.
  intros H.
   Admitted.
```
□

4.5.7 Conditionals

What sort of rule do we want for reasoning about conditional commands?

Certainly, if the same assertion Q holds after executing either of the branches, then it holds after the whole conditional. So we might be tempted to write:

[110] a=n
[111] Y=n

[112] c1 [113] [114] c2 [115]

---

[116] if b then c1 else c2 [117]

However, this is rather weak. For example, using this rule, we cannot show

[118] if X = 0 then Y := 2 else Y := X + 1 end [119]

since the rule doesn't tell us enough about the state in which the assignments take place in the "then" and "else" branches.

Fortunately, we can say something more precise. In the "then" branch, we know that the boolean expression b evaluates to true, and in the "else" branch, we know it evaluates to false. Making this information available in the premises of the rule gives us more information to work with when reasoning about the behavior of *c1* and *c2* (i.e., the reasons why they establish the postcondition Q).

[120] c1 [121] [122] c2 [123]

(hoare_if) [124] if b then c1 else c2 end [125]

To interpret this rule formally, we need to do a little work. Strictly speaking, the assertion we've written, $P \wedge b$, is the conjunction of an assertion and a boolean expression – i.e., it doesn't typecheck. To fix this, we need a way of formally "lifting" any bexp b to an assertion. We'll write bassertion b for the assertion "the boolean expression b evaluates to true (in the given state)."

```
Definition bassertion b : Assertion :=
  fun st ⇒ (beval st b = true).

Coercion bassertion : bexp >-> Assertion.

Arguments bassertion /.
```

A useful fact about bassertion:

```
Lemma bexp_eval_false : ∀ b st,
  beval st b = false → ¬ ((bassertion b) st).
Proof. congruence. Qed.

Hint Resolve bexp_eval_false : core.
```

---

[112] P
[113] Q
[114] P
[115] Q
[116] P
[117] Q
[118] True
[119] X<=Y
[120] P/\b
[121] Q
[122] P/\~b
[123] Q
[124] P
[125] Q

We mentioned the `congruence` tactic in passing in *Auto* when building the *find_rwd* tactic. Like *find_rwd*, `congruence` is able to automatically find that both beval *st b* = false and beval *st b* = `true` are being assumed, notice the contradiction, and `discriminate` to complete the proof.

Now we can formalize the Hoare proof rule for conditionals and prove it correct.

```
Theorem hoare_if : ∀ P Q (b:bexp) c1 c2,
  {{ P ∧ b }} c1 {{Q}} →
  {{ P ∧ ¬ b}} c2 {{Q}} →
  {{P}} if b then c1 else c2 end {{Q}}.
```

That is (unwrapping the notations):

Theorem hoare_if : forall P Q b c1 c2, [126] c1 [127] -> [128] c2 [129] -> [130] if b then c1 else c2 end [131].

```
Proof.
  intros P Q b c1 c2 HTrue HFalse st st' HE HP.
  inversion HE; subst; eauto.
Qed.
```

Example

Here is a formal proof that the program we used to motivate the rule satisfies the specification we wanted.

```
Example if_example :
  {{True}}
    if (X = 0)
      then Y := 2
      else Y := X + 1
    end
  {{X ≤ Y}}.
Proof.
  apply hoare_if.
  -
    eapply hoare_consequence_pre.
    + apply hoare_asgn.
    + assertion_auto.        unfold "->>", assertion_sub, t_update, bassertion.
      simpl. intros st [_ H]. apply eqb_eq in H.
      rewrite H. lia.
  -
```

[126] `funst=>Pst/\bassertionbst`
[127] `Q`
[128] `funst=>Pst/\~(bassertionbst)`
[129] `Q`
[130] `P`
[131] `Q`

```
    eapply hoare_consequence_pre.
    + apply hoare_asgn.
    + assertion_auto.
Qed.
```

As we did earlier, it would be nice to eliminate all the low-level proof script that isn't about the Hoare rules. Unfortunately, the *assertion_auto* tactic we wrote wasn't quite up to the job. Looking at the proof of if_example, we can see why. We had to unfold a definition (bassertion) and use a theorem (eqb_eq) that we didn't need in earlier proofs. So, let's add those into our tactic, and clean it up a little in the process.

```
Ltac assertion_auto' :=
  unfold "->>", assertion_sub, t_update, bassertion;
  intros; simpl in *;
  try rewrite → eqb_eq in *;
  auto; try lia.
```

Now the proof is quite streamlined.

```
Example if_example'' :
  {{True}}
    if X = 0
      then Y := 2
      else Y := X + 1
    end
  {{X ≤ Y}}.
Proof.
  apply hoare_if.
  - eapply hoare_consequence_pre.
    + apply hoare_asgn.
    + assertion_auto'.
  - eapply hoare_consequence_pre.
    + apply hoare_asgn.
    + assertion_auto'.
Qed.
```

We can even shorten it a little bit more.

```
Example if_example''' :
  {{True}}
    if X = 0
      then Y := 2
      else Y := X + 1
    end
  {{X ≤ Y}}.
Proof.
  apply hoare_if; eapply hoare_consequence_pre;
```

```
    try apply hoare_asgn; try assertion_auto'.
Qed.
```

For later proofs, it will help to extend *assertion_auto'* to handle inequalities, too.

```
Ltac assertion_auto'' :=
  unfold "->>", assertion_sub, t_update, bassertion;
  intros; simpl in *;
  try rewrite → eqb_eq in *;
  try rewrite → leb_le in *;
  auto; try lia.
```

Exercise: 2 stars, standard (if_minus_plus) Prove the theorem below using hoare_if. Do not use `unfold valid_hoare_triple`. The *assertion_auto''* tactic we just defined may be useful.

```
Theorem if_minus_plus :
  {{True}}
    if (X ≤ Y)
      then Z := Y - X
      else Y := X + Z
    end
  {{Y = X + Z}}.
Proof.
   Admitted.
   □
```

Exercise: One-sided conditionals

In this exercise we consider extending Imp with "one-sided conditionals" of the form *if1 b* `then` *c* `end`. Here *b* is a boolean expression, and *c* is a command. If *b* evaluates to `true`, then command *c* is evaluated. If *b* evaluates to `false`, then *if1 b* `then` *c* `end` does nothing.

We recommend that you complete this exercise before attempting the ones that follow, as it should help solidify your understanding of the material.

The first step is to extend the syntax of commands and introduce the usual notations. (We've done this for you, in a separate module to prevent polluting the global name space.)

```
Module If1.

Inductive com : Type :=
  | CSkip : com
  | CAsgn : string → aexp → com
  | CSeq : com → com → com
  | CIf : bexp → com → com → com
  | CWhile : bexp → com → com
  | CIf1 : bexp → com → com.
```

```
Notation "'if1' x 'then' y 'end'" :=
         (CIf1 x y)
             (in custom com at level 0, x custom com at level 99).
Notation "'skip'" := CSkip
  (in custom com at level 0) : com_scope.
Notation "x := y" := (CAsgn x y)
  (in custom com at level 0, x constr at level 0, y at level 85, no associativity,
    format "x := y") : com_scope.
Notation "x ; y" := (CSeq x y)
  (in custom com at level 90,
    right associativity,
    format "'[v' x ; '/' y ']'") : com_scope.
Notation "'if' x 'then' y 'else' z 'end'" := (CIf x y z)
  (in custom com at level 89, x at level 99, y at level 99, z at level 99,
    format "'[v' 'if' x 'then' '/ ' y '/' 'else' '/ ' z '/' 'end' ']'") : com_scope.
Notation "'while' x 'do' y 'end'" := (CWhile x y)
  (in custom com at level 89, x at level 99, y at level 99,
    format "'[v' 'while' x 'do' '/ ' y '/' 'end' ']'") : com_scope.
```

Exercise: 2 stars, standard, especially useful (if1_ceval) Add two new evaluation rules to relation **ceval**, below, for *if1*. Let the rules for `if` guide you.

```
Reserved Notation
         "st0 '=[' c ']=>' st1 '/' s"
         (at level 40, c custom com at level 99,
          st0 constr, st1 constr at next level,
          format "'[hv' st0 =[ '/ ' '[' c ']' '/' ]=> st1 / s ']'").

Inductive ceval : com → state → state → Prop :=
  | E_Skip : ∀ st,
      st =[ skip ]=> st
  | E_Asgn : ∀ st a1 n x,
      aeval st a1 = n →
      st =[ x := a1 ]=> (x !-> n ; st)
  | E_Seq : ∀ c1 c2 st st' st'',
      st =[ c1 ]=> st' →
      st' =[ c2 ]=> st'' →
      st =[ c1 ; c2 ]=> st''
  | E_IfTrue : ∀ st st' b c1 c2,
      beval st b = true →
      st =[ c1 ]=> st' →
      st =[ if b then c1 else c2 end ]=> st'
  | E_IfFalse : ∀ st st' b c1 c2,
      beval st b = false →
```

```
      st =[ c2 ]=> st' →
      st =[ if b then c1 else c2 end ]=> st'
  | E_WhileFalse : ∀ b st c,
      beval st b = false →
      st =[ while b do c end ]=> st
  | E_WhileTrue : ∀ st st' st'' b c,
      beval st b = true →
      st =[ c ]=> st' →
      st' =[ while b do c end ]=> st'' →
      st =[ while b do c end ]=> st''


where "st '=[' c ']=>' st'" := (ceval c st st').

Hint Constructors ceval : core.
```

The following unit tests should be provable simply by `eauto` if you have defined the rules for *if1* correctly.

```
Example if1true_test :
  empty_st =[ if1 X = 0 then X := 1 end ]=> (X !-> 1).
Proof. Admitted.

Example if1false_test :
  (X !-> 2) =[ if1 X = 0 then X := 1 end ]=> (X !-> 2).
Proof. Admitted.
```

□

Now we have to repeat the definition and notation of Hoare triples, so that they will use the updated **com** type.

```
Definition valid_hoare_triple
           (P : Assertion) (c : com) (Q : Assertion) : Prop :=
  ∀ st st',
       st =[ c ]=> st' →
       P st →
       Q st'.

Hint Unfold valid_hoare_triple : core.

Notation "{{ P }} c {{ Q }}" :=
  (valid_hoare_triple P c Q)
    (at level 2, P custom assn at level 99, c custom com at level 99, Q custom assn
at level 99)
    : hoare_spec_scope.
```

Exercise: 2 stars, standard (hoare_if1) Invent a Hoare logic proof rule for *if1*. State and prove a theorem named *hoare_if1* that shows the validity of your rule. Use hoare_if as a

guide. Try to invent a rule that is *complete*, meaning it can be used to prove the correctness of as many one-sided conditionals as possible. Also try to keep your rule *compositional*, meaning that any Imp command that appears in a premise should syntactically be a part of the command in the conclusion.

Hint: if you encounter difficulty getting Rocq to parse part of your rule as an assertion, try manually indicating that it should be in the assertion scope. For example, if you want e to be parsed as an assertion, write it as $(e)\%assertion$.

For example (hoare_if1_good) your rule should be strong enough to show the following Hoare triple is valid:

[132] if1 Y <> 0 then X := X + Y end [133]

```
Definition manual_grade_for_hoare_if1 : option (nat×string) := None.
```

□

Before the next exercise, we need to restate the Hoare rules of consequence (for preconditions) and assignment for the new **com** type.

```
Theorem hoare_consequence_pre : ∀ (P P' Q : Assertion) c,
  {{P'}} c {{Q}} →
  P -» P' →
  {{P}} c {{Q}}.
Proof.
  eauto.
Qed.

Theorem hoare_asgn : ∀ Q X a,
  {{Q [X |-> a]}} (X := a) {{Q}}.
Proof.
  intros Q X a st st' Heval HQ.
  inversion Heval; subst.
  auto.
Qed.
```

Exercise: 2 stars, standard (hoare_if1_good) Use your *if1* rule to prove the following (valid) Hoare triple.

Hint: *assertion_auto''* will once again get you most but not all the way to a completely automated proof. You can finish manually, or tweak the tactic further.

Hint: If you see a message like *Unable to unify* "*Imp.ceval Imp.CSkip st st'*" `with`..., it probably means you are using a definition or theorem *e.g.*, hoare_skip from above this exercise without re-proving it for the new version of Imp with if1.

```
Lemma hoare_if1_good :
  {{ X + Y = Z }}
    if1 Y ≠ 0 then
```

[132] X+Y=Z
[133] X=Z

```
      X := X + Y
    end
  {{ X = Z }}.
Proof. Admitted.
```
□

```
End IF1.
```

4.5.8 While Loops

The Hoare rule for *while* loops is based on the idea of a *command invariant* (or just *invariant*): an assertion whose truth is guaranteed after executing a command, assuming it is true before.

That is, an assertion P is a command invariant of c if

[134] c [135]

holds. Note that the command invariant might temporarily become false in the middle of executing c, but by the end of c it must be restored.

As a first attempt at a *while* rule, we could try:

[136] c [137]

---

while b do c end P[138]}

This rule is valid: if P is a command invariant of c, as the premise requires, then, no matter how many times the loop body executes, P is going to be true when the loop finally finishes.

But the rule also omits two crucial pieces of information. First, the loop terminates when b becomes false. So we can strengthen the postcondition in the conclusion:

[139] c [140]

---

while b do c end P / b[141]}

Second, the loop body will be executed only if b is true. So we can also strengthen the precondition in the premise:

[142] c [143]

(hoare_while) while b do c end P / b[144]}

---

[134] P
[135] P
[136] P
[137] P
[138] P
[139] P
[140] P
[141] P
[142] P/\b
[143] P
[144] P

That is the Hoare *while* rule. Note how it combines aspects of *skip* and conditionals:

- If the loop body executes zero times, the rule is like *skip* in that the precondition survives to become (part of) the postcondition.
- Like a conditional, we can assume guard b holds on entry to the subcommand.

```
Theorem hoare_while : ∀ P (b:bexp) c,
  {{P ∧ b}} c {{P}} →
  {{P}} while b do c end {{P ∧ ¬ b}}.
Proof.
  intros P b c Hhoare st st' Heval HP.
  remember <{while b do c end}> as original_command eqn:Horig.
  induction Heval;
    try (inversion Horig; subst; clear Horig);
    eauto.
Qed.
```

We call P a *loop invariant* of *while* b `do` c `end` if

[145] c [146]

is a valid Hoare triple.

This means that P will be true at the end of the loop body whenever the loop body executes. If P contradicts b, this holds trivially since the precondition is false.

For instance, X = 0 is a loop invariant of

while X = 2 do X := 1 end

since the program will never enter the loop.

The program

while Y > 10 do Y := Y - 1; Z := Z + 1 end

admits an interesting loop invariant:

X = Y + Z

Note that this doesn't contradict the loop guard but neither is it a command invariant of

Y := Y - 1; Z := Z + 1

since, if X = 5, Y = 0 and Z = 5, running the command will set Y + Z to 6. The loop guard Y > 10 guarantees that this will not be the case. We will see many such loop invariants in the following chapter.

```
Example while_example :
  {{X ≤ 3}}
    while (X ≤ 2) do
      X := X + 1
    end
```

[145] `P/\b`
[146] `P`

```
  {{X = 3}}.
 Proof.
  eapply hoare_consequence_post.
  - apply hoare_while.
    eapply hoare_consequence_pre.
    + apply hoare_asgn.
    + assertion_auto''.
  - assertion_auto''.
Qed.
```

If the loop never terminates, any postcondition will work.

```
Theorem always_loop_hoare : ∀ Q,
  {{True}} while true do skip end {{Q}}.
Proof.
  intros Q.
  eapply hoare_consequence_post.
  - apply hoare_while. apply hoare_post_true. auto.
  - simpl. intros st [Hinv Hguard]. congruence.
Qed.
```

Of course, this result is not surprising if we remember that the definition of valid_hoare_triple asserts that the postcondition must hold *only* when the command terminates. If the command doesn't terminate, we can prove anything we like about the post-condition.

Hoare rules that specify what happens *if* commands terminate, without proving that they do, are said to describe a logic of *partial* correctness. It is also possible to give Hoare rules for *total* correctness, which additionally specifies that commands must terminate. Total correctness is out of the scope of this textbook.

Exercise: *REPEAT*

Exercise: 4 stars, advanced, optional (hoare_repeat) In this exercise, we'll add a new command to our language of commands: *REPEAT* c `until` b `end`. You will write the evaluation rule for *REPEAT* and add a new Hoare rule to the language for programs involving it. (You may recall that the evaluation rule is given in an example in the *Auto* chapter. Try to figure it out yourself here rather than peeking.)

```
Module RepeatExercise.

Inductive com : Type :=
  | CSkip : com
  | CAsgn : string → aexp → com
  | CSeq : com → com → com
  | CIf : bexp → com → com → com
  | CWhile : bexp → com → com
  | CRepeat : com → bexp → com.
```

REPEAT behaves like *while*, except that the loop guard is checked *after* each execution of the body, with the loop repeating as long as the guard stays *false*. Because of this, the body will always execute at least once.

```
Notation "'repeat' e1 'until' b2 'end'" :=
          (CRepeat e1 b2)
              (in custom com at level 0,
               e1 custom com at level 99, b2 custom com at level 99).
Notation "'skip'" := CSkip
  (in custom com at level 0) : com_scope.
Notation "x := y" := (CAsgn x y)
  (in custom com at level 0, x constr at level 0, y at level 85, no associativity,
    format "x := y") : com_scope.
Notation "x ; y" := (CSeq x y)
  (in custom com at level 90,
    right associativity,
    format "'[v' x ; '/' y ']'") : com_scope.
Notation "'if' x 'then' y 'else' z 'end'" := (CIf x y z)
  (in custom com at level 89, x at level 99, y at level 99, z at level 99,
    format "'[v' 'if' x 'then' '/  ' y '/' 'else' '/  ' z '/' 'end' ']'") : com_scope.
Notation "'while' x 'do' y 'end'" := (CWhile x y)
  (in custom com at level 89, x at level 99, y at level 99,
    format "'[v' 'while' x 'do' '/  ' y '/' 'end' ']'") : com_scope.
```

Add new rules for *REPEAT* to **ceval** below. You can use the rules for *while* as a guide, but remember that the body of a *REPEAT* should always execute at least once, and that the loop ends when the guard becomes true.

```
Inductive ceval : state → com → state → Prop :=
  | E_Skip : ∀ st,
      st =[ skip ]=> st
  | E_Asgn : ∀ st a1 n x,
      aeval st a1 = n →
      st =[ x := a1 ]=> (x !-> n ; st)
  | E_Seq : ∀ c1 c2 st st' st'',
      st =[ c1 ]=> st' →
      st' =[ c2 ]=> st'' →
      st =[ c1 ; c2 ]=> st''
  | E_IfTrue : ∀ st st' b c1 c2,
      beval st b = true →
      st =[ c1 ]=> st' →
      st =[ if b then c1 else c2 end ]=> st'
  | E_IfFalse : ∀ st st' b c1 c2,
      beval st b = false →
      st =[ c2 ]=> st' →
```

```
      st =[ if b then c1 else c2 end ]=> st'
  | E_WhileFalse : ∀ b st c,
      beval st b = false →
      st =[ while b do c end ]=> st
  | E_WhileTrue : ∀ st st' st'' b c,
      beval st b = true →
      st =[ c ]=> st' →
      st' =[ while b do c end ]=> st'' →
      st =[ while b do c end ]=> st''


where "st '=[' c ']=>' st'" := (ceval st c st').
```

A couple of definitions from above, copied here so they use the new **ceval**.

```
Definition valid_hoare_triple (P : Assertion) (c : com) (Q : Assertion)
                          : Prop :=
  ∀ st st', st =[ c ]=> st' → P st → Q st'.

Notation "{{ P }} c {{ Q }}" :=
  (valid_hoare_triple P c Q)
    (at level 2, P custom assn at level 99, c custom com at level 99, Q custom assn
at level 99)
    : hoare_spec_scope.
```

To make sure you've got the evaluation rules for `repeat` right, prove that ex1_repeat evaluates correctly.

```
Definition ex1_repeat :=
  <{ repeat
       X := 1;
       Y := Y + 1
     until (X = 1) end }>.

Theorem ex1_repeat_works :
  empty_st =[ ex1_repeat ]=> (Y !-> 1 ; X !-> 1).
Proof.
   Admitted.
```

Now state and prove a theorem, *hoare_repeat*, that expresses an appropriate proof rule for `repeat` commands. Use hoare_while as a model, and try to make your rule as precise as possible.

For full credit, make sure (informally) that your rule can be used to prove the following valid Hoare triple:

[147] repeat Y := X; X := X - 1 until X = 0 end [148]

[147] `X>0`

[148] `X=0/\Y>0`

```
End RepeatExercise.

Definition manual_grade_for_hoare_repeat : option (nat×string) := None.
```
□

4.6 Summary

So far, we've introduced Hoare Logic as a tool for reasoning about Imp programs.

The rules of Hoare Logic are:

(hoare_asgn) [149] X:=a [150]

(hoare_skip) [151] skip [152]
[153] c1 [154] [155] c2 [156]

(hoare_seq) [157] c1;c2 [158]
[159] c1 [160] [161] c2 [162]

(hoare_if) [163] if b then c1 else c2 end [164]
[165] c [166]

(hoare_while) [167] while b do c end [168]
[169] c [170] P -» P' Q' -» Q

---

[149] `Q[X|->a]`
[150] `Q`
[151] `P`
[152] `P`
[153] `P`
[154] `Q`
[155] `Q`
[156] `R`
[157] `P`
[158] `R`
[159] `P/\b`
[160] `Q`
[161] `P/\~b`
[162] `Q`
[163] `P`
[164] `Q`
[165] `P/\b`
[166] `P`
[167] `P`
[168] `P/\~b`
[169] `P'`
[170] `Q'`

(hoare_consequence) [171] c [172]

Our main task in this chapter has been to *define* the rules of Hoare logic, and prove that the definitions are sound. Having done so, we can go on and work *within* Hoare logic to prove that particular programs satisfy particular Hoare triples. In the next chapter, we'll see how Hoare logic is can be used to prove that more interesting programs satisfy interesting specifications of their behavior.

Crucially, we will do so without ever again `unfold`ing the definition of Hoare triples – i.e., we will take the rules of Hoare logic as a closed world for reasoning about programs.

4.7 Additional Exercises

4.7.1 Havoc

In this exercise, we will derive proof rules for a *HAVOC* command, which is similar to the nondeterministic *any* expression from the the Imp chapter.

First, we enclose this work in a separate module, and recall the syntax and big-step semantics of Himp commands.

```
Module Himp.

Inductive com : Type :=
  | CSkip : com
  | CAsgn : string → aexp → com
  | CSeq : com → com → com
  | CIf : bexp → com → com → com
  | CWhile : bexp → com → com
  | CHavoc : string → com.

Notation "'havoc' l" := (CHavoc l)
                          (in custom com at level 60, l constr at level 0).
Notation "'skip'" := CSkip
  (in custom com at level 0) : com_scope.
Notation "x := y" := (CAsgn x y)
  (in custom com at level 0, x constr at level 0, y at level 85, no associativity,
    format "x := y") : com_scope.
Notation "x ; y" := (CSeq x y)
  (in custom com at level 90,
    right associativity,
    format "'[v' x ; '/' y ']'") : com_scope.
Notation "'if' x 'then' y 'else' z 'end'" := (CIf x y z)
  (in custom com at level 89, x at level 99, y at level 99, z at level 99,
    format "'[v' 'if' x 'then' '/  ' y '/' 'else' '/  ' z '/' 'end' ']'") : com_scope.
```

[171] P

[172] Q

```
Notation "'while' x 'do' y 'end'" := (CWhile x y)
  (in custom com at level 89, x at level 99, y at level 99,
    format "'[v' 'while' x 'do' '/  ' y '/' 'end' ']'") : com_scope.

Inductive ceval : com → state → state → Prop :=
  | E_Skip : ∀ st,
      st =[ skip ]=> st
  | E_Asgn : ∀ st a1 n x,
      aeval st a1 = n →
      st =[ x := a1 ]=> (x !-> n ; st)
  | E_Seq : ∀ c1 c2 st st' st'',
      st =[ c1 ]=> st' →
      st' =[ c2 ]=> st'' →
      st =[ c1 ; c2 ]=> st''
  | E_IfTrue : ∀ st st' b c1 c2,
      beval st b = true →
      st =[ c1 ]=> st' →
      st =[ if b then c1 else c2 end ]=> st'
  | E_IfFalse : ∀ st st' b c1 c2,
      beval st b = false →
      st =[ c2 ]=> st' →
      st =[ if b then c1 else c2 end ]=> st'
  | E_WhileFalse : ∀ b st c,
      beval st b = false →
      st =[ while b do c end ]=> st
  | E_WhileTrue : ∀ st st' st'' b c,
      beval st b = true →
      st =[ c ]=> st' →
      st' =[ while b do c end ]=> st'' →
      st =[ while b do c end ]=> st''
  | E_Havoc : ∀ st x n,
      st =[ havoc x ]=> (x !-> n ; st)

where "st '=[' c ']=>' st'" := (ceval c st st').

Hint Constructors ceval : core.
```

The definition of Hoare triples is exactly as before.

```
Definition valid_hoare_triple (P:Assertion) (c:com) (Q:Assertion) : Prop :=
  ∀ st st', st =[ c ]=> st' → P st → Q st'.

Hint Unfold valid_hoare_triple : core.

Notation "{{ P }} c {{ Q }}" :=
  (valid_hoare_triple P c Q)
    (at level 2, P custom assn at level 99, c custom com at level 99, Q custom assn
```

```
at level 99)
    : hoare_spec_scope.
```

And the precondition consequence rule is exactly as before.

```
Theorem hoare_consequence_pre : ∀ (P P' Q : Assertion) c,
  {{P'}} c {{Q}} →
  P -» P' →
  {{P}} c {{Q}}.
Proof. eauto. Qed.
```

Exercise: 3 stars, advanced (hoare_havoc) Complete the Hoare rule for *HAVOC* commands below by defining havoc_pre, and prove that the resulting rule is correct.

```
Definition havoc_pre (X : string) (Q : Assertion) (st : total_map nat) : Prop
  . Admitted.

Theorem hoare_havoc : ∀ (Q : Assertion) (X : string),
  {{ $(havoc_pre X Q) }} havoc X {{ Q }}.
Proof.
   Admitted.
   □
```

Exercise: 3 stars, advanced (havoc_post) Complete the following proof without changing any of the provided commands. If you find that it can't be completed, your definition of havoc_pre is probably too strong. Find a way to relax it so that havoc_post can be proved.

Hint: the *assertion_auto* tactics we've built won't help you here. You need to proceed manually.

```
Theorem havoc_post : ∀ (P : Assertion) (X : string),
  {{ P }} havoc X {{ $(fun st ⇒ ∃ (n:nat), ({{P [X |-> n] }}) st) }}.
Proof.
  intros P X. eapply hoare_consequence_pre.
  - apply hoare_havoc.
  - Admitted.

   □

End Himp.
```

4.7.2 Assert and Assume

Exercise: 4 stars, standard (assert_vs_assume) In this exercise, we will extend IMP with two commands, `assert` and *assume*. Both commands are ways to indicate that a certain assertion should hold any time this part of the program is reached. However they differ as follows:

- If an `assert` statement fails, it causes the program to go into an error state and exit.
- If an *assume* statement fails, the program fails to evaluate at all. In other words, the program gets stuck and has no final state.

The new set of commands is:

```
Module HoareAssertAssume.

Inductive com : Type :=
  | CSkip : com
  | CAsgn : string → aexp → com
  | CSeq : com → com → com
  | CIf : bexp → com → com → com
  | CWhile : bexp → com → com
  | CAssert : bexp → com
  | CAssume : bexp → com.

Notation "'assert' l" := (CAssert l)
                           (in custom com at level 8, l custom com at level 0).
Notation "'assume' l" := (CAssume l)
                          (in custom com at level 8, l custom com at level 0).
Notation "'skip'" := CSkip
  (in custom com at level 0) : com_scope.
Notation "x := y" := (CAsgn x y)
  (in custom com at level 0, x constr at level 0, y at level 85, no associativity,
    format "x := y") : com_scope.
Notation "x ; y" := (CSeq x y)
  (in custom com at level 90,
    right associativity,
    format "'[v' x ; '/' y ']'") : com_scope.
Notation "'if' x 'then' y 'else' z 'end'" := (CIf x y z)
  (in custom com at level 89, x at level 99, y at level 99, z at level 99,
    format "'[v' 'if' x 'then' '/  ' y '/' 'else' '/  ' z '/' 'end' ']'") : com_scope.
Notation "'while' x 'do' y 'end'" := (CWhile x y)
  (in custom com at level 89, x at level 99, y at level 99,
    format "'[v' 'while' x 'do' '/  ' y '/' 'end' ']'") : com_scope.
```

To define the behavior of `assert` and *assume*, we need to add notation for an error, which indicates that an assertion has failed. We modify the **ceval** relation, therefore, so that it relates a start state to either an end state or to *error*. The **result** type indicates the end value of a program, either a state or an error:

```
Inductive result : Type :=
  | RNormal : state → result
  | RError : result.
```

Now we are ready to give you the ceval relation for the new language.

```
Inductive ceval : com → state → result → Prop :=

  | E_Skip : ∀ st,
      st =[ skip ]=> RNormal st
  | E_Asgn : ∀ st a1 n x,
      aeval st a1 = n →
      st =[ x := a1 ]=> RNormal (x !-> n ; st)
  | E_SeqNormal : ∀ c1 c2 st st' r,
      st =[ c1 ]=> RNormal st' →
      st' =[ c2 ]=> r →
      st =[ c1 ; c2 ]=> r
  | E_SeqError : ∀ c1 c2 st,
      st =[ c1 ]=> RError →
      st =[ c1 ; c2 ]=> RError
  | E_IfTrue : ∀ st r b c1 c2,
      beval st b = true →
      st =[ c1 ]=> r →
      st =[ if b then c1 else c2 end ]=> r
  | E_IfFalse : ∀ st r b c1 c2,
      beval st b = false →
      st =[ c2 ]=> r →
      st =[ if b then c1 else c2 end ]=> r
  | E_WhileFalse : ∀ b st c,
      beval st b = false →
      st =[ while b do c end ]=> RNormal st
  | E_WhileTrueNormal : ∀ st st' r b c,
      beval st b = true →
      st =[ c ]=> RNormal st' →
      st' =[ while b do c end ]=> r →
      st =[ while b do c end ]=> r
  | E_WhileTrueError : ∀ st b c,
      beval st b = true →
      st =[ c ]=> RError →
      st =[ while b do c end ]=> RError

  | E_AssertTrue : ∀ st b,
      beval st b = true →
      st =[ assert b ]=> RNormal st
  | E_AssertFalse : ∀ st b,
      beval st b = false →
      st =[ assert b ]=> RError
  | E_Assume : ∀ st b,
```

```
      beval st b = true →
      st =[ assume b ]=> RNormal st

where "st '=[' c ']=>' r" := (ceval c st r).
```

We redefine hoare triples: Now, {{P}} c {{Q}} means that, whenever c is started in a state satisfying P, and terminates with result r, then r is not an error and the state of r satisfies Q.

```
Definition valid_hoare_triple
           (P : Assertion) (c : com) (Q : Assertion) : Prop :=
  ∀ st r,
     st =[ c ]=> r → P st →
     (∃ st', r = RNormal st' ∧ Q st').

Notation "{{ P }} c {{ Q }}" :=
  (valid_hoare_triple P c Q)
    (at level 2, P custom assn at level 99, c custom com at level 99, Q custom assn
at level 99)
    : hoare_spec_scope.
```

To test your understanding of this modification, give an example precondition and postcondition that are satisfied by the *assume* statement but not by the `assert` statement.

```
Theorem assert_assume_differ : ∃ (P:Assertion) b (Q:Assertion),
       ({{P}} assume b {{Q}})
  ∧ ¬ ({{P}} assert b {{Q}}).
   Admitted.
```

Then prove that any triple for an `assert` also works when `assert` is replaced by *assume*.

```
Theorem assert_implies_assume : ∀ P b Q,
     ({{P}} assert b {{Q}})
  → ({{P}} assume b {{Q}}).
Proof.
   Admitted.
```

Next, here are proofs for the old hoare rules adapted to the new semantics. You don't need to do anything with these.

```
Theorem hoare_asgn : ∀ Q X a,
  {{Q [X |-> a]}} X := a {{Q}}.
Proof.
  unfold valid_hoare_triple.
  intros Q X a st st' HE HQ.
  inversion HE. subst.
  ∃ (X !-> aeval st a ; st). split; try reflexivity.
  assumption. Qed.

Theorem hoare_consequence_pre : ∀ (P P' Q : Assertion) c,
```

```
  {{P'}} c {{Q}} →
  P -» P' →
  {{P}} c {{Q}}.
Proof.
  intros P P' Q c Hhoare Himp.
  intros st st' Hc HP. apply (Hhoare st st').
  - assumption.
  - apply Himp. assumption. Qed.

Theorem hoare_consequence_post : ∀ (P Q Q' : Assertion) c,
  {{P}} c {{Q'}} →
  Q' -» Q →
  {{P}} c {{Q}}.
Proof.
  intros P Q Q' c Hhoare Himp.
  intros st r Hc HP.
  unfold valid_hoare_triple in Hhoare.
  assert (∃ st', r = RNormal st' ∧ Q' st').
  { apply (Hhoare st); assumption. }
  destruct H as [st' [Hr HQ']].
  ∃ st'. split; try assumption.
  apply Himp. assumption.
Qed.

Theorem hoare_seq : ∀ P Q R c1 c2,
  {{Q}} c2 {{R}} →
  {{P}} c1 {{Q}} →
  {{P}} c1;c2 {{R}}.
Proof.
  intros P Q R c1 c2 H1 H2 st r H12 Pre.
  inversion H12; subst.
  - eapply H1.
    + apply H6.
    + apply H2 in H3. apply H3 in Pre.
      destruct Pre as [st'0 [Heq HQ]].
      inversion Heq; subst. assumption.
  -
    apply H2 in H5. apply H5 in Pre.
    destruct Pre as [st' [C _]].
    inversion C.
Qed.
```

Here are the other proof rules (sanity check)

```
Theorem hoare_skip : ∀ P,
  {{P}} skip {{P}}.
Proof.
```

```
  intros P st st' H HP. inversion H. subst.
  eexists. split.
  - reflexivity.
  - assumption.
Qed.

Theorem hoare_if : ∀ P Q (b:bexp) c1 c2,
  {{ P ∧ b}} c1 {{Q}} →
  {{ P ∧ ¬ b}} c2 {{Q}} →
  {{P}} if b then c1 else c2 end {{Q}}.
Proof.
  intros P Q b c1 c2 HTrue HFalse st st' HE HP.
  inversion HE; subst.
  -
    apply (HTrue st st').
    + assumption.
    + split; assumption.
  -
    apply (HFalse st st').
    + assumption.
    + split.
      × assumption.
      × apply bexp_eval_false. assumption.
Qed.

Theorem hoare_while : ∀ P (b:bexp) c,
  {{P ∧ b}} c {{P}} →
  {{P}} while b do c end {{ P ∧ ¬b}}.
Proof.
  intros P b c Hhoare st st' He HP.
  remember <{while b do c end}> as wcom eqn:Heqwcom.
  induction He;
    try (inversion Heqwcom); subst; clear Heqwcom.
  -
    eexists. split.
    + reflexivity.
    + split; try assumption.
      apply bexp_eval_false. assumption.
  -
    clear IHHe1.
    apply IHHe2.
    + reflexivity.
    + clear IHHe2 He2 r.
      unfold valid_hoare_triple in Hhoare.
```

```
      apply Hhoare in He1.
      × destruct He1 as [st1 [Heq Hst1] ].
        inversion Heq; subst.
        assumption.
      × split; assumption.
  -
    exfalso. clear IHHe.
    unfold valid_hoare_triple in Hhoare.
    apply Hhoare in He.
    + destruct He as [st' [C _] ]. inversion C.
    + split; assumption.
Qed.
```

Finally, state Hoare rules for `assert` and *assume* and use them to prove a simple program correct. Name your rules *hoare_assert* and *hoare_assume*.

Use your rules to prove the following triple.

```
Example assert_assume_example:
  {{True}}
    assume (X = 1);
    X := X + 1;
    assert (X = 2)
  {{True}}.
Proof.
  Admitted.

End HoareAssertAssume.
```

□

Chapter 5

Library SECF.Hoare2

5.1 Hoare2: Hoare Logic, Part II

```
Set Warnings "-notation-overridden".
Ltac intuition_solver ::= auto.
From Stdlib Require Import Strings.String.
From SECF Require Import Maps.
From Stdlib Require Import Bool.
From Stdlib Require Import Arith.
From Stdlib Require Import EqNat.
From Stdlib Require Import PeanoNat. Import Nat.
From Stdlib Require Import Lia.
From SECF Require Export Imp.
From SECF Require Import Hoare.

Definition FILL_IN_HERE := <{True}>.
```

5.2 Decorated Programs

The beauty of Hoare Logic is that it is *syntax directed*: the structure of proofs exactly follows the structure of programs.

We can record the essential ideas of a Hoare-logic proof – omitting low-level calculational details – by "decorating" a program with appropriate assertions on each of its commands.

Such a *decorated program* carries within itself an argument for its own correctness.

For example, consider the program:

X := m; Z := p; while X <> 0 do Z := Z - 1; X := X - 1 end Here is one possible specification for this program, in the form of a Hoare triple:

[1] X := m; Z := p; while X <> 0 do Z := Z - 1; X := X - 1 end [2] (Note the *parameters*

[1] `True`

[2] `Z=p-m`

m and p, which stand for fixed-but-arbitrary numbers. Formally, they are simply Rocq variables of type **nat**.)

Here is a decorated version of this program, embodying a proof of this specification:

[3] -» [4] X := m [5] -» [6]; Z := p; [7] -» [8] while X <> 0 do [9] -» [10] Z := Z - 1 [11]; X := X - 1 [12] end [13] -» [14]

Concretely, a decorated program consists of the program's text interleaved with assertions (sometimes multiple assertions separated by -»).

A decorated program can be viewed as a compact representation of a proof in Hoare Logic: the assertions surrounding each command specify the Hoare triple to be proved for that part of the program using one of the Hoare Logic rules, and the structure of the program itself shows how to assemble all these individual steps into a proof for the whole program.

Our goal is to verify such decorated programs "mostly automatically." But, before we can verify anything, we need to be able to *find* a proof for a given specification, and for this we need to discover the right assertions. This can be done in an almost mechanical way, with the exception of finding loop invariants. In the remainder of this section, we explain in detail how to construct decorations for several short programs, all of which are loop free or have simple loop invariants. We'll return to finding more interesting loop invariants later in the chapter.

5.2.1 Example: Swapping

Consider the following program, which swaps the values of two variables using addition and subtraction, instead of by assigning to a temporary variable.

X := X + Y; Y := X - Y; X := X - Y

We can give a proof, in the form of decorations, that this program is correct – i.e., it really swaps X and Y – as follows.

(1) [15] -» (2) [16] X := X + Y (3) [17]; Y := X - Y (4) [18]; X := X - Y (5) [19]

The decorations can be constructed as follows:

---

[3] `True`
[4] `m=m`
[5] `X=m`
[6] `X=m/\p=p`
[7] `X=m/\Z=p`
[8] `Z-X=p-m`
[9] `Z-X=p-m/\X<>0`
[10] `(Z-1)-(X-1)=p-m`
[11] `Z-(X-1)=p-m`
[12] `Z-X=p-m`
[13] `Z-X=p-m/\~(X<>0)`
[14] `Z=p-m`
[15] `X=m/\Y=n`
[16] `(X+Y)-((X+Y)-Y)=n/\(X+Y)-Y=m`
[17] `X-(X-Y)=n/\X-Y=m`
[18] `X-Y=n/\Y=m`
[19] `X=n/\Y=m`

- We begin with the undecorated program (the unnumbered lines).
- We add the specification – i.e., the outer precondition (1) and postcondition (5). In the precondition, we use parameters m and n to remember the initial values of variables X and Y so that we can refer to them in the postcondition (5).
- We work backwards, mechanically, starting from (5) and proceeding until we get to (2). At each step, we obtain the precondition of the assignment from its postcondition by substituting the assigned variable with the right-hand-side of the assignment. For instance, we obtain (4) by substituting X with X - Y in (5), and we obtain (3) by substituting Y with X - Y in (4).
- Finally, we verify that (1) logically implies (2) – i.e., that the step from (1) to (2) is a valid use of the law of consequence – by doing a bit of high-school algebra.

5.2.2 Example: Simple Conditionals

Here is a simple decorated program using conditionals:

(1) [20] if X <= Y then (2) [21] -» (3) [22] Z := Y - X (4) [23] else (5) [24] -» (6) [25] Z := X - Y (7) [26] end (8) [27]

These decorations can be constructed as follows:

- We start with the outer precondition (1) and postcondition (8).
- Following the format dictated by the hoare_if rule, we copy the postcondition (8) to (4) and (7). We conjoin the precondition (1) with the guard of the conditional to obtain (2). We conjoin (1) with the negated guard of the conditional to obtain (5).
- In order to use the assignment rule and obtain (3), we substitute Z by Y - X in (4). To obtain (6) we substitute Z by X - Y in (7).
- Finally, we verify that (2) implies (3) and (5) implies (6). Both of these implications crucially depend on the ordering of X and Y obtained from the guard. For instance, knowing that X $\leq$ Y ensures that subtracting X from Y and then adding back X produces Y, as required by the first disjunct of (3). Similarly, knowing that $\neg$ (X $\leq$ Y) ensures that subtracting Y from X and then adding back Y produces X, as needed by the second disjunct of (6). Note that n - m + m = n does *not* hold for arbitrary natural numbers n and m (for example, 3 - 5 + 5 = 5).

[20] `True`
[21] `True/\X<=Y`
[22] `(Y-X)+X=Y\/(Y-X)+Y=X`
[23] `Z+X=Y\/Z+Y=X`
[24] `True/\~(X<=Y)`
[25] `(X-Y)+X=Y\/(X-Y)+Y=X`
[26] `Z+X=Y\/Z+Y=X`
[27] `Z+X=Y\/Z+Y=X`

Exercise: 2 stars, standard, optional (if_minus_plus_reloaded) N.b.: Although this exercise is marked optional, it is an excellent warm-up for the (non-optional) if_minus_plus_correct exercise below!

Fill in valid decorations for the following program: Briefly justify each use of ->>.

□

5.2.3 Example: Reduce to Zero

Here is a *while* loop that is so simple that **True** suffices as a loop invariant.

(1) [28] while X <> 0 do (2) [29] ->> (3) [30] X := X - 1 (4) [31] end (5) [32] ->> (6) [33]

The decorations can be constructed as follows:

- Start with the outer precondition (1) and postcondition (6).
- Following the format dictated by the hoare_while rule, we copy (1) to (4). We conjoin (1) with the guard to obtain (2). We also conjoin (1) with the negation of the guard to obtain (5).
- Because the final postcondition (6) does not syntactically match (5), we add an implication between them.
- Using the assignment rule with assertion (4), we trivially substitute and obtain assertion (3).
- We add the implication between (2) and (3).

Finally we check that the implications do hold; both are trivial.

5.2.4 Example: Division

Let's do one more example of simple reasoning about a loop.

The following Imp program calculates the integer quotient and remainder of parameters m and n.

X := m; Y := 0; while n <= X do X := X - n; Y := Y + 1 end;

If we replace m and n by concrete numbers and execute the program, it will terminate with the variable X set to the remainder when m is divided by n and Y set to the quotient.

In order to give a specification to this program we need to remember that dividing m by n produces a remainder X and a quotient Y such that $n \times Y + X = m \wedge X < n$.

[28] `True`
[29] `True/\X<>0`
[30] `True`
[31] `True`
[32] `True/\~(X<>0)`
[33] `X=0`

It turns out that we get lucky with this program and don't have to think very hard about the loop invariant: the loop invariant is just the first conjunct, $n \times Y + X = m$, and we can use this to decorate the program.

(1) [34] -» (2) [35] X := m; (3) [36] Y := 0; (4) [37] while n <= X do (5) [38] -» (6) [39] X := X - n; (7) [40] Y := Y + 1 (8) [41] end (9) [42] -» (10) [43]

Assertions (4), (5), (8), and (9) are derived mechanically from the loop invariant and the loop's guard. Assertions (8), (7), and (6) are derived using the assignment rule going backwards from (8) to (6). Assertions (4), (3), and (2) are again backwards applications of the assignment rule.

Now that we've decorated the program it only remains to check that the uses of the consequence rule are correct – i.e., that (1) implies (2), that (5) implies (6), and that (9) implies (10). This is indeed the case:

- (1) -» (2): trivial, by algebra.
- (5) -» (6): because $n \leq X$, we are guaranteed that the subtraction in (6) does not get zero-truncated. We can therefore rewrite (6) as $n \times Y + n + X$ - n and cancel the ns, which results in the left conjunct of (5).
- (9) -» (10): if $\neg\ (n \leq X)$ then $X < n$. That's straightforward from high-school algebra.

So, we have a valid decorated program.

5.2.5 From Decorated Programs to Formal Proofs

From an informal proof in the form of a decorated program, it is "easy in principle" to read off a formal proof using the Rocq theorems corresponding to the Hoare Logic rules, but these proofs can be a bit long and fiddly.

Note that we do *not* unfold the definition of valid_hoare_triple anywhere in this proof: the point of the game we're playing now is to use the Hoare rules as a self-contained logic for reasoning about programs.

For example... `Definition reduce_to_zero : com :=`

```
<{ while X ≠ 0 do
     X := X - 1
   end }>.
```

[34] True
[35] n*0+m=m
[36] n*0+X=m
[37] n*Y+X=m
[38] n*Y+X=m/\n<=X
[39] n*(Y+1)+(X-n)=m
[40] n*(Y+1)+X=m
[41] n*Y+X=m
[42] n*Y+X=m/\~(n<=X)
[43] n*Y+X=m/\X<n

```
Theorem reduce_to_zero_correct' :
  {{True}}
    reduce_to_zero
  {{X = 0}}.
Proof.
  unfold reduce_to_zero.
  eapply hoare_consequence_post.
  - apply hoare_while.
    +

      eapply hoare_consequence_pre.
      × apply hoare_asgn.
      ×
        unfold assertion_sub, "-»". simpl. intros.
        exact I.
  -
    intros st [Inv GuardFalse].
    unfold bassertion in GuardFalse. simpl in GuardFalse.
    rewrite not_true_iff_false in GuardFalse.
    rewrite negb_false_iff in GuardFalse.
    apply eqb_eq in GuardFalse.
    apply GuardFalse.
Qed.
```

In Hoare we introduced a series of tactics named *assertion_auto* to automate proofs involving assertions.

The following declaration introduces a more sophisticated tactic that will help with proving assertions throughout the rest of this chapter. You don't need to understand the details, but briefly: it uses `split` repeatedly to turn all the conjunctions into separate subgoals, tries to use several theorems about booleans and (in)equalities, then uses `eauto` and *lia* to finish off as many subgoals as possible. What's left after *verify_assertion* does its thing should be just the "interesting parts" of the proof (which, if we're lucky, might be nothing at all!).

```
Ltac verify_assertion :=
  repeat split;
  simpl;
  unfold assert_implies;
  unfold bassertion in *; unfold beval in *; unfold aeval in *;
  unfold assertion_sub; intros;
  repeat (simpl in *;
          rewrite t_update_eq ||
          (try rewrite t_update_neq;
          [| (intro X; inversion X; fail)]));
  simpl in *;
```

```
  repeat match goal with [H : _ ∧ _ ⊢ _] ⇒
                          destruct H end;
  repeat rewrite not_true_iff_false in *;
  repeat rewrite not_false_iff_true in *;
  repeat rewrite negb_true_iff in *;
  repeat rewrite negb_false_iff in *;
  repeat rewrite eqb_eq in *;
  repeat rewrite eqb_neq in *;
  repeat rewrite leb_iff in *;
  repeat rewrite leb_iff_conv in *;
  try subst;
  simpl in *;
  repeat
    match goal with
      [st : state ⊢ _] ⇒
        match goal with
        | [H : st _ = _ ⊢ _] ⇒
            rewrite → H in *; clear H
        | [H : _ = st _ ⊢ _] ⇒
            rewrite ← H in *; clear H
        end
    end;
  try eauto;
  try lia.
```

This makes it pretty easy to verify reduce_to_zero:

```
Theorem reduce_to_zero_correct''' :
  {{True}}
    reduce_to_zero
  {{X = 0}}.
Proof.
  unfold reduce_to_zero.
  eapply hoare_consequence_post.
  - apply hoare_while.
    + eapply hoare_consequence_pre.
      × apply hoare_asgn.
      × verify_assertion.
  - verify_assertion.
Qed.
```

This example shows that it is conceptually straightforward to read off the main elements of a formal proof from a decorated program. Indeed, the process is so straightforward that it can be automated, as we will see next.

5.3 Formal Decorated Programs

Our informal conventions for decorated programs amount to a way of "displaying" Hoare triples, in which commands are annotated with enough embedded assertions that checking the validity of a triple is reduced to simple logical and algebraic calculations showing that some assertions imply others.

In this section, we show that this presentation style can be made completely formal – and indeed that checking the validity of decorated programs can be largely automated.

5.3.1 Syntax

The first thing we need to do is to formalize a variant of the syntax of Imp commands that includes embedded assertions, which we'll call "decorations." We call the new commands *decorated commands*, or **dcom**s.

The choice of exactly where to put assertions in the definition of **dcom** is a bit subtle. The simplest thing to do would be to annotate every **dcom** with a precondition and postcondition – something like this...

```
Module DComFirstTry.

Inductive dcom : Type :=
| DCSkip (P : Assertion)

| DCSeq (P : Assertion) (d1 : dcom) (Q : Assertion)
        (d2 : dcom) (R : Assertion)

| DCAsgn (X : string) (a : aexp) (Q : Assertion)

| DCIf (P : Assertion) (b : bexp) (P1 : Assertion) (d1 : dcom)
       (P2 : Assertion) (d2 : dcom) (Q : Assertion)
| DCWhile (P : Assertion) (b : bexp)
          (P1 : Assertion) (d : dcom) (P2 : Assertion)
          (Q : Assertion)
| DCPre (P : Assertion) (d : dcom)
| DCPost (d : dcom) (Q : Assertion).

End DComFirstTry.
```

But this would result in *very* verbose decorated programs with a lot of repeated annotations: a simple program like *skip*;*skip* would be decorated like this,

[44] ([45] skip [46]) ; ([47] skip [48]) [49]

with pre- and post-conditions around each *skip*, plus identical pre- and post-conditions on the semicolon!

In other words, we don't want both preconditions and postconditions on each command, because a sequence of two commands would contain redundant decorations–the postcondition of the first likely being the same as the precondition of the second.

Instead, our formal syntax of decorated commands will omit preconditions whenever possible and embed just postconditions.

- The *skip* command, for example, is decorated only with its postcondition

  skip [50]

  on the assumption that the precondition will be provided by somebody else.

  We carry the same assumption through the other syntactic forms: each decorated command is assumed to carry its own postcondition within itself but take its precondition from its context in which it is used.

- Sequences *d1* ; *d2* need no additional decorations.

  Why?

  Because inside *d2* there will be a postcondition, which also serves as the postcondition of *d1*;*d2*.

  Similarly, inside *d1* there will also be a postcondition, which additionally serves as the *precondition* for *d2*.

- An assignment X := *a* is decorated only with its postcondition:

  X := a [51]

- A conditional `if` *b* `then` *d1* `else` *d2* is decorated with a postcondition for the entire statement, as well as preconditions for each branch:

  if b then [52] d1 else [53] d2 end [54]

[44] P
[45] P
[46] P
[47] P
[48] P
[49] P
[50] Q
[51] Q
[52] P1
[53] P2
[54] Q

- A loop *while* *b* `do` *d* `end` is decorated with its final postcondition plus a precondition for the body:

  while b do [55] d end [56]

  The postcondition embedded in *d* serves as the loop invariant.

- Implications -» can be added as decorations either for a precondition...

  – » [57] d

  ...or for a postcondition:

  d -» [58]

  The former is waiting for another precondition to be supplied by the context; the latter relies on the postcondition already embedded in *d*.

Putting this all together gives us the formal syntax of decorated commands:

```
Inductive dcom : Type :=
| DCSkip (Q : Assertion)

| DCSeq (d1 d2 : dcom)

| DCAsgn (X : string) (a : aexp) (Q : Assertion)

| DCIf (b : bexp) (P1 : Assertion) (d1 : dcom)
       (P2 : Assertion) (d2 : dcom) (Q : Assertion)

| DCWhile (b : bexp) (P : Assertion) (d : dcom)
          (Q : Assertion)

| DCPre (P : Assertion) (d : dcom)

| DCPost (d : dcom) (Q : Assertion)
.
```

To provide the initial precondition that goes at the very top of a decorated program, we introduce a new type **decorated**:

```
Inductive decorated : Type :=
  | Decorated : Assertion → dcom → decorated.
```

[55] P
[56] Q
[57] P
[58] Q

To avoid clashing with the existing Notations for ordinary commands, we introduce these notations in a new grammar scope called **dcom**.

```
Declare Scope dcom_scope.
Notation "'skip' '{{' P '}}'" := (DCSkip P)
  (in custom com at level 0,
    P custom assn at level 99,
    format "'[v' 'skip' '/' '{{' P '}}' ']'") : dcom_scope.
Notation "l ':=' a '{{' P '}}'" := (DCAsgn l a P)
  (in custom com at level 0,
    l constr at level 0,
    a custom com at level 85,
    P custom assn at level 99,
    no associativity,
    format "'[v' l ':=' a '/' '{{' P '}}' ']'") : dcom_scope.
Notation "'while' b 'do' '{{' Pbody '}}' d 'end' '{{' Ppost '}}'" := (DCWhile
b Pbody d Ppost)
  (in custom com at level 89,
    b custom com at level 99,
    Pbody custom assn at level 99,
    Ppost custom assn at level 99,
    format "'[v' 'while' b 'do' '/ ' '{{' Pbody '}}' '/ ' d '/' 'end' '/' '{{' Ppost '}}' ']'") :
dcom_scope.
Notation "'if' b 'then' {{ P1 }} d1 'else' {{ P2 }} d2 'end' {{ Q }}" := (DCIf
b P1 d1 P2 d2 Q)
  (in custom com at level 89,
    b custom com at level 99,
    P1 custom assn at level 99,
    P2 custom assn at level 99,
    Q custom assn at level 99,
    format "'[v' 'if' b 'then' '/ ' '{{' P1 '}}' '/ ' d1 '/' 'else' '/ ' '{{' P2 '}}' '/ ' d2 '/' 'end'
'/' '{{' Q '}}' ']'"): dcom_scope.
Notation "'->>' {{ P }} d"
      := (DCPre P d)
          (in custom com at level 12, right associativity, P custom assn at level
99)
          : dcom_scope.
Notation "d '->>' {{ P }}"
      := (DCPost d P)
           (in custom com at level 10, right associativity, P custom assn at level
99)
           : dcom_scope.
Notation "x ; y" := (DCSeq x y)
```

```
  (in custom com at level 90,
    right associativity,
    format "'[v' x ; '/' y ']'") : dcom_scope.
Notation "{{ P }} d" := (Decorated P d)
  (in custom com at level 91,
    P custom assn at level 99,
    format "'[v' '{{' P '}}' '/' d ']'"): dcom_scope.

Local Open Scope dcom_scope.

Example dec0 : dcom :=
  <{ skip {{ True }} }>.
Example dec1 : dcom :=
  <{ while true do {{ True }} skip {{ True }} end {{ True }} }>.
```

Recall that you can `Set Printing All` to see how all that notation is desugared.

```
Set Printing All.
Print dec1.
Unset Printing All.
```

An example **decorated** program that decrements X to 0:

```
Example dec_while : decorated :=
  <{
  {{ True }}
    while X ≠ 0
    do
                  {{ True ∧ (X ≠ 0) }}
      X := X - 1
                  {{ True }}
    end
  {{ True ∧ X = 0}} -»
  {{ X = 0 }} }>.
```

It is easy to go from a **dcom** to a **com** by erasing all annotations.

```
Fixpoint erase (d : dcom) : com :=
  match d with
  | DCSkip _ ⇒ CSkip
  | DCSeq d1 d2 ⇒ CSeq (erase d1) (erase d2)
  | DCAsgn X a _ ⇒ CAsgn X a
  | DCIf b _ d1 _ d2 _ ⇒ CIf b (erase d1) (erase d2)
  | DCWhile b _ d _ ⇒ CWhile b (erase d)
  | DCPre _ d ⇒ erase d
  | DCPost d _ ⇒ erase d
  end.

Definition erase_d (dec : decorated) : com :=
```

```
  match dec with
  | Decorated P d ⇒ erase d
  end.

Example erase_while_ex :
    erase_d dec_while
  = <{while X ≠ 0 do X := X - 1 end}>.
Proof.
  unfold dec_while.
  reflexivity.
Qed.
```

It is also straightforward to extract the precondition and postcondition from a decorated program.

```
Definition precondition_from (dec : decorated) : Assertion :=
  match dec with
  | Decorated P d ⇒ P
  end.

Fixpoint post (d : dcom) : Assertion :=
  match d with
  | DCSkip P ⇒ P
  | DCSeq _ d2 ⇒ post d2
  | DCAsgn _ _ Q ⇒ Q
  | DCIf _ _ _ _ _ Q ⇒ Q
  | DCWhile _ _ _ Q ⇒ Q
  | DCPre _ d ⇒ post d
  | DCPost _ Q ⇒ Q
  end.

Definition postcondition_from (dec : decorated) : Assertion :=
  match dec with
  | Decorated P d ⇒ post d
  end.

Example precondition_from_while : precondition_from dec_while = True.
Proof. reflexivity. Qed.

Example postcondition_from_while : postcondition_from dec_while = {{ X = 0 }}.
Proof. reflexivity. Qed.
```

We can then express what it means for a decorated program to be correct as follows:

```
Definition outer_triple_valid (dec : decorated) :=
  {{$(precondition_from dec)}} erase_d dec {{$(postcondition_from dec)}}.
```

For example:

```
Example dec_while_triple_correct :
```

```
    outer_triple_valid dec_while
  =
    {{ True }}
      while X ≠ 0 do X := X - 1 end
    {{ X = 0 }}.
Proof. reflexivity. Qed.
```

The outer Hoare triple of a decorated program is just a `Prop`; thus, to show that it is *valid*, we need to produce a proof of this proposition.

We will do this by extracting "proof obligations" from the decorations sprinkled throughout the program.

These obligations are often called *verification conditions*, because they are the facts that must be verified to see that the decorations are locally consistent and thus constitute a proof of validity of the outer triple.

5.3.2 Extracting Verification Conditions

The function verification_conditions takes a decorated command d together with a precondition P and returns a *proposition* that, if it can be proved, implies that the triple

[59] erase d [60]

is valid.

It does this by walking over d and generating a big conjunction that includes

- local consistency checks for each form of command, plus
- uses of -» to bridge the gap between the assertions found inside a decorated command and the assertions imposed by the external precondition; these uses correspond to applications of the consequence rule.

Local consistency is defined as follows...

- The decorated command

  skip [61]

  is locally consistent with respect to a precondition P if P -» Q.

- The sequential composition of *d1* and *d2* is locally consistent with respect to P if *d1* is locally consistent with respect to P and *d2* is locally consistent with respect to the postcondition of *d1*.

[59] P
[60] postd
[61] Q

- An assignment

  X := a [62]

  is locally consistent with respect to a precondition P if:

  P -» Q X |-> a

- A conditional

  if b then [63] d1 else [64] d2 end [65]

  is locally consistent with respect to precondition P if

  (1) $P \wedge b$ -» $P1$

  (2) $P \wedge \neg b$ -» $P2$

  (3) $d1$ is locally consistent with respect to $P1$

  (4) $d2$ is locally consistent with respect to $P2$

  (5) post $d1$ -» Q

  (6) post $d2$ -» Q

- A loop

  while b do [66] d end [67]

  is locally consistent with respect to precondition P if:

  (1) P -» post d

  (2) post $d \wedge b$ -» Q

  (3) post $d \wedge \neg b$ -» R

  (4) d is locally consistent with respect to Q

- A command with an extra assertion at the beginning

  – » [68] d

  is locally consistent with respect to a precondition P if:

  (1) P -» Q

  (2) d is locally consistent with respect to Q

[62] Q
[63] P1
[64] P2
[65] Q
[66] Q
[67] R
[68] Q

- A command with an extra assertion at the end

  d -» [69]

  is locally consistent with respect to a precondition P if:

  (1) d is locally consistent with respect to P

  (2) post d -» Q

With all this in mind, we can write a *verification condition generator* that takes a decorated command and reads off a proposition saying that all its decorations are locally consistent.

Formally, since a decorated command is "waiting for its precondition" the main VC generator takes a **dcom** plus a given preondition as arguments.

```
Fixpoint verification_conditions (P : Assertion) (d : dcom) : Prop :=
  match d with
  | DCSkip Q ⇒
        (P -» Q)
  | DCSeq d1 d2 ⇒
        verification_conditions P d1
      ∧ verification_conditions (post d1) d2
  | DCAsgn X a Q ⇒
        P -» {{ Q [X |-> a] }}
  | DCIf b P1 d1 P2 d2 Q ⇒
        {{ P ∧ b }} -» P1
      ∧ {{ P ∧ ¬ b }} -» P2
      ∧ (post d1 -» Q) ∧ (post d2 -» Q)
      ∧ verification_conditions P1 d1
      ∧ verification_conditions P2 d2
  | DCWhile b Q d R ⇒

        (P -» post d)
      ∧ {{ $(post d) ∧ b }} -» Q
      ∧ {{ $(post d) ∧ ¬ b }} -» R
      ∧ verification_conditions Q d
  | DCPre P' d ⇒
        (P -» P')
      ∧ verification_conditions P' d
  | DCPost d Q ⇒
        verification_conditions P d
      ∧ (post d -» Q)
  end.
```

[69] Q

The following key theorem states that verification_conditions does its job correctly. Not surprisingly, each of the Hoare Logic rules plays a critical role at some point in the proof.

```
Theorem verification_correct : ∀ d P,
  verification_conditions P d → {{P}} erase d {{ $(post d) }}.
Proof.
  induction d; intros; simpl in *.
  -
    eapply hoare_consequence_pre.
      + apply hoare_skip.
      + assumption.
  -
    destruct H as [H1 H2].
    eapply hoare_seq.
      + apply IHd2. apply H2.
      + apply IHd1. apply H1.
  -
    eapply hoare_consequence_pre.
      + apply hoare_asgn.
      + assumption.
  -
    destruct H as [HPre1 [HPre2 [Hd1 [Hd2 [HThen HElse] ] ] ] ].
    apply IHd1 in HThen. clear IHd1.
    apply IHd2 in HElse. clear IHd2.
    apply hoare_if.
      + eapply hoare_consequence; eauto.
      + eapply hoare_consequence; eauto.
  -
    destruct H as [Hpre [Hbody1 [Hpost1 Hd] ] ].
    eapply hoare_consequence; eauto.
    apply hoare_while.
    eapply hoare_consequence_pre; eauto.
  -
    destruct H as [HP Hd].
    eapply hoare_consequence_pre; eauto.
  -
    destruct H as [Hd HQ].
    eapply hoare_consequence_post; eauto.
Qed.
```

Now that all the pieces are in place, we can define what it means to verify an entire program.

```
Definition verification_conditions_from
              (dec : decorated) : Prop :=
```

```
  match dec with
  | Decorated P d ⇒ verification_conditions P d
  end.
```

And this brings us to the main theorem of this section:

```
Corollary verification_conditions_correct : ∀ dec,
  verification_conditions_from dec →
  outer_triple_valid dec.
Proof.
  intros [P d]. apply verification_correct.
Qed.
```

5.3.3 More Automation

The propositions generated by verification_conditions are fairly big and contain many conjuncts that are essentially trivial.

```
Eval simpl in verification_conditions_from dec_while.
```

Fortunately, our *verify_assertion* tactic can generally take care of most (or sometimes all) of them. `Example vc_dec_while : verification_conditions_from dec_while.`
`Proof. verify_assertion. Qed.`

To automate the overall process of verification, we can use verification_correct to extract the verification conditions, use *verify_assertion* to verify them as much as it can, and finally tidy up any remaining bits by hand. `Ltac verify :=`

```
  intros;
  apply verification_correct;
  verify_assertion.
```

Here's the final, formal proof that dec_while is correct.

```
Theorem dec_while_correct :
  outer_triple_valid dec_while.
Proof. verify. Qed.
```

Similarly, here is the formal decorated program for the "swapping by adding and subtracting" example that we saw earlier.

```
Definition swap_dec (m n:nat) : decorated :=
  <{
    {{ X = m ∧ Y = n}} ->>
          {{ (X + Y) - ((X + Y) - Y) = n ∧ (X + Y) - Y = m }}
    X := X + Y
          {{ X - (X - Y) = n ∧ X - Y = m }};
    Y := X - Y
          {{ X - Y = n ∧ Y = m }};
    X := X - Y
```

```
    {{ X = n ∧ Y = m}}
  }>.

Theorem swap_correct : ∀ m n,
  outer_triple_valid (swap_dec m n).
Proof. verify. Qed.
```

And here is the formal decorated version of the "positive difference" program from earlier:

```
Definition positive_difference_dec :=
  <{
    {{True}}
    if X ≤ Y then
          {{True ∧ X ≤ Y}} ->»
          {{(Y - X) + X = Y ∨ (Y - X) + Y = X}}
      Z := Y - X
          {{Z + X = Y ∨ Z + Y = X}}
    else
          {{True ∧ ¬(X ≤ Y)}} ->»
          {{(X - Y) + X = Y ∨ (X - Y) + Y = X}}
      Z := X - Y
          {{Z + X = Y ∨ Z + Y = X}}
    end
    {{Z + X = Y ∨ Z + Y = X}}
  }>.

Theorem positive_difference_correct :
  outer_triple_valid positive_difference_dec.
Proof. verify. Qed.
```

Exercise: 2 stars, standard, especially useful (if_minus_plus_correct) Here is a skeleton of the formal decorated version of the if_minus_plus program that we saw earlier. Replace all occurrences of FILL_IN_HERE with appropriate assertions and fill in the proof (which should be just as straightforward as in the examples above).

```
Definition if_minus_plus_dec :=
  <{
  {{True}}
  if (X ≤ Y) then
              {{ FILL_IN_HERE }} ->»
              {{ FILL_IN_HERE }}
    Z := Y - X
              {{ FILL_IN_HERE }}
  else
              {{ FILL_IN_HERE }} ->»
              {{ FILL_IN_HERE }}
```

```
    Y := X + Z
              {{ FILL_IN_HERE }}
  end
  {{ Y = X + Z}} }>.

Theorem if_minus_plus_correct :
  outer_triple_valid if_minus_plus_dec.
Proof.
   Admitted.
   □
```

Exercise: 2 stars, standard, optional (div_mod_outer_triple_valid) Fill in appropriate assertions for the division program from above.

```
Definition div_mod_dec (a b : nat) : decorated :=
  <{
  {{ True }} ->>
  {{ FILL_IN_HERE }}
    X := a
              {{ FILL_IN_HERE }};
    Y := 0
              {{ FILL_IN_HERE }};
    while b ≤ X do
              {{ FILL_IN_HERE }} ->>
              {{ FILL_IN_HERE }}
      X := X - b
              {{ FILL_IN_HERE }};
      Y := Y + 1
              {{ FILL_IN_HERE }}
    end
  {{ FILL_IN_HERE }} ->>
  {{ FILL_IN_HERE }} }>.

Theorem div_mod_outer_triple_valid : ∀ a b,
  outer_triple_valid (div_mod_dec a b).
Proof.
   Admitted.
   □
```

5.4 Finding Loop Invariants

Once the outermost precondition and postcondition are chosen, the only creative part of a verifying program using Hoare Logic is finding the right loop invariants. The reason this is difficult is the same as the reason that inductive mathematical proofs are:

- Strengthening a *loop invariant* means that you have a stronger assumption to work with when trying to establish the postcondition of the loop body, but it also means that the loop body's postcondition is harder to prove.

- Similarly, strengthening an *induction hypothesis* means that you have a stronger assumption to work with when trying to complete the induction step of the proof, but it also means that the statement being proved inductively is harder to prove.

This section explains how to approach the challenge of finding loop invariants through a series of examples and exercises.

5.4.1 Example: Slow Subtraction

The following program subtracts the value of X from the value of Y by repeatedly decrementing both X and Y. We want to verify its correctness with respect to the pre- and postconditions shown:

[70] while X <> 0 do Y := Y - 1; X := X - 1 end [71]

To verify this program, we need to find an invariant *Inv* for the loop. As a first step we can leave *Inv* as an unknown and build a *skeleton* for the proof by applying the rules for local consistency, working from the end of the program to the beginning, as usual, and without doing any thinking at all yet.

This leads to the following skeleton:

(1) [72] -» (a) (2) [73] while X <> 0 do (3) [74] -» (c) (4) [75] Y := Y - 1; (5) [76] X := X - 1 (6) [77] end (7) [78] -» (b) (8) [79] Examining this skeleton, we can see that any valid *Inv* will have to respect three conditions:

- (a) it must be *weak* enough to be implied by the loop's precondition, i.e., (1) must imply (2);

- (b) it must be *strong* enough to imply the program's postcondition, i.e., (7) must imply (8);

- (c) it must be *preserved* by a single iteration of the loop, assuming that the loop guard also evaluates to true, i.e., (3) must imply (4).

[70] `X=m/\Y=n`
[71] `Y=n-m`
[72] `X=m/\Y=n`
[73] `Inv`
[74] `Inv/\X<>0`
[75] `Inv[X|->X-1][Y|->Y-1]`
[76] `Inv[X|->X-1]`
[77] `Inv`
[78] `Inv/\~(X<>0)`
[79] `Y=n-m`

These conditions are actually independent of the particular program and specification we are considering: every loop invariant has to satisfy them.

One way to find a loop invariant that simultaneously satisfies these three conditions is by using an iterative process: start with a "candidate" invariant (e.g., a guess or a heuristic choice) and check the three conditions above; if any of the checks fails, try to use the information that we get from the failure to produce another – hopefully better – candidate invariant, and repeat.

For instance, in the reduce-to-zero example above, we saw that, for a very simple loop, choosing **True** as a loop invariant did the job. Maybe it will work here too. To find out, let's try instantiating *Inv* with **True** in the skeleton above and see what we get...

(1) [80] -» (a - OK) (2) [81] while X <> 0 do (3) [82] -» (c - OK) (4) [83] Y := Y - 1; (5) [84] X := X - 1 (6) [85] end (7) [86] -» (b - WRONG!) (8) [87]

While conditions (a) and (c) are trivially satisfied, (b) is wrong: it is not the case that **True** ∧ X = 0 (7) implies Y = *n* - *m* (8). In fact, the two assertions are completely unrelated, so it is very easy to find a counterexample to the implication (say, Y = X = *m* = 0 and *n* = 1).

If we want (b) to hold, we need to strengthen the loop invariant so that it implies the postcondition (8). One simple way to do this is to let the loop invariant *be* the postcondition. So let's return to our skeleton, instantiate *Inv* with Y = *n* - *m*, and try checking conditions (a) to (c) again.

(1) [88] -» (a - WRONG!) (2) [89] while X <> 0 do (3) [90] -» (c - WRONG!) (4) [91] Y := Y - 1; (5) [92] X := X - 1 (6) [93] end (7) [94] -» (b - OK) (8) [95]

This time, condition (b) holds trivially, but (a) and (c) are broken. Condition (a) requires that (1) X = *m* ∧ Y = *n* implies (2) Y = *n* - *m*. If we substitute Y by *n* we have to show that *n* = *n* - *m* for arbitrary *m* and *n*, which is not the case (for instance, when *m* = *n* = 1). Condition (c) requires that *n* - *m* - 1 = *n* - *m*, which fails, for instance, for *n* = 1 and *m* = 0. So, although Y = *n* - *m* holds at the end of the loop, it does not hold from the start, and it doesn't hold on each iteration; it is not a correct loop invariant.

---

[80] `X=m/\Y=n`
[81] `True`
[82] `True/\X<>0`
[83] `True`
[84] `True`
[85] `True`
[86] `True/\~(X<>0)`
[87] `Y=n-m`
[88] `X=m/\Y=n`
[89] `Y=n-m`
[90] `Y=n-m/\X<>0`
[91] `Y-1=n-m`
[92] `Y=n-m`
[93] `Y=n-m`
[94] `Y=n-m/\~(X<>0)`
[95] `Y=n-m`

This failure is not very surprising: the variable Y changes during the loop, while m and n are constant, so the assertion we chose didn't have much chance of being a loop invariant!

To do better, we need to generalize (7) to some statement that is equivalent to (8) when X is 0, since this will be the case when the loop terminates, and that "fills the gap" in some appropriate way when X is nonzero. Looking at how the loop works, we can observe that X and Y are decremented together until X reaches 0. So, if X = 2 and Y = 5 initially, after one iteration of the loop we obtain X = 1 and Y = 4; after two iterations X = 0 and Y = 3; and then the loop stops. Notice that the difference between Y and X stays constant between iterations: initially, Y = n and X = m, and the difference is always n - m. So let's try instantiating *Inv* in the skeleton above with Y - X = n - m.

(1) [96] -» (a - OK) (2) [97] while X <> 0 do (3) [98] -» (c - OK) (4) [99] Y := Y - 1; (5) [100] X := X - 1 (6) [101] end (7) [102] -» (b - OK) (8) [103]

Success! Conditions (a), (b) and (c) all hold now. (To verify (c), we need to check that, under the assumption that X ≠ 0, we have Y - X = (Y - 1) - (X - 1); this holds for all natural numbers X and Y.)

Here is the final version of the decorated program:

```
Example subtract_slowly_dec (m : nat) (n : nat) : decorated :=
  <{
  {{ X = m ∧ Y = n }} ->>
  {{ Y - X = n - m }}
    while X ≠ 0 do
                  {{ Y - X = n - m ∧ X ≠ 0 }} ->>
                  {{ (Y - 1) - (X - 1) = n - m }}
       Y := Y - 1
                  {{ Y - (X - 1) = n - m }} ;
       X := X - 1
                  {{ Y - X = n - m }}
    end
  {{ Y - X = n - m ∧ X = 0 }} ->>
  {{ Y = n - m }} }>.

Theorem subtract_slowly_outer_triple_valid : ∀ m n,
  outer_triple_valid (subtract_slowly_dec m n).
Proof.
  verify. Qed.
```

[96] X=m/\Y=n
[97] Y-X=n-m
[98] Y-X=n-m/\X<>0
[99] (Y-1)-(X-1)=n-m
[100] Y-(X-1)=n-m
[101] Y-X=n-m
[102] Y-X=n-m/\~(X<>0)
[103] Y=n-m

5.4.2 Exercise: Slow Assignment

Exercise: 2 stars, standard (slow_assignment) A roundabout way of assigning a number currently stored in X to the variable Y is to start Y at 0, then decrement X until it hits 0, incrementing Y at each step. Here is a program that implements this idea. Fill in decorations and prove the decorated program correct. (The proof should be very simple.)

```
Example slow_assignment_dec (m : nat) : decorated :=
  <{
    {{ X = m }}
      Y := 0
                    {{ FILL_IN_HERE }} ->>
                    {{ FILL_IN_HERE }} ;
      while X ≠ 0 do
                    {{ FILL_IN_HERE }} ->>
                    {{ FILL_IN_HERE }}
         X := X - 1
                    {{ FILL_IN_HERE }} ;
         Y := Y + 1
                    {{ FILL_IN_HERE }}
      end
    {{ FILL_IN_HERE }} ->>
    {{ Y = m }}
  }>.

Theorem slow_assignment : ∀ m,
  outer_triple_valid (slow_assignment_dec m).
Proof. Admitted.
```

□

5.4.3 Example: Parity

Here is a cute way of computing the parity of a value initially stored in X, due to Daniel Cristofani.

[104] while 2 <= X do X := X - 2 end [105]

The parity function used in the specification is defined in Rocq as follows:

```
Fixpoint parity x :=
  match x with
  | 0 ⇒ 0
  | 1 ⇒ 1
  | S (S x') ⇒ parity x'
  end.
```

[104] X=m

[105] X=paritym

The postcondition does not hold at the beginning of the loop, since m = parity m does not hold for an arbitrary m, so we cannot hope to use that as a loop invariant. To find a loop invariant that works, let's think a bit about what this loop does. On each iteration it decrements X by 2, which preserves the parity of X. So the parity of X does not change, i.e., it is invariant. The initial value of X is m, so the parity of X is always equal to the parity of m. Using parity X = parity m as an invariant we obtain the following decorated program:

[106] -» (a - OK) [107] while 2 <= X do [108] -» (c - OK) [109] X := X - 2 [110] end [111] -» (b - OK) [112]

With this loop invariant, conditions (a), (b), and (c) are all satisfied. For verifying (b), we observe that, when X < 2, we have parity X = X (we can easily see this in the definition of parity). For verifying (c), we observe that, when 2 ≤ X, we have parity X = parity (X-2).

Exercise: 3 stars, standard, optional (parity) Translate the above informal decorated program into a formal one and prove it correct.

Hint: There are actually several possible loop invariants that all lead to good proofs; one that leads to a particularly simple proof is parity X = parity m – or more formally, using the # syntax to lift the application of the parity function into the syntax of assertions, {{ #parity X = #parity m }}.

```
Definition parity_dec (m:nat) : decorated :=
  <{
  {{ X = m }} ->»
  {{ FILL_IN_HERE }}
    while 2 ≤ X do
                  {{ FILL_IN_HERE }} ->»
                  {{ FILL_IN_HERE }}
      X := X - 2
                  {{ FILL_IN_HERE }}
    end
  {{ FILL_IN_HERE }} ->»
  {{ X = #parity m }} }>.
```

If you use the suggested loop invariant, you may find the following lemmas helpful (as well as *leb_complete* and *leb_correct*).

```
Lemma parity_ge_2 : ∀ x,
  2 ≤ x →
  parity (x - 2) = parity x.
```

---

[106] X=m
[107] parityX=paritym
[108] parityX=paritym/\2<=X
[109] parity(X-2)=paritym
[110] parityX=paritym
[111] parityX=paritym/\~(2<=X)
[112] X=paritym

```
Proof.
  destruct x; intros; simpl.
  - reflexivity.
  - destruct x; simpl.
    + lia.
    + rewrite sub_0_r. reflexivity.
Qed.

Lemma parity_lt_2 : ∀ x,
  ¬ 2 ≤ x →
  parity x = x.
Proof.
  induction x; intros; simpl.
  - reflexivity.
  - destruct x.
    + reflexivity.
    + lia.
Qed.

Theorem parity_outer_triple_valid : ∀ m,
  outer_triple_valid (parity_dec m).
Proof.
   Admitted.
```

□

5.4.4 Example: Finding Square Roots

The following program computes the integer square root of X by naive iteration:

[113] Z := 0; while (Z+1)*(Z+1) <= X do Z := Z+1 end [114]

As we did before, we can try to use the postcondition as a candidate loop invariant, obtaining the following decorated program:

(1) [115] -» (a - second conjunct of (2) WRONG!) (2) [116] Z := 0 (3) [117]; while (Z+1)*(Z+1) <= X do (4) [118] -» (c - WRONG!) (5) [119] Z := Z+1 (6) [120] end (7) [121] -» (b - OK) (8) [122]

This didn't work very well: conditions (a) and (c) both failed. Looking at condition (c), we see that the second conjunct of (4) is almost the same as the first conjunct of (5),

[113] X=m
[114] Z*Z<=m/\m<(Z+1)*(Z+1)
[115] X=m
[116] 0*0<=m/\m<(0+1)*(0+1)
[117] Z*Z<=m/\m<(Z+1)*(Z+1)
[118] Z*Z<=m/\m<(Z+1)*(Z+1)/\(Z+1)*(Z+1)<=X
[119] (Z+1)*(Z+1)<=m/\m<((Z+1)+1)*((Z+1)+1)
[120] Z*Z<=m/\m<(Z+1)*(Z+1)
[121] Z*Z<=m/\m<(Z+1)*(Z+1)/\~((Z+1)*(Z+1)<=X)
[122] Z*Z<=m/\m<(Z+1)*(Z+1)

except that (4) mentions X while (5) mentions m. But note that X is never assigned in this program, so we should always have X=m. We didn't propagate this information from (1) into the loop invariant, but we could!

Also, we don't need the second conjunct of (8), since we can obtain it from the negation of the guard – the third conjunct in (7) – again under the assumption that X=m. This allows us to simplify a bit.

So we now try X=m ∧ Z×Z ≤ m as the loop invariant:

[123] -» (a - OK) [124] Z := 0 [125]; while (Z+1)*(Z+1) <= X do [126] -» (c - OK) [127] Z := Z + 1 [128] end [129] -» (b - OK) [130]

This works, since conditions (a), (b), and (c) are now all rather trivially satisfied.

Very often, when a variable is used in a loop in a read-only fashion (i.e., it is referred to by the program or by the specification, and it is not changed by the loop), it is necessary to record the *fact* that it doesn't change in the loop invariant.

Exercise: 3 stars, standard, optional (sqrt) Translate the above informal decorated program into a formal one and prove it correct.

Hint: The loop invariant here must ensure that Z*Z is consistently less than or equal to X.

```
Definition sqrt_dec (m:nat) : decorated :=
  <{
    {{ X = m }} -»
    {{ FILL_IN_HERE }}
      Z := 0
                    {{ FILL_IN_HERE }};
      while ((Z+1)×(Z+1) ≤ X) do
                    {{ FILL_IN_HERE }} -»
                    {{ FILL_IN_HERE }}
        Z := Z + 1
                    {{ FILL_IN_HERE }}
      end
    {{ FILL_IN_HERE }} -»
    {{ Z×Z≤m ∧ m<(Z+1)×(Z+1) }}
  }>.

Theorem sqrt_correct : ∀ m,
  outer_triple_valid (sqrt_dec m).
```

[123] X=m
[124] X=m/\0*0<=m
[125] X=m/\Z*Z<=m
[126] X=m/\Z*Z<=m/\(Z+1)*(Z+1)<=X
[127] X=m/\(Z+1)*(Z+1)<=m
[128] X=m/\Z*Z<=m
[129] X=m/\Z*Z<=m/\~((Z+1)*(Z+1)<=X)
[130] Z*Z<=m/\m<(Z+1)*(Z+1)

```
Proof. Admitted.
```

5.4.5 Example: Squaring

Here is a program that squares X by repeated addition:

[131] Y := 0; Z := 0; while Y <> X do Z := Z + X; Y := Y + 1 end [132]

The first thing to note is that the loop reads X but doesn't change its value. As we saw in the previous example, it can be a good idea in such cases to add X = m to the loop invariant. The other thing that we know is often useful in the loop invariant is the postcondition, so let's add that too, leading to the candidate loop invariant Z = m × m ∧ X = m.

[133] -» (a - WRONG) [134] Y := 0 [135]; Z := 0 [136]; while Y <> X do [137] -» (c - WRONG) [138] Z := Z + X [139]; Y := Y + 1 [140] end [141] -» (b - OK) [142]

Conditions (a) and (c) fail because of the Z = m×m part. While Z starts at 0 and works itself up to m×m, we can't expect Z to be m×m from the start. If we look at how Z progresses in the loop, after the 1st iteration Z = m, after the 2nd iteration Z = 2*m, and at the end Z = m×m. Since the variable Y tracks how many times we go through the loop, this leads us to derive a new loop invariant candidate: Z = Y×m ∧ X = m.

[143] -» (a - OK) [144] Y := 0 [145]; Z := 0 [146]; while Y <> X do [147] -» (c - OK) [148] Z := Z + X [149]; Y := Y + 1 [150] end [151] -» (b - OK) [152]

This new loop invariant makes the proof go through: all three conditions are easy to check.

It is worth comparing the postcondition Z = m×m and the Z = Y×m conjunct of the loop invariant. It is often the case that one has to replace parameters with variables – or

---

[131] `X=m`
[132] `Z=m*m`
[133] `X=m`
[134] `0=m*m/\X=m`
[135] `0=m*m/\X=m`
[136] `Z=m*m/\X=m`
[137] `Z=m*m/\X=m/\Y<>X`
[138] `Z+X=m*m/\X=m`
[139] `Z=m*m/\X=m`
[140] `Z=m*m/\X=m`
[141] `Z=m*m/\X=m/\~(Y<>X)`
[142] `Z=m*m`
[143] `X=m`
[144] `0=0*m/\X=m`
[145] `0=Y*m/\X=m`
[146] `Z=Y*m/\X=m`
[147] `Z=Y*m/\X=m/\Y<>X`
[148] `Z+X=(Y+1)*m/\X=m`
[149] `Z=(Y+1)*m/\X=m`
[150] `Z=Y*m/\X=m`
[151] `Z=Y*m/\X=m/\~(Y<>X)`
[152] `Z=m*m`

with expressions involving both variables and parameters, like m - Y – when going from postconditions to loop invariants.

□

5.4.6 Exercise: Factorial

Exercise: 4 stars, advanced (factorial_correct) Recall that $n!$ denotes the factorial of n (i.e., $n!$ = 1*2*...*n). Formally, the factorial function is defined recursively in the Rocq standard library in a way that is equivalent to the following:

Fixpoint fact (n : nat) : nat := match n with | O => 1 | S n' => n * (fact n') end.

```
Compute fact 5.
```

First, write the Imp program *factorial* that calculates the factorial of the number initially stored in the variable X and puts it in the variable Y.

Using your definition *factorial* and slow_assignment_dec as a guide, write a formal decorated program factorial_dec that implements the factorial function. Hint: recall the use of # in assertions to apply a function to an Imp variable.

Fill in the blanks and finish the proof of correctness. Bear in mind that we are working with natural numbers, for which both division and subtraction can behave differently than with real numbers. Excluding both operations from your loop invariant is advisable!

Then state a theorem named factorial_correct that says factorial_dec is correct, and prove the theorem. If all goes well, *verify* will leave you with just two subgoals, each of which requires establishing some mathematical property of fact, rather than proving anything about your program.

Hint: if those two subgoals become tedious to prove, give some thought to how you could restate your assertions such that the mathematical operations are more amenable to manipulation in Rocq. For example, recall that 1 + ... is easier to work with than ... + 1.

```
Example factorial_dec (m:nat) : decorated
. Admitted.

Theorem factorial_correct: ∀ m,
  outer_triple_valid (factorial_dec m).
Proof. Admitted.
```

□

5.4.7 Exercise: Minimum

Exercise: 3 stars, advanced (minimum_correct) Fill in decorations for the following program and prove them correct. As with *factorial*, be careful about mathematical reasoning involving natural numbers, especially subtraction.

Also, remember that applications of Rocq functions in assertions need an *ap* or *ap2* to be parsed correctly. E.g., min *a b* needs to be written *ap2* min *a b* in an assertion.

You may find *andb_true_eq* useful (perhaps after using symmetry to get an equality the right way around).

```
Definition minimum_dec (a b : nat) : decorated :=
  <{
    {{ True }} ->>
    {{ FILL_IN_HERE }}
      X := a
            {{ FILL_IN_HERE }};
      Y := b
            {{ FILL_IN_HERE }};
      Z := 0
            {{ FILL_IN_HERE }};
      while X ≠ 0 && Y ≠ 0 do
            {{ FILL_IN_HERE }} ->>
            {{ FILL_IN_HERE }}
        X := X - 1
            {{ FILL_IN_HERE }};
        Y := Y - 1
            {{ FILL_IN_HERE }};
        Z := Z + 1
            {{ FILL_IN_HERE }}
      end
    {{ FILL_IN_HERE }} ->>
    {{ Z = #min a b }}
  }>.

Theorem minimum_correct : ∀ a b,
  outer_triple_valid (minimum_dec a b).
Proof. Admitted.
```

□

5.4.8 Exercise: Two Loops

Exercise: 3 stars, standard (two_loops) Here is a pretty inefficient way of adding 3 numbers:

X := 0; Y := 0; Z := c; while X <> a do X := X + 1; Z := Z + 1 end; while Y <> b do Y := Y + 1; Z := Z + 1 end

Show that it does what it should by completing the following decorated program.

```
Definition two_loops_dec (a b c : nat) : decorated :=
  <{
    {{ True }} ->>
    {{ FILL_IN_HERE }}
      X := 0
```

```
                {{ FILL_IN_HERE }};
    Y := 0
                {{ FILL_IN_HERE }};
    Z := c
                {{ FILL_IN_HERE }};
    while X ≠ a do
                {{ FILL_IN_HERE }} ->»
                {{ FILL_IN_HERE }}
      X := X + 1
                {{ FILL_IN_HERE }};
      Z := Z + 1
                {{ FILL_IN_HERE }}
    end
                {{ FILL_IN_HERE }} ->»
                {{ FILL_IN_HERE }};
    while Y ≠ b do
                {{ FILL_IN_HERE }} ->»
                {{ FILL_IN_HERE }}
      Y := Y + 1
                {{ FILL_IN_HERE }};
      Z := Z + 1
                {{ FILL_IN_HERE }}
    end
  {{ FILL_IN_HERE }} ->»
  {{ Z = a + b + c }}
}>.

Theorem two_loops : ∀ a b c,
  outer_triple_valid (two_loops_dec a b c).
Proof.
  Admitted.
```

□

5.4.9 Exercise: Power Series

Exercise: 4 stars, standard, optional (dpow2) Here is a program that computes the series: 1 + 2 + 2^2 + ... + 2^m = 2^(m+1) - 1

X := 0; Y := 1; Z := 1; while X <> m do Z := 2 * Z; Y := Y + Z; X := X + 1 end

Turn this into a decorated program and prove it correct.

```
Fixpoint pow2 n :=
  match n with
  | 0 ⇒ 1
  | S n' ⇒ 2 × (pow2 n')
```

```
  end.

Definition dpow2_dec (n : nat) :=
  <{
    {{ True }} ->>
    {{ FILL_IN_HERE }}
      X := 0
                {{ FILL_IN_HERE }};
      Y := 1
                {{ FILL_IN_HERE }};
      Z := 1
                {{ FILL_IN_HERE }};
      while X ≠ n do
                {{ FILL_IN_HERE }} ->>
                {{ FILL_IN_HERE }}
        Z := 2 × Z
                {{ FILL_IN_HERE }};
        Y := Y + Z
                {{ FILL_IN_HERE }};
        X := X + 1
                {{ FILL_IN_HERE }}
      end
    {{ FILL_IN_HERE }} ->>
    {{ Y = #pow2 (n+1) - 1 }}
  }>.
```

Some lemmas that you may find useful...

```
Lemma pow2_plus_1 : ∀ n,
  pow2 (n+1) = pow2 n + pow2 n.
Proof.
  induction n; simpl.
  - reflexivity.
  - lia.
Qed.

Lemma pow2_le_1 : ∀ n, pow2 n ≥ 1.
Proof.
  induction n; simpl; [constructor | lia].
Qed.
```

The main correctness theorem:

```
Theorem dpow2_down_correct : ∀ n,
  outer_triple_valid (dpow2_dec n).
Proof.
   Admitted.
```

□

Exercise: 2 stars, advanced, optional (fib_eqn) The Fibonacci function is usually written like this:

Fixpoint fib n := match n with | 0 => 1 | 1 => 1 | _=> fib (pred n) + fib (pred (pred n)) end.

This doesn't pass Rocq's termination checker, but here is a slightly clunkier definition that does:

```
Fixpoint fib n :=
  match n with
  | 0 ⇒ 1
  | S n' ⇒ match n' with
            | 0 ⇒ 1
            | S n'' ⇒ fib n' + fib n''
            end
  end.
```

Prove that fib satisfies the following equation. You will need this as a lemma in the next exercise.

```
Lemma fib_eqn : ∀ n,
  n > 0 →
  fib n + fib (pred n) = fib (1 + n).
Proof.
   Admitted.
```

□

Exercise: 4 stars, advanced, optional (fib) The following Imp program leaves the value of fib n in the variable Y when it terminates:

X := 1; Y := 1; Z := 1; while X <> 1 + n do T := Z; Z := Z + Y; Y := T; X := 1 + X end

Fill in the following definition of dfib and prove that it satisfies this specification: [153] dfib [154]

You will need many uses of *ap* in your assertions. If all goes well, your proof will be very brief.

```
Definition T : string := "T".

Definition dfib (n : nat) : decorated :=
  <{
    {{ True }} ->»
    {{ FILL_IN_HERE }}
```

[153] True

[154] Y=fibn

```
    X := 1
                {{ FILL_IN_HERE }} ;
    Y := 1
                {{ FILL_IN_HERE }} ;
    Z := 1
                {{ FILL_IN_HERE }} ;
    while X ≠ 1 + n do
                  {{ FILL_IN_HERE }} ->
                  {{ FILL_IN_HERE }}
      T := Z
                  {{ FILL_IN_HERE }};
      Z := Z + Y
                  {{ FILL_IN_HERE }};
      Y := T
                  {{ FILL_IN_HERE }};
      X := 1 + X
                  {{ FILL_IN_HERE }}
    end
    {{ FILL_IN_HERE }} ->
    {{ Y = #fib n }}
  }>.

Theorem dfib_correct : ∀ n,
  outer_triple_valid (dfib n).
Proof.
   Admitted.
   □
```

Exercise: 5 stars, advanced, optional (improve_dcom) The formal decorated programs defined above are intended to look as similar as possible to the informal ones defined earlier. If we drop this requirement, we can eliminate almost all annotations, just requiring final postconditions and loop invariants to be provided explicitly. Do this – i.e., define a new version of dcom with as few annotations as possible and adapt the rest of the formal development leading up to the verification_correct theorem.

5.5 Weakest Preconditions (Optional)

Some preconditions are more interesting than others. For example, the Hoare triple

[155] X := Y + 1 [156]

[155] False

[156] X<=5

is *not* very interesting: although it is perfectly valid , it tells us nothing useful. Since the precondition isn't satisfied by any state, it doesn't describe any situations where we can use the command X := Y + 1 to achieve the postcondition X ≤ 5.

By contrast,

[157] X := Y + 1 [158]

has a useful precondition: it tells us that, if we can somehow create a situation in which we know that Y ≤ 4 ∧ Z = 0, then running this command will produce a state satisfying the postcondition. However, this precondition is not as useful as it could be, because the Z = 0 clause in the precondition actually has nothing to do with the postcondition X ≤ 5.

The *most* useful precondition for this command is this one:

[159] X := Y + 1 [160]

The assertion Y ≤ 4 is called the *weakest precondition* of X := Y + 1 with respect to the postcondition X ≤ 5.

Assertion Y ≤ 4 is a *weakest precondition* of command X := Y + 1 with respect to postcondition X ≤ 5. Think of *weakest* here as meaning "easiest to satisfy": a weakest precondition is one that as many states as possible can satisfy.

P is a weakest precondition of command c for postcondition Q if

- P is a precondition, that is, $\{\{P\}\}\ c\ \{\{Q\}\}$; and
- P is at least as weak as all other preconditions, that is, if $\{\{P'\}\}\ c\ \{\{Q\}\}$ then P' -» P.

Note that weakest preconditions need not be unique. For example, Y ≤ 4 was a weakest precondition above, but so are the logically equivalent assertions Y < 5, Y ≤ 2 × 2, etc. It is easy to show that any two weakest preconditions P and P' of a command c with respect to postcondition Q are logically equivalent; that is, P «-» P'.

```
Definition is_wp P c Q :=
  {{P}} c {{Q}} ∧
  ∀ P', {{P'}} c {{Q}} → (P' -» P).
```

Exercise: 1 star, standard, optional (wp) What are weakest preconditions of the following commands for the following postconditions?

1) [161] skip [162]

2) [163] X := Y + Z [164]

---

[157] Y<=4/\Z=0
[158] X<=5
[159] Y<=4
[160] X<=5
[161] ?
[162] X=5
[163] ?
[164] X=5

3) [165] X := Y [166]
4) [167] if X = 0 then Y := Z + 1 else Y := W + 2 end [168]
5) [169] X := 5 [170]
6) [171] while true do X := 0 end [172]

Exercise: 3 stars, advanced, optional (is_wp) Prove formally, using the definition of valid_hoare_triple, that Y ≤ 4 is indeed a weakest precondition of X := Y + 1 with respect to postcondition X ≤ 5.

Note: we have to put parentheses around the inputs to is_wp to prevent Rocq from parsing those three things as a Hoare triple.

```
Theorem is_wp_example :
  is_wp ({{ Y ≤ 4 }}) (<{X := Y + 1}>) ({{ X ≤ 5 }}).
Proof.
  Admitted.
```
□

Exercise: 2 stars, advanced, optional (hoare_asgn_weakest) Show that the precondition in the rule hoare_asgn is in fact the weakest precondition.

```
Theorem hoare_asgn_weakest : ∀ Q X a,
  is_wp ({{ Q [X |-> a] }}) <{ X := a }> Q.
Proof.
  Admitted.
```
□

Exercise: 2 stars, advanced, optional (hoare_havoc_weakest) Show that your havoc_pre function from the *himp_hoare* exercise in the Hoare chapter returns a weakest precondition. `Module` HIMP2.
`Import` HIMP.

```
Lemma hoare_havoc_weakest : ∀ (P Q : Assertion) (X : string),
  {{ P }} havoc X {{ Q }} →
  P ->> havoc_pre X Q.
Proof.
  Admitted.
End HIMP2.
```

[165] ?
[166] X=Y
[167] ?
[168] Y=5
[169] ?
[170] X=0
[171] ?
[172] X=0

□

Chapter 6

Library SECF.Preface

6.1 Preface

6.2 Formal foundations for program security

This volume uses Rocq to lay down formal foundations for the security of programs, by (1) setting clear *security goals*, (2) investigating *enforcement mechanisms*, and (3) *proving* that the mechanisms achieve their goals. In a bit more detail:

(1) We set clear *security goals* by mathematically defining what it means for a program to be secure with respect to a precise attacker model. The security goals we investigate are various information-flow security properties, formalized as variants of noninterference. Noninterference precisely expresses what it means for a program to not leak secrets to attackers with various capabilities. For instance, we investigate attackers that can observe (part of) the final result or state of the computation, or explicit program outputs happening during execution, or side-channel observations (e.g., the branches and memory addresses accessed by the program, as assumed by cryptographic constant-time programming discipline). We also investigate attackers that can influence the program by causing speculative execution to take the wrong branch in a conditional.

(2) We investigate various static and dynamic information-flow control *enforcement mechanisms* aimed at achieving specific noninterference properties. In particular we investigate enforcement via various security type systems, secure multi-execution, and a program transformation called Speculative Load Hardening (SLH).

(3) Finally, we *prove* in Rocq that these enforcement mechanisms do indeed achieve their desired security goal. For instance, we prove that a standard type system that prevents branch conditions and memory accesses that may depend on secrets indeed achieves cryptographic constant-time security, and also that together with the SLH transformation it achieves speculative constant-time security.

6.3 Expected audience

This volume can be of interest to anyone curious about the security applications of the concepts from the *Logical Foundations* volume and, more generally, those interested in a formal approach to security that is solidly grounded in fully mechanized Rocq proofs. This volume can also serve as a start for research in this area, and the Postscript presents a few illustrative security research projects involving machine-checked proofs.

This volume directly builds on the material in the *Logical Foundations* volume and the two can be used together in a one-semester course. We try not to assume prior knowledge in security and would appreciate your feedback if you find places in the volume where this can be improved.

6.4 Further reading

This volume is intended to be self-contained, but readers looking for a deeper treatment of particular topics will find some suggestions for further reading as citations in the technical chapters and in the Postscript chapter. Bibliographic information for all cited works can be found in the Bib file.

6.5 Recommended citation format

If you want to refer to this volume in your own writing, please do so as follows:

@book {Hritcu:SF7, author = { Cătălin Hriţcu and Yonghyun Kim}, editor = {Benjamin C. Pierce}, title = "Security Foundations", series = "Software Foundations", volume = "7", year = "2026", publisher = "Electronic textbook", note = {Version 1.0, \URL{http://softwarefoundations.cis }, }

6.6 Feedback or any other contribution welcome

We plan to continue improving and expanding this volume, so any feedback on it or any other contribution would be much appreciated.

6.7 Thanks

This volume originated from teaching materials for courses we taught at Ruhr Uni Bochum and we would like to thank the students taking these courses for putting up with very rough early drafts of these materials. We also thank all people who have contributed to this volume (the people we remembered are listed on the cover page of this volume).

Chapter 7

Library SECF.Noninterference

7.1 Noninterference: Defining Secrecy and Secure Multi-Execution

```
Set Warnings "-notation-overridden,-parsing,-deprecated-hint-without-locality".
From Stdlib Require Import Bool.Bool.
From Stdlib Require Import Init.Nat.
From Stdlib Require Import Arith.Arith.
From Stdlib Require Import Arith.EqNat. Import Nat.
From SECF Require Import Maps.
From SECF Require Import Imp.
Set Default Goal Selector "!".

From Stdlib Require Import Lia.
```

Programmers have to be very careful about how information flows in the software they develop to prevent leaking secret data. For instance, in course management systems students shouldn't be able to obtain information about other student's grades. In crypto protocols the keys should be kept secret and not sent over the network in the clear.

Information-flow control tries to prevent leaking secret information. But how does one formalize that a program doesn't leak any information about the secret inputs to public outputs?

We first investigate this question in the very simple setting of Rocq functions taking two arguments, one we call the public input and the other one we call the secret input. Our functions return a pair where the first element is the public output and the second one the secret output.

Say we have the following function working on natural numbers:

```
Definition secure_f (pi si : nat) : nat×nat := (pi+1, pi+si×2).
```

This function seems intuitively secure, since the first output $pi+1$, which we assume to be public, only depends on the public input pi, but not on the secret input si. The second

output $pi+si*2$ depends on both the public input and the secret input, but that's okay, since we assume this second output to be secret.

Still, how can we mathematically define that this function is secure? Let's try it on a couple of inputs:

```
Example example1_secure_f : secure_f 0 0 = (1,0).
Proof. reflexivity. Qed.

Example example2_secure_f : secure_f 0 1 = (1,2).
Proof. reflexivity. Qed.

Example example3_secure_f : secure_f 1 2 = (2,5).
Proof. reflexivity. Qed.
```

In the last two cases the value of the public output is equal to the value of secret input. But that's just a coincidence, and has nothing to do with the public output leaking the secret input, which wasn't used at all in computing the public output.

7.2 Naive attempt at defining secrecy

So a naive security definition, which we'll only use as a strawman, is one that simply requires that public outputs are different from secret inputs:

```
Definition broken_sec_def (f : nat → nat → nat×nat) :=
  ∀ pi si, fst (f pi si) ≠ si.
```

As discussed above, this definition would reject our secure function above as insecure:

```
Lemma broken_sec_def_rejects_secure_f : ¬broken_sec_def secure_f.
Proof. intros Hc. apply (Hc 0 1). reflexivity. Qed.
```

Even worse, this broken definition of security would allow insecure functions, such as the following one whose public output is $si+1$:

```
Definition insecure_f (pi si : nat) : nat×nat := (si+1, pi+si×2).
```

This function's public output is never equal to its secret input, yet an attacker can easily compute one from the other by just subtracting 1. So the secret is entirely leaked, yet our broken definition accepts this:

```
Lemma broken_sec_def_accepts_insecure_f : broken_sec_def insecure_f.
Proof.
  unfold broken_sec_def. intros pi si. induction si as [| si' IH].
  - simpl. intros contra. discriminate contra.
  - simpl in *. intro Hc. injection Hc as Hc. apply IH. apply Hc.
Qed.
```

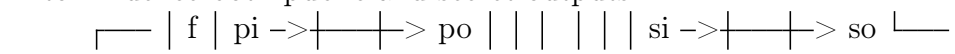
This attempt at defining secure information flow by looking at how inputs and outputs are related for a single execution of the program was a complete failure. In fact, it is well known in the formal security research community that secure information flow *cannot* be defined by looking at just one single program execution.

7.3 Noninterference for pure functions

The simplest correct way to define secure information flow is a property called *noninterference* *Sabelfeld* **and** *Myers* 2003 (in Bib.v), which in its most standard form looks at *two* program executions: for two different secret inputs the public outputs should not change:

```
Definition noninterferent {PI SI PO SO : Type} (f:PI→SI→PO×SO) :=
  ∀ (pi:PI) (si1 si2:SI), fst (f pi si1) = fst (f pi si2).
```

This definition prevents secret inputs from interfering with public outputs in any way. At the same time it allows secret inputs to influence secret outputs and also public inputs to influence both public and secret outputs:



The definition above defines noninterference for arbitrary types of inputs and outputs, so we can instantiate them to **nat** when looking at our example functions above:

```
Lemma noninterferent_secure_f : noninterferent secure_f.
Proof. unfold noninterferent, secure_f. simpl. reflexivity. Qed.

Lemma interferent_insecure_f : ¬noninterferent insecure_f.
Proof.
  unfold noninterferent. simpl. intros contra.
  specialize (contra 42 0 1). simpl in contra. discriminate contra.
Qed.
```

The secure_f function above is quite obviously noninterferent, because the expression $pi+1$ computing the public output doesn't syntactically mention the secret input at all. Since noninterference is a semantic property though (not a syntactic one), functions where the expression computing the public input does syntactically mention the secret input can still be noninterferent. Here is a first example:

```
Definition less_obvious_f1 (pi si : nat) : nat×nat := (si × 0, pi+si).
```

This function is noninterferent; since the public output is constant 0, so it can't depend on *si*, even if it syntactically mentions it:

```
Lemma noninterferent_less_obvious_f1 : noninterferent less_obvious_f1.
Proof.
  unfold noninterferent, less_obvious_f1. intros pi si1 si2.
  simpl. repeat rewrite ← mult_n_O. reflexivity.
Qed.
```

Here is another example of a function that is noninterferent, even if this is not syntactically obvious:

```
Definition less_obvious_f2 (pi si : nat) : nat×nat :=
  (if Nat.eqb si 1 then si × pi else pi, pi+si).
```

For proving this we show that the public output of this function is in fact always equal to just its public input:

```
Lemma aux_f2 : ∀ si pi, (if Nat.eqb si 1 then si × pi else pi) = pi.
Proof.
  intros si pi. destruct si; simpl.
  - reflexivity.
  - destruct si.
    + simpl. rewrite ← plus_n_O. reflexivity.
    + simpl. reflexivity.
Qed.

Lemma noninterferent_less_obvious_f2 : noninterferent less_obvious_f2.
Proof.
  unfold noninterferent, less_obvious_f2. intros pi si1 si2.
  repeat rewrite aux_f2. simpl. reflexivity.
Qed.
```

Branching on a secret can, however, be dangerous, since one can easily leak the secret this way, even if both the `then` and the `else` branches are public. For instance the following function leaks whether *si* is zero or not, so it is not noninterferent.

```
Definition less_obvious_f3 (pi si : nat) : nat×nat :=
  (if Nat.eqb si 0 then 1 else 0, pi+si).

Lemma interferent_less_obvious_f3 : ¬noninterferent less_obvious_f3.
Proof.
  unfold noninterferent, less_obvious_f3. simpl. intros contra.
  specialize (contra 42 0 1). simpl in contra. discriminate contra.
Qed.
```

7.3.1 Noninterference Exercises

Let's practice with some "prove or disprove noninterference" exercises, for which you are required to give constructive proofs, i.e. the use of classical axioms like excluded middle is not allowed.

Exercise: 1 star, standard (prove_or_disprove_obvious_f1)

```
Definition obvious_f1 (pi si : nat) : nat×nat := (0,0).

Lemma prove_or_disprove_obvious_f1 :
  noninterferent obvious_f1 ∨ ¬noninterferent obvious_f1.
Proof.
  Admitted.
```

□

Exercise: 1 star, standard (prove_or_disprove_obvious_f2)

```
Definition obvious_f2 (pi si : nat) : nat×nat := (pi+(2×si),(2×pi)+si).

Lemma prove_or_disprove_obvious_f2 :
```

```
  noninterferent obvious_f2 ∨ ¬noninterferent obvious_f2.
Proof.
  Admitted.
```
□

Exercise: 2 stars, standard (prove_or_disprove_less_obvious_f4)

```
Definition less_obvious_f4 (pi si : nat) : nat×nat :=
  (if Nat.eqb si 0 then si × pi else pi, pi+si).
```

Is the less_obvious_f4 function noninterferent or not?

```
Lemma prove_or_disprove_less_obvious_f4 :
  noninterferent less_obvious_f4 ∨ ¬noninterferent less_obvious_f4.
Proof.
  Admitted.
```
□

Exercise: 2 stars, standard (prove_or_disprove_less_obvious_f5)

```
Definition less_obvious_f5 (pi si : nat) : nat×nat :=
  (if Nat.eqb si 0 then si + pi else pi, pi+si).
```

Is the less_obvious_f5 function noninterferent or not?

```
Lemma prove_or_disprove_less_obvious_f5 :
  noninterferent less_obvious_f5 ∨ ¬noninterferent less_obvious_f5.
Proof.
  Admitted.
```
□

Exercise: 2 stars, standard (prove_or_disprove_less_obvious_f6)

```
Definition less_obvious_f6 (pi si : nat): nat×nat :=
  (if Nat.ltb si pi then 0 else pi, pi+si).
```

Is the less_obvious_f6 function noninterferent or not?

```
Lemma prove_or_disprove_less_obvious_f6 :
  noninterferent less_obvious_f6 ∨ ¬noninterferent less_obvious_f6.
Proof.
  Admitted.
```
□

Exercise: 3 stars, standard, optional (prove_or_disprove_less_obvious_f7)

```
Definition less_obvious_f7 (pi si : nat): nat×nat :=
  if Nat.eqb (si + pi) 0 then (si,pi) else (pi,si).

Lemma prove_or_disprove_less_obvious_f7 :
```

```
  noninterferent less_obvious_f7 ∨ ¬noninterferent less_obvious_f7.
Proof.
  Admitted.
  □
```

7.4 A too-strong secrecy definition

In the definition of noninterference above we pass the same public inputs to the two executions and this allows public outputs to depend on public inputs. To convince ourselves of this, let's look at the following overly strong definition of security:

```
Definition too_strong_sec_def {PI SI PO SO : Type} (f:PI→SI→PO×SO) :=
  ∀ (pi1 pi2:PI) (si1 si2:SI), fst (f pi1 si1) = fst (f pi2 si2).
```

This basically says that the public output of f can depend neither on the public input not on the secret input, so it has to be constant, which is not the case for our secure_f.

```
Lemma secure_f_rejected_again : ¬too_strong_sec_def secure_f.
Proof.
  unfold too_strong_sec_def, secure_f. simpl. intros contra.
  specialize (contra 0 1 0 0). discriminate contra.
Qed.
```

7.5 Noninterferent implies splittable

Noninterference is still a very strong property, though. In particular, f being noninterferent is equivalent to f being splittable into two different functions, one of which doesn't get the secret at all.

```
Definition splittable {PI SI PO SO : Type} (f:PI→SI→PO×SO) :=
  ∃ (pf : PI → PO) (sf : PI → SI → SO),
    ∀ pi si, f pi si = (pf pi, sf pi si).

Theorem splittable_noninterferent : ∀ PI SI PO SO : Type,
  ∀ f : PI → SI → PO×SO, splittable f → noninterferent f.
Proof.
  unfold splittable, noninterferent.
  intros PI SI PO SO f [pf [sf H]] pi si1 si2.
  rewrite H. rewrite H. simpl. reflexivity.
Qed.

Theorem noninterferent_splittable : ∀ PI SI PO SO : Type,
  ∀ some_si : SI,
  ∀ f : PI → SI → PO×SO, noninterferent f → splittable f.
Proof.
  unfold splittable, noninterferent.
```

```
  intros PI SI PO SO some_si f Hni.
  ∃ (fun pi ⇒ fst (f pi some_si)).
  ∃ (fun pi si ⇒ snd (f pi si)).
  intros pi si. rewrite (Hni _ _ si).
  destruct (f pi si) as [po so]. reflexivity.
Qed.
```

7.6 Secure Multi-Execution (SME)

The previous proof also captures the key idea behind Secure Multi-Execution (SME) *Devriese* **and** *Piessens* 2010 (in Bib.v), an enforcement mechanism that can make *any* function noninterferent. To achieve this SME runs the function twice, once passing a dummy secret as input to obtain the public output, and once using the real secret input to obtain the secret output.

```
Definition sme {PI SI PO SO : Type} (some_si : SI)
  (f:PI→SI→PO×SO) : PI→SI→PO×SO :=
    fun pi si ⇒ (fst (f pi some_si), snd (f pi si)).
```

Functions protected by sme are guaranteed to satisfy noninterference:

```
Theorem noninterferent_sme : ∀ PI SI PO SO : Type,
  ∀ some_si : SI,
  ∀ f : PI → SI → PO×SO,
    noninterferent (sme some_si f).
Proof. intros PI SI PO SO some_si f pi si1 si2. simpl. reflexivity. Qed.
```

Moreover, if the function we pass to sme is already noninterferent, then its behavior will not change; so we say that sme is a *transparent* enforcement mechanism for noninterference:

```
Theorem transparent_sme : ∀ PI SI PO SO : Type,
  ∀ some_si : SI,
  ∀ f : PI → SI → PO×SO,
    noninterferent f → ∀ pi si, f pi si = sme some_si f pi si.
Proof.
  unfold noninterferent, sme. intros PI SI PO SP some_si f Hni pi si.
  rewrite (Hni _ _ si).
  destruct (f pi si) as [po so]. reflexivity.
Qed.
```

It is interesting to look at what sme does for *interferent* functions, like insecure_f, whose public output was one plus its secret input:

```
Example example1_sme_insecure_f: sme 0 insecure_f 0 0 = (1, 0).
Proof. reflexivity. Qed.

Example example2_sme_insecure_f: sme 0 insecure_f 0 1 = (1, 2).
Proof. reflexivity. Qed.
```

```
Example example3_sme_insecure_f: sme 0 insecure_f 1 1 = (1, 3).
Proof. reflexivity. Qed.
```

Now the public output of sme insecure_f 0 is one plus the dummy constant 0, so always the constant 1.

```
Lemma constant_sme_insecure_f: ∀ pi si,
  fst (sme 0 insecure_f pi si) = 1.
Proof. reflexivity. Qed.
```

This is a secure behavior, but it is different from that of the original insecure_f function. So we are giving up some correctness for security. There is no free lunch!

Of course the public output of sme does not always become, since some functions still use the public input.

```
Definition another_insecure_f (pi si : nat) : nat×nat := (pi+si, pi+si).

Lemma sme_another_insecure_f : ∀ pi si,
  sme 0 (another_insecure_f) pi si = (pi,pi+si).
Proof. unfold sme, another_insecure_f.
  intros pi si. simpl. rewrite ← plus_n_O. reflexivity. Qed.
```

Exercise: 1 star, standard (sme_another_insecure_f2)

```
Definition another_insecure_f2
(pi si : nat) : nat×nat :=
  (if Nat.eqb si 0 then si × pi + pi else pi, pi+si).

Lemma sme_another_insecure_f2 : ∀ pi si,
    sme 0 (another_insecure_f2) pi si = (pi, pi+si).
Proof.
    Admitted.
```

□

Exercise: 2 stars, standard (sme_another_insecure_f3)

```
Definition another_insecure_f3
(pi si : nat) : nat×nat :=
  (if Nat.eqb si pi then si × pi else pi, pi+si).

Lemma interferent_another_insecure_f3 : ¬ noninterferent another_insecure_f3.
Proof.
  unfold noninterferent, another_insecure_f3. simpl.
  intros contra. specialize (contra 8 2 8). simpl in contra. discriminate contra.
Qed.

Lemma sme_another_insecure_f3 : ∀ pi si,
    sme 0 (another_insecure_f3) pi si = (pi, pi+si).
Proof.
    Admitted.
```

□

The other downside of sme is that we have to run the function twice for our two security levels, public and secret. In general, we need to run the program as many times as we have security levels, which is often an exponential number, say if we take our security levels to be sets of principals. This is inefficient!

Other information-flow control mechanisms overcome this downside, but have other downsides of their own, for instance:

- by requiring nontrivial manual proofs for each individual program (e.g., Relational Hoare Logic), or
- by using static overapproximations that reject some secure programs (security type systems), or
- by using dynamic overapproximations that unnecessarily change program behavior, for instance forcefully terminating even some secure programs to prevent leaks, in which case they are not transparent (dynamic information-flow control; an extension of dynamic taint tracking to also handle implicit flows).

Again, there is no free lunch!

7.7 Noninterference for state transformers

The development above is quite easy to adapt to Rocq functions that transform states (state→state), where we label each variable as either public or secret using a map of type pub_vars.

```
Print state.
```

```
Definition pub_vars := total_map bool.
```

Instead of requiring that the first elements of two pairs are equal, we require that the two states have equal values on the variables labeled public by the *pub* map.

```
Definition pub_equiv (pub : pub_vars) (s1 s2 : state) :=
  ∀ x:string, pub x = true → s1 x = s2 x.
```

This makes the definition more symmetric, since we can use pub_equiv both for the input states and the output states:

```
Definition noninterferent_state pub (f : state → state) :=
  ∀ s1 s2, pub_equiv pub s1 s2 → pub_equiv pub (f s1) (f s2).
```

We can prove an equivalence between noninterferent_state and our original noninterferent definition. For this we need to split and merge states.

We also need a few helper lemmas.

The way we define split_state and *merge_state* is a good example of programming with higher-order functions, and there's more of this in Maps.

The split_state function takes a state s and zeroes out the variables x for which $pub\ x$ is different than an argument bit b. So split_state s pub true keeps the public variables, and zeroes out the secret ones. Dually, split_state s pub false keeps the secret variables, and zeroes out the public ones.

```
Definition split_state (s:state) (pub:pub_vars) (b:bool) : state :=
  fun x : string ⇒ if Bool.eqb (pub x) b then s x else 0.
```

The *merge_state* function takes in two states s1 and s2 and produces a new state that contains the public variables from s1 and the private variables from s2.

```
Definition merge_states (s1 s2:state) (pub:pub_vars) : state :=
  fun x : string ⇒ if pub x then s1 x else s2 x.

Definition split_state_fun (pub : pub_vars) (mf : state → state) :=
  fun s1 s2 : state ⇒
    let ms := mf (merge_states s1 s2 pub) in
    (split_state ms pub true, split_state ms pub false).
```

The technical development needed for the equivalence proof between noninterferent_state and our original noninterferent definition is not that interesting though, and one can skip directly to the noninterferent_state_ni statement on first read.

```
Definition pub_equiv_split (pub : pub_vars) (s1 s2 : state) :=
  ∀ x:string, (split_state s1 pub true) x = (split_state s2 pub true) x.

Theorem pub_equiv_split_iff : ∀ pub s1 s2,
  pub_equiv pub s1 s2 ↔ pub_equiv_split pub s1 s2.
Proof.
  unfold pub_equiv, pub_equiv_split, split_state. intros. split.
  - intros H x. destruct (Bool.eqb_spec (pub x) true).
    + apply H. apply e.
    + reflexivity.
  - intros H x. specialize (H x). destruct (Bool.eqb_spec (pub x) true).
    + intros _. apply H.
    + contradiction.
Qed.

Theorem pub_equiv_merge_states : ∀ pub s z1 z2,
  pub_equiv pub (merge_states s z1 pub) (merge_states s z2 pub).
Proof.
  unfold pub_equiv, merge_states. intros pub s z1 z2 x Hx.
  rewrite Hx. reflexivity.
Qed.

From Stdlib Require Import FunctionalExtensionality.

Theorem merge_states_split_state : ∀ s pub,
  merge_states (split_state s pub true) (split_state s pub false) pub = s.
Proof.
```

```
  unfold merge_states, split_state. intros s pub.
  apply functional_extensionality. intro x.
  destruct (pub x) eqn:Heq; reflexivity.
Qed.
```

Now we can finally state our theorem about the equivalence between *non_interferent_state* and noninterferent:

```
Theorem noninterferent_state_ni : ∀ pub f,
  noninterferent_state pub f ↔
  noninterferent (split_state_fun pub f).
Proof.
  unfold noninterferent_state, noninterferent, split_state_fun.
  intros pub f. split.
  - intros H s z1 z2. simpl.
    assert (H' : pub_equiv pub (merge_states s z1 pub) (merge_states s z2 pub)).
      { apply pub_equiv_merge_states. }
    apply H in H'. rewrite pub_equiv_split_iff in H'.
    unfold pub_equiv_split in H'. apply functional_extensionality. apply H'.
  - intros H s1 s2 Hequiv. simpl in H.
    rewrite pub_equiv_split_iff in Hequiv. unfold pub_equiv_split in Hequiv.
    rewrite pub_equiv_split_iff. unfold pub_equiv_split. intro x.
    specialize (H (split_state s1 pub true)
                  (split_state s1 pub false)
                  (split_state s2 pub false)).
    rewrite merge_states_split_state in H.
    apply functional_extensionality in Hequiv. rewrite Hequiv in H.
    rewrite merge_states_split_state in H.
    rewrite H. reflexivity.
Qed.
```

7.8 SME for state transformers

We can use the split_state and merge_states functions above to also define SME for state transformers. We call the split_state below to zero out all secret variables before calling f the first time to obtain the final value of the public variables.

```
Definition sme_state (f : state → state) (pub:pub_vars) :=
  fun s ⇒ merge_states (f (split_state s pub true)) (f s) pub.
```

We will see examples of this in an upcoming section, but for now we prove the same two theorems as for sme above:

```
Theorem noninterferent_sme_state : ∀ pub f,
  noninterferent_state pub (sme_state f pub).
Proof.
```

```
  unfold noninterferent_state, sme_state.
  intros pub f s1 s2 Hequiv.
  rewrite pub_equiv_split_iff in Hequiv.
  unfold pub_equiv_split in Hequiv.
  apply functional_extensionality in Hequiv. rewrite Hequiv.
  apply pub_equiv_merge_states.
Qed.

Theorem transparent_sme_state : ∀ f pub,
  noninterferent_state pub f → ∀ s, f s = sme_state f pub s.
Proof.
  unfold noninterferent_state, sme_state.
  intros f pub Hni s.
  unfold merge_states, split_state. unfold pub_equiv in Hni.
  apply functional_extensionality. intro x.
  destruct (pub x) eqn:Eq.
  - apply Hni.
    + intros x' Hx'.
      destruct (Bool.eqb_spec (pub x') true).
      × reflexivity.
      × contradiction.
    + assumption.
  - reflexivity.
Qed.
```

One thing to note in this proof is that we used the lemma *Bool.eqb_spec* to do case analysis on whether the *pub x'* is equal to true. For more details on how this works, please check out the explanations about the **reflect** inductive predicate in *IndProp*.

7.8.1 Optional: Connection between sme and sme_state

We can formally relate sme amd sme_state, but this gets pretty technical, so the curious reader can directly skip to the two theorems at the end of this subsection.

```
Lemma split_merge_public: ∀ s pub,
    split_state s pub true = merge_states s (fun _ ⇒ 0) pub.
Proof.
  intros. eapply functional_extensionality. intro x.
  unfold split_state, merge_states.
  destruct (pub x) eqn:PUB; simpl; reflexivity.
Qed.

Lemma split_merge_split_true: ∀ s s' pub,
    split_state (merge_states s s' pub) pub true = split_state s pub true.
Proof.
  intros. eapply functional_extensionality. intro x.
```

```
  unfold split_state, merge_states.
  destruct (pub x) eqn:PUB; simpl; reflexivity.
Qed.

Lemma split_merge_split_false: ∀ s s' pub,
    split_state (merge_states s s' pub) pub false = split_state s' pub false.
Proof.
  intros. eapply functional_extensionality. intro x.
  unfold split_state, merge_states.
  destruct (pub x) eqn:PUB; simpl; reflexivity.
Qed.

Lemma merge_states_same: ∀ s pub,
    merge_states s s pub = s.
Proof.
  unfold merge_states. intros.
  eapply functional_extensionality. intro x.
  destruct (pub x); reflexivity.
Qed.

Lemma split_state_idem: ∀ s pub b,
    split_state (split_state s pub b) pub b = split_state s pub b.
Proof.
  unfold split_state. intros.
  eapply functional_extensionality. intro x.
  destruct (Bool.eqb (pub x) b); reflexivity.
Qed.

Lemma eqb_neg_distr_r: ∀ b1 b2,
    Bool.eqb b1 (negb b2) = negb (Bool.eqb b1 b2).
Proof. intros. destruct b1, b2; simpl; reflexivity. Qed.

Lemma split_state_orthogonal: ∀ s pub b,
    split_state (split_state s pub b) pub (negb b) = fun _ ⇒ 0.
Proof.
  unfold split_state. intros.
  eapply functional_extensionality. intro x.
  rewrite eqb_neg_distr_r.
  destruct (Bool.eqb (pub x) b) eqn:BOOL; simpl; reflexivity.
Qed.
```

First, we show a relationship between sme and sme_state using split_state_fun:

```
Theorem split_sme_state_sme: ∀ pub f,
    split_state_fun pub (sme_state f pub) = sme (fun _ ⇒ 0) (split_state_fun pub f).
Proof.
  intros.
  eapply functional_extensionality. intro PI.
```

```
  eapply functional_extensionality. intro SI.
  unfold split_state_fun, sme.
  rewrite pair_equal_spec. split.
  - simpl. unfold sme_state.
    rewrite ← split_merge_public.
    repeat rewrite split_merge_split_true. reflexivity.
  - simpl. unfold sme_state.
    rewrite split_merge_split_false. reflexivity.
Qed.
```

Second, we also show a relationship between sme and sme_state using merge_state_fun:

```
Definition merge_state_fun (pub : pub_vars) (sf : state → state → state×state) :=
  fun s : state ⇒
    let ps := sf (split_state s pub true) (split_state s pub false) in
    merge_states (fst ps) (snd ps) pub.

Theorem merge_sme_state_sme: ∀ pub f,
    sme_state (merge_state_fun pub f) pub = merge_state_fun pub (sme (fun _ ⇒ 0) f).
Proof.
  intros.
  eapply functional_extensionality. intro s.
  eapply functional_extensionality. intro x.
  unfold merge_state_fun. simpl.
  unfold sme_state. unfold merge_states.
  destruct (pub x) eqn:PUB.
  - rewrite split_state_idem. rewrite split_state_orthogonal. reflexivity.
  - reflexivity.
Qed.
```

7.9 Noninterference for Imp programs without loops

For programs without loops the "failed attempt" evaluation function from Imp works well and allows us to easily define a state transformer function for each Imp command.

```
Print ceval_fun_no_while.
Definition flip {A B C : Type} (f : A → B → C) := fun b a ⇒ f a b.
Definition cinterp : com → state → state := flip ceval_fun_no_while.

Definition noninterferent_no_while pub c : Prop :=
  noninterferent_state pub (cinterp c).
```

A command c without loops is noninterferent if the state transformer obtained by interpreting the command with cinterp maps public-equivalent States to public-equivalent states.

Let's use this definition to prove that the following command is noninterferent:

```
Definition xpub : pub_vars := (X !-> true; _ !-> false).
```

```
Definition secure_com : com :=
  <{ X := X+1;
     Y := (X-1)+Y×2 }>.
```

For proving secure_com noninterferent we first prove a few helper lemmas.

```
Lemma xpub_true : ∀ x, xpub x = true → x = X.
Proof.
  unfold xpub. intros x Hx.
  destruct (eqb_spec x X).
  - subst. reflexivity.
  - rewrite t_update_neq in Hx.
    + rewrite t_apply_empty in Hx. discriminate.
    + intro contra. subst. contradiction.
Qed.
```

Here we are using the t_update_neq and t_apply_empty lemmas from Maps

```
Lemma xpubX : xpub X = true.
Proof. reflexivity. Qed.
```

Using these lemmas the noninterference proof for secure_com is easy:

```
Lemma noninterferent_secure_com :
  noninterferent_no_while xpub secure_com.
Proof.
  unfold noninterferent_no_while, noninterferent_state, secure_com.
  intros s1 s2 PEQUIV x Hx.

  apply xpub_true in Hx. subst.

  specialize (PEQUIV X xpubX).

  simpl. rewrite PEQUIV. reflexivity.
Qed.
```

Exercise: 2 stars, standard (noninterferent_secure_ex1) Definition secure_ex1 :=

```
  <{ Y := Y - 1;
     X := 1 }>.
```

```
Lemma noninterferent_secure_ex1 :
  noninterferent_no_while xpub secure_ex1.
Proof.
  Admitted.
```

□

Exercise: 3 stars, standard, optional (noninterferent_secure_ex2) Definition secure_ex2 :=

```
  <{ if X = 0 then
```

```
    X := X + 5
  else
    Y := X
  end }>.

Lemma noninterferent_secure_ex2 :
  noninterferent_no_while xpub secure_ex2.
Proof.
  Admitted.
```

□

Now let's look at a couple of insecure commands:

```
Definition insecure_com1 : com :=
  <{ X := Y+1;
     Y := (X-1)+Y×2 }>.
```

An *explicit flow* is when a command directly assigns an expression depending on secret variables to a public variable, like the X := Y+1 assignment above. Explicit flows are easier to find automatically and even simple taint-tracking would be enough for discovering this.

We prove that insecure_com1 is interferent as follows:

```
Lemma interferent_insecure_com1 :
  ¬noninterferent_no_while xpub insecure_com1.
Proof.
  unfold noninterferent_no_while, noninterferent_state, insecure_com1.
  intro Hc.

  set (s1 := (X !-> 0 ; Y !-> 0)).
  set (s2 := (X !-> 0 ; Y !-> 1)).
  specialize (Hc s1 s2).

  assert (PEQUIV: pub_equiv xpub s1 s2).
  { clear Hc. intros x H. apply xpub_true in H. subst. reflexivity. }

  specialize (Hc PEQUIV X xpubX).

  simpl in Hc. unfold s1, s2, t_update in Hc. simpl in Hc.

  discriminate Hc.
Qed.
```

As we saw above, the set tactic allows us to give names to complex expression, making proofs more readable and manageable. It's particularly useful when constructing concrete counterexamples where one needs to work with specific values.

Exercise: 2 stars, standard (interferent_insecure_com_explicit) Definition insecure_com_explic
:=

```
  <{ X := Y + X;
     Y := Y - 1 }>.
```

```
Lemma interferent_insecure_com_explicit :
  ¬noninterferent_no_while xpub insecure_com_explicit.
Proof.
  Admitted.
```
□

Noninterference can be violated not only by explicit flows, but also by *implicit flows*, which leak secret information via the control-flow of the program. Here is a simple example:

```
Definition insecure_com2 : com :=
  <{ if Y = 0 then
       Y := 42
     else
       X := X+1
     end }>.
```

Here the expression X+1 we are assigning to X is public information, but we are doing this assignment after we branched on a secret condition Y = 0, so we are indirectly leaking information about the value of Y. In this case we can infer that if X gets incremented the value of Y is not 0.

```
Lemma interferent_insecure_com2 :
  ¬noninterferent_no_while xpub insecure_com2.
Proof.
  unfold noninterferent_no_while, noninterferent_state, insecure_com1.
  intro Hc.

  set (s1 := (X !-> 0 ; Y !-> 0)).
  set (s2 := (X !-> 0 ; Y !-> 1)).
  specialize (Hc s1 s2).

  assert (PEQUIV: pub_equiv xpub s1 s2).
  { clear Hc. intros x H. apply xpub_true in H. subst. reflexivity. }

  specialize (Hc PEQUIV X xpubX).

  simpl in Hc. unfold s1, s2, t_update in Hc. simpl in Hc.

  discriminate Hc.
Qed.
```

Exercise: 3 stars, standard (interferent_insecure_com_implicit) `Definition insecure_com_impli`
`:=`

```
  <{ if Y = 42 then
       X := X - 1
     else
       Y := 2 × Y
     end }>.
```

```
Lemma interferent_insecure_com_implicit :
  ¬noninterferent_no_while xpub insecure_com_implicit.
Proof.
    Admitted.
    □
```

We will return to explicit and implicit flows in the StaticIFC chapter.

7.10 SME for Imp programs without loops

We can use sme_state to execute such programs to obtain a noninterferent state transformer by running programs 2 times, once on a state where the secrets were zeroed out and once on the original input state, and then merging the final states.

```
Print sme_state.

Definition sme_cmd c : pub_vars→state→state := sme_state (cinterp c).
```

The result of applying sme_cmd to a program is not a program, but a state transformer. We prove noninterference and transparency for the state transformers obtained by sme_cmd using our noninterference and transparency theorems about sme_state:

```
Theorem noninterferent_sme_cmd : ∀ c pub,
  noninterferent_state pub (sme_cmd c pub).
Proof. intros c p. apply noninterferent_sme_state. Qed.

Theorem transparent_sme_cmd : ∀ c pub,
    noninterferent_state pub (fun s ⇒ ceval_fun_no_while s c) →
    ∀ s, cinterp c s = sme_cmd c pub s.
Proof.
  unfold sme_cmd. intros c pub NI. apply transparent_sme_state. apply NI.
Qed.
```

Perhaps more interesting is to look at how sme_cmd changes the behavior of some insecure commands:

```
Print insecure_com1. Definition secure_com1 : com :=
  <{ X := 1;
     Y := Y×3 }>.

Lemma sme_insecure_com1 : sme_cmd insecure_com1 xpub = cinterp secure_com1.
Proof.
  eapply functional_extensionality. intros st.
  unfold sme_cmd, sme_state, insecure_com1. simpl.
  eapply functional_extensionality. intros x.
  unfold merge_states. simpl.

  destruct (xpub x) eqn:HXP.
  { eapply xpub_true in HXP. subst.
```

```
    rewrite t_update_neq; try (intros Hcontra; discriminate).
    rewrite t_update_eq; simpl; auto. }

  destruct (eqb x Y) eqn:HY.
  { rewrite eqb_eq in HY. subst.
    repeat rewrite t_update_eq.
    repeat rewrite t_update_neq; try (intros Hcontra; discriminate).
    lia. }

  assert (HX: x ≠ X).
  { intros Hx. subst. rewrite xpubX in HXP. discriminate. }

  rewrite eqb_neq in HY.
  repeat (rewrite t_update_neq; auto).
Qed.
```

The example above shows that the effect of applying sme_cmd is hard to predict statically and it is not just a simple syntactic transformation of the original command. Here is another example of that:

```
Definition insecure_com2' : com :=
  <{ if Y = 0 then
       X := 42
     else
       X := X + 1
     end }>.

Definition secure_com2' : com :=
  <{ X := 42 }>.

Lemma sme_insecure_com2' : sme_cmd insecure_com2' xpub = cinterp secure_com2'.
Proof.
  eapply functional_extensionality. intros st.
  unfold sme_cmd. unfold sme_state. simpl.
  eapply functional_extensionality. intros x.
  unfold merge_states. simpl.

  destruct (xpub x) eqn:HXP.
  { eapply xpub_true in HXP. subst.
    rewrite t_update_eq; simpl; auto. }

  destruct (eqb x Y) eqn:HY.
  { rewrite eqb_eq in HY. subst.
    destruct (st Y =? 0);
      repeat rewrite t_update_neq;
      try (intros Hcontra; discriminate); reflexivity. }
```

```
  assert (HX: x ≠ X).
  { intros Hx. subst. rewrite xpubX in HXP. discriminate. }

  rewrite eqb_neq in HY.
  destruct (st Y =? 0).
  - rewrite t_update_neq; auto.
  - repeat rewrite t_update_neq; auto.
Qed.
```

For simplicity, above we looked at a modified insecure_com2'. What about the effect of sme_cmd on the *original* insecure_com2? `Print insecure_com2`.

This is more challenging, but it turns out there is a general and systematic way to characterize the effect of sme_cmd as a single program. This program is called a *self-composition* and it captures two executions of the original program (in this case the two executions performed by sme_cmd):

```
Definition pX := "pX"%string.
Definition pY := "pY"%string.
Definition secure_com2 :=
  <{
     pX := X;

     if Y = 0 then Y := 42
              else X := X+1 end;

     pY := 0;

     if pY = 0 then pY := 42
               else pX := pX+1 end;

     X := pX
}>.
```

Because in our simple Imp language we have no way to restore the pX and pY variables to their original state, the equivalence lemma below needs to account for the fact that their values will be different. We do this by reusing our old friend pub_equiv:

```
Definition psecret := (pX !-> false; pY !-> false; __ !-> true).

Lemma sme_insecure_com2 : ∀ st,
    pub_equiv psecret (sme_cmd insecure_com2 xpub st)
                      (cinterp secure_com2 st).
Proof.
  unfold pub_equiv. intros st x PSEC.
  unfold sme_cmd, sme_state, insecure_com2. simpl.
```

```
unfold merge_states. simpl.

destruct (xpub x) eqn:HXP.
{ eapply xpub_true in HXP. subst.
  rewrite t_update_neq; try discriminate.
  rewrite t_update_eq.
  unfold split_state. simpl.
  repeat (rewrite t_update_neq; try discriminate).
  destruct (st Y =? 0) eqn:HY0.
  - rewrite t_update_neq; try discriminate.
    rewrite t_update_eq. reflexivity.
  - rewrite t_update_neq; try discriminate.
    rewrite t_update_eq. reflexivity. }

destruct (eqb x Y) eqn:HY.
{ rewrite eqb_eq in HY. subst.
  destruct (st Y =? 0) eqn:HY0.
  - rewrite t_update_eq.
    repeat (rewrite t_update_neq; try discriminate).
    rewrite HY0.
    rewrite t_update_eq. reflexivity.
  - repeat (rewrite t_update_neq; try discriminate).
    rewrite HY0.
    repeat (rewrite t_update_neq; try discriminate).
    reflexivity. }

rewrite eqb_neq in HY.

assert (HX: x ≠ X).
{ intros Hx. subst. rewrite xpubX in HXP. discriminate. }

assert (HpXY: x ≠ pX ∧ x ≠ pY).
{ clear - PSEC.
  unfold psecret in PSEC.
  destruct (eqb x pX) eqn:HpX.
  - rewrite eqb_eq in HpX. subst.
    rewrite t_update_eq in PSEC. discriminate.
  - destruct (eqb x pY) eqn:HpY.
    + rewrite eqb_eq in HpY. subst.
      rewrite t_update_neq in PSEC; discriminate.
    + rewrite eqb_neq in HpX. subst.
      rewrite eqb_neq in HpY. subst. auto. }

destruct HpXY as [HpX HpY].
```

```
  repeat (rewrite t_update_neq; auto); try discriminate.
  destruct (st Y =? 0) eqn:HY0.
  - repeat (rewrite t_update_neq; auto).
  - repeat (rewrite t_update_neq; auto).
Qed.
```

By optimizing the self-composition program above quite a bit we can finally figure out what sme_cmd does for insecure_com2:

```
Definition secure_com2_simple :=
  <{ if Y = 0 then
       Y := 42
     else
       skip
     end
}>.

Lemma sme_insecure_com2_simple :
  sme_cmd insecure_com2 xpub = cinterp secure_com2_simple.
Proof.
  eapply functional_extensionality. intros st.
  unfold sme_cmd. unfold sme_state. simpl.
  eapply functional_extensionality. intros x.
  unfold merge_states. simpl.

  destruct (xpub x) eqn:HXP.
  { eapply xpub_true in HXP. subst.
    rewrite t_update_neq; try discriminate.
    unfold split_state. simpl.
    destruct (st Y =? 0).
    - rewrite t_update_neq; try discriminate.
      reflexivity.
    - reflexivity. }

  destruct (eqb x Y) eqn:HY.
  { rewrite eqb_eq in HY. subst.
    destruct (st Y =? 0).
    - repeat rewrite t_update_eq. reflexivity.
    - rewrite t_update_neq; try discriminate.
      reflexivity. }

  assert (HX: x ≠ X).
  { intros Hx. subst. rewrite xpubX in HXP. discriminate. }

  rewrite eqb_neq in HY.
```

```
  destruct (st Y =? 0).
  - reflexivity.
  - rewrite t_update_neq; auto.
Qed.
```

Self-composition and the more general concept of a *product program* are generally useful techniques of their own (e.g., for reducing relational properties proved by Relational Hoare Logic to regular properties proved by standard Hoare Logic), but we will not discuss them here any further.

7.11 Noninterference for Imp programs with loops

In the presence of loops, we need to define noninterference using the evaluation relation (**ceval**) of Imp:

```
Definition noninterferent_while pub c := ∀ s1 s2 s1' s2',
  pub_equiv pub s1 s2 →
  s1 =[ c ]=> s1' →
  s2 =[ c ]=> s2' →
  pub_equiv pub s1' s2'.

Ltac invert H := inversion H; subst; clear H.
```

We re-prove noninterference of secure_com for this new definition:

```
Lemma noninterferent_secure_com_a_bit_harder :
  noninterferent_while xpub secure_com.
Proof.
  unfold noninterferent_while, secure_com, pub_equiv.
  intros s1 s2 s1' s2' H H1 H2 x Hx.
  apply xpub_true in Hx. subst.
  invert H1. invert H4. invert H7.
  invert H2. invert H3. invert H6. simpl.
  rewrite (H X xpubX). reflexivity.
Qed.
```

The advantage of the new definition is that it also says something meaningful about programs with while loops.

For instance, we can prove that fact_in_coq from Imp does not leak the old value of Y and Z to X:

```
Print fact_in_coq.

Lemma noninterferent_fact_in_coq :
  noninterferent_while xpub fact_in_coq.
Proof.
  unfold noninterferent_while, fact_in_coq, pub_equiv.
  intros s1 s2 s1' s2' H H1 H2 x Hx.
```

```
  apply xpub_true in Hx. subst.
  assert (Hs: ∀ s s', s =[ Z := X; Y := 1; while Z ≠ 0 do Y := Y × Z; Z := Z - 1 end
]=> s' →
                                s X = s' X).
  { intros. clear -H0. invert H0. invert H5.
    invert H2. invert H1. simpl in H6.
    remember (Y !-> 1; Z !-> s X; s) as st.
    replace (s X) with (st X); cycle 1.
    { invert Heqst. rewrite t_update_neq; eauto. intros contra.
      discriminate contra. }
    clear -H6.
    remember <{ while Z ≠ 0 do Y := Y × Z; Z := Z - 1 end }> as loopdef
      eqn:Heqloopdef.
    revert Heqloopdef.
    induction H6; intros.
    - discriminate Heqloopdef.
    - discriminate Heqloopdef.
    - discriminate Heqloopdef.
    - discriminate Heqloopdef.
    - discriminate Heqloopdef.
    - reflexivity.
    - invert Heqloopdef.
      rewrite ← IHceval2; eauto.
      invert H6_. invert H5. invert H2. simpl.
      rewrite t_update_neq; cycle 1.
      { intros contra. discriminate contra. }
      rewrite t_update_neq; eauto.
      intros contra. discriminate contra. }
  eapply Hs in H1. eapply Hs in H2. rewrite ← H1, ← H2.
  rewrite (H X xpubX). reflexivity.
Qed.
```

7.12 SME for Imp programs with loops

To define SME in the presence of while loops we also need to use a relation, of a similar type to **ceval**:

```
Check ceval : com → state → state → Prop.

Definition sme_while (pub:pub_vars) (c:com) (s s':state) : Prop :=
  ∃ ps ss, split_state s pub true =[ c ]=> ps ∧
    s =[ c ]=> ss ∧
    merge_states ps ss pub = s'.
```

To state that sme_eval is secure, we need to generalize our noninterference definition, so that we can apply it not only to **ceval**, but with any evaluation relation, including sme_while *pub*.

```
Definition noninterferent_while_R (R:com→state→state→Prop) pub c :=
  ∀ s1 s2 s1' s2',
  pub_equiv pub s1 s2 →
  R c s1 s1' →
  R c s2 s2' →
  pub_equiv pub s1' s2'.
```

The proof that *while_sme* is noninterferent is as before, but now it relies on the determinism of **ceval**, which was obvious for state transformer functions, but is not obvious for evaluation relations.

```
Check ceval_deterministic : ∀ (c : com) (st st1 st2 : state),
     st =[ c ]=> st1 →
     st =[ c ]=> st2 →
     st1 = st2.

Theorem noninterferent_while_sme : ∀ pub c,
  noninterferent_while_R (sme_while pub) pub c.
Proof.
  unfold noninterferent_while_R, sme_while.
  intros pub c s1 s2 s1' s2' H [ps1 [ss1 [H1p [H1s H1m]]]]
   [ps2 [ss2 [H2p [H2s H2m]]]].
  subst. rewrite pub_equiv_split_iff in H. unfold pub_equiv_split in H.
  apply functional_extensionality in H. rewrite H in H1p.
  rewrite (ceval_deterministic _ _ _ _ H1p H2p).
  apply pub_equiv_merge_states.
Qed.
```

Turns out we can only prove a weak version of transparency for noninterferent programs, and this has to do with nontermination (more later).

But first we need a few lemmas:

```
Lemma pub_equiv_split_state : ∀ (pub:pub_vars) s,
  pub_equiv pub (split_state s pub true) s.
Proof.
  unfold pub_equiv, split_state.
  intros pub s x Hx. destruct (Bool.eqb_spec (pub x) true).
  - reflexivity.
  - contradiction.
Qed.

Lemma pub_equiv_sym : ∀ (pub:pub_vars) s1 s2,
  pub_equiv pub s1 s2 →
  pub_equiv pub s2 s1.
```

```
Proof.
  unfold pub_equiv. intros pub s1 s2 H x Hx.
  rewrite H.
  - reflexivity.
  - assumption.
Qed.

Lemma merge_state_pub_equiv : ∀ pub ss ps,
  pub_equiv pub ss ps →
  merge_states ps ss pub = ss.
Proof.
  unfold pub_equiv, merge_states.
  intros pub ss ps H. apply functional_extensionality.
  intros x. destruct (pub x) eqn:Heq.
  - rewrite H.
    + reflexivity.
    + assumption.
  - reflexivity.
Qed.
```

More specifically, we can only prove that an sme_while execution implies a **ceval** execution:

```
Theorem somewhat_transparent_while_sme : ∀ pub c,
  noninterferent_while pub c →
  (∀ s s', (sme_while pub) c s s' → s =[ c ]=> s').
Proof.
  unfold noninterferent_while, sme_while.
  intros pub c Hni s s' [ps [ss [Hp [Hs Hm]]]]. subst s'.
    assert(H:pub_equiv pub s (split_state s pub true)).
    { apply pub_equiv_sym. apply pub_equiv_split_state. }
    specialize (Hni s (split_state s pub true) ss ps H Hs Hp).
    apply merge_state_pub_equiv in Hni. rewrite Hni. apply Hs.
Qed.
```

But we cannot prove the reverse implication, since a command terminating when starting in state s, does not necessarily still terminate when starting in state split_state s pub true, as would be needed for proving sme_while.

Yet it seems we can still do most of the things as in the setting without while loops, including SME (just not fully transparent). So is there anything special about loops and nontermination?

Yes, there is! Let's look at our noninterference definition again:

Definition noninterferent_while pub c := forall s1 s2 s1' s2', pub_equiv pub s1 s2 -> s1 = c => s1' -> s2 = c => s2' -> pub_equiv pub s1' s2'.

It says that for any two *terminating* executions, if the initial states agree on their public

variables, then so do the final states. This is traditionally called *termination-insensitive* noninterference (TINI), since it doesn't consider nontermination to be observable to an attacker.

In particular, the following program is *secure* wrt TINI:

```
Definition termination_leak : com :=
  <{ if Y = 0 then
       Y := 42
     else
       while true do skip end
     end }>.
```

And we can prove it ...

```
Lemma Y_neq_X : (Y ≠ X).
Proof. intro contra. discriminate. Qed.
```

We use a lemma that is a homework exercise in Imp: `Check loop_never_stops : ∀ st st', ~(st =[ loop ]=> st').`

```
Definition tini_secure_termination_leak :
  noninterferent_while xpub termination_leak.
Proof.
  unfold noninterferent_while, termination_leak, pub_equiv.
  intros s1 s2 s1' s2' H H1 H2 x Hx. apply xpub_true in Hx.
  subst. specialize (H X xpubX).
  invert H1.
  + invert H8. simpl.
    rewrite (t_update_neq _ _ _ _ _ Y_neq_X).
    invert H2.
    × invert H8. simpl.
      rewrite (t_update_neq _ _ _ _ _ Y_neq_X). assumption.
    × apply loop_never_stops in H8. contradiction.
  + apply loop_never_stops in H8. contradiction.
Qed.
```

7.13 Termination-Sensitive Noninterference

We can give a stronger definition of security that disallows such nontermination leaks. It is traditionally called *termination-sensitive noninterference* (TSNI) and it is defined as follows:

```
Definition tsni_while_R (R:com→state→state→Prop) pub c :=
  ∀ s1 s2 s1',
  R c s1 s1' →
  pub_equiv pub s1 s2 →
  (∃ s2', R c s2 s2' ∧ pub_equiv pub s1' s2').
```

We can prove that termination_leak doesn't satisfy TSNI:

```
Definition tsni_insecure_termination_leak :
  ¬tsni_while_R ceval xpub termination_leak.
Proof.
  unfold tsni_while_R, termination_leak.
  intros Hc.
  specialize (Hc (X !-> 0 ; Y !-> 0) (X !-> 0 ; Y !-> 1)
                 (Y !-> 42; X !-> 0 ; Y !-> 0)).
  assert (HH : (X !-> 0; Y !-> 0) =[ termination_leak ]=>
               (Y !-> 42; X !-> 0; Y !-> 0)).
  { clear. unfold termination_leak. constructor.
    - reflexivity.
    - constructor. reflexivity. }
  specialize (Hc HH). clear HH.
  assert (H: ∀ x, xpub x = true →
                          (X !-> 0; Y !-> 0) x = (X !-> 0; Y !-> 1) x).
  { clear Hc. intros x H. apply xpub_true in H. subst. reflexivity. }
  specialize (Hc H). clear H.
  destruct Hc as [s2' [Hc _]].
  invert Hc.
  - simpl in H4. discriminate.
  - apply loop_never_stops in H5. contradiction.
Qed.
```

More generally, we can prove that TSNI is strictly stronger than TINI (noninterferent_while)

```
Lemma tsni_noninterferent : ∀ pub c,
  tsni_while_R ceval pub c →
  noninterferent_while_R ceval pub c.
Proof.
  unfold noninterferent_while_R, tsni_while_R.
  intros pub c Htsni s1 s2 s1' s2' Hequiv H1 H2.
  specialize (Htsni s1 s2 s1' H1 Hequiv).
  destruct Htsni as [s2'' [H2' Hequiv']].
  rewrite (ceval_deterministic _ _ _ _ H2 H2').
  apply Hequiv'.
Qed.
```

The reverse direction of the implication only works for programs that always terminate (such as most of our simple examples above).

```
Lemma terminating_noninterferent_tsni: ∀ pub c,
  (∀ s, ∃ s', s =[ c ]=> s') →
  noninterferent_while_R ceval pub c →
```

```
  tsni_while_R ceval pub c.
Proof.
  unfold noninterferent_while_R, tsni_while_R.
  intros pub c Hterminating Hni s1 s2 s1' H Eq.
  destruct (Hterminating s2) as [s2' H'].
  ∃ s2'; split.
  - assumption.
  - apply Hni with (s1 := s1) (s2 := s2).
    + assumption.
    + assumption.
    + assumption.
Qed.
```

Now for a more interesting use of TSNI: it turns out that sme_while is transparent for programs satisfying TSNI.

```
Theorem tsni_transparent_while_sme : ∀ pub c,
  tsni_while_R ceval pub c →
  (∀ s s', s =[ c ]=> s' ↔ (sme_while pub) c s s').
Proof.
  unfold tsni_while_R, sme_while.
  intros pub c Hni s s'.
  assert(HH:pub_equiv pub s (split_state s pub true)).
    { apply pub_equiv_sym. apply pub_equiv_split_state. }
  split.
  - intros H. specialize (Hni s (split_state s pub true) s' H HH).
    destruct Hni as [s'' [Heval Hequiv]].
    ∃ s''. ∃ s'. split.
    + assumption.
    + split.
      × assumption.
      × apply merge_state_pub_equiv. assumption.
  - intros [ps [ss [Hp [Hs Hm]]]]. subst s'.
    specialize (Hni s (split_state s pub true) ss Hs HH).
    destruct Hni as [s' [Hp' Hni]].
    rewrite (ceval_deterministic _ _ _ _ Hp Hp').
    apply merge_state_pub_equiv in Hni. rewrite Hni. apply Hs.
Qed.
```

Unfortunately sme_while does not *enforce* TSNI and this is hard to fix in our current setting, where programs only return a result in the end, a final state, so we had to merge the public and secret inputs into a single final state. Instead, SME is commonly defined in a setting with interactive IO, in which public outputs and secret outputs can be performed independently, during the execution *Devriese* **and** *Piessens* 2010 (in Bib.v). In that setting, it does transparently enforce a termination-insensitive version of noninterference later research

has called Indirect TSNI *Ngo et al* 2018 (in Bib.v).

7.13.1 Optional: Counterexample showing that SME doesn't enforce TSNI

We build a counterexample command that does not satisfy TSNI and for which the same publicly equivalent initial states s1 and s2 can be used to show that it still does not satisfy TSNI when run with sme_while.

In particular, we choose s1 below so that the command terminates and so that zeroing out the secret variable Y has no effect on s1. We choose s2 so that the command loops, which implies that it will still loop on s2 also when executed with sme_while.

```
Section TSNICOUNTER.

Definition counter : com := <{ while (Y = 1) do skip end; X := 1 }>.

Definition s1: state := X !-> 0; Y !-> 0; empty_st.
Definition s2: state := X !-> 0; Y !-> 1; empty_st.
Definition s1': state := X !-> 1; s1.

Lemma counter_s1_terminates_s1': s1 =[ counter ]=> s1'.
Proof.
  unfold counter, s1. eapply E_Seq.
  - eapply E_WhileFalse. simpl. reflexivity.
  - eapply E_Asgn. simpl. reflexivity.
Qed.

Lemma counter_s2_loops : ∀ s2',
  ¬ (s2 =[ counter ]=> s2').
Proof.
  unfold counter. intros s2' Hcontra.

  assert (NSTOP: ∀ s s', s Y = 1 →
                         s =[ while Y = 1 do skip end ]=> s' →
                         False).
  { clear. intros.
    remember <{ while Y = 1 do skip end }> as loopdef
             eqn:Heqloopdef.
    generalize dependent H.
    induction H0; try (discriminate Heqloopdef).
    - intros HY.
      injection Heqloopdef as H0 H1. subst.
      simpl in H. rewrite HY in H. discriminate H.
    - intros HY.
      injection Heqloopdef as H0 H1. subst.
      inversion H0_; subst. eapply IHceval2; eauto. }
```

```
  inversion Hcontra; subst. eapply NSTOP in H1; auto.
Qed.

Lemma initial_pub_equiv: pub_equiv xpub s1 s2.
Proof.
  unfold s1, s2, pub_equiv. intros.
  eapply xpub_true in H. subst.
  repeat rewrite t_update_eq. reflexivity.
Qed.

Lemma not_tsni_counter :
  ¬ (tsni_while_R ceval xpub counter).
Proof.
  intros Htsni. unfold tsni_while_R in Htsni.
  specialize (Htsni _ _ _ counter_s1_terminates_s1' initial_pub_equiv).
  destruct Htsni as [s2' [D _]].
  eapply counter_s2_loops. eassumption.
Qed.

Lemma sme_counter_s1_terminates_s1' : sme_while xpub counter s1 s1'.
Proof.
  unfold sme_while, counter.
  ∃ s1', s1'.
  split; [|split].
  - assert (Hsplit: split_state s1 xpub true = s1).
    { unfold split_state, s1, xpub.
      eapply functional_extensionality. intros x.
      destruct (Bool.eqb ((X !-> true; _ !-> false) x) true) eqn: B.
      - reflexivity.
      - destruct (eqb x Y) eqn:HY.
        + rewrite eqb_eq in HY. subst. rewrite t_update_neq.
          × rewrite t_update_eq. reflexivity.
          × intros Hcontra. inversion Hcontra.
        + rewrite eqb_neq in HY.
          destruct (eqb x X) eqn:HX.
          × rewrite eqb_eq in HX. subst.
            rewrite t_update_eq. reflexivity.
          × rewrite eqb_neq in HX.
            rewrite t_update_neq; eauto.
            rewrite t_update_neq; eauto. }
    rewrite Hsplit. eapply counter_s1_terminates_s1'.
  - eapply counter_s1_terminates_s1'.
  - eapply functional_extensionality. intros x.
    unfold merge_states, xpub.
    destruct ((X !-> true; _ !-> false) x); reflexivity.
```

```
Qed.

Lemma sme_counter_s2_loops: ∀ s2',
  ¬ (sme_while xpub counter s2 s2').
Proof.
  unfold not, sme_while. intros s2' H.
  destruct H as [ps [ss [A [B C]]]].
  eapply counter_s2_loops. eassumption.
Qed.

Lemma not_tsni_while_sme :
  ¬ (tsni_while_R (sme_while xpub) xpub counter).
Proof.
  intros Htsni. unfold tsni_while_R in Htsni.
  specialize (Htsni _ _ _ sme_counter_s1_terminates_s1' initial_pub_equiv).
  destruct Htsni as [s2' [D _]].
  eapply sme_counter_s2_loops. eassumption.
Qed.

End TSNICOUNTER.
```

Chapter 8

Library SECF.StaticIFC

8.1 StaticIFC: Information-Flow-Control Type Systems

```
Set Warnings "-notation-overridden,-parsing,-deprecated-hint-without-locality".
From Stdlib Require Import Strings.String.
From SECF Require Import Maps.
From Stdlib Require Import Bool.Bool.
From Stdlib Require Import Arith.Arith.
From Stdlib Require Import Arith.EqNat.
From Stdlib Require Import Arith.PeanoNat. Import Nat.
From Stdlib Require Import Lia.
From SECF Require Export Imp.
From Stdlib Require Import List. Import ListNotations.
Set Default Goal Selector "!".
```

8.2 Noninterference

As explained in the Noninterference chapter, data confidentiality is most often expressed formally as a property called *noninterference*.

To formalize this for Imp programs, we divide the variables as either public or secret by assuming a total map P : pub_vars between variables and Boolean labels:

```
Definition label := bool.

Definition public : label := true.
Definition secret : label := false.

Definition pub_vars := total_map label.
```

A noninterference attacker can only observe the final values of public variables, not of secret ones. We formalize this as a notion of *publicly equivalent states* that agree on the values of all public variables, which are thus indistinguishable to an attacker:

```
Definition pub_equiv (P : pub_vars) {X:Type} (s1 s2 : total_map X) :=
  ∀ x:string, P x = public → s1 x = s2 x.
```

For some total map P from variables to Boolean labels, pub_equiv P is an equivalence relation on states, so reflexive, symmetric, and transitive.

```
Lemma pub_equiv_refl : ∀ {X:Type} (P : pub_vars) (s : total_map X),
  pub_equiv P s s.
Proof. intros X P s x Hx. reflexivity. Qed.

Lemma pub_equiv_sym : ∀ {X:Type} (P : pub_vars) (s1 s2 : total_map X),
  pub_equiv P s1 s2 →
  pub_equiv P s2 s1.
Proof. unfold pub_equiv. intros X P s1 s2 H x Px. rewrite H; auto. Qed.

Lemma pub_equiv_trans : ∀ {X:Type} (P : pub_vars) (s1 s2 s3 : total_map X),
  pub_equiv P s1 s2 →
  pub_equiv P s2 s3 →
  pub_equiv P s1 s3.
Proof. unfold pub_equiv. intros X P s1 s2 s3 H12 H23 x Px.
       rewrite H12; try rewrite H23; auto. Qed.
```

Program c is *noninterferent* if whenever it has two terminating runs from two publicly equivalent initial states s1 and s2, the obtained final states s1' and $s2'$ are also publicly equivalent.

```
Definition noninterferent P c := ∀ s1 s2 s1' s2',
  pub_equiv P s1 s2 →
  s1 =[ c ]=> s1' →
  s2 =[ c ]=> s2' →
  pub_equiv P s1' s2'.
```

Intuitively, c is noninterferent when the value of the public variables in the final state can only depend on the value of public variables in the initial state, and do not depend on the initial value of secret variables.

In particular, changing the value of the secret variables in the initial state (as allowed by pub_equiv P s1 s2), should lead to no change in the final value of the public variables (as required by pub_equiv P s1' $s2'$).

For instance, consider the following command (taken from Noninterference):

```
Definition secure_com : com := <{ X := X+1; Y := X+Y×2 }>.
```

If we assume that variable X is public and variable Y is secret, we can state noninterference for secure_com as follows:

```
Definition xpub : pub_vars := (X !-> public; _ !-> secret).

Definition noninterferent_secure_com :=
  noninterferent xpub secure_com.
```

We have already proved that secure_com is indeed noninterferent both directly using the semantics (in Noninterference). This proof was manual though, while in this chapter we will show how this proof can be done more syntactically and automatically using several *information-flow-control* (IFC) type systems that enforce noninterference for all well-typed programs *Sabelfeld* **and** *Myers* 2003 (in Bib.v).

Not all programs are noninterferent though. For instance, a program that reads the contents of a secret variable and uses that to change the value of a public variable is unlikely to be noninterferent. We call this an *explicit flow* and all our type systems will prevent *all* explicit flows.

Here is a program that has an explicit flow, which in this case breaks noninterference (as we proved in Noninterference):

```
Definition insecure_com1 : com :=
  <{ X := Y+1;
     Y := X+Y×2 }>.

Definition interferent_insecure_com1 :=
  ¬noninterferent xpub insecure_com1.
```

Not all explicit flows break noninterference though. For instance, the following variant of insecure_com1 is noninterferent even if it has an explicit flow. The reason for this is that the variable X is overwritten with public information in a subsequent assignment.

```
Definition secure_com1' : com :=
  <{ X := Y+1;
     X := 42;
     Y := X+Y×2 }>.
```

Since our IFC type systems will prevent all explicit flows, they will also reject secure_com1', even if it is secure with respect to our simple attacker model for noninterference, in which the attacker only observes the *final* values of public variables.

Our type systems will only provide *sound syntactic overapproximations* of the semantic noninterference property.

Explicit flows are not the only way to leak secrets: one can also leak secrets using the control flow of the program, by branching on secrets and then assigning to public variables. We call these leaks *implicit flows*.

```
Definition insecure_com2 : com :=
  <{ if Y = 0
     then Y := 42
     else X := X+1
     end }>.
```

Here the expression X+1 we are assigning to X is public information, but we are doing this assignment after we branched on a secret condition Y = 0, so we are indirectly leaking information about the value of Y. In this case we can infer that if X gets incremented the value of Y is not 0. This program is insecure (as proved in Noninterference), so it will be

rejected by our type systems, which enforce noninterference by also preventing *all* implicit flows.

Not all implicit flows break noninterference though. Here is a program that is noninterferent, even though it contains both an explicit and an implicit flow:

```
Definition secure_p2 :=
  <{ if Y = 0
     then X := Y
     else X := 0
     end }>.
```

Intuitively, even if this program branches on the secret Y, it always assigns the value 0 to X, so no secret is leaked. We can prove this semantically:

```
Lemma xpub_true : ∀ x, xpub x = true → x = X.
Proof.
  unfold xpub. intros x Hx.
  destruct (String.eqb_spec x X).
  - subst. reflexivity.
  - rewrite t_update_neq in Hx.
    + rewrite t_apply_empty in Hx. discriminate.
    + intro contra. subst. contradiction.
Qed.
Ltac invert H := inversion H; subst; clear H.

Lemma noninterference_secure_p2 :
  noninterferent xpub secure_p2.
Proof.
  unfold noninterferent, secure_p2, pub_equiv.
  intros s1 s2 s1' s2' H H1 H2 x Hx.
  apply xpub_true in Hx. subst.
  invert H1.
  - invert H2.
    + invert H8. invert H9. simpl.
      repeat rewrite t_update_eq.
      invert H7. invert H6.
      repeat rewrite Nat.eqb_eq in *. rewrite H1, H2. auto.
    + invert H8. invert H9. simpl.
      repeat rewrite t_update_eq.
      invert H7.
      rewrite Nat.eqb_eq in H1. auto.
  - invert H2.
    + invert H8. invert H9. simpl.
      repeat rewrite t_update_eq.
      invert H6.
```

```
    rewrite Nat.eqb_eq in H1. auto.
  + invert H8. invert H9. simpl.
    repeat rewrite t_update_eq. auto.
Qed.
```

Still, our type systems will reject programs containing any explicit or implicit flows, this one included. C'est la vie!

8.3 Type system for noninterference of expressions

We will build a type system that prevents all explicit and implicit flows.

But first, let's start with something simpler, a type system for arithmetic expressions: our typing judgement `P ⊢a- a \in l` specifies the label `l` of an arithmetic expression `a` in terms of the labels of the variables read it reads.

In particular, `P ⊢a- a \in public` says that expression `a` only reads public variables, so it computes a public value. `P ⊢a- a \in secret` says that expression `a` reads some secret variable, so it computes a value that may depend on secrets.

Here are some examples:

- For a variable X we just look up its label in P, so `P ⊢a- X \in (P X)`.
- For a constant `n` the label is public, so `P ⊢a n \in public`.
- Given variable `X1` with label `l1` and variable `X2` with label `l2`, what should be the label of `X1 + X2` though?

8.3.1 Combining labels

We need a way to combine the labels of two sub-expressions, which we call the *join* (or least upper bound) of the two labels:

```
Definition join (l1 l2 : label) : label := l1 && l2.
```

Intuitively, if we add up two expressions `e1` labeled `l1` and `e2` labeled `l2`, the result of the addition will be labeled `join l1 l2`, which is public iff `l1` is public *and* `l2` is public.

```
Lemma join_commutative : ∀ {l1 l2},
  join l1 l2 = join l2 l1.
Proof. intros l1 l2. destruct l1; destruct l2; reflexivity. Qed.

Lemma join_public : ∀ {l1 l2},
  join l1 l2 = public → l1 = public ∧ l2 = public.
Proof. apply andb_prop. Qed.

Lemma join_public_l : ∀ {l},
  join public l = l.
```

```
Proof. reflexivity. Qed.

Lemma join_public_r : ∀ {l},
  join l public = l.
Proof. intros l. rewrite join_commutative. reflexivity. Qed.

Lemma join_secret_l : ∀ {l},
  join secret l = secret.
Proof. reflexivity. Qed.

Lemma join_secret_r : ∀ {l},
  join l secret = secret.
Proof. intros l. rewrite join_commutative. reflexivity. Qed.
```

We now define a set of rules for the IFC typing relation for arithmetic expressions `P ⊢a- a \in l`, which we read as follows: "given the public variables P expression a has label l:"

(T_Num) P |-a- n \in public

(T_Id) P |-a- X \in P X
  P |-a- a1 \in l1 P |-a- a2 \in l2

(T_Plus) P |-a- a1+a2 \in join l1 l2
  P |-a- a1 \in l1 P |-a- a2 \in l2

(T_Minus) P |-a- a1-a2 \in join l1 l2
  P |-a- a1 \in l1 P |-a- a2 \in l2

(T_Mult) P |-a- a1*a2 \in join l1 l2

```
Reserved Notation "P '|-a-' a \in l" (at level 40).

Inductive aexp_has_label (P:pub_vars) : aexp → label → Prop :=
  | T_Num : ∀ n,
        P ⊢a- (ANum n) \in public
  | T_Id : ∀ X,
        P ⊢a- (AId X) \in (P X)
  | T_Plus : ∀ a1 l1 a2 l2,
        P ⊢a- a1 \in l1 →
        P ⊢a- a2 \in l2 →
        P ⊢a- <{ a1 + a2 }> \in (join l1 l2)
  | T_Minus : ∀ a1 l1 a2 l2,
        P ⊢a- a1 \in l1 →
        P ⊢a- a2 \in l2 →
        P ⊢a- <{ a1 - a2 }> \in (join l1 l2)
  | T_Mult : ∀ a1 l1 a2 l2,
        P ⊢a- a1 \in l1 →
```

```
      P ⊢a- a2 \in l2 →
      P ⊢a- <{ a1 × a2 }> \in (join l1 l2)

where "P '|-a-' a '\in' l" := (aexp_has_label P a l).
```

8.3.2 Computing labels of arithmetic expressions

Beyond *specifying* when an expression has a certain label as an inductive relation, we can also easily *compute* the label of an expression:

```
Fixpoint label_of_aexp (P:pub_vars) (a:aexp) : label :=
  match a with
  | ANum n ⇒ public
  | AId X ⇒ P X
  | <{ a1 + a2 }>
  | <{ a1 - a2 }>
  | <{ a1 × a2 }> ⇒ join (label_of_aexp P a1) (label_of_aexp P a2)
  end.

Lemma label_of_aexp_sound : ∀ P a,
    P ⊢a- a \in label_of_aexp P a.
Proof. intros P a. induction a; constructor; eauto. Qed.

Lemma label_of_aexp_unique : ∀ P a l,
  P ⊢a- a \in l →
  l = label_of_aexp P a.
Proof.
  intros P a l H. induction H; simpl in *; subst; auto.
Qed.

Theorem noninterferent_aexp : ∀ {P s1 s2 a},
  pub_equiv P s1 s2 →
  P ⊢a- a \in public →
  aeval s1 a = aeval s2 a.
Proof.
  intros P s1 s2 a Heq Ht. remember public as l.
  induction Ht; simpl.
  - reflexivity.
  - apply Heq. apply Heql.
  - destruct (join_public Heql) as [H1 H2].
    rewrite (IHHt1 H1). rewrite (IHHt2 H2). reflexivity.
  - destruct (join_public Heql) as [H1 H2].
    rewrite (IHHt1 H1). rewrite (IHHt2 H2). reflexivity.
  - destruct (join_public Heql) as [H1 H2].
    rewrite (IHHt1 H1). rewrite (IHHt2 H2). reflexivity.
Qed.
```

(T_True) P |-b- true \in public

(T_False) P |-b- false \in public
  P |-a- a1 \in l1 P |-a- a2 \in l2

(T_Eq) P |-b- a1=a2 \in join l1 l2
  ...
  P |-b- b \in l

(T_Not) P |-b- ~b \in l
  P |-b- b1 \in l1 P |-b- b2 \in l2

(T_And) P |-b- b1&&b2 \in join l1 l2

```
Reserved Notation "P '|-b-' b \in l" (at level 40).

Inductive bexp_has_label (P:pub_vars) : bexp → label → Prop :=
  | T_True :
        P ⊢b- <{ true }> \in public
  | T_False :
        P ⊢b- <{ false }> \in public
  | T_Eq : ∀ a1 l1 a2 l2,
        P ⊢a- a1 \in l1 →
        P ⊢a- a2 \in l2 →
        P ⊢b- <{ a1 = a2 }> \in (join l1 l2)
  | T_Neq : ∀ a1 l1 a2 l2,
        P ⊢a- a1 \in l1 →
        P ⊢a- a2 \in l2 →
        P ⊢b- <{ a1 ≠ a2 }> \in (join l1 l2)
  | T_Le : ∀ a1 l1 a2 l2,
        P ⊢a- a1 \in l1 →
        P ⊢a- a2 \in l2 →
        P ⊢b- <{ a1 ≤ a2 }> \in (join l1 l2)
  | T_Gt : ∀ a1 l1 a2 l2,
        P ⊢a- a1 \in l1 →
        P ⊢a- a2 \in l2 →
        P ⊢b- <{ a1 > a2 }> \in (join l1 l2)
  | T_Not : ∀ b l,
        P ⊢b- b \in l →
        P ⊢b- <{ ¬b }> \in l
  | T_And : ∀ b1 l1 b2 l2,
        P ⊢b- b1 \in l1 →
        P ⊢b- b2 \in l2 →
```

```
      P ⊢b- <{ b1 && b2 }> \in (join l1 l2)

where "P '|-b-' b '\in' l" := (bexp_has_label P b l).
```

8.3.3 Computing labels of boolean expressions

```
Fixpoint label_of_bexp (P:pub_vars) (a:bexp) : label :=
  match a with
  | <{ true }> | <{ false }> ⇒ public
  | <{ a1 = a2 }>
  | <{ a1 ≠ a2 }>
  | <{ a1 ≤ a2 }>
  | <{ a1 > a2 }> ⇒ join (label_of_aexp P a1) (label_of_aexp P a2)
  | <{ ¬b }> ⇒ label_of_bexp P b
  | <{ b1 && b2 }> ⇒ join (label_of_bexp P b1) (label_of_bexp P b2)
  end.

Lemma label_of_bexp_sound : ∀ P b,
    P ⊢b- b \in label_of_bexp P b.
Proof.
  intros P b. induction b; constructor;
    eauto using label_of_aexp_sound. Qed.

Lemma label_of_bexp_unique : ∀ P b l,
  P ⊢b- b \in l →
  l = label_of_bexp P b.
Proof.
  intros P a l H. induction H; simpl in *;
  (repeat match goal with
    | [H : _ ⊢a- _ \in _ ⊢ _] ⇒
        apply label_of_aexp_unique in H
    | [Heql : _ = _ ⊢ _] ⇒ rewrite Heql in *
   end); eauto.
Qed.

Theorem noninterferent_bexp : ∀ {P s1 s2 b},
  pub_equiv P s1 s2 →
  P ⊢b- b \in public →
  beval s1 b = beval s2 b.
Proof.
  intros P s1 s2 b Heq Ht. remember public as l.
  induction Ht; simpl; try reflexivity;
    try (destruct (join_public Heql) as [H1 H2];
         rewrite H1 in *; rewrite H2 in *).
  - rewrite (noninterferent_aexp Heq H).
```

```
    rewrite (noninterferent_aexp Heq H0).
    reflexivity.
  - rewrite (noninterferent_aexp Heq H).
    rewrite (noninterferent_aexp Heq H0).
    reflexivity.
  - rewrite (noninterferent_aexp Heq H).
    rewrite (noninterferent_aexp Heq H0).
    reflexivity.
  - rewrite (noninterferent_aexp Heq H).
    rewrite (noninterferent_aexp Heq H0).
    reflexivity.
  - rewrite (IHHt Heql). reflexivity.
  - rewrite (IHHt1 Logic.eq_refl).
    rewrite (IHHt2 Logic.eq_refl). reflexivity.
Qed.
```

8.4 Restrictive type system prohibiting branching on secrets

For commands, we start with a simple type system that doesn't allow any branching on secrets, which is so strong that on its own prevents all implicit flows.

For preventing explicit flows when typing assignments, we need to define when it is okay for information to flow from an expression with label *l1* to a variable with label *l1*.

```
Definition can_flow (l1 l2 : label) : bool := l1 || negb l2.
```

One way to read this is as boolean implication from *l2* to *l1*, so *l1* can flow to *l2* iff *l2* is public implies that *l1* is public as well. In particular, this disallows that the value of secret expressions be assigned to public variables:

```
Lemma cannot_flow_secret_public : can_flow secret public = false.
Proof. reflexivity. Qed.
```

This allows public information to flow everywhere, and secret information to flow to secret variables:

```
Lemma can_flow_public : ∀ l, can_flow public l = true.
Proof. reflexivity. Qed.
Lemma can_flow_secret : can_flow secret secret = true.
Proof. reflexivity. Qed.

Lemma can_flow_refl : ∀ l,
  can_flow l l = true.
Proof. intros []; reflexivity. Qed.

Lemma can_flow_trans : ∀ l1 l2 l3,
```

```
  can_flow l1 l2 = true →
  can_flow l2 l3 = true →
  can_flow l1 l3 = true.
Proof. intros l1 l2 l3 H12 H23.
  destruct l1; destruct l2; simpl in *; auto. discriminate H12. Qed.

Lemma can_flow_join_1 : ∀ l1 l2 l,
  can_flow (join l1 l2) l = true →
  can_flow l1 l = true.
Proof. intros l1 l2 l. destruct l1; [reflexivity | auto ]. Qed.

Lemma can_flow_join_2 : ∀ l1 l2 l,
  can_flow (join l1 l2) l = true →
  can_flow l2 l = true.
Proof. intros l1 l2 l. destruct l1; auto. destruct l2; auto. Qed.

Lemma can_flow_join_l : ∀ l1 l2 l,
  can_flow l1 l = true →
  can_flow l2 l = true →
  can_flow (join l1 l2) l = true.
Proof. intros l1 l2 l H1 H2. destruct l1; simpl in *; auto. Qed.

Lemma can_flow_join_r1 : ∀ l l1 l2,
  can_flow l l1 = true →
  can_flow l (join l1 l2) = true.
Proof. intros l l1 l2 H. destruct l; destruct l1; simpl in *; auto.
       discriminate H. Qed.

Lemma can_flow_join_r2 : ∀ l l1 l2,
  can_flow l l2 = true →
  can_flow l (join l1 l2) = true.
Proof. intros l l1 l2 H. destruct l; destruct l1; simpl in *; auto. Qed.
```

For commands we use the previous relations to define a **cf_well_typed** relation inductively using the following rules:

(CFWT_Skip) P |-cf- skip
  P |-a- a \in l can_flow l (P X) = true

(CFWT_Asgn) P |-cf- X := a
  P |-cf- c1 P |-cf- c2

(CFWT_Seq) P |-cf- c1;c2
  P |-b- b \in public P |-cf- c1 P |-cf- c2

(CFWT_If) P |-cf- if b then c1 else c2
  P |-b- b \in public P |-cf- c

(CFWT_While) P |-cf- while b then c end

Intuitively, explicit flows are prevented by the can_flow requirement in the assignment rule and implicit flows are prevented by the requirement that the boolean condition of `if` and *while* has to be a public expression.

```
Reserved Notation "P '|-cf-' c" (at level 40).

Inductive cf_well_typed (P:pub_vars) : com → Prop :=
  | CFWT_Com :
      P ⊢cf- <{ skip }>
  | CFWT_Asgn : ∀ X a l,
      P ⊢a- a \in l →
      can_flow l (P X) = true →
      P ⊢cf- <{ X := a }>
  | CFWT_Seq : ∀ c1 c2,
      P ⊢cf- c1 →
      P ⊢cf- c2 →
      P ⊢cf- <{ c1 ; c2 }>
  | CFWT_If : ∀ b c1 c2,
      P ⊢b- b \in public →
      P ⊢cf- c1 →
      P ⊢cf- c2 →
      P ⊢cf- <{ if b then c1 else c2 end }>
  | CFWT_While : ∀ b c1,
      P ⊢b- b \in public →
      P ⊢cf- c1 →
      P ⊢cf- <{ while b do c1 end }>

where "P '|-cf-' c" := (cf_well_typed P c).
```

8.4.1 Typechecker for cf_well_typed

```
Fixpoint cf_typechecker (P:pub_vars) (c:com) : bool :=
  match c with
  | <{ skip }> ⇒ true
  | <{ X := a }> ⇒ can_flow (label_of_aexp P a) (P X)
  | <{ c1 ; c2 }> ⇒ cf_typechecker P c1 && cf_typechecker P c2
  | <{ if b then c1 else c2 end }> ⇒
      Bool.eqb (label_of_bexp P b) public &&
      cf_typechecker P c1 && cf_typechecker P c2
  | <{ while b do c1 end }> ⇒
      Bool.eqb (label_of_bexp P b) public && cf_typechecker P c1
  end.
```

This typechecker is sound and complete with respect to the **cf_well_typed** relation.

```
Lemma cf_typechecker_sound : ∀ P c,
  cf_typechecker P c = true →
  P ⊢cf- c.
Proof.
  intros P c. induction c; simpl in *; econstructor;
    try rewrite andb_true_iff in *; try tauto;
    eauto using label_of_aexp_sound, label_of_bexp_sound.
  - destruct H as [H1 H2]. rewrite andb_true_iff in H1; try tauto.
    destruct H1 as [H11 H12]. apply Bool.eqb_prop in H11.
    rewrite ← H11. apply label_of_bexp_sound.
  - destruct H as [H1 H2]. rewrite andb_true_iff in H1; tauto.
  - destruct H as [H1 H2]. apply Bool.eqb_prop in H1.
    rewrite ← H1. apply label_of_bexp_sound.
Qed.

Lemma cf_typechecker_complete : ∀ P c,
  cf_typechecker P c = false →
  ¬P ⊢cf- c.
Proof.
  intros P c H Hc. induction Hc; simpl in *;
    try rewrite andb_false_iff in *;
    try tauto; try congruence.
  - apply label_of_aexp_unique in H0.
    rewrite H0 in *. congruence.
  - destruct H; eauto. rewrite andb_false_iff in H.
    destruct H; eauto. rewrite eqb_false_iff in H.
    apply label_of_bexp_unique in H0. congruence.
  - destruct H; eauto. rewrite eqb_false_iff in H.
    apply label_of_bexp_unique in H0. congruence.
Qed.
```

It is worth noting that, while our type-checker is sound and complete wrt the **cf_well_typed** relation, this relation is only a sound overapproximation of noninterference (proved below), but not complete. So the type-checker is also not complete wrt noninterference, but is still provides an efficient way of proving it. For a start, let's use the type-checker to prove or disprove the **cf_well_typed** relation for concrete programs by computation:

8.4.2 Secure program that is cf_well_typed:

```
Example cf_wt_secure_com :
  xpub ⊢cf- <{ X := X+1;
               Y := X+Y×2
            }>.
```

```
Proof. apply cf_typechecker_sound. reflexivity. Qed.
```

8.4.3 Explicit flow prevented by cf_well_typed:

```
Example not_cf_wt_insecure_com1 :
  ¬ xpub ⊢cf- <{ X := Y+1;
                 Y := X+Y×2
               }>.
Proof. apply cf_typechecker_complete. reflexivity. Qed.
```

8.4.4 Implicit flow prevented by cf_well_typed:

```
Example not_cf_wt_insecure_com2 :
  ¬ xpub ⊢cf- <{ if Y=0
                 then Y := 42
                 else X := X+1
                 end }>.
Proof. apply cf_typechecker_complete. reflexivity. Qed.
```

8.4.5 Noninterference enforced by cf_well_typed

We show that all cf_well_typed commands are noninterferent.

```
Theorem cf_well_typed_noninterferent : ∀ P c,
  P ⊢cf- c →
  noninterferent P c.
Proof.
  intros P c Hwt s1 s2 s1' s2' Heq Heval1 Heval2.
  generalize dependent s2'. generalize dependent s2.
  induction Heval1; intros s2 Heq s2' Heval2;
    inversion Heval2; inversion Hwt; subst.
  - assumption.
  - intros y Hy. destruct (String.eqb_spec x y) as [Hxy | Hxy].
    + rewrite Hxy. do 2 rewrite t_update_eq.
      unfold can_flow in H8. apply orb_prop in H8. destruct H8 as [Hl | Hx].
      × rewrite Hl in *. apply (noninterferent_aexp Heq H7).
      × subst. rewrite Hy in Hx. discriminate Hx.
    + do 2 rewrite (t_update_neq _ _ _ _ _ Hxy).
      apply Heq. apply Hy.
  - eapply IHHeval1_2; try eassumption. eapply IHHeval1_1; eassumption.
  - eapply IHHeval1; eassumption.
  - rewrite (noninterferent_bexp Heq H10) in H.
    rewrite H in H5. discriminate H5.
```

```
  - rewrite (noninterferent_bexp Heq H10) in H.
    rewrite H in H5. discriminate H5.
  - eapply IHHeval1; eassumption.
  - assumption.
  - rewrite (noninterferent_bexp Heq H9) in H.
    rewrite H in H2. discriminate H2.
  - rewrite (noninterferent_bexp Heq H7) in H.
    rewrite H in H4. discriminate H4.
  - eapply IHHeval1_2; try eassumption. eapply IHHeval1_1; eassumption.
Qed.
```

Remember the definition of noninterferent is as follows:

forall s1 s2 s1' s2', pub_equiv P s1 s2 -> s1 = *c* => s1' -> s2 = *c* => s2' -> pub_equiv P s1' s2'.

The main intuition is that the two executions will proceed "in lockstep", because all the branch conditions are enforced to be public, so they will execute to the same Boolean in both executions.

The proof is by induction on s1 =[*c*]=> s1' and inversion on s2 =[*c*]=> *s2'* and *P* ⊢*cf*- *c*. Here is a sketch of the two most interesting cases:

- In the conditional case we have that *c* is `if b then c1 else c2`, *P* ⊢*cf*- *c1*, *P* ⊢*cf*- *c2*, and *P* ⊢*b*- *b* \in public. Given this last fact we can apply noninterference of boolean expressions to show that beval *st1* *b* = beval *st2* *b*. If they are both true, we use the induction hypothesis for *c1*, and if they are both false we use the induction hypothesis for *c2* to conclude.

- In the assignment case we have that *c* is X := *a*, *P* ⊢*a*- *a* \in *l*, and can_flow *l* (*P* X) = true, which expands out to *l* == public ∨ *P* X == secret.

  If *l* == public then by noninterference of arithmetic expressions then aeval *st1* *a* = aeval s2 *a*, so we are assigning the same value to X, which leads to public equivalent final states (since the initial states were public equivalent).

  If *P* X == secret then the value of X doesn't matter for determining whether the final states are pub_equiv.

8.4.6 cf_well_typed too strong for noninterference

While we have just proved that **cf_well_typed** implies noninterference, this type system is too restrictive for enforcing just noninterference. For instance, the following program is rejected by the type system just because it branches on a secret:

Exercise: 1 star, standard (not_cf_wt_noninterferent_com) Use the type-checker to prove that the following program is not **cf_well_typed** (Hint: This can be proved very easily, if stuck see examples above):

```
Example not_cf_wt_noninterferent_com :
  ¬ xpub ⊢cf- <{ if Y=0
                 then Z := 0
                 else skip
                 end }>.
Proof.
  Admitted.
```

□

Yet this program contains no explicit flows and no implicit flows (since the assigned variable Z is secret), so it is intuitively noninterferent, and with a bit more work we can prove this formally:

```
Example not_cf_wt_noninterferent_com_is_noninterferent:
  noninterferent xpub <{ if Y=0
                         then Z := 0
                         else skip
                         end }>.
Proof.
  unfold noninterferent.
  intros s1 s2 s1' s2' H red1 red2.
  inversion red1; inversion red2; subst; clear red1 red2;
  inversion H6; subst; clear H6; inversion H13; subst; clear H13; intros x Px;
  destruct (String.eqb_spec x Z); subst; try discriminate.
  - rewrite !t_update_neq; auto.
  - rewrite !t_update_neq; auto.
  - rewrite !t_update_neq; auto.
  - eapply H; eauto.
Qed.
```

We will later show that **cf_well_typed** enforces not just noninterference, but also a security notion called Control Flow security, which prevents some side-channel attacks and which also serves as the base for cryptographic constant-time.

8.5 IFC type system allowing branching on secrets

Let's now investigate a more permissive type system for noninterference in which we do allow branching on secrets *Volpano et al* 1996 (in Bib.v).

Now to prevent implicit flows we need to track whether we have branched on secrets. We do this with a *program counter* (*pc*) label, which records the labels of the branches we have taken at the current point in the execution (joined together).

(NIWT_Skip) P ;; pc |-ni- skip
  P |-a- a \in l can_flow (join pc l) (P X) = true

(NIWT_Asgn) P ;; pc |-ni- X := a
  P ;; pc |-ni- c1 P ;; pc |-ni- c2

(NIWT_Seq) P ;; pc |-ni- c1;c2
  P |-b- b \in l P ;; join pc l |-ni- c1 P ;; join pc l |-ni- c2

(NIWT_If) P ;; pc |-ni- if b then c1 else c2
  P |-b- b \in l P ;; join pc l |-ni- c

(NIWT_While) P ;; pc |-ni- while b then c end

```
Reserved Notation "P ';;' pc '|-ni-' c" (at level 40).

Inductive ni_well_typed (P:pub_vars) : label → com → Prop :=
  | NIWT_Com : ∀ pc,
       P ;; pc ⊢ni- <{ skip }>
  | NIWT_Asgn : ∀ pc X a l,
       P ⊢a- a \in l →
       can_flow (join pc l) (P X) = true →
       P ;; pc ⊢ni- <{ X := a }>
  | NIWT_Seq : ∀ pc c1 c2,
       P ;; pc ⊢ni- c1 →
       P ;; pc ⊢ni- c2 →
       P ;; pc ⊢ni- <{ c1 ; c2 }>
  | NIWT_If : ∀ pc b l c1 c2,
       P ⊢b- b \in l →
       P ;; (join pc l) ⊢ni- c1 →
       P ;; (join pc l) ⊢ni- c2 →
       P ;; pc ⊢ni- <{ if b then c1 else c2 end }>
  | NIWT_While : ∀ pc b l c1,
       P ⊢b- b \in l →
       P ;; (join pc l) ⊢ni- c1 →
       P ;; pc ⊢ni- <{ while b do c1 end }>

where "P ';;' pc '|-ni-' c" := (ni_well_typed P pc c).
```

We now allow branching on arbitrary boolean expressions in `if` and *while*, but join the label of the branch expression to the *pc*. Then in the assignment rule we require that also the *pc* label flows to the label of the assigned variable, in order to still prevent implicit flows.

8.5.1 Typechecker for ni_well_typed relation.

```
Fixpoint ni_typechecker (P:pub_vars) (pc:label) (c:com) : bool :=
  match c with
  | <{ skip }> ⇒ true
  | <{ X := a }> ⇒ can_flow (join pc (label_of_aexp P a)) (P X)
  | <{ c1 ; c2 }> ⇒ ni_typechecker P pc c1 && ni_typechecker P pc c2
  | <{ if b then c1 else c2 end }> ⇒
      ni_typechecker P (join pc (label_of_bexp P b)) c1 &&
      ni_typechecker P (join pc (label_of_bexp P b)) c2
  | <{ while b do c1 end }> ⇒
      ni_typechecker P (join pc (label_of_bexp P b)) c1
  end.

Lemma ni_typechecker_sound : ∀ P pc c,
  ni_typechecker P pc c = true →
  P ;; pc ⊢ni- c.
Proof.
  intros P pc c. generalize dependent pc.
  induction c; intros pc H; simpl in *; econstructor;
    try rewrite andb_true_iff in *;
    try destruct H; try tauto;
    eauto using label_of_aexp_sound, label_of_bexp_sound.
Qed.

Lemma ni_typechecker_complete : ∀ P pc c,
  ni_typechecker P pc c = false →
  ¬ P ;; pc ⊢ni- c.
Proof.
  intros P pc c H Hc. induction Hc; simpl in *;
    try rewrite andb_false_iff in *; try tauto; try congruence.
  - apply label_of_aexp_unique in H0.
    rewrite H0 in *. congruence.
  - destruct H; apply label_of_bexp_unique in H0; subst; eauto.
  - destruct H; apply label_of_bexp_unique in H0; subst; eauto.
Qed.
```

With this more permissive type system we can accept more noninterferent programs that were rejected by cf_well_typed.

```
Example ni_noninterferent_com :
  xpub ;; public ⊢ni-
    <{ if Y=0
       then Z := 0
       else skip
       end }>.
```

```
Proof. apply ni_typechecker_sound. reflexivity. Qed.
```

And we still prevent implicit flows:

```
Example not_ni_insecure_com2 :
  ¬ xpub ;; public ⊢ni-
    <{ if Y=0
       then Y := 42
       else X := X+1
       end }>.
Proof. apply ni_typechecker_complete. reflexivity. Qed.

Lemma weaken_pc : ∀ {P pc1 pc2 c},
  P ;; pc1 ⊢ni- c →
  can_flow pc2 pc1 = true→
  P ;; pc2 ⊢ni- c.
Proof.
  intros P pc1 pc2 c H. generalize dependent pc2.
  induction H; subst; intros pc2 Hcan_flow.
  - constructor.
  - econstructor; try eassumption. apply can_flow_join_l.
    + apply can_flow_join_1 in H0. eapply can_flow_trans; eassumption.
    + apply can_flow_join_2 in H0. assumption.
  - constructor; auto.
  - econstructor; try eassumption.
    + apply IHni_well_typed1. apply can_flow_join_l.
      × apply can_flow_join_r1. assumption.
      × apply can_flow_join_r2. apply can_flow_refl.
    + apply IHni_well_typed2. apply can_flow_join_l.
      × apply can_flow_join_r1. assumption.
      × apply can_flow_join_r2. apply can_flow_refl.
  - econstructor; try eassumption. apply IHni_well_typed. apply can_flow_join_l.
      × apply can_flow_join_r1. assumption.
      × apply can_flow_join_r2. apply can_flow_refl.
Qed.
```

8.5.2 Dealing with unsynchronized executions running different code

The different_code corollary below is crucial for proving that the type system above still enforces noninterference even if it allows branching on secrets, and its proof follows easily from the following basic lemma:

```
Lemma secret_run : ∀ {P c s s'},
  P ;; secret ⊢ni- c →
  s =[ c ]=> s' →
```

```
  pub_equiv P s s'.
Proof.
  intros P c s s' Hwt Heval. induction Heval; inversion Hwt;
    subst; eauto using pub_equiv_trans, pub_equiv_refl.
  -
    intros y Hy. destruct (String.eqb_spec x y) as [Hxy | Hxy].
    +
      subst. rewrite join_secret_l in H4. rewrite Hy in H4. discriminate H4.
    + rewrite t_update_neq; auto.
Qed.

Corollary different_code : ∀ P c1 c2 s1 s2 s1' s2',
  P ;; secret ⊢ni- c1 →
  P ;; secret ⊢ni- c2 →
  pub_equiv P s1 s2 →
  s1 =[ c1 ]=> s1' →
  s2 =[ c2 ]=> s2' →
  pub_equiv P s1' s2'.
Proof.
  intros P c1 c2 s1 s2 s1' s2' Hwt1 Hwt2 Hequiv Heval1 Heval2.
  eapply secret_run in Hwt1; [| eassumption].
  eapply secret_run in Hwt2; [| eassumption].
  apply pub_equiv_sym in Hwt1.
  eapply pub_equiv_trans; try eassumption.
  eapply pub_equiv_trans; eassumption.
Qed.
```

8.5.3 We show that ni_well_typed commands are noninterferent.

```
Theorem ni_well_typed_noninterferent : ∀ P c,
  P ;; public ⊢ni- c →
  noninterferent P c.
Proof.
  intros P c Hwt s1 s2 s1' s2' Heq Heval1 Heval2.
  generalize dependent s2'. generalize dependent s2.
  induction Heval1; intros s2 Heq s2' Heval2;
    inversion Heval2; inversion Hwt; subst; try rewrite join_public_l in *.
  - assumption.
  - intros y Hy. destruct (String.eqb_spec x y) as [Hxy | Hxy].
    + rewrite Hxy. do 2 rewrite t_update_eq.
      unfold can_flow in H9.
      apply orb_prop in H9. destruct H9 as [Hl | Hx].
      × rewrite Hl in *. apply (noninterferent_aexp Heq H8).
```

```
      × subst. rewrite Hy in Hx. discriminate Hx.
    + do 2 rewrite (t_update_neq _ _ _ _ _ Hxy).
      apply Heq. apply Hy.
  - eapply IHHeval1_2; try eassumption. eapply IHHeval1_1; eassumption.
  -
    eapply IHHeval1; try eassumption.
    eapply weaken_pc; try eassumption. apply can_flow_public.
  - destruct l.
    + rewrite (noninterferent_bexp Heq H11) in H.
      rewrite H in H5. discriminate H5.
    + eapply different_code with (c1:=c1) (c2:=c2); eassumption.
  - destruct l.
    + rewrite (noninterferent_bexp Heq H11) in H.
      rewrite H in H5. discriminate H5.
    + eapply different_code with (c1:=c2) (c2:=c1); eassumption.
  -
    eapply IHHeval1; try eassumption.
    eapply weaken_pc; try eassumption. apply can_flow_public.
  - assumption.
  - destruct l.
    + rewrite (noninterferent_bexp Heq H10) in H.
      rewrite H in H2. discriminate H2.
    + eapply different_code with (c1:=<{skip}>) (c2:=<{c;while b do c end}>);
        repeat (try eassumption; try econstructor).
  - destruct l.
    + rewrite (noninterferent_bexp Heq H8) in H.
      rewrite H in H4. discriminate H4.
    + eapply different_code with (c1:=<{c;while b do c end}>) (c2:=<{skip}>);
        repeat (try eassumption; try econstructor).
  -
    eapply IHHeval1_2; try eassumption. eapply IHHeval1_1; try eassumption.
    eapply weaken_pc; try eassumption. apply can_flow_public.
Qed.
```

The noninterference proof is still relatively simple, since the cases in which we take different branches based on secret information are all handled by the different_code lemma.

Another key ingredient for having a simple noninterference proof is working with a big-step semantics for Imp.

8.6 Type system for termination-sensitive noninterference

The noninterference notion we used above was "termination insensitive". If we prevent loop conditions depending on secrets we can actually enforce termination-sensitive noninterference (TSNI), which we defined in Noninterference as follows:

```
Definition tsni P c :=
  ∀ s1 s2 s1',
  s1 =[ c ]=> s1' →
  pub_equiv P s1 s2 →
  (∃ s2', s2 =[ c ]=> s2' ∧ pub_equiv P s1' s2').
```

We could prove that **cf_well_typed** enforces TSNI, but that typing relation is too restrictive, since for TSNI we can allow if-then-else conditions to depend on secrets. So we define another type system that only prevents *loop* conditions from depending on secrets *Volpano* **and** *Smith* 1997 (in Bib.v).

8.6.1 We just need to update the while rule of ni_well_typed:

Old rule for noninterference:
P |-b- b \in l P ;; join pc l |-ni- c

(NIWT_While) P ;; pc |-ni- while b then c end
New rule for termination-sensitive noninterference:
P |-b- b \in public P ;; public |-ts- c

(TSWT_While) P ;; public |-ts- while b then c end
Beyond requiring the label of *b* to be public, this rule also requires that once one branches on secrets with if-then-else (i.e. pc=secret) no while loops are allowed.

```
Reserved Notation "P ';;' pc '|-ts-' c" (at level 40).

Inductive ts_well_typed (P:pub_vars) : label → com → Prop :=
  | TSWT_Com : ∀ pc,
      P ;; pc ⊢ts- <{ skip }>
  | TSWT_Asgn : ∀ pc X a l,
      P ⊢a- a \in l →
      can_flow (join pc l) (P X) = true →
      P ;; pc ⊢ts- <{ X := a }>
  | TSWT_Seq : ∀ pc c1 c2,
      P ;; pc ⊢ts- c1 →
      P ;; pc ⊢ts- c2 →
      P ;; pc ⊢ts- <{ c1 ; c2 }>
  | TSWT_If : ∀ pc b l c1 c2,
```

```
      P ⊢b- b \in l →
      P ;; (join pc l) ⊢ts- c1 →
      P ;; (join pc l) ⊢ts- c2 →
      P ;; pc ⊢ts- <{ if b then c1 else c2 end }>
  | TSWT_While : ∀ b c1,
      P ⊢b- b \in public →
      P ;; public ⊢ts- c1 →
      P ;; public ⊢ts- <{ while b do c1 end }>

where "P ';;' pc '|-ts-' c" := (ts_well_typed P pc c).
```

8.6.2 TSNI Type-Checker

In the following exercises you will write a type-checker for the TSNI type system above and prove your type-checker sound and complete.

Exercise: 2 stars, standard (ts_typechecker)

```
Fixpoint ts_typechecker (P:pub_vars) (pc:label) (c:com) : bool :=
  match c with
  | <{ skip }> ⇒ true
  | <{ X := a }> ⇒ can_flow (join pc (label_of_aexp P a)) (P X)
  | <{ c1 ; c2 }> ⇒ ts_typechecker P pc c1 && ts_typechecker P pc c2
  | <{ if b then c1 else c2 end }> ⇒
      ts_typechecker P (join pc (label_of_bexp P b)) c1 &&
      ts_typechecker P (join pc (label_of_bexp P b)) c2

  | _ ⇒ false
  end.
```

□

Exercise: 2 stars, standard (ts_typechecker_sound)

```
Lemma ts_typechecker_sound :
∀ P pc c,
  ts_typechecker P pc c = true →
  P ;; pc ⊢ts- c.
Proof.
  Admitted.
```

□

Exercise: 2 stars, standard (ts_typechecker_complete)

```
Lemma ts_typechecker_complete
: ∀ P pc c,
  ts_typechecker P pc c = false →
  ¬ P ;; pc ⊢ts- c.
```

```
Proof.
    Admitted.
    □
```

With this termination-sensitive type-checker, we reject programs where the termination behavior itself leaks secret information. The following example shows a command that either runs forever or terminates depending on the value of a secret variable (Y).

```
Definition termination_leak : com :=
    <{ if Y=0
       then (while true do skip end)
       else skip
       end }>.
```

Our previous termination-insensitive type system accepts this program:

```
Example ni_termination_leak :
  xpub ;; public ⊢ni- termination_leak.
Proof. apply ni_typechecker_sound. reflexivity. Qed.
```

But our new termination-sensitive type system rejects it, and you can use your new type-checker to prove it:

Exercise: 1 star, standard (not_ts_non_termination_com) `Example not_ts_non_termination_com :`

```
  ¬ xpub ;; public ⊢ts- termination_leak.
Proof.
    Admitted.
    □
```

8.6.3 We prove that ts_well_typed enforces TSNI.

For this we show that **ts_well_typed** implies **ni_well_typed**, so by our previous theorem also (termination-insensitive) *noninterference*.

Then we show that P;; secret $\vdash ts\text{-}\ c$ implies termination.

We use this to show that **ts_well_typed** implies equitermination, which together with noninterference implies termination-sensitive noninterference.

```
Theorem ts_well_typed_ni_well_typed : ∀ P c pc,
  P ;; pc ⊢ts- c →
  P ;; pc ⊢ni- c.
Proof.
  intros P c pc H. induction H; econstructor; eassumption.
Qed.

Theorem ts_well_typed_noninterferent : ∀ P c,
  P ;; public ⊢ts- c →
  noninterferent P c.
```

```
Proof.
  intros P c H. apply ni_well_typed_noninterferent.
  apply ts_well_typed_ni_well_typed. apply H.
Qed.

Lemma ts_secret_run_terminating : ∀ {P c s},
  P;; secret ⊢ts- c →
  ∃ s', s =[ c ]=> s'.
Proof.
  intros P c s Hwt. remember secret as l.
  generalize dependent s. induction Hwt; intro s.
  - eexists. econstructor.
  - eexists. econstructor. reflexivity.
  - destruct (IHHwt1 Heql s) as [s' IH1].
    destruct (IHHwt2 Heql s') as [s''IH2]. eexists. econstructor; eassumption.
  - rewrite Heql in *. rewrite join_secret_l in *.
    destruct (IHHwt1 Logic.eq_refl s) as [s1 IH1].
    destruct (IHHwt2 Logic.eq_refl s) as [s2 IH2].
    destruct (beval s b) eqn:Heq; eexists; econstructor; eassumption.
  - discriminate Heql.
Qed.

Theorem ts_well_typed_equitermination : ∀ {P c s1 s2 s1'},
  P;; public ⊢ts- c →
  s1 =[ c ]=> s1' →
  pub_equiv P s1 s2 →
  ∃ s2', s2 =[ c ]=> s2'.
Proof.
  intros P C s1 s2 s1' Hwt Heval. generalize dependent s2.
  induction Heval; intros s2 Heq; inversion Hwt; subst.
  - eexists. constructor.
  - eexists. econstructor. reflexivity.
  - destruct (IHHeval1 H2 _ Heq) as [s2' IH1].
    assert (Heq' : pub_equiv P st' s2').
    { eapply ts_well_typed_noninterferent;
        [ | eassumption | eassumption | eassumption]. assumption. }
    destruct (IHHeval2 H3 _ Heq') as [s2'' IH2].
    eexists. econstructor; eassumption.
  - rewrite join_public_l in *. destruct l.
    + destruct (IHHeval H5 _ Heq) as [s2' IH1].
      eexists. apply E_IfTrue; [ | eassumption ].
      × eapply noninterferent_bexp in Heq; [ | eassumption ]. congruence.
    + eapply ts_secret_run_terminating in H5. destruct H5 as [s1' H5].
      eapply ts_secret_run_terminating in H6. destruct H6 as [s2' H6].
```

```
      destruct (beval s2 b) eqn:Heq2; eexists; econstructor; eassumption.
  - rewrite join_public_l in *. destruct l.
    + destruct (IHHeval H6 _ Heq) as [s2' IH1].
      eexists. apply E_IfFalse; [ | eassumption ].
      × eapply noninterferent_bexp in Heq; [ | eassumption ]. congruence.
    + eapply ts_secret_run_terminating in H5. destruct H5 as [s1' H5].
      eapply ts_secret_run_terminating in H6. destruct H6 as [s2' H6].
      destruct (beval s2 b) eqn:Heq2; eexists; econstructor; eassumption.
  - eapply noninterferent_bexp in Heq; [ | eassumption ].
    eexists. apply E_WhileFalse. congruence.
  - destruct (IHHeval1 H3 _ Heq) as [s2' IH1].
    assert (Heq' : pub_equiv P st' s2').
    { eapply ts_well_typed_noninterferent;
        [ | eassumption | eassumption | eassumption]. assumption. }
    destruct (IHHeval2 Hwt _ Heq') as [s2'' IH2].
    eapply noninterferent_bexp in Heq; [ | eassumption ].
    eexists. eapply E_WhileTrue; try congruence; eassumption.
Qed.

Corollary ts_well_typed_tsni : ∀ P c,
  P ;; public ⊢ts- c →
  tsni P c.
Proof.
  intros P c Hwt s1 s2 s1' Heval1 Heq.
  destruct (ts_well_typed_equitermination Hwt Heval1 Heq) as [s2' Heval2].
  ∃ s2'. split; [assumption| ].
  eapply ts_well_typed_noninterferent; eassumption.
Qed.
```

8.7 Control Flow security

Especially for cryptographic code one is also worried about side-channel attacks, in which secrets are for instance leaked via the execution time of the program. For instance, most processors have instruction caches, which make executing cached instructions faster than non-cached ones.

To prevent such attacks, cryptographic code is normally written without branching on any secrets. To formalize this we introduce a security notion called *Control Flow (CF) security* (sometimes called PC security *Molnar et al* 2005 (in Bib.v)), which considers the program's branching visible to the attacker. More precisely, we instrument the operational semantics of Imp to also record the control-flow decisions of the program.

```
Definition branches := list bool.
```

8.7.1 Instrumented semantics with branches

(CFE_Skip) st = *skip* => st, □
  aeval st a = n

(CFE_Asgn) st = *x* := *a* => (x !-> n ; st), □
  st = *c1* => st', bs1 st' = *c2* => st'', bs2

(CFE_Seq) st = *c1*;*c2* => st'', (bs1++bs2)
  beval st b = true st = *c1* => st', bs1

(CFE_IfTrue) st = `if` *b* `then` *c1* `else` *c2* `end` => st', true::bs1
  beval st b = false st = *c2* => st', bs2

(CFE_IfFalse) st = `if` *b* `then` *c1* `else` *c2* `end` => st', false::bs2
  st = `if` *b* `then` *c*; *while b* `do` *c* `end` `else` *skip* `end` => st', os

---

(CFE_While) st = *while b* `do` *c* `end` => st', os

```
Reserved Notation
         "st '=[' c ']=>' st' , bs"
         (at level 40, c custom com at level 99,
          st constr, st' constr at next level).

Inductive cf_ceval : com → state → state → branches → Prop :=
  | CFE_Skip : ∀ st,
      st =[ skip ]=> st, []
  | CFE_Asgn : ∀ st a n x,
      aeval st a = n →
      st =[ x := a ]=> (x !-> n ; st), []
  | CFE_Seq : ∀ c1 c2 st st' st'' bs1 bs2,
      st =[ c1 ]=> st', bs1 →
      st' =[ c2 ]=> st'', bs2 →
      st =[ c1 ; c2 ]=> st'', (bs1++bs2)
  | CFE_IfTrue : ∀ st st' b c1 c2 bs1,
      beval st b = true →
      st =[ c1 ]=> st', bs1 →
      st =[ if b then c1 else c2 end]=> st', (true::bs1)
  | CFE_IfFalse : ∀ st st' b c1 c2 bs1,
      beval st b = false →
      st =[ c2 ]=> st', bs1 →
      st =[ if b then c1 else c2 end]=> st', (false::bs1)
  | CFE_While : ∀ b st st' os c,
```

```
      st =[ if b then c; while b do c end else skip end ]=> st', os →
      st =[ while b do c end ]=> st', os

  where "st =[ c ]=> st' , bs" := (cf_ceval c st st' bs).

Lemma cf_ceval_ceval : ∀ c st st' bs,
  st =[ c ]=> st', bs →
  st =[ c ]=> st'.
Proof.
  intros c st st' bs H. induction H; try (econstructor; eassumption).
  -
    inversion IHcf_ceval.
    + inversion H6. subst. eapply E_WhileTrue; eauto.
    + subst. invert H6. eapply E_WhileFalse; eauto.
Qed.
```

8.7.2 Control Flow security definition

Using the instrumented semantics we define Control Flow (CF) security:

```
Definition cf_secure P c := ∀ s1 s2 s1' s2' bs1 bs2,
  pub_equiv P s1 s2 →
  s1 =[ c ]=> s1', bs1 →
  s2 =[ c ]=> s2', bs2 →
  bs1 = bs2.
```

CF security is mostly orthogonal to noninterference and instead of relating the final states it requires the branches of the program to be independent of secrets.

Our restrictive **cf_well_typed** relation enforces both noninterference (as we already proved at the beginning of the chapter) and CF security:

```
Theorem cf_well_typed_cf_secure : ∀ P c,
  P ⊢cf- c →
  cf_secure P c.
Proof.
  intros P c Hwt s1 s2 s1' s2' bs1 bs2 Heq Heval1 Heval2.
  generalize dependent s2'. generalize dependent s2.
  generalize dependent bs2.
  induction Heval1; intros bs2' s2 Heq s2' Heval2;
    inversion Heval2; inversion Hwt; subst.
  - reflexivity.
  - reflexivity.
  - destruct (IHHeval1_1 H8 bs0 s2 Heq st'0 H1).
    assert (Heq': pub_equiv P st' st'0).
    { eapply cf_ceval_ceval in Heval1_1.
```

```
      eapply cf_ceval_ceval in H1.
      eapply cf_well_typed_noninterferent with (c:=c1); eauto. }
    erewrite IHHeval1_2; eauto.
  - f_equal. eapply IHHeval1; try eassumption.
  - rewrite (noninterferent_bexp Heq H11) in H.
    rewrite H in H6. discriminate H6.
  - rewrite (noninterferent_bexp Heq H11) in H.
    rewrite H in H6. discriminate H6.
  - f_equal. eapply IHHeval1; eassumption.
  - eapply IHHeval1; try eassumption. repeat constructor; eassumption.
Qed.
```

The proof does rely on **cf_well_typed** implying noninterference for the sequencing case (and indirectly for the while case too, since in our semantics of while evaluates to a sequence).

Control flow security forms the foundation on which we will define cryptographic constant time in the SpecCT chapter.

Exercise: 4 stars, standard (cf_well_typed_ts_cf_secure) We can also define a stronger, termination-sensitive version of control flow security:

```
Definition ts_cf_secure P c := ∀ s1 s2 s1' bs1,
  pub_equiv P s1 s2 →
  s1 =[ c ]=> s1', bs1 →
  ∃ s2', s2 =[ c ]=> s2', bs1.
```

In this exercise, you have to prove that **cf_well_typed** also implies ts_cf_secure. The while case should actually be quite easy, if you exploit how we reduced evaluation of while to sequencing and `if-then-else` in rule CFE_While above.

```
Theorem cf_well_typed_ts_cf_secure : ∀ P c,
  P ⊢cf- c →
  ts_cf_secure P c.
Proof.
  Admitted.
```

□

8.8 Exercise: Adding public outputs

Exercise: 5 stars, standard (public_outputs) Imp, the simple imperative language we considered so far, doesn't have an output operation. In practice, however, programs often need to produce publicly-observable outputs. In this exercise, we extend our language with an output command and introduce an additional security property to be enforced for such programs.

```
Module OUTPUT.
```

```
Definition outputs := list nat.

Inductive com : Type :=
  | Skip
  | Asgn (x : string) (a : aexp)
  | Seq (c1 c2 : com)
  | If (b : bexp) (c1 c2 : com)
  | While (b : bexp) (c : com)
  | Output (a: aexp).

Open Scope com_scope.

Notation "'skip'" :=
  Skip (in custom com at level 0) : com_scope.
Notation "x := y" :=
  (Asgn x y)
    (in custom com at level 0, x constr at level 0,
      y custom com at level 85, no associativity) : com_scope.
Notation "x ; y" :=
  (Seq x y)
    (in custom com at level 90, right associativity) : com_scope.
Notation "'if' x 'then' y 'else' z 'end'" :=
  (If x y z)
    (in custom com at level 89, x custom com at level 99,
     y at level 99, z at level 99) : com_scope.
Notation "'while' x 'do' y 'end'" :=
  (While x y)
    (in custom com at level 89, x custom com at level 99, y at level 99) : com_scope.

Notation "'output' x" :=
  (Output x)
    (in custom com at level 89, x at level 99) : com_scope.

Check <{ skip }>.
Check <{ output 42 }>.

Reserved Notation
         "st '=[' c ']=>' st' , pn"
         (at level 40, c custom com at level 99,
          st constr, st' constr at next level).
```

We modify the command evaluation to explicitly track outputs. Instead of the previous evaluation relation *st* =[*c*]=> *st'*, we now use the *st* =[*c*]=> *st'*, *os* relation below, where *os* represents the sequence of outputs produced during evaluation.

```
Inductive oceval : com → state → state → outputs → Prop :=
  | OE_Skip : ∀ st,
      st =[ skip ]=> st, []
```

```
  | OE_Asgn : ∀ st a n x,
      aeval st a = n →
      st =[ x := a ]=> (x !-> n ; st), []
  | OE_Seq : ∀ c1 c2 st st' st'' pn1 pn2,
      st =[ c1 ]=> st', pn1 →
      st' =[ c2 ]=> st'', pn2 →
      st =[ c1 ; c2 ]=> st'', (pn1++pn2)
  | OE_If : ∀ st st' b c1 c2 pn,
      let c := if (beval st b) then c1 else c2 in
      st =[ c ]=> st', pn →
      st =[ if b then c1 else c2 end]=> st', pn
  | OE_While : ∀ b st st' pn c,
      st =[ if b then c; while b do c end else skip end ]=> st', pn →
      st =[ while b do c end ]=> st', pn
  | OE_Output : ∀ st a n,
      aeval st a = n →
      st =[ output a ]=> st, [n]
  where "st =[ c ]=> st' , pn" := (oceval c st st' pn).
```

The original noninterference definition, which only compares final states, does not guarantee security of the publicly-observable outputs.

Although output_insecure_com1 and output_insecure_com2 below obviously leak secret through their outputs they still satisfy noninterference.

```
Definition noninterferent P c := ∀ s1 s2 s1' o1 s2' o2,
  pub_equiv P s1 s2 →
  s1 =[ c ]=> s1', o1 →
  s2 =[ c ]=> s2', o2 →
  pub_equiv P s1' s2'.

Definition output_insecure_com1 : com :=
  <{ output Y }>.

Lemma noninterferent_output_insecure_com1 :
  noninterferent xpub output_insecure_com1.
Proof.
  unfold noninterferent. intros.
  invert H0. invert H1. auto.
Qed.

Definition output_insecure_com2 : com :=
  <{ if Y=0 then (output 1) else skip end }>.

Lemma noninterferent_output_insecure_com2 :
  noninterferent xpub output_insecure_com2.
Proof.
  unfold noninterferent. intros.
```

```
  invert H0. invert H1. simpl in *.
  destruct (s1 Y), (s2 Y);
  simpl in *; subst c c0; invert H8; invert H7; auto.
Qed.
```

We define an output security property inspired by control flow security. Instead of relating final states like noninterference, we require that a program's outputs be independent of secrets.

```
Definition output_secure P c := ∀ s1 s2 s1' o1 s2' o2,
  pub_equiv P s1 s2 →
  s1 =[ c ]=> s1', o1 →
  s2 =[ c ]=> s2', o2 →
  o1 = o2.
```

This property disallows programs whose outputs depend on secrets:

```
Lemma output_insecure_output_insecure_com1 :
  ¬ output_secure xpub output_insecure_com1.
Proof.
  unfold output_secure, output_insecure_com1.
  intro Hc.

  set (s1 := Y !-> 0).
  set (s2 := Y !-> 1).

  specialize (Hc s1 s2).

  assert (PEQUIV: pub_equiv xpub s1 s2).
  { clear Hc. intros x H. apply xpub_true in H. subst. reflexivity. }

  specialize (Hc s1 [0] s2 [1] PEQUIV). subst s1 s2.

  assert (Hcontra: [0] = [1]).
  { eapply Hc; econstructor; simpl; auto. }

  discriminate Hcontra.
Qed.

Lemma output_insecure_output_insecure_com2 :
  ¬ output_secure xpub output_insecure_com2.
Proof.
  unfold output_secure, output_insecure_com2.
  intro Hc.

  set (s1 := Y !-> 0).
  set (s2 := Y !-> 1).

  specialize (Hc s1 s2).

  assert (PEQUIV: pub_equiv xpub s1 s2).
```

```
  { clear Hc. intros x H. apply xpub_true in H. subst. reflexivity. }

  specialize (Hc s1 [1] s2 [] PEQUIV). subst s1 s2.
  assert (Hcontra: [1] = []).
  { eapply Hc.
    - repeat econstructor; simpl; auto.
    - eapply OE_If; simpl; auto. econstructor. }

  discriminate Hcontra.
Qed.
```

In the following tasks, you will define a type system enforcing both noninterference and output security. Then, you will write a type-checker and prove that it is sound and complete with respect to the type system. Finally, you will prove that your type system implies both noninterference and output security.

All lemmas and theorems marked as *Admitted* provide partial credit, even if you cannot prove everything.

```
Reserved Notation "P ';;' pc '|-ni-' c" (at level 40).

Inductive oni_well_typed (P:pub_vars) : label → com → Prop :=
  | ONIWT_Com : ∀ pc,
       P ;; pc ⊢ni- <{ skip }>
  | ONIWT_Asgn : ∀ pc X a l,
       P ⊢a- a \in l →
       can_flow (join pc l) (P X) = true →
       P ;; pc ⊢ni- <{ X := a }>
  | ONIWT_Seq : ∀ pc c1 c2,
       P ;; pc ⊢ni- c1 →
       P ;; pc ⊢ni- c2 →
       P ;; pc ⊢ni- <{ c1 ; c2 }>
  | ONIWT_If : ∀ pc b l c1 c2,
       P ⊢b- b \in l →
       P ;; (join pc l) ⊢ni- c1 →
       P ;; (join pc l) ⊢ni- c2 →
       P ;; pc ⊢ni- <{ if b then c1 else c2 end }>
  | ONIWT_While : ∀ pc b l c1,
       P ⊢b- b \in l →
       P ;; (join pc l) ⊢ni- c1 →
       P ;; pc ⊢ni- <{ while b do c1 end }>



where "P ';;' pc '|-ni-' c" := (oni_well_typed P pc c).
```

```
Fixpoint oni_typechecker (P:pub_vars) (pc:label) (c:com) : bool :=
  match c with
  | <{ skip }> ⇒ true
  | <{ X := a }> ⇒ can_flow (join pc (label_of_aexp P a)) (P X)
  | <{ c1 ; c2 }> ⇒ oni_typechecker P pc c1 && oni_typechecker P pc c2
  | <{ if b then c1 else c2 end }> ⇒
      oni_typechecker P (join pc (label_of_bexp P b)) c1 &&
      oni_typechecker P (join pc (label_of_bexp P b)) c2
  | <{ while b do c1 end }> ⇒
      oni_typechecker P (join pc (label_of_bexp P b)) c1

   | _ ⇒ false
    end.

Lemma oni_typechecker_sound : ∀ P pc c,
  oni_typechecker P pc c = true →
  P ;; pc ⊢ni- c.
Proof.
  intros P pc c. generalize dependent pc.
  induction c; intros pc H; simpl in *; try econstructor;
    try repeat rewrite andb_true_iff in *;
    try destruct H; try tauto;
    eauto using label_of_aexp_sound, label_of_bexp_sound.
  Admitted.

Lemma oni_typechecker_complete : ∀ P pc c,
  oni_typechecker P pc c = false →
  ¬ P ;; pc ⊢ni- c.
Proof.
  intros P pc c H Hc. induction Hc; simpl in *;
    try rewrite andb_false_iff in *; try tauto; try congruence.
  - apply label_of_aexp_unique in H0.
    rewrite H0 in *. congruence.
  - destruct H; apply label_of_bexp_unique in H0; subst; eauto.
  - apply label_of_bexp_unique in H0. subst. auto.
  Admitted.

Example not_ni_wt_output1 :
  ¬ xpub ;; public ⊢ni- output_insecure_com1.
Proof.
  Admitted.

Example not_ni_wt_output2 :
  ¬ xpub ;; public ⊢ni- output_insecure_com2.
Proof.
```

```
  Admitted.
```

The noninterference proof follows the same structure as for **ni_well_typed**:

```
Lemma weaken_pc : ∀ {P pc1 pc2 c},
  P ;; pc1 ⊢ni- c →
  can_flow pc2 pc1 = true→
  P ;; pc2 ⊢ni- c.
Proof.
  intros P pc1 pc2 c H. generalize dependent pc2.
  induction H; subst; intros pc2 Hcan_flow.
  - constructor.
  - econstructor; try eassumption. apply can_flow_join_l.
    + apply can_flow_join_1 in H0. eapply can_flow_trans; eassumption.
    + apply can_flow_join_2 in H0. assumption.
  - constructor; auto.
  - econstructor; try eassumption.
    + apply IHoni_well_typed1. apply can_flow_join_l.
      × apply can_flow_join_r1. assumption.
      × apply can_flow_join_r2. apply can_flow_refl.
    + apply IHoni_well_typed2. apply can_flow_join_l.
      × apply can_flow_join_r1. assumption.
      × apply can_flow_join_r2. apply can_flow_refl.
  Admitted.

Lemma secret_run : ∀ {P c s s' os},
  P ;; secret ⊢ni- c →
  s =[ c ]=> s', os →
  pub_equiv P s s'.
Proof.
  intros P c s s' os Hwt Heval. induction Heval; inversion Hwt;
    subst; eauto using pub_equiv_trans, pub_equiv_refl.
  -
    intros y Hy. destruct (String.eqb_spec x y) as [Hxy | Hxy].
    +
      subst. rewrite join_secret_l in H4. rewrite Hy in H4. discriminate H4.
    + rewrite t_update_neq; auto.
  - simpl in *. destruct (beval st b); eapply IHHeval; eauto.
  - rewrite join_secret_l in H3.
    eapply IHHeval. econstructor; eauto; simpl; econstructor; eauto.
Qed.

Lemma secret_run_no_output : ∀ {P c s s' os},
  P ;; secret ⊢ni- c →
  s =[ c ]=> s', os →
```

```
  os = [].
Proof.
  Admitted.

Corollary different_code : ∀ P c1 c2 s1 s2 s1' s2' os1 os2,
  P;; secret ⊢ni- c1 →
  P;; secret ⊢ni- c2 →
  pub_equiv P s1 s2 →
  s1 =[ c1 ]=> s1', os1 →
  s2 =[ c2 ]=> s2', os2 →
  pub_equiv P s1' s2'.
Proof.
  intros P c1 c2 s1 s2 s1' s2' os1 os2 Hwt1 Hwt2 Hequiv Heval1 Heval2.
  eapply secret_run in Hwt1; [| eassumption].
  eapply secret_run in Hwt2; [| eassumption].
  apply pub_equiv_sym in Hwt1.
  eapply pub_equiv_trans; try eassumption.
  eapply pub_equiv_trans; eassumption.
Qed.

Theorem oni_well_typed_noninterferent : ∀ P c,
  P;; public ⊢ni- c →
  noninterferent P c.
Proof.
  intros P c Hwt s1 s2 s1' o1 s2' o2 Heq Heval1 Heval2.
  generalize dependent s2'. generalize dependent o2. generalize dependent s2.
  induction Heval1; intros s2 Heq o2 s2' Heval2; invert Heval2; auto.
  - invert Hwt. intros y Hy.
    destruct (String.eqb_spec x y) as [Hxy | Hxy].
    + subst. do 2 rewrite t_update_eq.
      apply orb_prop in H3. destruct H3 as [Hl | Hx].
      × eapply join_public in Hl. invert Hl. eapply (noninterferent_aexp Heq H2).
      × subst. rewrite Hy in Hx. discriminate Hx.
    + do 2 rewrite (t_update_neq _ _ _ _ _ Hxy).
      apply Heq. apply Hy.
  - invert Hwt. eapply IHHeval1_2; try eassumption. eapply IHHeval1_1; eassumption.
  Admitted.
```

To prove output_secure you can use a similar corollary to different_code, but about the outputs:

```
Corollary different_code_no_output : ∀ P c1 c2 s1 s2 s1' s2' os1 os2,
  P;; secret ⊢ni- c1 →
  P;; secret ⊢ni- c2 →
  pub_equiv P s1 s2 →
```

```
  s1 =[ c1 ]=> s1', os1 →
  s2 =[ c2 ]=> s2', os2 →
  os1 = os2.
Proof.
  intros P c1 c2 s1 s2 s1' s2' os1 os2 Hwt1 Hwt2 Hequiv Heval1 Heval2.
  eapply secret_run_no_output in Hwt1; [| eassumption].
  eapply secret_run_no_output in Hwt2; [| eassumption].
  subst. auto.
Qed.

Theorem oni_well_typed_output_secure : ∀ P c,
  P ;; public ⊢ni- c →
  output_secure P c.
Proof.
   Admitted.
   □

End OUTPUT.
```

Chapter 9

Library SECF.SpecCT

9.1 SpecCT: Cryptographic Constant-Time and Speculative Constant-Time

```
Set Warnings "-notation-overridden,-parsing,-deprecated-hint-without-locality".
From Stdlib Require Import Strings.String.
From SECF Require Import Maps.
From Stdlib Require Import Bool.Bool.
From Stdlib Require Import Arith.Arith.
From Stdlib Require Import Arith.EqNat.
From Stdlib Require Import Arith.PeanoNat. Import Nat.
From Stdlib Require Import Lia.
From Stdlib Require Import List. Import ListNotations.
Set Default Goal Selector "!".
```

This chapter starts by presenting the cryptographic constant-time (CCT) discipline, which we statically enforce using a simple type system. This static discipline is, however, not enough to protect cryptographic programs against speculative execution attacks. To secure CCT programs against this more powerful attacker model we additionally use a program transformation called *Speculative Load Hardening* (SLH). We prove formally that CCT programs protected by SLH achieve speculative constant-time security.

9.2 Cryptographic constant-time

Cryptographic constant-time (CCT) is a software countermeasure against timing side-channel attacks that is widely deployed for cryptographic implementations, for instance to prevent leakage of crypto keys *Barthe et al* 2019 (in Bib.v).

More generally, each program input has to be identified as public or secret, and intuitively the execution time of the program should not depend on secret inputs, even on processors with instruction and data caches.

We, however, do not want to explicitly model execution time or caches, since

- it would be very hard to do right, and
- it would bring in too many extremely low-level details of the concrete compiler (Clang/LLVM 20.1.6) and hardware microarchitecture (Intel Core i7-8650U).

Instead CCT works with a *more abstract model of leakage*, which simply assumes that:

- *all branches the program takes are leaked*;
  - since the path the program takes can greatly influence how long execution takes
  - this is exactly like in the Control Flow (CF) security model from StaticIFC
- *all accessed memory addresses are leaked*;
  - since timing attacks can also exploit the latency difference between hits and misses in the data cache
- *the operands influencing timing of variable-time operations are leaked*;
  - as an exercise we will add a division operation that leaks both operands.

To ensure security against this leakage model, the CCT discipline requires that:

- *the control flow of the program does not depend on secrets*;
  - intuitively this prevents the execution time of different program paths from directly depending on secrets:
    if Wsecret then ... slow computation ... else skip
- *the accessed memory addresses do not depend on secrets*;
  - intuitively this prevents secrets from leaking into the data cache:
    Vsecret <- AP *Wsecret*
- *the operands leaked by variable-time operations do not depend on secrets.*
  - this prevents leaking information about secrets e.g., via division:
    Usecret := div Vsecret Wsecret

To model memory accesses that depend on secrets we will make the Imp language more realistic by adding arrays.

We need such an extension, since otherwise variable accesses in the original Imp map to memory operations at constant locations, which thus cannot depend on secrets, so in Imp CCT trivially holds for all CF well-typed programs. Array indices on the other hand are computed at runtime, which leads to accessing memory addresses that can depend on secrets, making CCT non-trivial for Imp with arrays.

For instance, here is a simple program that is CF secure (since it does no branches), but not CCT secure (since it accesses memory based on secret information):

- *Vsecret* ← *A*[*Wsecret*]

9.2.1 Adding constant-time conditional and refactoring expressions

But first, we add a *b* ? *e1* : *e2* conditional expression that executes in constant time (for instance by being compiled to a special constant-time conditional move instruction). This constant-time conditional will also be used in our SLH countermeasure below.

Technically, adding such conditionals to Imp arithmetic expressions would make them dependent on boolean expressions. But boolean expressions are already dependent on arithmetic expressions.

To avoid making the definitions of arithmetic and boolean expressions mutually inductive, we drop boolean expressions altogether and encode them using arithmetic expressions. Our encoding of bools in terms of nats is similar to that of C, where zero means false, and non-zero means true.

We also refactor the semantics of binary operators in terms of the **binop** enumeration below, to avoid the duplication in Imp:

```
Inductive binop : Type :=
  | BinPlus
  | BinMinus
  | BinMult
  | BinEq
  | BinLe
  | BinAnd
  | BinImpl.
```

We define the semantics of **binop**s directly on nats. We are careful to allow other representations of true (any non-zero number).

```
Definition not_zero (n : nat) : bool := negb (n =? 0).
Definition bool_to_nat (b : bool) : nat := if b then 1 else 0.

Definition eval_binop (o:binop) (n1 n2 : nat) : nat :=
  match o with
  | BinPlus ⇒ n1 + n2
```

```
  | BinMinus ⇒ n1 - n2
  | BinMult ⇒ n1 × n2
  | BinEq ⇒ bool_to_nat (n1 =? n2)
  | BinLe ⇒ bool_to_nat (n1 <=? n2)
  | BinAnd ⇒ bool_to_nat (not_zero n1 && not_zero n2)
  | BinImpl ⇒ bool_to_nat (negb (not_zero n1) || not_zero n2)
  end.

Inductive exp : Type :=
  | ANum (n : nat)
  | AId (x : string)
  | ABin (o : binop) (e1 e2 : exp)
  | ACTIf (b : exp) (e1 e2 : exp).
```

We encode all the previous arithmetic and boolean operations:

```
Definition APlus := ABin BinPlus.
Definition AMinus := ABin BinMinus.
Definition AMult := ABin BinMult.
Definition BTrue := ANum 1.
Definition BFalse := ANum 0.
Definition BAnd := ABin BinAnd.
Definition BImpl := ABin BinImpl.
Definition BNot b := BImpl b BFalse.
Definition BOr e1 e2 := BImpl (BNot e1) e2.
Definition BEq := ABin BinEq.
Definition BNeq e1 e2 := BNot (BEq e1 e2).
Definition BLe := ABin BinLe.
Definition BGt e1 e2 := BNot (BLe e1 e2).
Definition BLt e1 e2 := BGt e2 e1.

Hint Unfold eval_binop : core.
Hint Unfold APlus AMinus AMult : core.
Hint Unfold BTrue BFalse : core.
Hint Unfold BAnd BImpl BNot BOr BEq BNeq BLe BGt BLt : core.
```

The notations we use for expressions are the same as in Imp, except the notation for *be*?*e1*:*e2* which is new: `Definition U : string := "U".`

```
Definition V : string := "V".
Definition W : string := "W".
Definition X : string := "X".
Definition Y : string := "Y".
Definition Z : string := "Z".
Definition AP : string := "AP".
Definition AS : string := "AS".

Coercion AId : string >-> exp.
```

```
Coercion ANum : nat >-> exp.

Declare Custom Entry com.
Declare Scope com_scope.

Notation "<{ e }>" := e (at level 0, e custom com at level 99) : com_scope.
Notation "( x )" := x (in custom com, x at level 99) : com_scope.
Notation "x" := x (in custom com at level 0, x constr at level 0) : com_scope.
Notation "f x .. y" := (.. (f x) .. y)
                    (in custom com at level 0, only parsing,
                    f constr at level 0, x constr at level 9,
                    y constr at level 9) : com_scope.
Notation "x + y" := (APlus x y) (in custom com at level 50, left associativity).
Notation "x - y" := (AMinus x y) (in custom com at level 50, left associativity).
Notation "x * y" := (AMult x y) (in custom com at level 40, left associativity).
Notation "'true'" := true (at level 1).
Notation "'true'" := BTrue (in custom com at level 0).
Notation "'false'" := false (at level 1).
Notation "'false'" := BFalse (in custom com at level 0).
Notation "x <= y" := (BLe x y) (in custom com at level 70, no associativity).
Notation "x > y" := (BGt x y) (in custom com at level 70, no associativity).
Notation "x < y" := (BLt x y) (in custom com at level 70, no associativity).
Notation "x = y" := (BEq x y) (in custom com at level 70, no associativity).
Notation "x <> y" := (BNeq x y) (in custom com at level 70, no associativity).
Notation "x && y" := (BAnd x y) (in custom com at level 80, left associativity).
Notation "'~' b" := (BNot b) (in custom com at level 75, right associativity).

Open Scope com_scope.

Notation "be '?' e1 ':' e2" := (ACTIf be e1 e2)
                  (in custom com at level 20, no associativity).
```

9.2.2 Adding arrays

Now back to adding array loads and stores to commands:

```
Inductive com : Type :=
  | Skip
  | Asgn (x : string) (e : exp)
  | Seq (c1 c2 : com)
  | If (be : exp) (c1 c2 : com)
  | While (be : exp) (c : com)
  | ALoad (x : string) (a : string) (i : exp)
  | AStore (a : string) (i : exp) (e : exp) .

Notation "<{{ e }}>" := e (at level 0, e custom com at level 99) : com_scope.
Notation "( x )" := x (in custom com, x at level 99) : com_scope.
```

```
Notation "x" := x (in custom com at level 0, x constr at level 0) : com_scope.
Notation "f x .. y" := (.. (f x) .. y)
                    (in custom com at level 0, only parsing,
                    f constr at level 0, x constr at level 9,
                    y constr at level 9) : com_scope.

Open Scope com_scope.

Notation "'skip'" :=
  Skip (in custom com at level 0) : com_scope.
Notation "x := y" :=
  (Asgn x y)
    (in custom com at level 0, x constr at level 0,
      y custom com at level 85, no associativity) : com_scope.
Notation "x ; y" :=
  (Seq x y)
    (in custom com at level 90, right associativity) : com_scope.
Notation "'if' x 'then' y 'else' z 'end'" :=
  (If x y z)
    (in custom com at level 89, x custom com at level 99,
     y at level 99, z at level 99) : com_scope.
Notation "'while' x 'do' y 'end'" :=
  (While x y)
    (in custom com at level 89, x custom com at level 99, y at level 99) : com_scope.

Notation "x '<-' a '[[' i ']]'" := (ALoad x a i)
     (in custom com at level 0, x constr at level 0,
      a at level 85, i custom com at level 85, no associativity) : com_scope.
Notation "a '[' i ']' '<-' e" := (AStore a i e)
     (in custom com at level 0, a constr at level 0,
      i custom com at level 0, e custom com at level 85,
         no associativity) : com_scope.

Definition state := total_map nat.
Definition mem := total_map (list nat).

Fixpoint eval (st : state) (e: exp) : nat :=
  match e with
  | ANum n ⇒ n
  | AId x ⇒ st x
  | ABin b e1 e2 ⇒ eval_binop b (eval st e1) (eval st e2)
  | <{b ? e1 : e2}> ⇒ if not_zero (eval st b) then eval st e1
                                 else eval st e2
  end.
```

A couple of obvious lemmas that will be useful in the proofs:

```
Lemma not_zero_eval_S : ∀ b n st,
```

```
  eval st b = S n →
  not_zero (eval st b) = true.
Proof. intros b n st H. rewrite H. reflexivity. Qed.

Lemma not_zero_eval_O : ∀ b st,
  eval st b = O →
  not_zero (eval st b) = false.
Proof. intros b st H. rewrite H. reflexivity. Qed.
```

We also define an array update operation, to be used in the semantics of array stores below:

```
Fixpoint upd (i:nat) (ns:list nat) (n:nat) : list nat :=
  match i, ns with
  | 0, _ :: ns' ⇒ n :: ns'
  | S i', n' :: ns' ⇒ n' :: upd i' ns' n
  | _, _ ⇒ ns
  end.
```

9.2.3 Instrumenting semantics with observations

In addition to the boolean branches, which are observable in the CF security model, for CCT security also the index of array loads and stores are observable:

```
Inductive observation : Type :=
  | OBranch (b : bool)
  | OALoad (a : string) (i : nat)
  | OAStore (a : string) (i : nat).

Definition obs := list observation.
```

We define an instrumented big-step operational semantics producing these observations:

- <(*st*, *m*)> =[*c*]=> <(*st'*, *m'*, *os*)>

Intuitively, variables act like registers (not observable), while arrays act like the memory (addresses observable).

(CTE_Skip) <(st, m)> = *skip* => <(st, m, □)>
  eval st e = n

(CTE_Asgn) <(st, m)> = *x* := *e* => <(x !-> n; st, m, □)>
  <(st, m)> = *c1* => <(st', m', os1)> <(st', m')> = *c2* => <(st", m", os2)>

(CTE_Seq) <(st, m)> = *c1* ; *c2* => <(st", m", os1 ++ os2)>
  let c := if not_zero (eval st be) then c1 else c2 in <(st,m)> = *c* => <(st',m',os1)>

(CTE_If) <(st, m)> = if *be* then *c1* else *c2* end=> <(st', m', OBranch (not_zero (eval *st be*)) ++ os1)>
<(st,m)> = if *be* then *c*; *while be* do *c* end else *skip* end => <(st',m',os)>

(CTE_While) <(st,m)> = *while be* do *c* end => <(st', m', os)>
eval st ie = i i < length (m a)

(CTE_ALoad) <(st,m)> = *x* ← *a*[[*ie*]] => <(x!->nth i (m a) 0; st, m,OALoad *a i*)>
eval st e = n eval st ie = i i < length (m a)

(CTE_AStore) <(st,m)> = *a*[*ie*] ← *e* => <(st, a!->upd i (m a) n; m,OAStore *a i*)>

```
Reserved Notation
        "'<(' st , m ')>' '=[' c ']=>' '<(' stt , mt , os ')>'"
        (at level 40, c custom com at level 99,
         st constr, m constr, stt constr, mt constr at next level).

Inductive cteval : com → state → mem → state → mem → obs → Prop :=
  | CTE_Skip : ∀ st m,
      <(st , m)> =[ skip ]=> <(st, m, [])>
  | CTE_Asgn : ∀ st m e n x,
      eval st e = n →
      <(st, m)> =[ x := e ]=> <(x !-> n; st, m, [])>
  | CTE_Seq : ∀ c1 c2 st m st' m' st'' m'' os1 os2,
      <(st, m)> =[ c1 ]=> <(st', m', os1)> →
      <(st', m')> =[ c2 ]=> <(st'', m'', os2)> →
      <(st, m)> =[ c1 ; c2 ]=> <(st'', m'', os1++os2)>
  | CTE_If : ∀ st m st' m' be c1 c2 os1,
      let c := if not_zero (eval st be) then c1 else c2 in
      <(st, m)> =[ c ]=> <(st', m', os1)> →
      <(st, m)> =[ if be then c1 else c2 end]=>
      <(st', m', [OBranch (not_zero (eval st be))] ++ os1)>
  | CTE_While : ∀ b st m st' m' os c,
      <(st,m)> =[ if b then c; while b do c end else skip end ]=>
      <(st', m', os)> →
      <(st,m)> =[ while b do c end ]=> <(st', m', os)>
  | CTE_ALoad : ∀ st m x a ie i,
      eval st ie = i →
      i < length (m a) →
      <(st, m)> =[ x ← a[[ie]] ]=> <(x !-> nth i (m a) 0; st, m, [OALoad a i])>
  | CTE_AStore : ∀ st m a ie i e n,
      eval st e = n →
      eval st ie = i →
      i < length (m a) →
```

```
      <(st, m)> =[ a[ie] ← e ]=> <(st, a !-> upd i (m a) n; m, [OAStore a i])>

  where "<( st , m )> =[ c ]=> <( stt , mt , os )>" := (cteval c st m stt mt os).
Hint Constructors cteval : core.
```

9.2.4 Constant-time security definition

```
Definition label := bool.

Definition public : label := true.
Definition secret : label := false.

Definition pub_vars := total_map label.
Definition pub_arrs := total_map label.

Definition pub_equiv (P : total_map label) {X:Type} (s1 s2 : total_map X) :=
  ∀ x:string, P x = true → s1 x = s2 x.

Lemma pub_equiv_refl :
  ∀ {X:Type} (P : total_map label) (s : total_map X),
  pub_equiv P s s.
Proof. intros X P s x Hx. reflexivity. Qed.

Lemma pub_equiv_sym :
  ∀ {X:Type} (P : total_map label) (s1 s2 : total_map X),
  pub_equiv P s1 s2 →
  pub_equiv P s2 s1.
Proof.
  unfold pub_equiv. intros X P s1 s2 H x Px.
  rewrite H; auto.
Qed.

Lemma pub_equiv_trans :
  ∀ {X:Type} (P : total_map label) (s1 s2 s3 : total_map X),
  pub_equiv P s1 s2 →
  pub_equiv P s2 s3 →
  pub_equiv P s1 s3.
Proof.
  unfold pub_equiv. intros X P s1 s2 s3 H12 H23 x Px.
  rewrite H12; try rewrite H23; auto.
Qed.

Lemma pub_equiv_update_secret :
  ∀ {X: Type} (P : total_map label) (s1 s2 : total_map X)
         (x: string) (e1 e2: X),
  pub_equiv P s1 s2 →
  P x = secret →
```

```
  pub_equiv P (x !-> e1; s1) (x !-> e2; s2).
Proof.
  unfold pub_equiv. intros X P s1 s2 x e H Pe Px y Py.
  destruct (String.eqb_spec x y) as [Hxy | Hxy]; subst.
  - rewrite Px in Py. discriminate.
  - repeat rewrite t_update_neq; auto.
Qed.

Lemma pub_equiv_update_public :
  ∀ {X: Type} (P : total_map label) (s1 s2 : total_map X)
          (x: string) {e1 e2: X},
  pub_equiv P s1 s2 →
  e1 = e2 →
  pub_equiv P (x !-> e1; s1) (x !-> e2; s2).
Proof.
  unfold pub_equiv. intros X P s1 s2 x e1 e2 H Eq y Py.
  destruct (String.eqb_spec x y) as [Hxy | Hxy]; subst.
  - repeat rewrite t_update_eq; auto.
  - repeat rewrite t_update_neq; auto.
Qed.

Definition cct_secure P PA c :=
  ∀ st1 st2 m1 m2 st1' st2' m1' m2' os1 os2,
    pub_equiv P st1 st2 →
    pub_equiv PA m1 m2 →
    <(st1, m1)> =[ c ]=> <(st1', m1', os1)> →
    <(st2, m2)> =[ c ]=> <(st2', m2', os2)> →
    os1 = os2.
```

9.2.5 Example CF secure program that is not CCT secure

```
Definition cct_insecure_prog :=
   <{{ V ← AP[[W]] }}> .
```

Let's assume that W and V are secret variables. This program is trivially CF secure, because it does not branch at all. But it is not CCT secure.

This is proved below. We first define the public variables and arrays, which we will use in this kind of examples:

```
Definition XYZpub : pub_vars :=
  (X!-> public; Y!-> public; Z!-> public; _ !-> secret).
Definition APpub : pub_arrs :=
  (AP!-> public; _ !-> secret).

Lemma XYZpub_true : ∀ x, XYZpub x = true → x = X ∨ x = Y ∨ x = Z.
Proof.
```

```
  unfold XYZpub. intros x Hxyz.
  destruct (String.eqb_spec x X); auto.
  rewrite t_update_neq in Hxyz; auto.
  destruct (String.eqb_spec x Y); auto.
  rewrite t_update_neq in Hxyz; auto.
  destruct (String.eqb_spec x Z); auto.
  rewrite t_update_neq in Hxyz; auto.
  rewrite t_apply_empty in Hxyz. discriminate.
Qed.

Lemma APpub_true : ∀ a, APpub a = true → a = AP.
Proof.
  unfold APpub. intros a Ha.
  destruct (String.eqb_spec a AP); auto.
  rewrite t_update_neq in Ha; auto. discriminate Ha.
Qed.

Lemma XYZpubXYZ : ∀ x, x = X ∨ x = Y ∨ x = Z → XYZpub x = true.
Proof.
  intros x Hx.
  destruct Hx as [HX | HYZ]; subst.
  - reflexivity.
  - destruct HYZ as [HY | HZ]; subst; reflexivity.
Qed.

Example cct_insecure_prog_is_not_cct_secure :
  ¬ (cct_secure XYZpub APpub cct_insecure_prog).
Proof.
  unfold cct_secure, cct_insecure_prog; intros CTSEC.
  remember (W !-> 1; _ !-> 0) as st1.
  remember (W !-> 2; _ !-> 0) as st2.
  remember (AP !-> [1;2;3]; _ !-> []) as m.
  specialize (CTSEC st1 st2 m m).

  assert (Contra: [OALoad AP 1] = [OALoad AP 2]).
  { eapply CTSEC; subst.
    - apply pub_equiv_update_secret; auto.
      apply pub_equiv_refl.
    - apply pub_equiv_refl.
    - eapply CTE_ALoad; simpl; auto.
    - eapply CTE_ALoad; simpl; auto. }

  discriminate.
Qed.
```

Exercise: 2 stars, standard (cct_insecure_prog'_is_not_cct_secure) Show that also the following program is not CCT secure. `Definition cct_insecure_prog' := <{{ AS[W] ← 42 }}> .`

```
Example cct_insecure_prog'_is_not_cct_secure :
  ¬ (cct_secure XYZpub APpub cct_insecure_prog').
Proof.
  Admitted.
```

□

9.2.6 Type system for cryptographic constant-time programming

In our CCT type system, the label assigned to the result of a constant-time conditional expression simply joins the labels of the 3 involved expressions:

P |- be \in l P |- e1 \in l1 P |- e2 \in l2

(T_CTIf) P |- be?e1:e2 \in join l (join l1 l2)

The rules for the other expressions are standard, and a lot fewer because of our refactoring:

(T_Num) P |- n \in public

(T_Id) P |- X \in (P X)

P |- e1 \in l1 P |- e2 \in l2

(T_Bin) P |- (e1 ‘op‘ e2) \in (join l1 l2)

```
Definition join (l1 l2 : label) : label := l1 && l2.

Lemma join_public : ∀ {l1 l2},
  join l1 l2 = public → l1 = public ∧ l2 = public.
Proof. apply andb_prop. Qed.

Lemma join_public_l : ∀ {l},
  join public l = l.
Proof. reflexivity. Qed.

Definition can_flow (l1 l2 : label) : bool := l1 || negb l2.

Reserved Notation "P '|-' a \in l" (at level 40).

Inductive exp_has_label (P:pub_vars) : exp → label → Prop :=
  | T_Num : ∀ n,
        P ⊢ (ANum n) \in public
  | T_Id : ∀ X,
        P ⊢ (AId X) \in (P X)
  | T_Bin : ∀ op e1 l1 e2 l2,
        P ⊢ e1 \in l1 →
        P ⊢ e2 \in l2 →
```

```
      P ⊢ (ABin op e1 e2) \in (join l1 l2)
  | T_CTIf : ∀ be l e1 l1 e2 l2,
      P ⊢ be \in l →
      P ⊢ e1 \in l1 →
      P ⊢ e2 \in l2 →
      P ⊢ <{ be ? e1 : e2 }> \in (join l (join l1 l2))

where "P '|-' e '\in' l" := (exp_has_label P e l).

Hint Constructors exp_has_label : core.

Theorem noninterferent_exp : ∀ {P s1 s2 e},
  pub_equiv P s1 s2 →
  P ⊢ e \in public →
  eval s1 e = eval s2 e.
Proof.
  intros P s1 s2 e Heq Ht. remember public as l.
  generalize dependent Heql.
  induction Ht; simpl; intros.
  - reflexivity.
  - eapply Heq; auto.
  - eapply join_public in Heql.
    destruct Heql as [HP1 HP2]. subst.
    rewrite IHHt1, IHHt2; reflexivity.
  - eapply join_public in Heql.
    destruct Heql as [HP HP']. subst.
    eapply join_public in HP'.
    destruct HP' as [HP1 HP2]. subst.
    rewrite IHHt1, IHHt2, IHHt3; reflexivity.
Qed.
```

All rules for commands are exactly the same as for **cf_well_typed** (from StaticIFC), except CCT_ALoad and CCT_AStore, which are new.

(CCT_Skip) P ;; PA |-ct- skip
  P |- e \in l can_flow l (P X) = true

(CCT_Asgn) P ;; PA |-ct- X := e
  P ;; PA |-ct- c1 P ;; PA |-ct- c2

(CCT_Seq) P ;; PA |-ct- c1;c2
  P |- be \in public P ;; PA |-ct- c1 P ;; PA |-ct- c2

(CCT_If) P ;; PA |-ct- if be then c1 else c2
  P |- be \in public P |-ct- c

(CCT_While) P ;; PA |-ct- while be then c end
P |- i \in public can_flow (PA a) (P x) = true

(CCT_ALoad) P ;; PA |-ct- x <- a[i]
P |- i \in public P |- e \in l can_flow l (PA a) = true

(CCT_AStore) P ;; PA |-ct- ai <- e

```
Reserved Notation "P ';;' PA '|-ct-' c" (at level 40).

Inductive cct_well_typed (P:pub_vars) (PA:pub_arrs) : com → Prop :=
  | CCT_Skip :
      P ;; PA ⊢ct- <{{ skip }}>
  | CCT_Asgn : ∀ X e l,
      P ⊢ e \in l →
      can_flow l (P X) = true →
      P ;; PA ⊢ct- <{{ X := e }}>
  | CCT_Seq : ∀ c1 c2,
      P ;; PA ⊢ct- c1 →
      P ;; PA ⊢ct- c2 →
      P ;; PA ⊢ct- <{{ c1 ; c2 }}>
  | CCT_If : ∀ b c1 c2,
      P ⊢ b \in public →
      P ;; PA ⊢ct- c1 →
      P ;; PA ⊢ct- c2 →
      P ;; PA ⊢ct- <{{ if b then c1 else c2 end }}>
  | CCT_While : ∀ b c1,
      P ⊢ b \in public →
      P ;; PA ⊢ct- c1 →
      P ;; PA ⊢ct- <{{ while b do c1 end }}>
  | CCT_ALoad : ∀ x a i,
      P ⊢ i \in public →
      can_flow (PA a) (P x) = true →
      P ;; PA ⊢ct- <{{ x ← a[[i]] }}>
  | CCT_AStore : ∀ a i e l,
      P ⊢ i \in public →
      P ⊢ e \in l →
      can_flow l (PA a) = true →
      P ;; PA ⊢ct- <{{ a[i] ← e }}>

where "P ;; PA '|-ct-' c" := (cct_well_typed P PA c).
Hint Constructors cct_well_typed : core.
```

9.2.7 Exercise: CCT Type-Checker

In these exercises you will write a type-checker for the CCT type system above and prove your type-checker sound and complete. If you get stuck, you can take inspiration in the similar type-checkers from StaticIFC.

Exercise: 1 star, standard (label_of_exp) `Fixpoint label_of_exp (P:pub_vars) (e:exp) : label . Admitted.` □

Exercise: 1 star, standard (label_of_exp_sound) `Lemma label_of_exp_sound : ∀ P e, P ⊢ e \in label_of_exp P e. Proof. Admitted.` □

Exercise: 1 star, standard (label_of_exp_unique) `Lemma label_of_exp_unique : ∀ P e l, P ⊢ e \in l → l = label_of_exp P e. Proof. Admitted.` □

Exercise: 2 stars, standard (cct_typechecker) `Fixpoint cct_typechecker (P PA:pub_vars) (c:com) : bool . Admitted.` □

Exercise: 2 stars, standard (cct_typechecker_sound) `Theorem cct_typechecker_sound : ∀ P PA c, cct_typechecker P PA c = true → P ;; PA ⊢ct- c. Proof. Admitted.` □

Exercise: 2 stars, standard (cct_typechecker_complete) `Theorem cct_typechecker_complete : ∀ P PA c, cct_typechecker P PA c = false →`

```
  ¬ (P ;; PA ⊢ct- c).
Proof.
    Admitted.
    □
```

Finally, we use the type-checker to show that the cct_insecure_prog and cct_insecure_prog' examples above are not well-typed.

```
Print cct_insecure_prog. Print XYZpub. Print APpub.
```

Exercise: 1 star, standard (cct_insecure_prog_ill_typed) `Theorem` cct_insecure_prog_ill_typed :

```
  ~(XYZpub ;; APpub ⊢ct- cct_insecure_prog).
Proof.
    Admitted.
    □
```

Exercise: 1 star, standard (cct_insecure_prog'_ill_typed) `Theorem` cct_insecure_prog'_ill_typed :

```
  ~(XYZpub ;; APpub ⊢ct- cct_insecure_prog').
Proof.
    Admitted.
    □
```

9.2.8 Noninterference lemma

To prove the security of our type system, we first show a noninterference lemma, which is not that hard, given that our very restrictive type system ensures the two executions run in lock-step, since it disallows branching on secrets.

```
Lemma cct_well_typed_noninterferent :
  ∀ P PA c st1 st2 m1 m2 st1' st2' m1' m2' os1 os2,
  P ;; PA ⊢ct- c →
  pub_equiv P st1 st2 →
  pub_equiv PA m1 m2 →
  <(st1, m1)> =[ c ]=> <(st1', m1', os1)> →
  <(st2, m2)> =[ c ]=> <(st2', m2', os2)> →
  pub_equiv P st1' st2' ∧ pub_equiv PA m1' m2'.
Proof.
  intros P PA c st1 st2 m1 m2 st1' st2' m1' m2' os1 os2
    Hwt Heq Haeq Heval1 Heval2.
  generalize dependent st2'. generalize dependent st2.
  generalize dependent m2'. generalize dependent m2.
  generalize dependent os2.
  induction Heval1;
```

```
    intros os2' m2 Haeq m2' st2 Heq st2' Heval2;
    inversion Heval2; inversion Hwt; subst.
  - split; auto.
  - split; auto. destruct l.
    + rewrite (noninterferent_exp Heq H10).
      eapply pub_equiv_update_public; auto.
    + simpl in H11. rewrite negb_true_iff in H11.
      eapply pub_equiv_update_secret; auto.
  - edestruct IHHeval1_2; eauto.
    + eapply IHHeval1_1; eauto.
    + eapply IHHeval1_1; eauto.
  - eapply IHHeval1; eauto.
    + subst c. destruct (eval st be); simpl; auto.
    + subst c c4.
      rewrite (noninterferent_exp Heq H11); eauto.
  - eapply IHHeval1; eauto.
  -
    split; eauto.
    erewrite noninterferent_exp; eauto.
    destruct (PA a) eqn:PAa.
    + eapply pub_equiv_update_public; auto.
      eapply Haeq in PAa. rewrite PAa. reflexivity.
    + simpl in H15. rewrite negb_true_iff in H15.
      eapply pub_equiv_update_secret; auto.
  -
    split; eauto.
    destruct (PA a) eqn:PAa; simpl in *.
    + eapply Haeq in PAa. rewrite PAa.
      destruct l; [|discriminate].
      eapply pub_equiv_update_public; auto.
      repeat erewrite (noninterferent_exp Heq); auto.
    + eapply pub_equiv_update_secret; auto.
Qed.
```

9.2.9 Final theorem: cryptographic constant-time security

```
Module Remember.
Definition cct_secure P PA c :=
  ∀ st1 st2 m1 m2 st1' st2' m1' m2' os1 os2,
    pub_equiv P st1 st2 →
    pub_equiv PA m1 m2 →
    <(st1 , m1)> =[ c ]=> <(st1', m1', os1)> →
```

```
    <(st2, m2)> =[ c ]=> <(st2', m2', os2)> →
    os1 = os2.
End Remember.

Theorem cct_well_typed_secure : ∀ P PA c,
  P ;; PA ⊢ct- c →
  cct_secure P PA c.
Proof.
  unfold cct_secure.
  intros P PA c Hwt st1 st2 m1 m2 st1' st2' m1' m2' os1 os2
    Heq Haeq Heval1 Heval2.
  generalize dependent st2'. generalize dependent st2.
  generalize dependent m2'. generalize dependent m2.
  generalize dependent os2.
  induction Heval1; intros os2' a2 Haeq a2' s2 Heq s2' Heval2;
    inversion Heval2; inversion Hwt; subst.
  - reflexivity.
  - reflexivity.
  - erewrite IHHeval1_2; [erewrite IHHeval1_1 | | | |];
      try reflexivity; try eassumption.
    + eapply cct_well_typed_noninterferent with (c:=c1); eauto.
    + eapply cct_well_typed_noninterferent with (c:=c1); eauto.
  - rewrite (noninterferent_exp Heq H11).
    f_equal; auto. eapply IHHeval1; eauto.
    + subst c. destruct (eval st be); simpl; auto.
    + subst c c4.
      rewrite (noninterferent_exp Heq H11); eauto.
  - eapply IHHeval1; eauto.
  -
    f_equal. f_equal. eapply noninterferent_exp; eassumption.
  -
    f_equal. f_equal. eapply noninterferent_exp; eassumption.
Qed.
```

Most cases of this proof are similar to the security proof for **cf_well_typed** from StaticIFC. In particular, *noninterference* is used to prove the sequence case in both proofs.

The only new cases here are for array operations, and they follow immediately from noninterferent_exp, since the CCT type system requires array indices to be public.

9.2.10 Exercise: Adding division (non-constant-time operation)

The CCT discipline also prevents passing secrets to operations that are not constant time. For instance, division often takes time that depends on the values of the two operands. In this exercise we will add a new $x := e1\ div\ e2$ command for division, add corresponding

evaluation and typing rules, and extend the security proofs with the new division case.

```
Module Div.

Inductive com : Type :=
| Skip
| Asgn (x : string) (e : exp)
| Seq (c1 c2 : com)
| If (be : exp) (c1 c2 : com)
| While (be : exp) (c : com)
| ALoad (x : string) (a : string) (i : exp)
| AStore (a : string) (i : exp) (e : exp)
| Div (x: string) (e1 e2: exp).

Open Scope com_scope.
```

Notations for the old commands are the same as before: `Notation "'skip'" := Skip (in custom com at level 0) : com_scope.`

```
Notation "x := y" :=
  (Asgn x y)
    (in custom com at level 0, x constr at level 0,
      y custom com at level 85, no associativity) : com_scope.
Notation "x ; y" :=
  (Seq x y)
    (in custom com at level 90, right associativity) : com_scope.
Notation "'if' x 'then' y 'else' z 'end'" :=
  (If x y z)
    (in custom com at level 89, x custom com at level 99,
     y at level 99, z at level 99) : com_scope.
Notation "'while' x 'do' y 'end'" :=
  (While x y)
    (in custom com at level 89, x custom com at level 99, y at level 99) : com_scope.
Notation "x '<-' a '[[' i ']]'" := (ALoad x a i)
     (in custom com at level 0, x constr at level 0,
      a at level 85, i custom com at level 85, no associativity) : com_scope.
Notation "a '[' i ']' '<-' e" := (AStore a i e)
     (in custom com at level 0, a constr at level 0,
      i custom com at level 0, e custom com at level 85,
         no associativity) : com_scope.
```

Notation for division: `Notation "x := y 'div' z" :=`

```
  (Div x y z)
    (in custom com at level 0, x constr at level 0,
        y custom com at level 85, z custom com at level 85, no associativity) :
com_scope.

Inductive observation : Type :=
```

```
| OBranch (b : bool)
| OALoad (a : string) (i : nat)
| OAStore (a : string) (i : nat)
| ODiv (n1 n2: nat).

Definition obs := list observation.
```

We add a new rule to the big-step operational semantics that produces an ODiv observation:

eval st e1 = n1 eval st e2 = n2

(CTE_Div) <(st,m)> =x := e1 div e2=> <(x!->(n1/n2);st,m,ODiv n1 n2)>

Formally this looks as follows:

```
Reserved Notation
        "'<(' st , m ')>' '=[' c ']=>' '<(' stt , mt , os ')>'"
        (at level 40, c custom com at level 99,
         st constr, m constr, stt constr, mt constr at next level).

Inductive cteval : com → state → mem → state → mem → obs → Prop :=
  | CTE_Skip : ∀ st m,
      <(st , m)> =[ skip ]=> <(st, m, [])>
  | CTE_Asgn : ∀ st m e n x,
      eval st e = n →
      <(st, m)> =[ x := e ]=> <(x !-> n; st, m, [])>
  | CTE_Seq : ∀ c1 c2 st m st' m' st'' m'' os1 os2,
      <(st, m)> =[ c1 ]=> <(st', m', os1)> →
      <(st', m')> =[ c2 ]=> <(st'', m'', os2)> →
      <(st, m)> =[ c1 ; c2 ]=> <(st'', m'', os1++os2)>
  | CTE_If : ∀ st m st' m' be c1 c2 os1,
      let c := if not_zero (eval st be) then c1 else c2 in
      <(st, m)> =[ c ]=> <(st', m', os1)> →
      <(st, m)> =[ if be then c1 else c2 end]=>
      <(st', m', [OBranch (not_zero (eval st be))] ++ os1)>
  | CTE_While : ∀ b st m st' m' os c,
      <(st,m)> =[ if b then c; while b do c end else skip end ]=>
      <(st', m', os)> →
      <(st,m)> =[ while b do c end ]=> <(st', m', os)>
  | CTE_ALoad : ∀ st m x a ie i,
      eval st ie = i →
      i < length (m a) →
      <(st, m)> =[ x ← a[[ie]] ]=> <(x !-> nth i (m a) 0; st, m, [OALoad a i])>
  | CTE_AStore : ∀ st m a ie i e n,
      eval st e = n →
      eval st ie = i →
```

```
      i < length (m a) →
      <(st, m)> =[ a[ie] ← e ]=> <(st, a !-> upd i (m a) n; m, [OAStore a i])>
  | CTE_Div : ∀ st m e1 n1 e2 n2 x,
      eval st e1 = n1 →
      eval st e2 = n2 →
      <(st, m)> =[ x := e1 div e2 ]=> <(x !-> (n1 / n2)%nat; st, m, [ODiv n1 n2]
)>

  where "<( st , m )> =[ c ]=> <( stt , mt , os )>" := (cteval c st m stt mt os).

Hint Constructors cteval : core.

Reserved Notation "P ';;' PA '|-ct-' c" (at level 40).
```

Exercise: 1 star, standard (cct_well_typed_div) Add a new typing rule for division to **cct_well_typed** below. Your rule should prevent leaking secret division operands via observations.

```
Inductive cct_well_typed (P:pub_vars) (PA:pub_arrs) : com → Prop :=
  | CCT_Skip :
      P ;; PA ⊢ct- <{{ skip }}>
  | CCT_Asgn : ∀ X e l,
      P ⊢ e \in l →
      can_flow l (P X) = true →
      P ;; PA ⊢ct- <{{ X := e }}>
  | CCT_Seq : ∀ c1 c2,
      P ;; PA ⊢ct- c1 →
      P ;; PA ⊢ct- c2 →
      P ;; PA ⊢ct- <{{ c1 ; c2 }}>
  | CCT_If : ∀ b c1 c2,
      P ⊢ b \in public →
      P ;; PA ⊢ct- c1 →
      P ;; PA ⊢ct- c2 →
      P ;; PA ⊢ct- <{{ if b then c1 else c2 end }}>
  | CCT_While : ∀ b c1,
      P ⊢ b \in public →
      P ;; PA ⊢ct- c1 →
      P ;; PA ⊢ct- <{{ while b do c1 end }}>
  | CCT_ALoad : ∀ x a i,
      P ⊢ i \in public →
      can_flow (PA a) (P x) = true →
      P ;; PA ⊢ct- <{{ x ← a[[i]] }}>
  | CCT_AStore : ∀ a i e l,
      P ⊢ i \in public →
      P ⊢ e \in l →
```

```
      can_flow l (PA a) = true →
      P ;; PA ⊢ct- <{{ a[i] ← e }}>

  where "P ;; PA '|-ct-' c" := (cct_well_typed P PA c).
Definition manual_grade_for_cct_well_typed_div : option (nat×string) := None.
```
□

```
Hint Constructors cct_well_typed : core.
```

Exercise: 2 stars, standard (cct_well_typed_div_noninterferent) `Theorem cct_well_typed_div_no`

```
:
  ∀ P PA c st1 st2 m1 m2 st1' st2' m1' m2' os1 os2,
  P ;; PA ⊢ct- c →
  pub_equiv P st1 st2 →
  pub_equiv PA m1 m2 →
  <(st1, m1)> =[ c ]=> <(st1', m1', os1)> →
  <(st2, m2)> =[ c ]=> <(st2', m2', os2)> →
  pub_equiv P st1' st2' ∧ pub_equiv PA m1' m2'.
Proof.
  intros P PA c st1 st2 m1 m2 st1' st2' m1' m2' os1 os2
    Hwt Heq Haeq Heval1 Heval2.
  generalize dependent st2'. generalize dependent st2.
  generalize dependent m2'. generalize dependent m2.
  generalize dependent os2.
  induction Heval1;
    intros os2' m2 Haeq m2' st2 Heq st2' Heval2;
    inversion Heval2; inversion Hwt; subst.
  - split; auto.
  - split; auto. destruct l.
    + rewrite (noninterferent_exp Heq H10).
      eapply pub_equiv_update_public; auto.
    + simpl in H11. rewrite negb_true_iff in H11.
      eapply pub_equiv_update_secret; auto.
  - edestruct IHHeval1_2; eauto.
    + eapply IHHeval1_1; eauto.
    + eapply IHHeval1_1; eauto.
  - eapply IHHeval1; eauto.
    + subst c. destruct (eval st be); simpl; auto.
    + subst c c4.
      rewrite (noninterferent_exp Heq H11); eauto.
  - eapply IHHeval1; eauto.
  - split; eauto.
```

```
    erewrite noninterferent_exp; eauto.
    destruct (PA a) eqn:PAa.
    + eapply pub_equiv_update_public; auto.
      eapply Haeq in PAa. rewrite PAa. reflexivity.
    + simpl in H15. rewrite negb_true_iff in H15.
      eapply pub_equiv_update_secret; auto.
  - split; eauto.
    destruct (PA a) eqn:PAa; simpl in *.
    + eapply Haeq in PAa. rewrite PAa.
      destruct l; [|discriminate].
      eapply pub_equiv_update_public; auto.
      repeat erewrite (noninterferent_exp Heq); auto.
    + eapply pub_equiv_update_secret; auto.
  Admitted.
```

□

We need to redefine cct_secure for our new command definition

```
Definition cct_secure P PA c :=
  ∀ st1 st2 m1 m2 st1' st2' m1' m2' os1 os2,
    pub_equiv P st1 st2 →
    pub_equiv PA m1 m2 →
    <(st1, m1)> =[ c ]=> <(st1', m1', os1)> →
    <(st2, m2)> =[ c ]=> <(st2', m2', os2)> →
    os1 = os2.
```

Exercise: 2 stars, standard (cct_well_typed_div_secure) Reprove CCT security of the type system. Hint: If this proof doesn't go through easily, you may need to go back and fix your div rule.

```
Theorem cct_well_typed_div_secure : ∀ P PA c,
  P ;; PA ⊢ct- c →
  cct_secure P PA c.
Proof.
  unfold cct_secure.
  intros P PA c Hwt st1 st2 m1 m2 st1' st2' m1' m2' os1 os2
    Heq Haeq Heval1 Heval2.
  generalize dependent st2'. generalize dependent st2.
  generalize dependent m2'. generalize dependent m2.
  generalize dependent os2.
  induction Heval1; intros os2' a2 Haeq a2' s2 Heq s2' Heval2;
    inversion Heval2; inversion Hwt; subst.
  - reflexivity.
  - reflexivity.
  - erewrite IHHeval1_2; [erewrite IHHeval1_1 | | | |];
      try reflexivity; try eassumption.
```

```
    + eapply cct_well_typed_div_noninterferent with (c:=c1); eauto.
    + eapply cct_well_typed_div_noninterferent with (c:=c1); eauto.
  - rewrite (noninterferent_exp Heq H11).
    f_equal; auto. eapply IHHeval1; eauto.
    + subst c. destruct (eval st be); simpl; auto.
    + subst c c4.
      rewrite (noninterferent_exp Heq H11); eauto.
  - eapply IHHeval1; eauto.
  - f_equal. f_equal. eapply noninterferent_exp; eassumption.
  - f_equal. f_equal. eapply noninterferent_exp; eassumption.
  Admitted.
  □ End Div.
```

9.3 Speculative constant-time (text under development)

This second part of the chapter is based on the Spectre Declassified paper *Shivakumar et al* 2023 (in Bib.v) in simplified form (e.g., without declassification). Like in this paper, we only look at a class of speculative execution attacks called Spectre v1.

The Rocq development below is complete, but the text about it is still under development and gets sparse after the first 3-4 subsections, especially for the security proof. Readers can skip the security proof, or if they have access to the slides associated to this chapter (i.e. the TERSE version) look there for a high-level overview of the security proof.

9.3.1 CCT programs can be insecure under speculative execution

All variables mentioned in the program below (X, Y, AP) are *public*, so this program respects the CCT discipline, yet this program is not secure under speculative execution.

The size of public array AP is 3 and we check we're in bounds, yet this check can misspeculate!

```
Definition spec_insecure_prog :=
  <{{ if Y < 3 then
        X ← AP[[Y]];
        if X ≤ 5 then X := 5 else skip end
      else skip end }}> .

Example spec_insecure_prog_is_ct_well_typed :
  XYZpub ;; APpub ⊢ct- spec_insecure_prog.
Proof.
  unfold spec_insecure_prog.
  - apply CCT_If; auto.
    + rewrite ← join_public_l.
      eapply T_Bin.
```

```
      × rewrite ← join_public_l.
        eapply T_Bin; auto.
      × eapply T_Num.
    + eapply CCT_Seq.
      × eapply CCT_ALoad; auto.
      × eapply CCT_If; auto.
        { rewrite ← join_public_l.
          eapply T_Bin; auto. }
        { eapply CCT_Asgn; eauto. }
Qed.
```

Here is a more realistic version of this example:

```
Definition spec_insecure_prog_2 :=
  <{{ X := 0;
      Y := 0;
      while Y < 3 do
        Z ← AP[[Y]];
        X := X + Z;
        Y := Y + 1
      end;
      if X ≤ 5 then X := 5 else skip end }}> .

Example spec_insecure_prog_2_is_ct_well_typed :
  XYZpub ;; APpub ⊢ct- spec_insecure_prog_2.
Proof.
  apply CCT_Seq.
  - eapply CCT_Asgn; auto.
  - apply CCT_Seq.
    + eapply CCT_Asgn; auto.
    + eapply CCT_Seq.
      { apply CCT_While.
        - rewrite ← join_public_l.
          apply T_Bin; auto.
          + rewrite ← join_public_l.
            apply T_Bin; auto.
          + unfold BFalse. auto.
        - eapply CCT_Seq.
          + eapply CCT_ALoad; auto.
          + eapply CCT_Seq.
            × eapply CCT_Asgn with (l:= public).
              { rewrite ← join_public_l.
                eapply T_Bin; auto. }
              { reflexivity. }
            × eapply CCT_Asgn with (l:= public).
```

```
                { rewrite ← join_public_l.
                  eapply T_Bin; auto. }
                { reflexivity. } }
        { apply CCT_If; auto.
          - rewrite ← join_public_l.
            eapply T_Bin; auto.
          - eapply CCT_Asgn; auto. }
Qed.
```

All variables mentioned in the program are again public, so also this program respects the CCT discipline, yet it is also not secure under speculative execution.

This example is formalized at the end of the chapter.

9.3.2 Speculative semantics

To reason about the security of these examples against Spectre v1 we will introduce a speculative semantics. To model leakage the semantics uses the same CCT observations as above (OBranch, OALoad, and OAStore).

More interestingly, to model speculative execution we add to the semantics adversary-provided *directions*, which control the speculation behavior:

```
Inductive direction :=
| DStep
| DForce
| DLoad (a : string) (i : nat)
| DStore (a : string) (i : nat).

Definition dirs := list direction.
```

This gives us a very high-level model of speculation that abstracts away low-level details such as the compiler, branch predictors, memory layout, speculation window, rollbacks, etc. We do this in a way that tries to overapproximate the adversary's power.

This kind of speculation model is actually used by the Jasmin language for high-assurance crypto.

Compared to the CCT semantics with observations as output, we now add the directions as input to the evaluation judgement and we also track a misspeculation bit b.

(Spec_Skip) <(st,m,b,□)> =*skip*=> <(st,m,b,□)>
eval st e = n

(Spec_Asgn) <(st,m,b,□)> =x:=e=> <(x!->n;st,m,b,□)>
<(st,m,b,ds1)> =*c1*=> <(st',m',b',os1)> <(st',m',b',ds2)> =*c2*=> <(st",m",b",os2)>

(Spec_Seq) <(st,m,b,ds1++ds2)> =*c1*;*c2*=> <(st",m",b",os1++os2)>
<(st,m,b,ds)> = `if` *be* `then` *c*; *while be* `do` *c* `end` => <(st',m',b',os)>

(Spec_While) <(st,m,b,ds)> = *while be* `do` *c* `end` => <(st',m',b',os)>
let c := if not_zero (eval st be) then c1 else c2 in <(st,m,b,ds)> = *c* => <(st',m',b',os1)>

(Spec_If) <(st,m,b, DStep::ds)> = `if` *be* `then` *c1* `else` *c2* `end` => <(st',m',b', OBranch (not_zero (`eval` *st be*))++os1)>
let c := if not_zero (eval st be) then c2 else c1 in <(st,m,true,ds)> = *c* => <(st',m',b',os1)>

(Spec_If_F) <(st,m,b, DForce::ds)> = `if` *be* `then` *c1* `else` *c2* `end` => <(st',m',b', OBranch (not_zero (`eval` *st be*))++os1)>
eval st ie = i i < length(m a)

(Spec_ALoad) <(st, m, b, DStep)> = *x* ← *a*[[*ie*]] => <(x !-> nth i (m a) 0; st, m, b, OALoad *a i*)>
eval st ie = i i >= length(m a) i' < length(m a')

(Spec_ALoad_U) <(st, m, true, DLoad *a' i'*)> = *x* ← *a*[[*ie*]] => <(x !-> nth i' (m a') 0; st, m, true, OALoad *a i*)>
eval st e = n eval st ie = i i < length(m a)

(Spec_AStore) <(st, m, b, DStep)> = *a*[*ie*] ← *e* => <(st, a !-> upd i (m a) n; m, b, OAStore *a i*)>
eval st e = n eval st ie = i i >= length(m a) i' < length(m a')

(Spec_AStore_U) <(st, m, true, DStore *a' i'*)> = *a*[*ie*] ← *e* => <(st, a' !-> upd i' (m a') n; m, true, OAStore *a i*)>
Formally this definition looks as follows:

```
Reserved Notation
  "'<(' st , m , b , ds ')>' '=[' c ']=>' '<(' stt , mt , bb , os ')>'"
  (at level 40, c custom com at level 99,
   st constr, m constr, stt constr, mt constr at next level).

Inductive spec_eval : com → state → mem → bool → dirs →
                                  state → mem → bool → obs → Prop :=
  | Spec_Skip : ∀ st m b,
      <(st, m, b, [])> =[ skip ]=> <(st, m, b, [])>
  | Spec_Asgn : ∀ st m b e n x,
      eval st e = n →
      <(st, m, b, [])> =[ x := e ]=> <(x !-> n; st, m, b, [])>
  | Spec_Seq : ∀ c1 c2 st m b st' m' b' st'' m'' b'' os1 os2 ds1 ds2,
      <(st, m, b, ds1)> =[ c1 ]=> <(st', m', b', os1)> →
      <(st', m', b', ds2)> =[ c2 ]=> <(st'', m'', b'', os2)> →
      <(st, m, b, ds1++ds2)> =[ c1 ; c2 ]=> <(st'', m'', b'', os1++os2)>
```

```
  | Spec_If : ∀ st m b st' m' b' be c1 c2 os1 ds,
      let c := (if (not_zero (eval st be)) then c1 else c2) in
      <(st, m, b, ds)> =[ c ]=> <(st', m', b', os1)> →
      <(st, m, b, DStep :: ds)> =[ if be then c1 else c2 end ]=>
      <(st', m', b', [OBranch (not_zero (eval st be))] ++ os1)>
  | Spec_If_F : ∀ st m b st' m' b' be c1 c2 os1 ds,
      let c := (if (not_zero (eval st be)) then c2 else c1) in
      <(st, m, true, ds)> =[ c ]=> <(st', m', b', os1)> →
      <(st, m, b, DForce :: ds)> =[ if be then c1 else c2 end ]=>
      <(st', m', b', [OBranch (not_zero (eval st be))] ++ os1)>
  | Spec_While : ∀ be st m b ds st' m' b' os c,
      <(st, m, b, ds)> =[ if be then c; while be do c end else skip end ]=>
      <(st', m', b', os)> →
      <(st, m, b, ds)> =[ while be do c end ]=> <(st', m', b', os)>
  | Spec_ALoad : ∀ st m b x a ie i,
      eval st ie = i →
      i < length (m a) →
      <(st, m, b, [DStep])> =[ x ← a[[ie]] ]=>
      <(x !-> nth i (m a) 0; st, m, b, [OALoad a i])>
  | Spec_ALoad_U : ∀ st m x a ie i a' i',
      eval st ie = i →
      i ≥ length (m a) →
      i' < length (m a') →
      <(st, m, true, [DLoad a' i'])> =[ x ← a[[ie]] ]=>
      <(x !-> nth i' (m a') 0; st, m, true, [OALoad a i])>
  | Spec_AStore : ∀ st m b a ie i e n,
      eval st e = n →
      eval st ie = i →
      i < length (m a) →
      <(st, m, b, [DStep])> =[ a[ie] ← e ]=>
      <(st, a !-> upd i (m a) n; m, b, [OAStore a i])>
  | Spec_AStore_U : ∀ st m a ie i e n a' i',
      eval st e = n →
      eval st ie = i →
      i ≥ length (m a) →
      i' < length (m a') →
      <(st, m, true, [DStore a' i'])> =[ a[ie] ← e ]=>
      <(st, a' !-> upd i' (m a') n; m, true, [OAStore a i])>

  where "<( st , m , b , ds )> =[ c ]=> <( stt , mt , bb , os )>" :=
    (spec_eval c st m b ds stt mt bb os).

Hint Constructors spec_eval : core.
```

9.3.3 Speculative constant-time security definition

The definition of speculative constant-time security is very similar to CCT security, but applied to the speculative semantics. The two executions receive the same directions *ds*:

```
Definition spec_ct_secure P PA c :=
  ∀ st1 st2 m1 m2 st1' st2' m1' m2' b1' b2' os1 os2 ds,
    pub_equiv P st1 st2 →
    pub_equiv PA m1 m2 →
    <(st1, m1, false, ds)> =[ c ]=> <(st1', m1', b1', os1)> →
    <(st2, m2, false, ds)> =[ c ]=> <(st2', m2', b2', os2)> →
    os1 = os2.
```

We can use this definition to show that our first example is speculatively insecure:

```
Print spec_insecure_prog.
```

For this we build a counterexample where the attacker chooses an out-of-bounds index Y = 3 and then passes the directions: [DForce; DLoad AS 0; DStep]. This causes the two executions to load different values for X from index 0 of secret array AS. If the different values loaded from AS[0] are well chosen (e.g., $4 \leq 5$ in the first execution and $7 > 5$ in the second) this causes two different observations: - [OBranch false; OALoad AP 3; OBranch true] and - [OBranch false; OALoad AP 3; OBranch false].

```
Example spec_insecure_prog_is_spec_insecure :
  ~(spec_ct_secure XYZpub APpub spec_insecure_prog).
Proof.
  unfold spec_insecure_prog. intros Hcs.
  remember (Y!-> 3; __ !-> 0) as st.
  remember (AP!-> [0;1;2]; AS!-> [4;1]; __ !-> []) as m1.
  remember (AP!-> [0;1;2]; AS!-> [7;1]; __ !-> []) as m2.
  remember (DForce :: ([DLoad AS 0] ++ [DStep])) as ds.
  remember (([OBranch false] ++ ([OALoad AP 3]) ++ [OBranch true])) as os1.
  remember (([OBranch false] ++ ([OALoad AP 3])++ [OBranch false])) as os2.

  assert (Heval1:
            <(st, m1, false, ds )> =[ spec_insecure_prog ]=>
            <( X!-> 5; X!-> 4; st, m1, true, os1)>).
  { unfold spec_insecure_prog; subst.
    eapply Spec_If_F. eapply Spec_Seq.
    - eapply Spec_ALoad_U; simpl; eauto.
    - rewrite ← app_nil_l with (l:=[OBranch true]).
      eapply Spec_If; simpl. eapply Spec_Asgn; eauto. }

  assert (Heval2:
            <(st, m2, false, ds )> =[ spec_insecure_prog ]=>
            <( X!-> 7; st, m2, true, os2)>).
```

```
    { unfold spec_insecure_prog; subst.
      eapply Spec_If_F. eapply Spec_Seq.
      - eapply Spec_ALoad_U; simpl; eauto.
      - rewrite ← app_nil_l with (l:=[OBranch false]).
        eapply Spec_If; simpl. auto. }

  subst. eapply Hcs in Heval1.
  + eapply Heval1 in Heval2. inversion Heval2.
  + eapply pub_equiv_refl.
  + apply pub_equiv_update_public; auto.
    apply pub_equiv_update_secret; auto.
    apply pub_equiv_refl.
Qed.
```

Exercise: 1 star, standard (speculation_bit_monotonic) As mentioned above, our speculative semantics is very high-level, and doesn't have to deal with detecting misspeculation and rolling back. So in our semantics once the misspeculation bit is set to true, it will stay set:

```
Lemma speculation_bit_monotonic :
  ∀ c s a b ds s' a' b' os,
  <(s, a, b, ds)> =[ c ]=> <(s', a', b', os)> →
  b = true →
  b' = true.
Proof.
    Admitted.
    □

Lemma speculation_needs_force :
  ∀ c s a b ds s' a' b' os,
  <(s, a, b, ds)> =[ c ]=> <(s', a', b', os)> →
  b = false →
  b' = true →
  In DForce ds.
Proof.
  intros c s a b ds s' a' b' os Heval Hb Hb'.
  induction Heval; subst; simpl; eauto; try discriminate.
  apply in_or_app. destruct b'; eauto.
Qed.
```

We can recover sequential execution from **spec_eval** if there is no speculation, so all directives are DStep and misspeculation flag starts set to false.

```
Definition seq_spec_eval (c :com) (st :state) (m :mem)
    (st' :state) (m' :mem) (os :obs) : Prop :=
```

```
  ∃ ds, (∀ d, In d ds → d = DStep) ∧
    <(st, m, false, ds)> =[ c ]=> <(st', m', false, os)>.

Lemma cteval_equiv_seq_spec_eval : ∀ c st m st' m' os,
  seq_spec_eval c st m st' m' os ↔ <(st, m)> =[ c ]=> <(st', m', os)>.
Proof.
  intros c st m st' m' os. unfold seq_spec_eval. split; intros H.
  -
    destruct H as [ds [Hstep Heval] ].
    induction Heval; try (now econstructor; eauto).
    +
      eapply CTE_Seq.
      × eapply IHHeval1. intros d HdIn.
        assert (L: In d ds1 ∨ In d ds2) by (left; assumption).
        eapply in_or_app in L. eapply Hstep in L. assumption.
      × eapply IHHeval2. intros d HdIn.
        assert (L: In d ds1 ∨ In d ds2) by (right; assumption).
        eapply in_or_app in L. eapply Hstep in L. assumption.
    +
      eapply CTE_If. destruct (eval st be) eqn:Eqbe.
      × eapply IHHeval. intros d HdIn.
        apply (in_cons DStep d) in HdIn.
        apply Hstep in HdIn. assumption.
      × eapply IHHeval. intros d HdIn.
        apply (in_cons DStep d) in HdIn.
        apply Hstep in HdIn. assumption.
    +
      exfalso.
      assert (L: ~(DForce = DStep)) by discriminate.
      apply L. apply (Hstep DForce). apply in_eq.
    +
      exfalso.
      assert (L: ~(DLoad a' i' = DStep)) by discriminate.
      apply L. apply (Hstep (DLoad a' i')). apply in_eq.
    +
      exfalso.
      assert (L: ~(DStore a' i' = DStep)) by discriminate.
      apply L. apply (Hstep (DStore a' i')). apply in_eq.
  -
    induction H.
    +
      ∃ []; split; [| eapply Spec_Skip].
      simpl. intros d Contra; destruct Contra.
```

```
  + ∃ []; split; [| eapply Spec_Asgn; assumption].
    simpl. intros d Contra; destruct Contra.
  + destruct IHcteval1 as [ds1 [Hds1 Heval1] ].
    destruct IHcteval2 as [ds2 [Hds2 Heval2] ].
    ∃ (ds1 ++ ds2). split; [| eapply Spec_Seq; eassumption].
    intros d HdIn. apply in_app_or in HdIn.
    destruct HdIn as [Hin1 | Hin2].
    × apply Hds1 in Hin1. assumption.
    × apply Hds2 in Hin2. assumption.
  + destruct IHcteval as [ds [Hds Heval] ].
    ∃ (DStep :: ds). split.
    × intros d HdIn. apply in_inv in HdIn.
      destruct HdIn as [Heq | HIn];
        [symmetry; assumption | apply Hds; assumption].
    × subst c. eapply Spec_If. eauto.
  + destruct IHcteval as [ds [Hds Heval] ].
    ∃ ds. split; [assumption |].
    eapply Spec_While; assumption.
  + ∃ [DStep]. split.
    × simpl. intros d HdIn.
      destruct HdIn as [Heq | Contra]; [| destruct Contra].
      symmetry. assumption.
    × eapply Spec_ALoad; assumption.
  + ∃ [DStep]. split.
    × simpl. intros d HdIn.
      destruct HdIn as [Heq | Contra]; [| destruct Contra].
      symmetry. assumption.
    × eapply Spec_AStore; assumption.
Qed.
```

Exercise: 1 star, standard (ct_well_typed_seq_spec_eval_ct_secure) Lemma ct_well_typed_seq_s
:

```
∀ P PA c st1 st2 m1 m2 st1' st2' m1' m2' os1 os2,
P ;; PA ⊢ct- c →
pub_equiv P st1 st2 →
pub_equiv PA m1 m2 →
```

```
  seq_spec_eval c st1 m1 st1' m1' os1 →
  seq_spec_eval c st2 m2 st2' m2' os2 →
  os1 = os2.
Proof.
  Admitted.
```
□

9.3.4 Selective SLH transformation

Now how can we make CCT programs secure against speculative execution attacks? It turns out that we can protect such programs against Spectre v1 by doing only two things:

- Keep track of a misspeculation flag using constant-time conditionals;
- Use this flag to mask the value of misspeculated public loads.

We implement this as a *Selective Speculative Load Hardening* (SLH) transformation that we will show enforces speculative constant-time security for all CCT programs.

This SLH transformation is "selective", since it only masks *public* loads. A non-selective SLH transformation was invented in LLVM, but what they implement is anyway much more complicated.

```
Definition msf : string := "msf".

Fixpoint sel_slh (P:pub_vars) (c:com) :=
  match c with
  | <{{skip}}> ⇒ <{{skip}}>
  | <{{x := e}}> ⇒ <{{x := e}}>
  | <{{c1; c2}}> ⇒ <{{sel_slh P c1; sel_slh P c2}}>
  | <{{if be then c1 else c2 end}}> ⇒
      <{{if be then msf := (be ? msf : 1); sel_slh P c1
                else msf := (be ? 1 : msf); sel_slh P c2 end}}>
  | <{{while be do c end}}> ⇒
      <{{while be do msf := (be ? msf : 1); sel_slh P c end;
         msf := (be ? 1 : msf)}}>
  | <{{x ← a[[i]]}}> ⇒
      if P x then <{{x ← a[[i]]; x := (msf ≠ 0) ? 0 : x}}>
             else <{{x ← a[[i]]}}>
  | <{{a[i] ← e}}> ⇒ <{{a[i] ← e}}>
  end.

Print spec_insecure_prog.

Definition sel_slh_spec_insecure_prog :=
<{{ if Y < 3 then
      msf := ((Y < 3) ? msf : 1);
```

```
    (X ← AP[[Y]]; X := (msf ≠ 0) ? 0 : X);
    if X ≤ 5 then
      msf := ((X ≤ 5) ? msf : 1);
      X := 5
    else msf := ((X ≤ 5) ? 1 : msf); skip end
  else msf := ((Y < 3) ? 1 : msf); skip end }}>.

Lemma sel_slh_spec_insecure_prog_check:
  sel_slh XYZpub spec_insecure_prog = sel_slh_spec_insecure_prog.
Proof. reflexivity. Qed.
```

When misspeculation occurs in the first condition if $Z < 1$, the transformation detects this misspeculation and sets msf (misspeculation flag) to 1. Then, although the secret value gets loaded into X via the out-of-bounds access $X \leftarrow AP[[Z]]$, it is immediatly overwritten with 0 due to the masking code $X := (msf \neq 0)\ ?\ 0 : X$ that follows. As a result, all subsequent operations like if $X \leq 5$ only uses the masked value 0 instead of the actual secret.

9.3.5 Main proof idea: use compiler correctness wrt ideal semantics

To prove this transformation secure, Spectre Declassified uses an ideal semantics, capturing selective speculative load hardening more abstractly. The proof effort is decomposed into:

- a speculative constant-time proof for the ideal semantics;
- a compiler correctness proof for the sel_slh transformation, taking source programs which are executed using the ideal semantics, to target programs executed using the speculative semantics.

In a little bit more detail, we're intuitively trying to prove:
forall P PA c, P;;PA |-ct- c -> spec_ct_secure P PA (sel_slh P c),
where the conclusion looks as follows:

```
forall st1 st2 m1 m2 st1' st2' m1' m2' b1' b2' os1 os2 ds,
  pub_equiv P st1 st2 ->
  pub_equiv PA m1 m2 ->
  <(st1,m1,false,ds)> =[ sel_slh P c ]=> <(st1',m1',b1',os1)> ->
  <(st2,m2,false,ds)> =[ sel_slh P c ]=> <(st2',m2',b2',os2)> ->
  os1 = os2

    Compiler correctness allows us to get rid of [sel_slh P c] in the premises
    and instead get an execution in terms of the ideal semantics:

  <(st,m,b,ds)> =[ sel_slh P c ]=> <(st',m',b',os)> ->
    P |-i <(st,m,b,ds)> =[ c ]=> <(msf!->st msf;st',m',b',os)>
```

```
*)

(** One thing to note is that the ideal semantics doesn't track misspeculation
    in the [msf] variable, but instead directly uses the misspeculation bit in
    the speculative semantics for masking. This allows us to keep the ideal
    semantics simple, and then we show that [msf] correctly tracks misspeculation
    in our compiler correctness proof . *)

(* ================================================================= *)
(** ** Ideal semantics definition *)

(** All rules of the ideal semantics are the same as for the speculative
    semantics, except the ones for array loads, which add the extra
    masking done by [sel_slh] on top of the speculative semantics.

      eval st ie = i      i < length(m a)
 ------------------------------------------------- (Ideal_ALoad)
let v := if b && P x then 0 else nth i (m a) 0 in
P |-i <(st, m, b, [DStep])> =[ x <- a[[ie]] ]=>
      <(x !-> v; st, m, b, [OALoad a i])>

eval st ie = i   i >= length(m a)   i' < length(m a')
------------------------------------------------------------ (Ideal_ALoad_U)
let v := if P x then 0 else nth i' (m a') 0 in
P |-i <(st, m, true, [DLoad a' i'])> =[ x <- a[[ie]] ]=>
      <(x !-> v; st, m, true, [OALoad a i])>
*)

Reserved Notation
  "P '|-i' '<(' st , m , b , ds ')>' '=[' c ']=>' '<(' stt , mt , bb , os ')>'"
  (at level 40, c custom com at level 99,
   st constr, m constr, stt constr, mt constr at next level).

Inductive ideal_eval (P:pub_vars) :
    com -> state -> mem -> bool -> dirs ->
           state -> mem -> bool -> obs -> Prop :=
  | Ideal_Skip : forall st m b,
      P |-i <(st, m, b, [])> =[ skip ]=> <(st, m, b, [])>
  | Ideal_Asgn  : forall st m b e n x,
      eval st e = n ->
      P |-i <(st, m, b, [])> =[ x := e ]=> <(x !-> n; st, m, b, [])>
  | Ideal_Seq : forall c1 c2 st m b st' m' b' st'' m'' b'' os1 os2 ds1 ds2,
```

```
    P |-i <(st, m, b, ds1)> =[ c1 ]=> <(st', m', b', os1)>  ->
    P |-i <(st', m', b', ds2)> =[ c2 ]=> <(st'', m'', b'', os2)> ->
    P |-i <(st, m, b, ds1++ds2)>  =[ c1 ; c2 ]=> <(st'', m'', b'', os1++os2)>
| Ideal_If : forall st m b st' m' b' be c1 c2 os1 ds,
    let c := (if (not_zero (eval st be)) then c1 else c2) in
    P |-i <(st, m, b, ds)> =[ c ]=> <(st', m', b', os1)> ->
    P |-i <(st, m, b, DStep :: ds)> =[ if be then c1 else c2 end ]=>
      <(st', m', b', [OBranch (not_zero (eval st be))] ++ os1 )>
| Ideal_If_F : forall st m b st' m' b' be c1 c2 os1 ds,
    let c := (if (not_zero (eval st be)) then c2 else c1) in (* <-- branches swapped *
    P |-i <(st, m, true, ds)> =[ c ]=> <(st', m', b', os1)> ->
    P |-i <(st, m, b, DForce :: ds)> =[ if be then c1 else c2 end ]=>
      <(st', m', b', [OBranch (not_zero (eval st be))] ++ os1)>
| Ideal_While : forall be st m b ds st' m' b' os c,
    P |-i <(st, m, b, ds)> =[ if be then c; while be do c end else skip end ]=>
      <(st', m', b', os)> ->
    P |-i <(st, m, b, ds)> =[ while be do c end ]=> <(st', m', b', os)>
| Ideal_ALoad : forall st m b x a ie i,
    eval st ie = i ->
    i < length (m a) ->
    let v := if b && P x then 0 else nth i (m a) 0 in
    P |-i <(st, m, b, [DStep])> =[ x <- a[[ie]] ]=>
      <(x !-> v; st, m, b, [OALoad a i])>
| Ideal_ALoad_U : forall st m x a ie i a' i',
    eval st ie = i ->
    i >= length (m a) ->
    i' < length (m a') ->
    let v := if P x then 0 else nth i' (m a') 0 in
    P |-i <(st, m, true, [DLoad a' i'])> =[ x <- a[[ie]] ]=>
      <(x !-> v; st, m, true, [OALoad a i])>
| Ideal_AStore : forall st m b a ie i e n,
    eval st e = n ->
    eval st ie = i ->
    i < length (m a) ->
    P |-i <(st, m, b, [DStep])> =[ a[ie] <- e ]=>
      <(st, a !-> upd i (m a) n; m, b, [OAStore a i])>
| Ideal_AStore_U : forall st m a ie i e n a' i',
    eval st e = n ->
    eval st ie = i ->
    i >= length (m a) ->
    i' < length (m a') ->
    P |-i <(st, m, true, [DStore a' i'])> =[ a[ie] <- e ]=>
```

```
        <(st, a' !-> upd i' (m a') n; m, true, [OAStore a i])>

  where "P |-i <( st , m , b , ds )> =[ c ]=> <( stt , mt , bb , os )>" :=
    (ideal_eval P c st m b ds stt mt bb os).

Hint Constructors ideal_eval : core.

(* ================================================================= *)
(** ** Ideal semantics enforces speculative constant-time *)

(** Let's now prove that the ideal semantics does enforce speculative
    constant-time.  As in the proofs we did before for constant-time and CF
    security, we rely on a proof of noninterference. For our ideal semantics
    this noninterference proof requires interesting generalization of the
    induction hypothesis (see [ct_well_typed_ideal_noninterferent_general]). *)

(** Generalization 1: We need to also deal with executions ending with [b=true],
    but in that case we cannot ensure that the array states are publicly
    equivalent, since our selective SLH does not mask misspeculated stores (for
    efficiency, since it's not needed for security). This requires to generalize
    the [pub_equiv PA m1 m2] premise of our statements too. *)

(** Generalization 2: To show that the two executions run in lock-step the proof
    uses not only the CCT type system (not branching on secrets) but also the
    fact that the directions are the same, which we need to establish as an
    extra invariant though. *)

Definition prefix {X:Type} (xs ys : list X) : Prop :=
  exists zs, xs ++ zs = ys.

Lemma prefix_refl : forall {X:Type} {ds : list X},
  prefix ds ds.
Proof. intros X ds. exists []. apply app_nil_r. Qed.

Lemma prefix_nil : forall {X:Type} (ds : list X),
  prefix [] ds.
Proof. intros X ds. unfold prefix. eexists. simpl. reflexivity. Qed.

Lemma prefix_heads_and_tails : forall {X:Type} (h1 h2 : X) (t1 t2 : list X),
  prefix (h1::t1) (h2::t2) -> h1 = h2 /\ prefix t1 t2.
Proof.
  intros X h1 h2 t1 t2. unfold prefix. intros Hpre.
```

```
  destruct Hpre as [zs Hpre]; simpl in Hpre.
  inversion Hpre; subst. eauto.
Qed.

Lemma prefix_heads : forall {X:Type} (h1 h2 : X) (t1 t2 : list X),
  prefix (h1::t1) (h2::t2) -> h1 = h2.
Proof.
  intros X h1 h2 t1 t2 H. apply prefix_heads_and_tails in H; tauto.
Qed.

Lemma prefix_or_heads : forall {X:Type} (x y : X) (xs ys : list X),
  prefix (x :: xs) (y :: ys) \/ prefix (y :: ys) (x :: xs) ->
  x = y.
Proof.
  intros X x y xs ys H.
  destruct H as [H | H]; apply prefix_heads in H; congruence.
Qed.

Lemma prefix_cons : forall {X:Type} (d :X) (ds1 ds2: list X),
 prefix ds1 ds2 <->
 prefix (d::ds1) (d::ds2).
Proof.
  intros X d ds1 ds2. split; [unfold prefix| ]; intros H.
  - destruct H; subst.
    eexists; simpl; eauto.
  - apply prefix_heads_and_tails in H. destruct H as [_ H]. assumption.
Qed.

Lemma prefix_app : forall {X:Type} {ds1 ds2 ds0 ds3 : list X},
  prefix (ds1 ++ ds2) (ds0 ++ ds3) ->
  prefix ds1 ds0 \/ prefix ds0 ds1.
Proof.
  intros X ds1. induction ds1 as [| d1 ds1' IH]; intros ds2 ds0 ds3 H.
  - left. apply prefix_nil.
  - destruct ds0 as [| d0 ds0'] eqn:D0.
    + right. apply prefix_nil.
    + simpl in H; apply prefix_heads_and_tails in H.
      destruct H as [Heq Hpre]; subst.
      apply IH in Hpre; destruct Hpre; [left | right];
      apply prefix_cons; assumption.
Qed.
```

```
Lemma prefix_append_front : forall {X:Type} {ds1 ds2 ds3 : list X},
  prefix (ds1 ++ ds2) (ds1 ++ ds3) ->
  prefix ds2 ds3.
Proof.
  intros X ds1. induction ds1 as [| d1 ds1' IH]; intros ds2 ds3 H.
  - auto.
  - simpl in H; apply prefix_cons in H. apply IH in H. assumption.
Qed.

Lemma app_eq_prefix : forall {X:Type} {ds1 ds2 ds1' ds2' : list X},
  ds1 ++ ds2 = ds1' ++ ds2' ->
  prefix ds1 ds1' \/ prefix ds1' ds1.
Proof.
  intros X ds1. induction ds1 as [| h1 t1 IH]; intros ds2 ds1' ds2' H.
  - left. apply prefix_nil.
  - destruct ds1' as [| h1' t1'] eqn:D1'.
    + right. apply prefix_nil.
    + simpl in H; inversion H; subst.
      apply IH in H2. destruct H2 as [HL | HR];
      [left | right]; apply prefix_cons; auto.
Qed.

Ltac split4 := split; [|split; [| split] ].

Lemma ct_well_typed_ideal_noninterferent_general : forall P PA c,
  forall st1 st2 m1 m2 b st1' st2' m1' m2' b1' b2' os1 os2 ds1 ds2,
    P ;; PA |-ct- c ->
    pub_equiv P st1 st2 ->
    (b = false -> pub_equiv PA m1 m2) -> (* Generalization 1 *)
    (prefix ds1 ds2 \/ prefix ds2 ds1) -> (* <- Generalization 2 *)
    P |-i <(st1, m1, b, ds1)> =[ c ]=> <(st1', m1', b1', os1)> ->
    P |-i <(st2, m2, b, ds2)> =[ c ]=> <(st2', m2', b2', os2)> ->
    pub_equiv P st1' st2' /\ b1' = b2' /\
      (b1' = false -> pub_equiv PA m1' m2') /\ (* <- Generalization 1 *)
      ds1 = ds2.  (* <- Generalization 2 *)
Proof.
  intros P PA c st1 st2 m1 m2 b st1' st2' m1' m2' b1' b2' os1 os2 ds1 ds2
    Hwt Heq Haeq Hds Heval1 Heval2.
  generalize dependent st2'. generalize dependent st2.
  generalize dependent m2'. generalize dependent m2.
  generalize dependent os2. generalize dependent b2'.
  generalize dependent ds2.
```

```
induction Heval1; intros ds2X Hds b2' os2' a2 Haeq a2' s2 Heq s2' Heval2;
  inversion Heval2; inversion Hwt; subst.
- (* Skip *) auto.
- (* Asgn *) split4; auto.
  destruct (P x) eqn:EqPx.
  + eapply pub_equiv_update_public; eauto.
    eapply noninterferent_exp; eauto.
    destruct l; [auto | simpl in H14; discriminate].
  + eapply pub_equiv_update_secret; eauto.
- (* Seq *)
  destruct Hds as [Hpre | Hpre]; apply prefix_app in Hpre as Hds1.
  + (* prefix (ds1 ++ ds2) (ds0 ++ ds3) *)
    eapply IHHeval1_1 in Hds1; eauto.
    destruct Hds1 as [ Hstates [Hbits [Hmates Hdirections] ] ]. subst.
    eapply prefix_append_front in Hpre as Hds2.
    eapply IHHeval1_2 in H14; eauto. firstorder. subst. reflexivity.
  + (* prefix (ds0 ++ ds3) (ds1 ++ ds2) *)
    eapply IHHeval1_1 with (ds2:=ds0) in H13; eauto; [| tauto].
    destruct H13 as [ Hstates [Hbits [Hmates Hdirections] ] ]. subst.
    eapply prefix_append_front in Hpre as Hds2.
    eapply IHHeval1_2 in H14; eauto. firstorder; subst; reflexivity.
- (* If *)
  remember (if not_zero (eval st be) then c1 else c2) as c5.
  assert(G : P ;; PA |-ct- c5).
  { subst c5. destruct (eval st be); assumption. }
  assert(Gds : prefix ds ds0 \/ prefix ds0 ds).
  { destruct Hds as [Hds | Hds]; apply prefix_cons in Hds; tauto. }
  subst c4 c5. erewrite noninterferent_exp in H10.
  + specialize (IHHeval1 G _ Gds _ _ _ Haeq _ _ Heq _ H10).
    firstorder; congruence.
  + apply pub_equiv_sym. eassumption.
  + eassumption.
- (* IF; contra *)
  apply prefix_or_heads in Hds; inversion Hds.
- (* IF; contra *)
   apply prefix_or_heads in Hds; inversion Hds.
- (* If_F; analog to If *)
  remember (if not_zero (eval st be) then c2 else c1) as c5.
  assert(G : P ;; PA |-ct- c5).
  { subst c5. destruct (eval st be); assumption. }
  assert(Gds : prefix ds ds0 \/ prefix ds0 ds).
  { destruct Hds as [Hds | Hds]; apply prefix_cons in Hds; tauto. }
```

```
  subst c4 c5. erewrite noninterferent_exp in H10.
  + assert(GG: true = false -> pub_equiv PA m a2). (* <- only difference *)
    { intro Hc. discriminate. }
    specialize (IHHeval1 G _ Gds _ _ _ GG _ _ Heq _ H10).
    firstorder; congruence.
  + apply pub_equiv_sym. eassumption.
  + eassumption.
- (* While *) eapply IHHeval1; try eassumption. repeat constructor; eassumption.
- (* ALoad *) split4; eauto.
  destruct (P x) eqn:EqPx; simpl.
  + eapply pub_equiv_update_public; eauto.
    destruct b2' eqn:Eqb2'; simpl; [reflexivity |].
    unfold can_flow in H18. eapply orb_true_iff in H18.
    destruct H18 as [Hapub | Contra]; [| simpl in Contra; discriminate].
    subst v v1 v2. eapply Haeq in Hapub; [| reflexivity]. rewrite Hapub.
    eapply noninterferent_exp in Heq; eauto. rewrite Heq.
    reflexivity.
  + eapply pub_equiv_update_secret; eauto.
- (* ALoad_U *)
  split4; eauto.
  + destruct (P x) eqn:EqPx.
    * simpl. eapply pub_equiv_update_public; eauto.
    * eapply pub_equiv_update_secret; eauto.
  + apply prefix_or_heads in Hds. inversion Hds.
- (* ALoad *)
  split4; eauto.
  + destruct (P x) eqn:EqPx.
    * eapply pub_equiv_update_public; eauto.
    * eapply pub_equiv_update_secret; eauto.
  + apply prefix_or_heads in Hds. inversion Hds.
- (* ALoad_U *)
  split4; eauto.
  + destruct (P x) eqn:EqPx.
    * eapply pub_equiv_update_public; eauto.
    * eapply pub_equiv_update_secret; eauto.
  + apply prefix_or_heads in Hds. inversion Hds. reflexivity.
- (* AStore *)
  split4; eauto. intro Hb2'.
  destruct (PA a) eqn:EqPAa.
  + eapply pub_equiv_update_public; eauto.
    destruct l eqn:Eql.
    * eapply noninterferent_exp in H19, H20; eauto. rewrite H19, H20.
```

```
        apply Haeq in Hb2'. apply Hb2' in EqPAa. rewrite EqPAa. reflexivity.
      * simpl in H21. discriminate.
    + eapply pub_equiv_update_secret; eauto.
  - (* AStore_U; contra *) apply prefix_or_heads in Hds. inversion Hds.
  - (* AStore; contra *) apply prefix_or_heads in Hds. inversion Hds.
  - (* AStore_U; contra *)
    split4; eauto.
    + intro contra. discriminate contra.
    + apply prefix_or_heads in Hds. inversion Hds. reflexivity.
Qed.

Corollary ct_well_typed_ideal_noninterferent :
  forall P PA c st1 st2 m1 m2 b st1' st2' m1' m2' b1' b2' os1 os2 ds,
    P ;; PA |-ct- c ->
    pub_equiv P st1 st2 ->
    (b = false -> pub_equiv PA m1 m2) ->
    P |-i <(st1, m1, b, ds)> =[ c ]=> <(st1', m1', b1', os1)> ->
    P |-i <(st2, m2, b, ds)> =[ c ]=> <(st2', m2', b2', os2)> ->
    pub_equiv P st1' st2' /\ b1' = b2' /\ (b1' = false -> pub_equiv PA m1' m2').
Proof.
  intros P PA c st1 st2 m1 m2 b st1' st2' m1' m2' b1' b2' os1 os2 ds
    Hwt Heq Haeq Heval1 Heval2.
  eapply ct_well_typed_ideal_noninterferent_general in Heval2; eauto; try tauto.
  left. apply prefix_refl.
Qed.

(** This corollary (used below in the sequence case) also follows from
    [noninterferent_general] *)
Corollary aux : forall P PA st1 st2 m1 m2 b ds1 ds2 c st1' st2' m1' m2' b1 b2 os1 os2 ds
  ds2 ++ ds2' = ds1 ++ ds1' ->
  P ;; PA |-ct- c ->
  pub_equiv P st1 st2 ->
  (b = false -> pub_equiv PA m1 m2) ->
  P |-i <(st1, m1, b, ds1)> =[ c ]=> <(st1', m1', b1, os1)>  ->
  P |-i <(st2, m2, b, ds2)> =[ c ]=> <(st2', m2', b2, os2)> ->
  ds1 = ds2 /\ ds1' = ds2'.
Proof.
  intros P PA st1 st2 m1 m2 b ds1 ds2 c st1' st2' m1' m2' b1 b2 os1 os2 ds1' ds2'
    Hds Hwt Heq Haeq Heval1 Heval2.
  pose proof Hds as H.
  symmetry in H.
  apply app_eq_prefix in H.
```

```
  eapply ct_well_typed_ideal_noninterferent_general in H;
    [ | | | | apply Heval1 | apply Heval2]; try eassumption.
  - destruct H as [ _ [ _ [ _ H] ] ]. subst. split; [reflexivity|].
    apply app_inv_head in Hds. congruence.
Qed.

Theorem ideal_spec_ct_secure :
  forall P PA c st1 st2 m1 m2 b st1' st2' m1' m2' b1' b2' os1 os2 ds,
    P ;; PA |-ct- c ->
    pub_equiv P st1 st2 ->
    (b = false -> pub_equiv PA m1 m2) ->
    P |-i <(st1, m1, b, ds)> =[ c ]=> <(st1', m1', b1', os1)> ->
    P |-i <(st2, m2, b, ds)> =[ c ]=> <(st2', m2', b2', os2)> ->
    os1 = os2.
Proof.
  intros P PA c st1 st2 m1 m2 b st1' st2' m1' m2' b1' b2' os1 os2 ds
    Hwt Heq Haeq Heval1 Heval2.
  generalize dependent st2'. generalize dependent st2.
  generalize dependent m2'. generalize dependent m2.
  generalize dependent os2. generalize dependent b2'.
  induction Heval1; intros b2' os2' m2 Haeq m2' st2 Heq st2' Heval2;
    inversion Heval2; inversion Hwt; subst.
  - (* Skip *) reflexivity.
  - (* Skip *) reflexivity.
  - (* Seq *)
    eapply aux in H1; [| | | | apply Heval1_1 | apply H5 ]; eauto.
    destruct H1 as [H1 H1']. subst.
    assert(NI1 : pub_equiv P st' st'0 /\ b' = b'0 /\ (b' = false -> pub_equiv PA m' m'0)
    { eapply ct_well_typed_ideal_noninterferent; [ | | | eassumption | eassumption]; eau
    destruct NI1 as [NI1eq [NIb NIaeq] ]. subst.
    erewrite IHHeval1_2; [erewrite IHHeval1_1 | | | |];
      try reflexivity; try eassumption.
  - (* If *)
    f_equal.
    + f_equal. eapply noninterferent_exp in Heq; [| eassumption].
      rewrite Heq. reflexivity.
    + eapply IHHeval1; try eassumption; try (destruct (eval st be); eassumption).
      subst c c4. erewrite (noninterferent_exp Heq H14); eassumption.
  - (* If_F *)
    f_equal.
    + f_equal. eapply noninterferent_exp in Heq; [| eassumption].
      rewrite Heq. reflexivity.
```

```
    + eapply IHHeval1; try eassumption; try (destruct (eval st be); eassumption).
      * intro contra. discriminate contra.
      * subst c c4. erewrite noninterferent_exp; eassumption.
  - (* While *) eapply IHHeval1; eauto.
  - (* ALoad *) f_equal. f_equal. eapply noninterferent_exp; eassumption.
  - (* ALoad_U *) f_equal. f_equal. eapply noninterferent_exp; eassumption.
  - (* AStore *) f_equal. f_equal. eapply noninterferent_exp; eassumption.
  - (* AStore *) f_equal. f_equal. eapply noninterferent_exp; eassumption.
Qed.

(* ================================================================= *)
(** ** Correctness of sel_slh as a compiler from ideal to speculative semantics *)

(** We now prove that the ideal semantics correctly captures the programs
    produced by [sel_slh] when executed using the speculative semantics. We
    phrase this as a backwards compiler correctness proof for [sel_slh],
    which intuitively looks as follows:

    <(st,m,b,ds)> =[[ sel_slh P c ]]=> <(st',m',b',os)> ->
    P |-i <(st,m,b,ds)> =[[ c ]]=> <(msf!->st msf;st',m',b',os)>
*)

(** All results about [sel_slh] below assume that the original [c] doesn't
    already use the variable [msf] needed by the [sel_slh] translation. *)

Fixpoint e_unused (x:string) (e:exp) : Prop :=
  match e with
  | ANum n      => True
  | AId y       => y <> x
  | ABin _ e1 e2 => e_unused x e1 /\ e_unused x e2
  | <{b ? e1 : e2}> => e_unused x b /\ e_unused x e1 /\ e_unused x e2
  end.

Fixpoint unused (x:string) (c:com) : Prop :=
  match c with
  | <{{skip}}> => True
  | <{{y := e}}> => y <> x /\ e_unused x e
  | <{{c1; c2}}> => unused x c1 /\ unused x c2
  | <{{if be then c1 else c2 end}}> =>
      e_unused x be /\ unused x c1 /\ unused x c2
  | <{{while be do c end}}> => e_unused x be /\ unused x c
  | <{{y <- a[[i]]}}> => y <> x /\ e_unused x i
```

```
  | <{{a[i] <- e}}> => e_unused x i /\ e_unused x e
  end.

(** As a warm-up we prove that [sel_slh] properly updates the variable msf. *)

(** Proving this by induction on [com] or [spec_eval] leads to induction
    hypotheses, that are not strong enough to prove the [Spec_While]
    case. Therefore we will prove it by induction on the [size]
    of a the pair of the [(c:com)] and the [(ds:dirs)]. *)

Fixpoint com_size (c:com) : nat :=
  match c with
  | <{{ c1; c2 }}> => 1 + (com_size c1) + (com_size c2)
  | <{{ if be then ct else cf end }}> => 1 + max (com_size ct) (com_size cf)
  | <{{ while be do cw end }}> => 1 + (com_size cw)
  | <{{ skip }}> => 1
  | _  => 1
  end.

Definition size (c:com) (ds:dirs) : nat := com_size c + length ds.

(** We prove a helpful induction principle on [size]: *)

Check measure_induction.

Lemma size_ind : forall P : com -> dirs -> Prop,
  (forall c ds,
    (forall c' ds', size c' ds' < size c ds -> P c' ds') ->
    P c ds) ->
  (forall c ds, P c ds).
Proof.
  intros.
  remember (fun c_ds => P (fst c_ds) (snd c_ds)) as P'.
  replace (P c ds) with (P' (c, ds)) by now rewrite HeqP'.
  eapply measure_induction with (f:=fun c_ds => size (fst c_ds) (snd c_ds)).
  intros. rewrite HeqP'.
  apply H. intros.
  remember (c', ds') as c_ds'.
  replace (P c' ds') with (P' c_ds') by now rewrite Heqc_ds', HeqP'.
  apply H0. now rewrite Heqc_ds'.
Qed.
```

```
(** The proof of [sel_slh_flag] *)

Lemma size_decreasing: forall c1 ds1 c2 ds2,
  (com_size c1 < com_size c2 /\ length ds1 <= length ds2 ) \/
  (com_size c1 <= com_size c2 /\ length ds1 < length ds2) ->
  size c1 ds1 < size c2 ds2.
Proof.
  intros c1 ds1 c2 ds2 [ [Hcom Hdir] | [Hcom Hdir] ];
  unfold size; lia.
Qed.

(** Based on the Lemma [size_decreasing] we can build a tactic to solve
    the subgoals in the form of [size c' ds' < size c ds],
    which will be produced by [size_ind].*)

Ltac size_auto :=
  try ( apply size_decreasing; left; split; simpl;
        [| repeat rewrite length_app]; lia );
  try ( apply size_decreasing; right; split; simpl;
        [| repeat rewrite length_app]; lia);
  try ( apply size_decreasing; left; split; simpl;
        [auto | repeat rewrite length_app; lia] ).

(** To properly apply [size_ind], we need to state [sel_slh_flag]
    as a proposition of type [com -> dirs -> Prop]. Therefore we define the
    following: *)

Definition sel_slh_flag_prop (c :com) (ds :dirs) :Prop :=
  forall P st m (b:bool) st' m' (b':bool) os,
  unused msf c ->
  st msf = (if b then 1 else 0) ->
  <(st, m, b, ds)> =[ sel_slh P c ]=> <(st', m', b', os)> ->
  st' msf = (if b' then 1 else 0).

Lemma sel_slh_flag : forall c ds,
  sel_slh_flag_prop c ds.
Proof.
  eapply size_ind. unfold sel_slh_flag_prop.
  intros c ds IH P st m b st' m' b' os Hunused Hstb Heval.
  destruct c; simpl in *; try (now inversion Heval; subst; eauto).
  - (* Asgn *)
    inversion Heval; subst. rewrite t_update_neq; tauto.
```

```
- (* Seq *)
  inversion Heval; subst; clear Heval.
  apply IH in H1; try tauto.
  + apply IH in H10; try tauto. size_auto.
  + size_auto.
- (* IF *)
  inversion Heval; subst; clear Heval.
  + (* Spec_If *)
    destruct (eval st be) eqn:Eqnbe.
    * inversion H10; subst; clear H10.
      inversion H1; subst; clear H1.
      apply IH in H11; try tauto.
      { size_auto. }
      { rewrite t_update_eq. simpl. rewrite Eqnbe. assumption. }
    * (* analog to true case *)
      inversion H10; subst; clear H10.
      inversion H1; subst; clear H1.
      apply IH in H11.
      { auto. }
      { size_auto. }
      { tauto. }
      { rewrite t_update_eq. simpl. rewrite Eqnbe. assumption. }
  + (* Spec_If_F; analog to Spec_If case *)
    destruct (eval st be) eqn:Eqnbe.
    * inversion H10; subst; clear H10.
      inversion H1; subst; clear H1.
      apply IH in H11; try tauto.
      { size_auto. }
      { rewrite t_update_eq. simpl. rewrite Eqnbe. simpl. reflexivity. }
    * inversion H10; subst; clear H10.
      inversion H1; subst; clear H1.
      apply IH in H11; try tauto.
      { size_auto. }
      { rewrite t_update_eq. simpl. rewrite Eqnbe. simpl. reflexivity. }
- (* While *)
    inversion Heval; subst; clear Heval.
    inversion H1; subst; clear H1.
    inversion H11; subst; clear H11.
    + (* non-speculative *)
      destruct (eval st be) eqn:Eqnbe.
      * inversion H12; subst; clear H12.
        inversion H10; subst; simpl.
```

```
          rewrite t_update_eq, Eqnbe; simpl. assumption.
        * inversion H12; subst; clear H12.
          assert(Hwhile: <(st'1, m'1, b'1, (ds0 ++ ds2)%list)>
              =[ sel_slh P <{{while be do c end}}> ]=> <(st', m', b', (os3++os2)%list)>
          { simpl. eapply Spec_Seq; eassumption. }
          apply IH in Hwhile; eauto.
          { size_auto. }
          { clear Hwhile; clear H11.
            inversion H1; subst; clear H1.
            inversion H2; subst; clear H2. simpl in H12.
            apply IH in H12; try tauto.
            - size_auto.
            - rewrite t_update_eq, Eqnbe; simpl. assumption. }
      + (* speculative; analog to non_speculative case *)
        destruct (eval st be) eqn:Eqnbe.
        * inversion H12; subst; clear H12.
          assert(Hwhile: <(st'1, m'1, b'1, (ds0 ++ ds2)%list)>
              =[sel_slh P <{{while be do c end}}>]=> <(st', m', b', (os3++os2)%list )>).
          { simpl. eapply Spec_Seq; eassumption. }
          apply IH in Hwhile; eauto.
          { size_auto. }
          { clear Hwhile; clear H11.
            inversion H1; subst; clear H1.
            inversion H2; subst; clear H2. simpl in H12.
            apply IH in H12; try tauto.
            - size_auto.
            - rewrite t_update_eq, Eqnbe; simpl. reflexivity. }
        * inversion H12; subst; clear H12.
          inversion H10; subst; simpl.
          rewrite t_update_eq, Eqnbe; simpl. reflexivity.
  - (* ALoad *)
    destruct (P x) eqn:Eqnbe.
    + inversion Heval; subst; clear Heval.
      inversion H10; subst; clear H10.
      rewrite t_update_neq; [| tauto].
      inversion H1; subst;
      try (rewrite t_update_neq; [assumption| tauto]).
    + inversion Heval; subst;
      try (rewrite t_update_neq; [assumption| tauto]).
Qed.

(** We need a few more lemmas before we prove backwards compiler correctness *)
```

```
Lemma eval_unused_update : forall X st n,
  (forall ae, e_unused X ae ->
    eval (X !-> n; st) ae = eval st ae).
Proof.
  intros X st n. induction ae; intros; simpl in *; try reflexivity.
  - rewrite t_update_neq; eauto.
  - destruct H.
    rewrite IHae1; [| tauto]. rewrite IHae2; [| tauto].
    reflexivity.
  - destruct H. destruct H0.
    rewrite IHae1, IHae2, IHae3; auto.
Qed.

Lemma ideal_unused_overwrite: forall P st m b ds c st' m' b' os X n,
  unused X c ->
  P |-i <(st, m, b, ds)> =[ c ]=> <(st', m', b', os)> ->
  P |-i <(X !-> n; st, m, b, ds)> =[ c ]=> <(X !-> n; st', m', b', os)>.
Proof.
  intros P st m b ds c st' m' b' os X n Hu H.
  induction H; simpl in Hu.
  - (* Skip *) econstructor.
  - (* Asgn *)
    rewrite t_update_permute; [| tauto].
    econstructor. rewrite eval_unused_update; tauto.
  - (* Seq *)
    econstructor.
    + apply IHideal_eval1; tauto.
    + apply IHideal_eval2; tauto.
  - (* If *)
    rewrite <- eval_unused_update with (X:=X) (n:=n); [| tauto].
    econstructor.
    rewrite eval_unused_update; [ | tauto].
    destruct (eval st be) eqn:D; apply IHideal_eval; tauto.
  - (* If_F *)
    rewrite <- eval_unused_update with (X:=X) (n:=n); [| tauto].
    econstructor.
    rewrite eval_unused_update; [ | tauto].
    destruct (eval st be) eqn:D; apply IHideal_eval; tauto.
  - (* While *)
    econstructor. apply IHideal_eval. simpl; tauto.
  - (* ALoad *)
```

```
    rewrite t_update_permute; [| tauto]. econstructor; [ | assumption].
    rewrite eval_unused_update; tauto.
  - (* ALoad_U *)
    rewrite t_update_permute; [| tauto]. econstructor; try assumption.
    rewrite eval_unused_update; tauto.
  - (* AStore *)
    econstructor; try assumption.
    + rewrite eval_unused_update; tauto.
    + rewrite eval_unused_update; tauto.
  - (* AStore_U *)
    econstructor; try assumption.
    + rewrite eval_unused_update; tauto.
    + rewrite eval_unused_update; tauto.
Qed.

Lemma ideal_unused_update : forall P st m b ds c st' m' b' os X n,
  unused X c ->
  P |-i <(X !-> n; st, m, b, ds)> =[ c ]=> <(X !-> n; st', m', b', os)> ->
  P |-i <(st, m, b, ds)> =[ c ]=> <(X !-> st X; st', m', b', os)>.
Proof.
  intros P st m b ds c st' m' b' os X n Hu Heval.
  eapply ideal_unused_overwrite with (X:=X) (n:=(st X)) in Heval; [| assumption].
  do 2 rewrite t_update_shadow in Heval. rewrite t_update_same in Heval. assumption.
Qed.

Lemma ideal_unused_update_rev : forall P st m b ds c st' m' b' os X n,
  unused X c ->
  P |-i <(st, m, b, ds)> =[ c ]=> <(X!-> st X; st', m', b', os)> ->
  P |-i <(X !-> n; st, m, b, ds)> =[ c ]=> <(X !-> n; st', m', b', os)>.
Proof.
  intros P st m b ds c st' m' b' os X n Hu H.
  eapply ideal_unused_overwrite in H; [| eassumption].
  rewrite t_update_shadow in H. eassumption.
Qed.

(** The backwards compiler correctness proof uses [size_ind]: *)

Definition sel_slh_compiler_correctness_prop (c:com) (ds:dirs) : Prop :=
  forall P st m (b: bool) st' m' b' os,
  unused msf c ->
  st msf = (if b then 1 else 0) ->
  <(st, m, b, ds)> =[ sel_slh P c ]=> <(st', m', b', os)> ->
```

```
  P |-i <(st, m, b, ds)> =[ c ]=> <(msf !-> st msf; st', m', b', os)>.

Lemma sel_slh_compiler_correctness : forall c ds,
  sel_slh_compiler_correctness_prop c ds.
Proof.
  apply size_ind. unfold sel_slh_compiler_correctness_prop.
  intros c ds IH P st m b st' m' b' os Hunused Hstb Heval.
  destruct c; simpl in *; inversion Heval; subst; clear Heval;
  try (destruct (P x); discriminate).
  - (* Skip *)
    rewrite t_update_same. apply Ideal_Skip.
  - (* Asgn *)
    rewrite t_update_permute; [| tauto].
    rewrite t_update_same.
    constructor. reflexivity.
  - (* Seq *)
    eapply Ideal_Seq.
    + apply IH in H1; try tauto.
      * eassumption.
      * size_auto.
    + apply sel_slh_flag in H1 as Hstb'0; try tauto.
      apply IH in H10; try tauto.
      * eapply ideal_unused_update_rev; try tauto.
      * size_auto.
  (* IF *)
  - (* non-speculative *)
    destruct (eval st be) eqn:Eqnbe; inversion H10;
    inversion H1; subst; clear H10; clear H1; simpl in *.
    + apply IH in H11; try tauto.
      * rewrite <- Eqnbe. apply Ideal_If. rewrite Eqnbe in *.
        rewrite t_update_same in H11. apply H11.
      * size_auto.
      * rewrite t_update_eq. rewrite Eqnbe. assumption.
    + (* analog to false case *)
      apply IH in H11; try tauto.
      * rewrite <- Eqnbe. apply Ideal_If. rewrite Eqnbe in *.
        rewrite t_update_same in H11. apply H11.
      * size_auto.
      * rewrite t_update_eq. rewrite Eqnbe. assumption.
  - (* speculative *)
    destruct (eval st be) eqn:Eqnbe; inversion H10; inversion H1;
    subst; simpl in *; clear H10; clear H1; rewrite Eqnbe in H11.
```

```
  + rewrite <- Eqnbe. apply Ideal_If_F. rewrite Eqnbe. apply IH in H11; try tauto.
    * rewrite t_update_eq in H11.
      apply ideal_unused_update in H11; try tauto.
    * size_auto.
  + (* analog to false case *)
    rewrite <- Eqnbe. apply Ideal_If_F. rewrite Eqnbe. apply IH in H11; try tauto.
    * rewrite t_update_eq in H11.
      apply ideal_unused_update in H11; try tauto.
    * size_auto.
- (* While *)
  eapply Ideal_While.
  inversion H1; subst; clear H1.
  inversion H11; subst; clear H11; simpl in *.
  + (* non-speculative *)
    assert(Lnil: os2 = [] /\ ds2 = []).
    { inversion H10; subst; eauto. }
    destruct Lnil; subst; simpl.
    apply Ideal_If.
    destruct (eval st be) eqn:Eqnbe.
    * inversion H12; subst; clear H12.
      inversion H10; subst; clear H10; simpl in *.
      rewrite Eqnbe. do 2 rewrite t_update_same.
      apply Ideal_Skip.
    * inversion H12; subst; clear H12.
      inversion H1; subst; clear H1.
      inversion H2; subst; clear H2; simpl in *.
      assert(Hwhile: <(st'1, m'1, b'1, ds2)>
        =[ sel_slh P <{{while be do c end}}> ]=> <(st', m', b', os2)> ).
      { simpl. replace ds2 with (ds2 ++ [])%list by (rewrite app_nil_r; reflexivity).
        replace os2 with (os2 ++ [])%list by (rewrite app_nil_r; reflexivity).
        eapply Spec_Seq; eassumption. }
      do 2 rewrite app_nil_r. eapply Ideal_Seq.
      { rewrite Eqnbe in H13. rewrite t_update_same in H13.
        apply IH in H13; try tauto.
        - eassumption.
        - size_auto. }
      { apply IH in Hwhile; auto.
        - eapply ideal_unused_update_rev; eauto.
        - size_auto.
        - apply sel_slh_flag in H13; try tauto.
          rewrite t_update_eq. rewrite Eqnbe. assumption. }
  + (* speculative; analog to non_speculative *)
```

```
    assert(Lnil: os2 = [] /\ ds2 = []).
    { inversion H10; subst; eauto. }
    destruct Lnil; subst; simpl.
    apply Ideal_If_F.
    destruct (eval st be) eqn:Eqnbe.
    * inversion H12; subst; clear H12.
      inversion H1; subst; clear H1.
      inversion H2; subst; clear H2; simpl in *.
      assert(Hwhile: <(st'1, m'1, b'1, ds2)>
        =[ sel_slh P <{{while be do c end}}> ]=> <(st', m', b', os2)> ).
      { simpl. replace ds2 with (ds2 ++ [])%list by (rewrite app_nil_r; reflexivity).
        replace os2 with (os2 ++ [])%list by (rewrite app_nil_r; reflexivity).
        eapply Spec_Seq; eassumption. }
      do 2 rewrite app_nil_r. eapply Ideal_Seq.
      { rewrite Eqnbe in H13.
        apply IH in H13; try tauto.
        - rewrite t_update_eq in H13.
          apply ideal_unused_update in H13; [| tauto].
          eassumption.
        - size_auto. }
      { apply IH in Hwhile; auto.
        - rewrite Eqnbe in H13.
          apply IH in H13; try tauto.
          + apply ideal_unused_update_rev; eauto.
          + size_auto.
        - size_auto.
        - apply sel_slh_flag in H13; try tauto.
          rewrite Eqnbe. rewrite t_update_eq. reflexivity. }
    * inversion H12; subst; clear H12.
      inversion H10; subst; clear H10; simpl in *.
      rewrite Eqnbe. rewrite t_update_shadow. rewrite t_update_same.
      apply Ideal_Skip.
(* ALoad *)
- (* Spec_ALoad; public *)
  destruct (P x) eqn:Heq; try discriminate H.
  injection H; intros; subst; clear H.
  inversion H1; clear H1; subst. rewrite <- app_nil_r in *.
  inversion H0; clear H0; subst; simpl in *.
  * (* Ideal_ALoad *)
    rewrite t_update_neq; [| tauto]. rewrite Hstb.
    rewrite t_update_shadow. rewrite t_update_permute; [| tauto].
    rewrite t_update_eq. simpl.
```

```
      rewrite <- Hstb at 1. rewrite t_update_same.
      replace (not_zero (bool_to_nat (negb (not_zero
        (bool_to_nat ((if b' then 1 else 0) =? 0)%nat)) || not_zero 0))) with (b' && (P
          by (rewrite Heq; destruct b'; simpl; reflexivity).
        eapply Ideal_ALoad; eauto.
    * (* Ideal_ALoad_U *)
      rewrite t_update_neq; [| tauto]. rewrite Hstb.
      rewrite t_update_shadow. rewrite t_update_permute; [| tauto].
      simpl. rewrite <- Hstb at 1. rewrite t_update_same.
      replace (x !-> 0; st) with (x !-> if P x then 0 else nth i' (m' a') 0; st)
        by (rewrite Heq; reflexivity).
      eapply Ideal_ALoad_U; eauto.
  - (* Spec_ALoad; secret*)
    destruct (P x) eqn:Heq; try discriminate H. inversion H; clear H; subst.
    rewrite t_update_permute; [| tauto]. rewrite t_update_same.
    replace (x !-> nth (eval st i) (m' a) 0; st)
      with (x !-> if b' && P x then 0 else nth (eval st i) (m' a) 0; st)
        by (rewrite Heq; destruct b'; reflexivity).
    eapply Ideal_ALoad; eauto.
  - (* Spec_ALoad_U *)
    destruct (P x) eqn:Heq; try discriminate H. inversion H; clear H; subst.
    rewrite t_update_permute; [| tauto]. rewrite t_update_same.
    replace (x !-> nth i' (m' a') 0; st)
      with (x !-> if P x then 0 else nth i' (m' a') 0; st)
        by (rewrite Heq; reflexivity).
    eapply Ideal_ALoad_U; eauto.
  (* AStore *)
  - (* Spec_AStore *)
    rewrite t_update_same. apply Ideal_AStore; tauto.
  - (* Spec_AStore_U *)
    rewrite t_update_same. apply Ideal_AStore_U; tauto.
Qed.

(* ================================================================= *)
(** ** Speculative constant-time security for Selective SLH *)

(** Finally, we use compiler correctness and [spec_ct_secure] for the ideal
    semantics to prove [spec_ct_secure] for [sel_slh]. *)

Theorem sel_slh_spec_ct_secure :
  forall P PA c st1 st2 m1 m2 st1' st2' m1' m2' b1' b2' os1 os2 ds,
  P ;; PA |-ct- c ->
```

```
  unused msf c ->
  st1 msf = 0 ->
  st2 msf = 0 ->
  pub_equiv P st1 st2 ->
  pub_equiv PA m1 m2 ->
  <(st1, m1, false, ds)> =[ sel_slh P c ]=> <(st1', m1', b1', os1)> ->
  <(st2, m2, false, ds)> =[ sel_slh P c ]=> <(st2', m2', b2', os2)> ->
  os1 = os2.
Proof.
  intros P PA c st1 st2 m1 m2 st1' st2' m1' m2' b1' b2' os1 os2 ds
    Hwt Hunused Hs1b Hs2b Hequiv Haequiv Heval1 Heval2.
  eapply sel_slh_compiler_correctness in Heval1; try assumption.
  eapply sel_slh_compiler_correctness in Heval2; try assumption.
  eapply ideal_spec_ct_secure; eauto.
Qed.

(* ################################################################# *)
(** * Monadic interpreter for speculative semantics (optional; text missing) *)

Module SpecCTInterpreter.

(** Since manually constructing directions for the proofs of examples is very
    time consuming, we introduce a sound monadic interpreter, which can be used
    to simplify the proofs of the examples. *)

(** The Rocq development below is complete, but the text about it is missing.
    Readers not familiar with monadic interpreters can safely skip this section. *)

Definition prog_st : Type :=  state * mem * bool * dirs * obs.

Inductive output_st (A : Type): Type :=
| OST_Error : output_st A
| OST_OutOfFuel : output_st A
| OST_Finished : A -> prog_st -> output_st A.

Definition evaluator (A : Type): Type := prog_st -> (output_st A).
Definition interpreter : Type := evaluator unit.

Definition ret {A : Type} (value : A) : evaluator A :=
  fun (pst: prog_st) => OST_Finished A value pst.

Definition bind {A : Type} {B : Type} (e : evaluator A) (f : A -> evaluator B): evaluato
```

```
  fun (pst: prog_st) =>
    match e pst with
    | OST_Finished _ value (st', m', b', ds', os1)  =>
        match (f value) (st', m', b', ds', os1) with
        | OST_Finished _ value (st'', m'', b'', ds'', os2) =>
            OST_Finished B value (st'', m'', b'', ds'', os2)
        | ret => ret
        end
    | OST_Error _ => OST_Error B
    | OST_OutOfFuel _ => OST_OutOfFuel B
    end.

Notation "e >>= f" := (bind e f) (at level 58, left associativity).
Notation "e >> f" := (bind e (fun _ => f)) (at level 58, left associativity).

(* ================================================================= *)
(** ** Helper functions for individual instructions *)

Definition finish : interpreter := ret tt.

Definition get_var (name : string): evaluator nat :=
  fun (pst : prog_st) =>
    let
      '(st, _, _, _, _) := pst
    in
      ret (st name) pst.

Definition set_var (name : string) (value : nat) : interpreter :=
  fun (pst: prog_st) =>
    let
      '(st, m, b, ds, os) := pst
    in
      let
        new_st := (name !-> value; st)
      in
        finish (new_st, m, b, ds, os).

Definition get_arr (name : string): evaluator (list nat) :=
  fun (pst: prog_st) =>
    let
      '(_, m, _, _, _) := pst
    in
```

```
      ret (m name) pst.

Definition set_arr (name : string) (value : list nat) : interpreter :=
  fun (pst : prog_st) =>
    let '(st, m, b, ds, os) := pst in
    let new_m := (name !-> value ; m) in
    finish (st, new_m, b, ds, os).

Definition start_speculating : interpreter :=
  fun (pst : prog_st) =>
    let '(st, m, _, ds, os) := pst in
    finish (st, m, true, ds, os).

Definition is_speculating : evaluator bool :=
  fun (pst : prog_st) =>
    let '(_, _, b, _, _) := pst in
    ret b pst.

Definition eval_exp (a : exp) : evaluator nat :=
  fun (pst: prog_st) =>
    let '(st, _, _, _, _) := pst in
    let v := eval st a in
    ret v pst.

Definition raise_error : interpreter :=
  fun _ => OST_Error unit.

Definition observe (o : observation) : interpreter :=
  fun (pst : prog_st) =>
    let '(st, m, b, ds, os) := pst in
    OST_Finished unit tt (st, m, b, ds, (os ++ [o])%list).

Definition fetch_direction : evaluator (option direction) :=
  fun (pst : prog_st) =>
    let '(st, m, b, ds, os) := pst in
    match ds with
    | d::ds' =>
        ret (Some d) (st, m, b, ds', os)
    | [] => ret None (st, m, b, [], os)
    end.

(* ================================================================= *)
```

```
(** ** The actual speculative interpreter *)

Fixpoint spec_eval_engine_aux (fuel : nat) (c : com) : interpreter :=
  match fuel with
  | O => fun _ => OST_OutOfFuel unit
  | S fuel =>
    match c with
    | <{ skip }> => finish
    | <{ x := e }> => eval_exp e >>= fun v => set_var x v
    | <{ c1 ; c2 }> =>
        spec_eval_engine_aux fuel c1 >>
        spec_eval_engine_aux fuel c2
    | <{ if be then ct else cf end }> =>
        eval_exp be >>= fun bool_value =>
          observe (OBranch (not_zero bool_value)) >> fetch_direction >>=
        fun dop =>
          match dop with
          | Some DStep =>
              if not_zero bool_value then spec_eval_engine_aux fuel ct
              else spec_eval_engine_aux fuel cf
          | Some DForce =>
              start_speculating >>
              if not_zero bool_value then spec_eval_engine_aux fuel cf
              else spec_eval_engine_aux fuel ct
          | _ => raise_error
          end
    | <{ while be do c end }> =>
        spec_eval_engine_aux fuel <{if be then c; while be do c end else skip end}>
    | <{ x <- a[[ie]] }> =>
        eval_exp ie >>= fun i => observe (OALoad a i) >> get_arr a >>=
        fun arr_a => is_speculating >>= fun b => fetch_direction >>=
        fun dop =>
          match dop with
          | Some DStep =>
              if (i <? List.length arr_a)%nat then set_var x (nth i arr_a 0)
              else raise_error
          | Some (DLoad a' i') =>
              get_arr a' >>= fun arr_a' =>
                if negb (i <? List.length arr_a)%nat && (i' <? List.length arr_a')%nat &
                  set_var x (nth i' arr_a' 0)
                else raise_error
          | _ => raise_error
```

```
          end
    | <{ a[ie] <- e }> =>
        eval_exp ie >>= fun i => observe (OAStore a i) >> get_arr a >>=
        fun arr_a => eval_exp e >>= fun n => is_speculating >>= fun b => fetch_direction
        fun dop =>
          match dop with
          | Some DStep =>
              if (i <? List.length arr_a)%nat then set_arr a (upd i arr_a n)
              else raise_error
          | Some (DStore a' i') =>
              get_arr a' >>= fun arr_a' =>
                if negb (i <? List.length arr_a)%nat && (i' <? List.length arr_a')%nat &
                  set_arr a' (upd i' arr_a' n)
                else raise_error
          | _ => raise_error
          end
    end
end.

Definition compute_fuel (c :com) (ds :dirs) : nat :=
  2 +
    match ds with
    | [] => com_size c
    | _ => length ds * com_size c
    end.

Definition spec_eval_engine (c : com) (st : state) (m : mem) (b : bool) (ds : dirs)
      : option (state * mem * bool * obs) :=
    match spec_eval_engine_aux (compute_fuel c ds) c (st, m, b, ds, []) with
    | OST_Finished _ _ (st', m', b', ds', os) =>
        if ((length ds') =? 0)%nat then Some (st', m', b', os)
        else None
    | _ => None
    end.

(* ================================================================= *)
(** ** Soundness of the interpreter *)

Lemma ltb_reflect : forall n m :nat,
  reflect (n < m) (n <? m)%nat.
Proof.
  intros n m. apply iff_reflect. rewrite ltb_lt. reflexivity.
```

```
Qed.

Lemma eqb_reflect: forall n m :nat,
  reflect (n = m ) (n =? m)%nat.
Proof.
  intros n m. apply iff_reflect. rewrite eqb_eq. reflexivity.
Qed.

Lemma spec_eval_engine_aux_sound : forall n c st m b ds os st' m' b' ds' os' u,
  spec_eval_engine_aux n c (st, m, b, ds, os)
    = OST_Finished unit u (st', m', b', ds', os') ->
  (exists dsn osn,
  (dsn++ds')%list = ds /\ (os++osn)%list = os' /\
      <(st, m, b, dsn)> =[ c ]=> <(st', m', b', osn)> ).
Proof.
  induction n as [| n' IH]; intros c st m b ds os st' m' b' ds' os' u Haux;
  simpl in Haux; [discriminate |].
  destruct c as [| X e | c1 c2 | be ct cf | be cw | X a ie | a ie e ] eqn:Eqnc;
  unfold ">>=" in Haux; simpl in Haux.
  - (* Skip *)
    inversion Haux; subst.
    exists []; exists []; split;[| split].
    + reflexivity.
    + rewrite app_nil_r. reflexivity.
    + apply Spec_Skip.
  - (* Asgn *)
    simpl in Haux. inversion Haux; subst.
    exists []; exists []; split;[| split].
    + reflexivity.
    + rewrite app_nil_r. reflexivity.
    + apply Spec_Asgn. reflexivity.
  - destruct (spec_eval_engine_aux _ c1 _) eqn:Hc1;
    try discriminate; simpl in Haux.
    destruct p as [ [ [ [stm mm] bm] dsm] osm]; simpl in Haux.
    destruct (spec_eval_engine_aux _ c2 _) eqn:Hc2;
    try discriminate; simpl in Haux.
    destruct p as [ [ [ [stt mt] bt] dst] ost]; simpl in Haux.
    apply IH in Hc1. destruct Hc1 as [ds1 [ os1 [Hds1 [Hos1 Heval1] ] ] ].
    apply IH in Hc2. destruct Hc2 as [ds2 [ os2 [Hds2 [Hos2 Heval2] ] ] ].
    inversion Haux; subst. exists (ds1++ds2)%list; exists (os1++os2)%list;
    split; [| split].
    + rewrite <- app_assoc. reflexivity.
```

```
  + rewrite <- app_assoc. reflexivity.
  + eapply Spec_Seq; eauto.
- (* IF *)
  destruct ds as [| d ds_tl] eqn:Eqnds; simpl in Haux; try discriminate.
  destruct d eqn:Eqnd; try discriminate; simpl in Haux.
  + (* DStep *)
    destruct (eval st be) eqn:Eqnbe.
    * unfold obs, dirs, not_zero in Haux. simpl in Haux.
      destruct (spec_eval_engine_aux n' cf (st, m, b, ds_tl, (os ++ [OBranch false])%l
      try discriminate; simpl in Haux.
      destruct p as [ [ [ [stt mt] bt] dst] ost]; simpl in Haux.
      inversion Haux; subst. apply IH in Hcf.
      destruct Hcf as [dst [ ost [Hds [Hos Heval] ] ] ].
      exists (DStep :: dst); exists ([OBranch false]++ost)%list; split;[| split].
      { simpl. rewrite Hds. reflexivity. }
      { rewrite app_assoc. rewrite Hos. reflexivity. }
      { erewrite <- not_zero_eval_0; [| eassumption].
        apply Spec_If. rewrite Eqnbe. apply Heval. }
    * unfold obs, dirs, not_zero in Haux. simpl in Haux.
      destruct (spec_eval_engine_aux n' ct (st, m, b, ds_tl, (os ++ [OBranch true])%li
      try discriminate; simpl in Haux.
      destruct p as [ [ [ [stt mt] bt] dst] ost]; simpl in Haux.
      inversion Haux; subst. apply IH in Hct.
      destruct Hct as [dst [ ost [Hds [Hos Heval] ] ] ].
      exists (DStep :: dst); exists ([OBranch true]++ost)%list; split;[| split].
      { simpl. rewrite Hds. reflexivity. }
      { rewrite app_assoc. rewrite Hos. reflexivity. }
      { erewrite <- not_zero_eval_S; [| eassumption].
        apply Spec_If. rewrite Eqnbe. apply Heval. }
  + (* DForce *)
    destruct (eval st be) eqn:Eqnbe.
    * unfold obs, dirs, not_zero in Haux. simpl in Haux.
      destruct (spec_eval_engine_aux n' ct (st, m, true, ds_tl, (os ++ [OBranch false]
      try discriminate; simpl in Haux.
      destruct p as [ [ [ [stt mt] bt] dst] ost]; simpl in Haux.
      inversion Haux; subst. apply IH in Hcf.
      destruct Hcf as [dst [ ost [Hds [Hos Heval] ] ] ].
      exists (DForce :: dst); exists ([OBranch false]++ost)%list; split;[| split].
      { simpl. rewrite Hds. reflexivity. }
      { rewrite app_assoc. rewrite Hos. reflexivity. }
      { erewrite <- not_zero_eval_0; [| eassumption].
        apply Spec_If_F. rewrite Eqnbe. apply Heval. }
```

```
    * unfold obs, dirs, not_zero in Haux. simpl in Haux.
      destruct (spec_eval_engine_aux n' cf (st, m, true, ds_tl, (os ++ [OBranch true])
      destruct p as [ [ [ [stt mt] bt] dst] ost]; simpl in Haux.
      inversion Haux; subst. apply IH in Hct.
      destruct Hct as [dst [ ost [Hds [Hos Heval] ] ] ].
      exists (DForce :: dst); exists ([OBranch true]++ost)%list; split;[| split].
      { simpl. rewrite Hds. reflexivity. }
      { rewrite app_assoc. rewrite Hos. reflexivity. }
      { erewrite <- not_zero_eval_S; [| eassumption].
        apply Spec_If_F. rewrite Eqnbe. apply Heval. }
- (* While *)
  apply IH in Haux. destruct Haux as [dst [ ost [Hds [Hos Heval] ] ] ].
  exists dst; exists ost; split; [| split]; eauto.
- (* ALoad *)
  destruct ds as [| d ds_tl] eqn:Eqnds; simpl in Haux; try discriminate.
  destruct d eqn:Eqnd; try discriminate; simpl in Haux.
  + (* DStep *)
    destruct (eval st ie <? Datatypes.length (m a))%nat eqn:Eqnindex; try discriminate
    destruct (observe (OALoad a (eval st ie)) (st, m, b, ds_tl, os)) eqn:Eqbobs; try d
    simpl in Haux. inversion Haux; subst.
    eexists [DStep]; eexists [OALoad a (eval st ie)]; split;[| split]; try reflexivity
    eapply Spec_ALoad; eauto. destruct (ltb_reflect (eval st ie) (length (m' a))) as [
    * apply Hlt.
    * discriminate.
  + (* DForce *)
    destruct (negb (eval st ie <? Datatypes.length (m a))%nat) eqn:Eqnindex1;
    destruct ((i <? Datatypes.length (m a0))%nat) eqn:Eqnindex2;
    destruct b eqn:Eqnb; try discriminate; simpl in Haux. inversion Haux; subst.
    eexists [DLoad a0 i ]; eexists [OALoad a (eval st ie)]; split;[| split]; try refle
    eapply Spec_ALoad_U; eauto.
    * destruct (ltb_reflect (eval st ie) (length (m' a))) as [Hlt | Hgeq].
      { discriminate. }
      { apply not_lt in Hgeq. apply Hgeq. }
    * destruct (ltb_reflect i (length (m' a0))) as [Hlt | Hgeq].
      { apply Hlt. }
      { discriminate. }
- (* AStore *)
destruct ds as [| d ds_tl] eqn:Eqnds; simpl in Haux; try discriminate.
destruct d eqn:Eqnd; try discriminate; simpl in Haux.
+ (* DStep *)
  destruct ((eval st ie <? Datatypes.length (m a))%nat) eqn:Eqnindex; try discriminate
  destruct (observe (OAStore a (eval st ie)) (st, m, b, ds_tl, os)) eqn:Eqbobs; try di
```

```
    simpl in Haux. inversion Haux; subst.
    eexists [DStep]; eexists [OAStore a (eval st' ie)]; split;[| split]; try reflexivity
    eapply Spec_AStore; eauto. destruct (ltb_reflect (eval st' ie) (length (m a))) as [H
    * apply Hlt.
    * discriminate.
  + (* DForce *)
    destruct  (negb (eval st ie <? Datatypes.length (m a))%nat) eqn:Eqnindex1;
    destruct (i <? Datatypes.length (m a0))%nat eqn:Eqnindex2;
    destruct b eqn:Eqnb; try discriminate; simpl in Haux. inversion Haux; subst.
    eexists [DStore a0 i]; eexists [OAStore a (eval st' ie)]; split;[| split]; try refle
    eapply Spec_AStore_U; eauto.
    * destruct (ltb_reflect (eval st' ie) (length (m a))) as [Hlt | Hgeq].
      { discriminate. }
      {  apply not_lt in Hgeq. apply Hgeq. }
    * destruct (ltb_reflect i (length (m a0))) as [Hlt | Hgeq].
      { apply Hlt. }
      { discriminate. }
Qed.

Theorem spec_eval_engine_sound: forall c st m b ds st' m' b' os',
  spec_eval_engine c st m b ds = Some (st', m', b', os') ->
  <(st, m, b, ds)> =[ c ]=> <(st', m', b', os')> .
Proof.
  intros c st m b ds st' m' b' os' Hengine.
  unfold spec_eval_engine in Hengine.
  destruct (spec_eval_engine_aux _ c _) eqn:Eqnaux;
  try discriminate. destruct p as [ [ [ [stt mt] bt] dst] ost].
  destruct ((Datatypes.length dst =? 0)%nat) eqn:Eqnds; try discriminate.
  apply spec_eval_engine_aux_sound in Eqnaux.
  destruct Eqnaux as [dsn [osn [Hdsn [Hosn Heval] ] ] ].
  inversion Hengine; subst. rewrite app_nil_l.
  destruct (eqb_reflect (length dst) 0) as [Heq | Hneq].
  + apply length_zero_iff_nil in Heq. rewrite Heq. rewrite app_nil_r. apply Heval.
  + discriminate.
Qed.

(* ================================================================= *)
(** ** Back to showing that our example is not speculative constant-time *)

Example spec_insecure_prog_2_is_spec_insecure :
  ~(spec_ct_secure XYZpub APpub spec_insecure_prog_2).
Proof.
```

```
  unfold spec_insecure_prog_2.
  (* program is insecure under speculative execution. *)
  remember (__ !-> 0) as st.
  remember (AP!-> [0;1;2]; AS !-> [0;0;0;0]; __ !-> []) as m1.
  remember (AP!-> [0;1;2]; AS !-> [4;5;6;7]; __ !-> []) as m2.
  remember ([DStep; DStep; DStep; DStep; DStep; DStep; DForce; DLoad AS 3; DStep; DStep]
  intros Hsecure.
  assert (L: exists stt1 mt1 bt1 os1 stt2 mt2 bt2 os2,
    <(st, m1, false, ds )> =[ spec_insecure_prog_2 ]=> <( stt1, mt1, bt1, os1)> /\
    <(st, m2, false, ds )> =[ spec_insecure_prog_2 ]=> <( stt2, mt2, bt2, os2)> /\
    os1 <> os2 ).
  { eexists; eexists; eexists; eexists; eexists; eexists; eexists; eexists.
    split; [| split].
    - apply spec_eval_engine_sound. unfold spec_insecure_prog_2, spec_eval_engine;
      subst; simpl; reflexivity.
    - apply spec_eval_engine_sound. unfold spec_insecure_prog_2, spec_eval_engine;
      subst; simpl; reflexivity.
    - intros Contra; inversion Contra. }
  destruct L as [stt1 [mt1 [bt1 [os1 [stt2 [mt2 [bt2 [os2 [Heval1 [Heval2 Hneq] ] ] ] ]
  eapply Hsecure in Heval1; eauto.
  - apply pub_equiv_refl.
  - subst. apply pub_equiv_update_public; auto.
    apply pub_equiv_update_secret; auto.
    apply pub_equiv_refl.
Qed.

End SpecCTInterpreter.

(* 2026-01-07 13:37 *)
```

Chapter 10

Library SECF.Postscript

10.1 Postscript

10.2 Looking Back

Here is a quick summary of the topics we covered in this volume:

10.2.1 Noninterference

- definitions for pure functions, state transformers, and imperative programs
- termination-insensitive noninterference (TINI)
- termination-sensitive noninterference (TSNI)

10.2.2 Secure multi-execution

- sound and transparent dynamic enforcement of TINI

10.2.3 Information-flow control type systems

- type-checkers enforcing TINI and TSNI
- for imperative programs with state and outputs

10.2.4 Side channels

- control flow security and type system enforcing it
- cryptographic constant-time security and type system enforcing it

10.2.5 Speculative execution attacks

- speculative constant-time security definition
- speculative load hardening (SLH) transformation achieving it

10.3 Looking Around

The topics above have found practical applications in system security. Below we highlight a few recent research projects involving machine-checked proofs:

10.3.1 Proving Noninterference by Parametricity

While in StaticIFC we showed how to build specialized type systems for noninterference, research has shown that in functional programming languages with strong type-abstraction mechanisms, information-flow control can be implemented as a library. Recent research has shown that in this setting simple and elegant noninterference proofs can be built by relying on the theory of parametricity, both for libraries doing static enforcement *Algehed* **and** *Bernardy* 2019 (in Bib.v) and for ones doing dynamic enforcement *Algehed et al* 2021 (in Bib.v). These noninterference proofs have been machine-checked in Agda.

10.3.2 Formal verification of a constant-time preserving C compiler

This work *Barthe et al* 2020 (in Bib.v) shows that a mildly modified version of the CompCert verified C compiler preserves cryptographic constant-time security. In particular the authors prove in Rocq that the compiler does not introduce secret dependencies into control flow or memory access. Their Rocq formalization is aimed at maximizing reuse of the CompCert correctness proof, through the use of novel proof techniques for constant-time preservation.

10.3.3 Jasmin programming language and compiler

Jasmin is a low-level domain-specific language for implementing high-assurance and high-speed cryptography. Jasmin programs can be verified for correctness, cryptographic security, and side-channel resistance by translation to the EasyCrypt proof assistant *Almeida et al* 2020 (in Bib.v). The Jasmin compiler was formally verified in Rocq to be correct *Almeida et al* 2020 (in Bib.v) and to preserve constant-time security *Barthe et al* 2021 (in Bib.v). In more recent work a core compiler inspired by Jasmin was proved in Rocq to also preserve speculative constant-time *Arranz-Olmos et al* 2025 (in Bib.v).

10.3.4 Flexible Mechanized Speculative Load Hardening

The SpecCT chapter and the two projects above are aimed at achieving security for cryptographic code. Yet Spectre attacks are also a serious threat for non-cryptographic code, since without any defenses attackers can construct "universal read gadgets" that leak a sensitive program's entire memory. SLH is, however, not strong enough for protecting code that does not respect the constant-time discipline, leading to the introduction of Ultimate SLH *Zhang et al* 2023 (in Bib.v), which provides protection for arbitrary programs, but has too large overhead for general use, since it conservatively assumes that all data is secret. More recent work introduces Flexible SLH *Baumann et al* 2025 (in Bib.v), which achieves the best of both worlds by generalizing both the selective SLH variant from SpecCT and Ultimate SLH. Baumann et al prove in Rocq that Flexible SLH and Ultimate SLH satisfy a relative security property: any transformed program running with speculation must not leak more than what the source program leaks sequentially. Their Rocq formalization originated as an extension of the simple development from the SpecCT chapter.

10.3.5 Strong Timing Isolation of Hardware Enclaves

This work *Lau et al* 2024 (in Bib.v) introduced a RISC-V processor design that is formally verified in Rocq to achieve strong timing isolation for enclaves, which is formalized in terms of "air-gaped machines".

10.4 Looking Forward

For readers interesting in research, here are the main conferences publishing papers on formal foundations on security:

- Computer Security Foundations (CSF)
- Principles of Programming Languages (POPL)
- International Conference on Functional Programming (ICFP)
- Certified Programs and Proofs (CPP)
- Interactive Theorem Proving (ITP)
- Computer and Communications Security (CCS)
  - Formal Methods and Programming Languages track
- IEEE Security and Privacy (SP)
- Principles of Secure Compilation Workshop (PriSC)

Chapter 11

Library SECF.Bib

11.1 Bib: Bibliography

11.2 Resources cited in this volume

Algehed and *Bernardy* 2019 Simple noninterference from parametricity. Maximilian Algehed, Jean-Philippe Bernardy. Proc. ACM Program. Lang. 3(ICFP): 89:1-89:22. 2019. {https://doi.org/10.1145/3341693}

Algehed et al 2021 Dynamic IFC Theorems for Free! Maximilian Algehed, Jean-Philippe Bernardy, Catalin Hritcu. CSF 2021. 1-14. {https://arxiv.org/abs/2005.04722}

Almeida et al 2020 The Last Mile: High-Assurance and High-Speed Cryptographic Implementations. José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, Pierre-Yves Strub. IEEE Symposium on Security and Privacy. 2020. {https://doi.org/10.1109/SP40000.2020.00028}

Arranz-Olmos et al 2025 Preservation of Speculative Constant-Time by Compilation. Santiago Arranz-Olmos, Gilles Barthe, Lionel Blatter, Benjamin Grégoire, Vincent Laporte. Proc. ACM Program. Lang. 9(POPL): 1293-1325. 2025. {https://doi.org/10.1145/3704880}

Barthe et al 2019 System-Level Non-interference of Constant-Time Cryptography. Part I: Model. Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Luna. J. Autom. Reason. 63(1): 1-51. 2019. {https://doi.org/10.1007/s10817-017-9441-5}

Barthe et al 2020 Formal verification of a constant-time preserving C compiler. Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, Alix Trieu. Proc. ACM Program. Lang. 4(POPL): 7:1-7:30. 2020. {https://doi.org/10.1145/3371075}

Barthe et al 2021 Structured Leakage and Applications to Cryptographic Constant-Time and Cost. Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Swarn Priya. ACM SIGSAC Conference on Computer and Communications Security (CCS). 2021. {https://doi.org/10.1145/3460120.34

Baumann et al 2025 FSLH: Flexible Mechanized Speculative Load Hardening. Jonathan Baumann, Roberto Blanco, Léon Ducruet, Sebastian Harwig, and Cătălin Hriţcu. IEEE Computer Security Foundations Symposium (CSF). 2025. {http://arxiv.org/abs/2502.03203}

Devriese **and** *Piessens* 2010 Noninterference through Secure Multi-execution. Dominique Devriese, Frank Piessens. IEEE Symposium on Security and Privacy. 2010. {https://doi.org/10.1109/SP.20

Lau et al 2024 Specification and Verification of Strong Timing Isolation of Hardware Enclaves. Stella Lau, Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala. ACM SIGSAC Conference on Computer and Communications Security (CCS). 2024. {https://doi.org/10.1145/3658644.36

Molnar et al 2005 The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. David Molnar, Matt Piotrowski, David Schultz, and David Wagner. ICISC 2005. {https://eprint.iacr.org/2005/368}

Ngo et al 2018 Impossibility of Precise and Sound Termination-Sensitive Security Enforcements. Minh Ngo, Frank Piessens, Tamara Rezk. IEEE Symposium on Security and Privacy. 2018. {https://doi.org/10.1109/SP.2018.00048}

Sabelfeld **and** *Myers* 2003 Language-based information-flow security. Andrei Sabelfeld and Andrew C. Myers. IEEE Journal on Selected Areas in Communications 21 (1), 5-19. 2003. {https://www.cs.cornell.edu/andru/papers/jsac/sm-jsac03.pdf}

Shivakumar et al 2023 Spectre Declassified: Reading from the Right Place at the Wrong Time. B. A. Shivakumar, J. Barnes, G. Barthe, S. Cauligi, C. Chuengsatiansup, D. Genkin, S. O'Connell, P. Schwabe, R. Q. Sim, and Y. Yarom. IEEE Symposium on Security and Privacy. 2023. {http://dx.doi.org/10.1109/SP46215.2023.10179355}

Volpano et al 1996 A Sound Type System for Secure Flow Analysis. Dennis M. Volpano, Cynthia E. Irvine, Geoffrey Smith. J. Comput. Secur. 4(2/3): 167-188. 1996. {https://people.mpi-sws.org/~dg/teaching/lis2014/modules/ifc-1-volpano96.pdf}

Volpano **and** *Smith* 1997 Eliminating Covert Flows with Minimum Typings. Dennis M. Volpano, Geoffrey Smith. Computer Security Foundations Workshop (CSFW). 1997. {https://ifc-challenge.appspot.com/static/pdfs/csfw97.pdf}

Zhang et al 2023 Ultimate SLH: Taking Speculative Load Hardening to the Next Level. Zhiyuan Zhang, Gilles Barthe, Chitchanok Chuengsatiansup, Peter Schwabe, Yuval Yarom. USENIX Security Symposium. 2023. {https://www.usenix.org/conference/usenixsecurity23/presentation/zhiyuan-slh}

Date

Chapter 12

Library SECF.MapsTest

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib Require Export String.
From SECF Require Import Maps.

Parameter MISSING: Type.

Module CHECK.

Ltac check_type A B :=
    match type of A with
    | context[MISSING] ⇒ idtac "Missing:" A
    | ?T ⇒ first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
")"]
    end.

Ltac print_manual_grade A :=
    match eval compute in A with
    | Some (_ ?S ?C) ⇒
        idtac "Score:" S;
        match eval compute in C with
          | ""%string ⇒ idtac "Comment: None"
          | _ ⇒ idtac "Comment:" C
        end
    | None ⇒
        idtac "Score: Ungraded";
        idtac "Comment: None"
    end.

End CHECK.

From SECF Require Import Maps.
Import CHECK.

Goal True.

idtac "————————- t_update_same ————————".
```

```
idtac " ".

idtac "#> t_update_same".
idtac "Possible points: 2".
check_type @t_update_same (
(∀ (A : Type) (m : total_map A) (x : string),
 @eq (∀ _ : string, A) (@t_update A m x (m x)) m)).
idtac "Assumptions:".
Abort.
Print Assumptions t_update_same.
Goal True.
idtac " ".

idtac "———————- t_update_permute ———————".
idtac " ".

idtac "#> t_update_permute".
idtac "Possible points: 3".
check_type @t_update_permute (
(∀ (A : Type) (m : total_map A) (v1 v2 : A) (x1 x2 : string)
    (_ : not (@eq string x2 x1)),
 @eq (∀ _ : string, A) (@t_update A (@t_update A m x2 v2) x1 v1)
    (@t_update A (@t_update A m x1 v1) x2 v2))).
idtac "Assumptions:".
Abort.
Print Assumptions t_update_permute.
Goal True.
idtac " ".

idtac " ".

idtac "Max points - standard: 5".
idtac "Max points - advanced: 5".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "".
idtac "".
idtac "********** Summary **********".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
```

```
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "********** Standard **********".
idtac "------- t_update_same -------".
Print Assumptions t_update_same.
idtac "------- t_update_permute -------".
Print Assumptions t_update_permute.
idtac "".
idtac "********** Advanced **********".
Abort.
```

Chapter 13

Library SECF.ImpTest

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib Require Export String.
From SECF Require Import Imp.

Parameter MISSING: Type.

Module CHECK.

Ltac check_type A B :=
    match type of A with
    | context[MISSING] ⇒ idtac "Missing:" A
    | ?T ⇒ first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
")"]
    end.

Ltac print_manual_grade A :=
    match eval compute in A with
    | Some (_ ?S ?C) ⇒
        idtac "Score:" S;
        match eval compute in C with
          | ""%string ⇒ idtac "Comment: None"
          | _ ⇒ idtac "Comment:" C
        end
    | None ⇒
        idtac "Score: Ungraded";
        idtac "Comment: None"
    end.

End CHECK.

From SECF Require Import Imp.
Import CHECK.

Goal True.

idtac "————————- optimize_0plus_b_sound ————————".
```

```
idtac " ".

idtac "#> AExp.optimize_0plus_b_test1".
idtac "Possible points: 0.5".
check_type @AExp.optimize_0plus_b_test1 (
(@eq AExp.bexp
   (AExp.optimize_0plus_b
      (AExp.BNot
         (AExp.BGt (AExp.APlus (AExp.ANum 0) (AExp.ANum 4)) (AExp.ANum 8))))
   (AExp.BNot (AExp.BGt (AExp.ANum 4) (AExp.ANum 8))))).
idtac "Assumptions:".
Abort.
Print Assumptions AExp.optimize_0plus_b_test1.
Goal True.
idtac " ".

idtac "#> AExp.optimize_0plus_b_test2".
idtac "Possible points: 0.5".
check_type @AExp.optimize_0plus_b_test2 (
(@eq AExp.bexp
   (AExp.optimize_0plus_b
      (AExp.BAnd
         (AExp.BLe (AExp.APlus (AExp.ANum 0) (AExp.ANum 4)) (AExp.ANum 5))
         AExp.BTrue))
   (AExp.BAnd (AExp.BLe (AExp.ANum 4) (AExp.ANum 5)) AExp.BTrue))).
idtac "Assumptions:".
Abort.
Print Assumptions AExp.optimize_0plus_b_test2.
Goal True.
idtac " ".

idtac "#> AExp.optimize_0plus_b_sound".
idtac "Possible points: 2".
check_type @AExp.optimize_0plus_b_sound (
(∀ b : AExp.bexp,
 @eq bool (AExp.beval (AExp.optimize_0plus_b b)) (AExp.beval b))).
idtac "Assumptions:".
Abort.
Print Assumptions AExp.optimize_0plus_b_sound.
Goal True.
idtac " ".

idtac "———————- bevalR ———————".
idtac " ".

idtac "#> AExp.bevalR_iff_beval".
```

```
idtac "Possible points: 3".
check_type @AExp.bevalR_iff_beval (
(∀ (b : AExp.bexp) (bv : bool),
 iff (AExp.bevalR b bv) (@eq bool (AExp.beval b) bv))).
idtac "Assumptions:".
Abort.
Print Assumptions AExp.bevalR_iff_beval.
Goal True.
idtac " ".

idtac "———————- ceval_example2 ———————".
idtac " ".

idtac "#> ceval_example2".
idtac "Possible points: 2".
check_type @ceval_example2 (
(ceval (CSeq (CAsgn X (ANum 0)) (CSeq (CAsgn Y (ANum 1)) (CAsgn Z (ANum 2))))
    empty_st
    (@Maps.t_update nat
        (@Maps.t_update nat (@Maps.t_update nat empty_st X 0) Y 1) Z 2))).
idtac "Assumptions:".
Abort.
Print Assumptions ceval_example2.
Goal True.
idtac " ".

idtac "———————- loop_never_stops ———————".
idtac " ".

idtac "#> loop_never_stops".
idtac "Possible points: 3".
check_type @loop_never_stops ((∀ st st' : state, not (ceval loop st st'))).
idtac "Assumptions:".
Abort.
Print Assumptions loop_never_stops.
Goal True.
idtac " ".

idtac "———————- no_whiles_eqv ———————".
idtac " ".

idtac "#> no_whiles_eqv".
idtac "Possible points: 3".
check_type @no_whiles_eqv (
(∀ c : com, iff (@eq bool (no_whiles c) true) (no_whilesR c))).
idtac "Assumptions:".
Abort.
```

```
Print Assumptions no_whiles_eqv.
Goal True.
idtac " ".

idtac "———————- no_whiles_terminating ———————".
idtac " ".

idtac "#> Manually graded: no_whiles_terminating".
idtac "Possible points: 6".
print_manual_grade manual_grade_for_no_whiles_terminating.
idtac " ".

idtac "———————- stack_compiler ———————".
idtac " ".

idtac "#> s_execute1".
idtac "Possible points: 1".
check_type @s_execute1 (
(@eq (list nat)
    (s_execute empty_st (@nil nat)
        (@cons sinstr (SPush 5)
            (@cons sinstr (SPush 3)
                (@cons sinstr (SPush 1) (@cons sinstr SMinus (@nil sinstr))))))
    (@cons nat 2 (@cons nat 5 (@nil nat))))).
idtac "Assumptions:".
Abort.
Print Assumptions s_execute1.
Goal True.
idtac " ".

idtac "#> s_execute2".
idtac "Possible points: 0.5".
check_type @s_execute2 (
(@eq (list nat)
    (s_execute (@Maps.t_update nat empty_st X 3)
        (@cons nat 3 (@cons nat 4 (@nil nat)))
        (@cons sinstr (SPush 4)
            (@cons sinstr (SLoad X)
                (@cons sinstr SMult (@cons sinstr SPlus (@nil sinstr))))))
    (@cons nat 15 (@cons nat 4 (@nil nat))))).
idtac "Assumptions:".
Abort.
Print Assumptions s_execute2.
Goal True.
idtac " ".

idtac "#> s_compile1".
```

```
idtac "Possible points: 1.5".
check_type @s_compile1 (
(@eq (list sinstr) (s_compile (AMinus (AId X) (AMult (ANum 2) (AId Y))))
    (@cons sinstr (SLoad X)
        (@cons sinstr (SPush 2)
            (@cons sinstr (SLoad Y)
                (@cons sinstr SMult (@cons sinstr SMinus (@nil sinstr))))))).
idtac "Assumptions:".
Abort.
Print Assumptions s_compile1.
Goal True.
idtac " ".

idtac "————————- execute_app ————————".
idtac " ".

idtac "#> execute_app".
idtac "Possible points: 3".
check_type @execute_app (
(∀ (st : state) (p1 p2 : list sinstr) (stack : list nat),
 @eq (list nat) (s_execute st stack (@app sinstr p1 p2))
    (s_execute st (s_execute st stack p1) p2))).
idtac "Assumptions:".
Abort.
Print Assumptions execute_app.
Goal True.
idtac " ".

idtac "————————- stack_compiler_correct ————————".
idtac " ".

idtac "#> s_compile_correct_aux".
idtac "Possible points: 2.5".
check_type @s_compile_correct_aux (
(∀ (st : state) (e : aexp) (stack : list nat),
 @eq (list nat) (s_execute st stack (s_compile e))
    (@cons nat (aeval st e) stack))).
idtac "Assumptions:".
Abort.
Print Assumptions s_compile_correct_aux.
Goal True.
idtac " ".

idtac "#> s_compile_correct".
idtac "Possible points: 0.5".
check_type @s_compile_correct (
```

```
(∀ (st : state) (e : aexp),
  @eq (list nat) (s_execute st (@nil nat) (s_compile e))
     (@cons nat (aeval st e) (@nil nat)))).
idtac "Assumptions:".
Abort.
Print Assumptions s_compile_correct.
Goal True.
idtac " ".

idtac " ".

idtac "Max points - standard: 29".
idtac "Max points - advanced: 29".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "".
idtac "".
idtac "********** Summary **********".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "********** Standard **********".
idtac "---------- AExp.optimize_0plus_b_test1 ----------".
Print Assumptions AExp.optimize_0plus_b_test1.
idtac "---------- AExp.optimize_0plus_b_test2 ----------".
Print Assumptions AExp.optimize_0plus_b_test2.
idtac "---------- AExp.optimize_0plus_b_sound ----------".
Print Assumptions AExp.optimize_0plus_b_sound.
idtac "---------- AExp.bevalR_iff_beval ----------".
Print Assumptions AExp.bevalR_iff_beval.
idtac "---------- ceval_example2 ----------".
Print Assumptions ceval_example2.
idtac "---------- loop_never_stops ----------".
Print Assumptions loop_never_stops.
idtac "---------- no_whiles_eqv ----------".
Print Assumptions no_whiles_eqv.
```

```
idtac "————— no_whiles_terminating —————".
idtac "MANUAL".
idtac "————— s_execute1 —————".
Print Assumptions s_execute1.
idtac "————— s_execute2 —————".
Print Assumptions s_execute2.
idtac "————— s_compile1 —————".
Print Assumptions s_compile1.
idtac "————— execute_app —————".
Print Assumptions execute_app.
idtac "————— s_compile_correct_aux —————".
Print Assumptions s_compile_correct_aux.
idtac "————— s_compile_correct —————".
Print Assumptions s_compile_correct.
idtac "".
idtac "********** Advanced **********".
Abort.
```

Chapter 14

Library SECF.EquivTest

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib Require Export String.
From SECF Require Import Equiv.

Parameter MISSING: Type.

Module CHECK.

Ltac check_type A B :=
    match type of A with
    | context[MISSING] ⇒ idtac "Missing:" A
    | ?T ⇒ first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
")"]
    end.

Ltac print_manual_grade A :=
    match eval compute in A with
    | Some (_ ?S ?C) ⇒
        idtac "Score:" S;
        match eval compute in C with
          | ""%string ⇒ idtac "Comment: None"
          | _ ⇒ idtac "Comment:" C
        end
    | None ⇒
        idtac "Score: Ungraded";
        idtac "Comment: None"
    end.

End CHECK.

From SECF Require Import Equiv.
Import CHECK.

Goal True.

idtac "————————- skip_right ————————".
```

```
idtac " ".
idtac "#> skip_right".
idtac "Possible points: 2".
check_type @skip_right ((∀ c : com, cequiv (CSeq c CSkip) c)).
idtac "Assumptions:".
Abort.
Print Assumptions skip_right.
Goal True.
idtac " ".
idtac "———————- if_false ———————".
idtac " ".
idtac "#> if_false".
idtac "Possible points: 2".
check_type @if_false (
(∀ (b : bexp) (c1 c2 : com) (_ : bequiv b BFalse),
 cequiv (CIf b c1 c2) c2)).
idtac "Assumptions:".
Abort.
Print Assumptions if_false.
Goal True.
idtac " ".
idtac "———————- swap_if_branches ———————".
idtac " ".
idtac "#> swap_if_branches".
idtac "Possible points: 3".
check_type @swap_if_branches (
(∀ (b : bexp) (c1 c2 : com), cequiv (CIf b c1 c2) (CIf (BNot b) c2 c1))).
idtac "Assumptions:".
Abort.
Print Assumptions swap_if_branches.
Goal True.
idtac " ".
idtac "———————- while_true ———————".
idtac " ".
idtac "#> while_true".
idtac "Possible points: 2".
check_type @while_true (
(∀ (b : bexp) (c : com) (_ : bequiv b BTrue),
 cequiv (CWhile b c) (CWhile BTrue CSkip))).
idtac "Assumptions:".
Abort.
```

```
Print Assumptions while_true.
Goal True.
idtac " ".

idtac "———————- assign_aequiv ———————".
idtac " ".

idtac "#> assign_aequiv".
idtac "Possible points: 2".
check_type @assign_aequiv (
(∀ (X : String.string) (a : aexp) (_ : aequiv (AId X) a),
 cequiv CSkip (CAsgn X a))).
idtac "Assumptions:".
Abort.
Print Assumptions assign_aequiv.
Goal True.
idtac " ".

idtac "———————- CIf_congruence ———————".
idtac " ".

idtac "#> CIf_congruence".
idtac "Possible points: 3".
check_type @CIf_congruence (
(∀ (b b' : bexp) (c1 c1' c2 c2' : com) (_ : bequiv b b')
    (_ : cequiv c1 c1') (_ : cequiv c2 c2'),
 cequiv (CIf b c1 c2) (CIf b' c1' c2'))).
idtac "Assumptions:".
Abort.
Print Assumptions CIf_congruence.
Goal True.
idtac " ".

idtac "———————- not_congr ———————".
idtac " ".

idtac "#> Manually graded: not_congr".
idtac "Advanced".
idtac "Possible points: 3".
print_manual_grade manual_grade_for_not_congr.
idtac " ".

idtac "———————- fold_constants_com_sound ———————".
idtac " ".

idtac "#> fold_constants_com_sound".
idtac "Possible points: 3".
check_type @fold_constants_com_sound ((ctrans_sound fold_constants_com)).
```

```
idtac "Assumptions:".
Abort.
Print Assumptions fold_constants_com_sound.
Goal True.
idtac " ".

idtac "———————- inequiv_exercise ———————".
idtac " ".

idtac "#> inequiv_exercise".
idtac "Possible points: 3".
check_type @inequiv_exercise ((not (cequiv (CWhile BTrue CSkip) CSkip))).
idtac "Assumptions:".
Abort.
Print Assumptions inequiv_exercise.
Goal True.
idtac " ".

idtac "———————- himp_ceval ———————".
idtac " ".

idtac "#> Manually graded: Himp.Check_rule_for_HAVOC".
idtac "Possible points: 2".
print_manual_grade Himp.manual_grade_for_Check_rule_for_HAVOC.
idtac " ".

idtac "———————- havoc_swap ———————".
idtac " ".

idtac "#> Himp.pXY_cequiv_pYX".
idtac "Possible points: 3".
check_type @Himp.pXY_cequiv_pYX (
(or (Himp.cequiv Himp.pXY Himp.pYX) (not (Himp.cequiv Himp.pXY Himp.pYX)))).
idtac "Assumptions:".
Abort.
Print Assumptions Himp.pXY_cequiv_pYX.
Goal True.
idtac " ".

idtac "———————- p1_p2_term ———————".
idtac " ".

idtac "#> Himp.p1_may_diverge".
idtac "Advanced".
idtac "Possible points: 3".
check_type @Himp.p1_may_diverge (
(∀ (st : ∀ _ : String.string, nat) (st' : state)
    (_ : not (@eq nat (st X) 0)),
```

```
 not (Himp.ceval Himp.p1 st st'))).
idtac "Assumptions:".
Abort.
Print Assumptions Himp.p1_may_diverge.
Goal True.
idtac " ".

idtac "#> Himp.p2_may_diverge".
idtac "Advanced".
idtac "Possible points: 3".
check_type @Himp.p2_may_diverge (
(∀ (st : ∀ _ : String.string, nat) (st' : state)
    (_ : not (@eq nat (st X) 0)),
 not (Himp.ceval Himp.p2 st st'))).
idtac "Assumptions:".
Abort.
Print Assumptions Himp.p2_may_diverge.
Goal True.
idtac " ".

idtac "———————- p1_p2_equiv ———————".
idtac " ".

idtac "#> Himp.p1_p2_equiv".
idtac "Advanced".
idtac "Possible points: 6".
check_type @Himp.p1_p2_equiv ((Himp.cequiv Himp.p1 Himp.p2)).
idtac "Assumptions:".
Abort.
Print Assumptions Himp.p1_p2_equiv.
Goal True.
idtac " ".

idtac "———————- p3_p4_inequiv ———————".
idtac " ".

idtac "#> Himp.p3_p4_inequiv".
idtac "Advanced".
idtac "Possible points: 6".
check_type @Himp.p3_p4_inequiv ((not (Himp.cequiv Himp.p3 Himp.p4))).
idtac "Assumptions:".
Abort.
Print Assumptions Himp.p3_p4_inequiv.
Goal True.
idtac " ".

idtac " ".
```

```
idtac "Max points - standard: 25".
idtac "Max points - advanced: 46".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "".
idtac "".
idtac "********** Summary **********".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "********** Standard **********".
idtac "---------- skip_right ----------".
Print Assumptions skip_right.
idtac "---------- if_false ----------".
Print Assumptions if_false.
idtac "---------- swap_if_branches ----------".
Print Assumptions swap_if_branches.
idtac "---------- while_true ----------".
Print Assumptions while_true.
idtac "---------- assign_aequiv ----------".
Print Assumptions assign_aequiv.
idtac "---------- CIf_congruence ----------".
Print Assumptions CIf_congruence.
idtac "---------- fold_constants_com_sound ----------".
Print Assumptions fold_constants_com_sound.
idtac "---------- inequiv_exercise ----------".
Print Assumptions inequiv_exercise.
idtac "---------- Check_rule_for_HAVOC ----------".
idtac "MANUAL".
idtac "---------- Himp.pXY_cequiv_pYX ----------".
Print Assumptions Himp.pXY_cequiv_pYX.
idtac "".
idtac "********** Advanced **********".
idtac "---------- not_congr ----------".
```

```
idtac "MANUAL".
idtac "———- Himp.p1_may_diverge ———".
Print Assumptions Himp.p1_may_diverge.
idtac "———- Himp.p2_may_diverge ———".
Print Assumptions Himp.p2_may_diverge.
idtac "———- Himp.p1_p2_equiv ———".
Print Assumptions Himp.p1_p2_equiv.
idtac "———- Himp.p3_p4_inequiv ———".
Print Assumptions Himp.p3_p4_inequiv.
Abort.
```

Chapter 15

Library SECF.HoareTest

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib Require Export String.
From SECF Require Import Hoare.

Parameter MISSING: Type.

Module CHECK.

Ltac check_type A B :=
    match type of A with
    | context[MISSING] ⇒ idtac "Missing:" A
    | ?T ⇒ first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
")"]
    end.

Ltac print_manual_grade A :=
    match eval compute in A with
    | Some (_ ?S ?C) ⇒
        idtac "Score:" S;
        match eval compute in C with
          | ""%string ⇒ idtac "Comment: None"
          | _ ⇒ idtac "Comment:" C
        end
    | None ⇒
        idtac "Score: Ungraded";
        idtac "Comment: None"
    end.

End CHECK.

From SECF Require Import Hoare.
Import CHECK.

Goal True.

idtac "————————- hoare_post_true ————————".
```

```
idtac " ".

idtac "#> hoare_post_true".
idtac "Possible points: 1".
check_type @hoare_post_true (
(∀ (P Q : Assertion) (c : com) (_ : ∀ st : state, Q st),
 valid_hoare_triple P c Q)).
idtac "Assumptions:".
Abort.
Print Assumptions hoare_post_true.
Goal True.
idtac " ".

idtac "———————- hoare_asgn_wrong ———————".
idtac " ".

idtac "#> hoare_asgn_wrong".
idtac "Possible points: 2".
check_type @hoare_asgn_wrong (
(@ex aexp
   (fun a : aexp ⇒
    not
      (valid_hoare_triple (fun _ : state ⇒ True)
         (CAsgn X a)
         (fun st : state ⇒
          @eq nat ((Aexp_of_aexp (AId X) : Aexp) st)
            ((Aexp_of_aexp a : Aexp) st))))).
idtac "Assumptions:".
Abort.
Print Assumptions hoare_asgn_wrong.
Goal True.
idtac " ".

idtac "———————- hoare_asgn_examples_2 ———————".
idtac " ".

idtac "#> assertion_sub_ex1'".
idtac "Possible points: 1".
check_type @assertion_sub_ex1' (
(valid_hoare_triple
   (fun st : state ⇒
    le ((Aexp_of_aexp (AId X) : Aexp) st) ((Aexp_of_nat 5 : Aexp) st))
   (CAsgn X (AMult (ANum 2) (AId X)))
   (fun st : state ⇒
    le ((Aexp_of_aexp (AId X) : Aexp) st) ((Aexp_of_nat 10 : Aexp) st)))).
idtac "Assumptions:".
```

```
Abort.
Print Assumptions assertion_sub_ex1'.
Goal True.
idtac " ".

idtac "#> assertion_sub_ex2'".
idtac "Possible points: 1".
check_type @assertion_sub_ex2' (
(valid_hoare_triple
    (fun st : state ⇒
     and
       (((fun st0 : state ⇒
          le ((Aexp_of_nat 0 : Aexp) st0) ((Aexp_of_nat 3 : Aexp) st0))
         :
         Assertion) st)
       (((fun st0 : state ⇒
          le ((Aexp_of_nat 3 : Aexp) st0) ((Aexp_of_nat 5 : Aexp) st0))
         :
         Assertion) st))
    (CAsgn X (ANum 3))
    (fun st : state ⇒
     and
       (((fun st0 : state ⇒
          le ((Aexp_of_nat 0 : Aexp) st0) ((Aexp_of_aexp (AId X) : Aexp) st0))
         :
         Assertion) st)
       (((fun st0 : state ⇒
          le ((Aexp_of_aexp (AId X) : Aexp) st0) ((Aexp_of_nat 5 : Aexp) st0))
         :
         Assertion) st)))).
idtac "Assumptions:".
Abort.
Print Assumptions assertion_sub_ex2'.
Goal True.
idtac " ".

idtac "———————- hoare_asgn_example4 ———————".
idtac " ".

idtac "#> hoare_asgn_example4".
idtac "Possible points: 2".
check_type @hoare_asgn_example4 (
(valid_hoare_triple (fun _ : state ⇒ True)
    (CSeq (CAsgn X (ANum 1)) (CAsgn Y (ANum 2)))
    (fun st : state ⇒
```

```
    and
      (((fun st0 : state ⇒
          @eq nat ((Aexp_of_aexp (AId X) : Aexp) st0)
            ((Aexp_of_nat 1 : Aexp) st0))
        :
        Assertion) st)
      (((fun st0 : state ⇒
          @eq nat ((Aexp_of_aexp (AId Y) : Aexp) st0)
            ((Aexp_of_nat 2 : Aexp) st0))
        :
        Assertion) st)))).
idtac "Assumptions:".
Abort.
Print Assumptions hoare_asgn_example4.
Goal True.
idtac " ".

idtac "———————- swap_exercise ———————".
idtac " ".

idtac "#> swap_exercise".
idtac "Possible points: 3".
check_type @swap_exercise (
(valid_hoare_triple
    (fun st : state ⇒
     le ((Aexp_of_aexp (AId X) : Aexp) st) ((Aexp_of_aexp (AId Y) : Aexp) st))
    swap_program
    (fun st : state ⇒
     le ((Aexp_of_aexp (AId Y) : Aexp) st) ((Aexp_of_aexp (AId X) : Aexp) st)))).
idtac "Assumptions:".
Abort.
Print Assumptions swap_exercise.
Goal True.
idtac " ".

idtac "———————- invalid_triple ———————".
idtac " ".

idtac "#> invalid_triple".
idtac "Advanced".
idtac "Possible points: 6".
check_type @invalid_triple (
(not
    (∀ (a : aexp) (n : nat),
     valid_hoare_triple
```

```
        (fun st : state ⇒
         @eq nat ((Aexp_of_aexp a : Aexp) st) ((Aexp_of_nat n : Aexp) st))
        (CSeq (CAsgn X (ANum 3)) (CAsgn Y a))
        (fun st : state ⇒
         @eq nat ((Aexp_of_aexp (AId Y) : Aexp) st) ((Aexp_of_nat n : Aexp) st))))).
idtac "Assumptions:".
Abort.
Print Assumptions invalid_triple.
Goal True.
idtac " ".

idtac "———————- if_minus_plus ———————".
idtac " ".

idtac "#> if_minus_plus".
idtac "Possible points: 2".
check_type @if_minus_plus (
(valid_hoare_triple (fun _ : state ⇒ True)
    (CIf (BLe (AId X) (AId Y)) (CAsgn Z (AMinus (AId Y) (AId X)))
       (CAsgn Y (APlus (AId X) (AId Z))))
    (fun st : state ⇒
     @eq nat ((Aexp_of_aexp (AId Y) : Aexp) st)
       (((fun st0 : state ⇒
           PeanoNat.Nat.add ((Aexp_of_aexp (AId X) : Aexp) st0)
             ((Aexp_of_aexp (AId Z) : Aexp) st0))
         :
         Aexp) st)))).
idtac "Assumptions:".
Abort.
Print Assumptions if_minus_plus.
Goal True.
idtac " ".

idtac "———————- if1_ceval ———————".
idtac " ".

idtac "#> If1.if1true_test".
idtac "Possible points: 1".
check_type @If1.if1true_test (
(If1.ceval (If1.CIf1 (BEq (AId X) (ANum 0)) (If1.CAsgn X (ANum 1))) empty_st
    (@Maps.t_update nat empty_st X 1))).
idtac "Assumptions:".
Abort.
Print Assumptions If1.if1true_test.
Goal True.
```

```
idtac " ".

idtac "#> If1.if1false_test".
idtac "Possible points: 1".
check_type @If1.if1false_test (
(If1.ceval (If1.CIf1 (BEq (AId X) (ANum 0)) (If1.CAsgn X (ANum 1)))
    (@Maps.t_update nat empty_st X 2) (@Maps.t_update nat empty_st X 2))).
idtac "Assumptions:".
Abort.
Print Assumptions If1.if1false_test.
Goal True.
idtac " ".

idtac "———————- hoare_if1 ———————".
idtac " ".

idtac "#> Manually graded: If1.hoare_if1".
idtac "Possible points: 2".
print_manual_grade If1.manual_grade_for_hoare_if1.
idtac " ".

idtac "———————- hoare_if1_good ———————".
idtac " ".

idtac "#> If1.hoare_if1_good".
idtac "Possible points: 2".
check_type @If1.hoare_if1_good (
(If1.valid_hoare_triple
    (fun st : state ⇒
     @eq nat
       (((fun st0 : state ⇒
           PeanoNat.Nat.add ((Aexp_of_aexp (AId X) : Aexp) st0)
             ((Aexp_of_aexp (AId Y) : Aexp) st0))
         :
         Aexp) st)
       ((Aexp_of_aexp (AId Z) : Aexp) st))
    (If1.CIf1 (BNeq (AId Y) (ANum 0)) (If1.CAsgn X (APlus (AId X) (AId Y))))
    (fun st : state ⇒
     @eq nat ((Aexp_of_aexp (AId X) : Aexp) st)
       ((Aexp_of_aexp (AId Z) : Aexp) st)))).
idtac "Assumptions:".
Abort.
Print Assumptions If1.hoare_if1_good.
Goal True.
idtac " ".

idtac "———————- hoare_havoc ———————".
```

```
idtac " ".

idtac "#> Himp.hoare_havoc".
idtac "Advanced".
idtac "Possible points: 3".
check_type @Himp.hoare_havoc (
(∀ (Q : Assertion) (X : String.string),
 Himp.valid_hoare_triple (Himp.havoc_pre X Q) (Himp.CHavoc X) Q)).
idtac "Assumptions:".
Abort.
Print Assumptions Himp.hoare_havoc.
Goal True.
idtac " ".

idtac "———————- havoc_post ———————".
idtac " ".

idtac "#> Himp.havoc_post".
idtac "Advanced".
idtac "Possible points: 3".
check_type @Himp.havoc_post (
(∀ (P : Assertion) (X : String.string),
 Himp.valid_hoare_triple P (Himp.CHavoc X)
   (fun st : state ⇒ @ex nat (fun n : nat ⇒ assertion_sub X (ANum n) P st)))).
idtac "Assumptions:".
Abort.
Print Assumptions Himp.havoc_post.
Goal True.
idtac " ".

idtac "———————- assert_vs_assume ———————".
idtac " ".

idtac "#> HoareAssertAssume.assert_assume_differ".
idtac "Possible points: 1".
check_type @HoareAssertAssume.assert_assume_differ (
(@ex Assertion
   (fun P : Assertion ⇒
    @ex bexp
      (fun b : bexp ⇒
       @ex Assertion
         (fun Q : Assertion ⇒
          and
            (HoareAssertAssume.valid_hoare_triple P
               (HoareAssertAssume.CAssume b) Q)
            (not
```

```
                    (HoareAssertAssume.valid_hoare_triple P
                       (HoareAssertAssume.CAssert b) Q)))))))).
idtac "Assumptions:".
Abort.
Print Assumptions HoareAssertAssume.assert_assume_differ.
Goal True.
idtac " ".

idtac "#> HoareAssertAssume.assert_implies_assume".
idtac "Possible points: 1".
check_type @HoareAssertAssume.assert_implies_assume (
(∀ (P : Assertion) (b : bexp) (Q : Assertion)
   (_ : HoareAssertAssume.valid_hoare_triple P (HoareAssertAssume.CAssert b)
          Q),
 HoareAssertAssume.valid_hoare_triple P (HoareAssertAssume.CAssume b) Q)).
idtac "Assumptions:".
Abort.
Print Assumptions HoareAssertAssume.assert_implies_assume.
Goal True.
idtac " ".

idtac "#> HoareAssertAssume.assert_assume_example".
idtac "Possible points: 4".
check_type @HoareAssertAssume.assert_assume_example (
(HoareAssertAssume.valid_hoare_triple (fun _ : state ⇒ True)
   (HoareAssertAssume.CSeq (HoareAssertAssume.CAssume (BEq (AId X) (ANum 1)))
      (HoareAssertAssume.CSeq
         (HoareAssertAssume.CAsgn X (APlus (AId X) (ANum 1)))
         (HoareAssertAssume.CAssert (BEq (AId X) (ANum 2)))))
   (fun _ : state ⇒ True))).
idtac "Assumptions:".
Abort.
Print Assumptions HoareAssertAssume.assert_assume_example.
Goal True.
idtac " ".

idtac " ".

idtac "Max points - standard: 24".
idtac "Max points - advanced: 36".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "".
```

```
idtac "".
idtac "********** Summary **********".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "********** Standard **********".
idtac "---------- hoare_post_true ----------".
Print Assumptions hoare_post_true.
idtac "---------- hoare_asgn_wrong ----------".
Print Assumptions hoare_asgn_wrong.
idtac "---------- assertion_sub_ex1' ----------".
Print Assumptions assertion_sub_ex1'.
idtac "---------- assertion_sub_ex2' ----------".
Print Assumptions assertion_sub_ex2'.
idtac "---------- hoare_asgn_example4 ----------".
Print Assumptions hoare_asgn_example4.
idtac "---------- swap_exercise ----------".
Print Assumptions swap_exercise.
idtac "---------- if_minus_plus ----------".
Print Assumptions if_minus_plus.
idtac "---------- If1.if1true_test ----------".
Print Assumptions If1.if1true_test.
idtac "---------- If1.if1false_test ----------".
Print Assumptions If1.if1false_test.
idtac "---------- hoare_if1 ----------".
idtac "MANUAL".
idtac "---------- If1.hoare_if1_good ----------".
Print Assumptions If1.hoare_if1_good.
idtac "---------- HoareAssertAssume.assert_assume_differ ----------".
Print Assumptions HoareAssertAssume.assert_assume_differ.
idtac "---------- HoareAssertAssume.assert_implies_assume ----------".
Print Assumptions HoareAssertAssume.assert_implies_assume.
idtac "---------- HoareAssertAssume.assert_assume_example ----------".
Print Assumptions HoareAssertAssume.assert_assume_example.
idtac "".
idtac "********** Advanced **********".
```

```
idtac "------- invalid_triple -------".
Print Assumptions invalid_triple.
idtac "------- Himp.hoare_havoc -------".
Print Assumptions Himp.hoare_havoc.
idtac "------- Himp.havoc_post -------".
Print Assumptions Himp.havoc_post.
Abort.
```

Chapter 16

Library SECF.Hoare2Test

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib Require Export String.
From SECF Require Import Hoare2.

Parameter MISSING: Type.

Module CHECK.

Ltac check_type A B :=
    match type of A with
    | context[MISSING] ⇒ idtac "Missing:" A
    | ?T ⇒ first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
")"]
    end.

Ltac print_manual_grade A :=
    match eval compute in A with
    | Some (_ ?S ?C) ⇒
        idtac "Score:" S;
        match eval compute in C with
          | ""%string ⇒ idtac "Comment: None"
          | _ ⇒ idtac "Comment:" C
        end
    | None ⇒
        idtac "Score: Ungraded";
        idtac "Comment: None"
    end.

End CHECK.

From SECF Require Import Hoare2.
Import CHECK.

Goal True.

idtac "————————- if_minus_plus_correct ————————".
```

```
idtac " ".
idtac "#> if_minus_plus_correct".
idtac "Possible points: 2".
check_type @if_minus_plus_correct ((outer_triple_valid if_minus_plus_dec)).
idtac "Assumptions:".
Abort.
Print Assumptions if_minus_plus_correct.
Goal True.
idtac " ".
idtac "———————- slow_assignment ———————".
idtac " ".
idtac "#> slow_assignment".
idtac "Possible points: 2".
check_type @slow_assignment (
(∀ m : nat, outer_triple_valid (slow_assignment_dec m))).
idtac "Assumptions:".
Abort.
Print Assumptions slow_assignment.
Goal True.
idtac " ".
idtac "———————- factorial_correct ———————".
idtac " ".
idtac "#> factorial_correct".
idtac "Advanced".
idtac "Possible points: 6".
check_type @factorial_correct ((∀ m : nat, outer_triple_valid (factorial_dec m))).
idtac "Assumptions:".
Abort.
Print Assumptions factorial_correct.
Goal True.
idtac " ".
idtac "———————- minimum_correct ———————".
idtac " ".
idtac "#> minimum_correct".
idtac "Advanced".
idtac "Possible points: 3".
check_type @minimum_correct ((∀ a b : nat, outer_triple_valid (minimum_dec a b))).
idtac "Assumptions:".
Abort.
Print Assumptions minimum_correct.
Goal True.
```

```
idtac " ".
idtac "———————- two_loops ———————".
idtac " ".
idtac "#> two_loops".
idtac "Possible points: 3".
check_type @two_loops ((∀ a b c : nat, outer_triple_valid (two_loops_dec a b c))).
idtac "Assumptions:".
Abort.
Print Assumptions two_loops.
Goal True.
idtac " ".
idtac " ".
idtac "Max points - standard: 7".
idtac "Max points - advanced: 16".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "".
idtac "".
idtac "********** Summary **********".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "********** Standard **********".
idtac "———— if_minus_plus_correct ————".
Print Assumptions if_minus_plus_correct.
idtac "———— slow_assignment ————".
Print Assumptions slow_assignment.
idtac "———— two_loops ————".
Print Assumptions two_loops.
idtac "".
idtac "********** Advanced **********".
idtac "———— factorial_correct ————".
Print Assumptions factorial_correct.
```

```
idtac "------- minimum_correct -------".
Print Assumptions minimum_correct.
Abort.
```

Chapter 17

Library SECF.PrefaceTest

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib Require Export String.
From SECF Require Import Preface.

Parameter MISSING: Type.

Module CHECK.

Ltac check_type A B :=
    match type of A with
    | context[MISSING] ⇒ idtac "Missing:" A
    | ?T ⇒ first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
")"]
    end.

Ltac print_manual_grade A :=
    match eval compute in A with
    | Some (_ ?S ?C) ⇒
        idtac "Score:" S;
        match eval compute in C with
          | ""%string ⇒ idtac "Comment: None"
          | _ ⇒ idtac "Comment:" C
        end
    | None ⇒
        idtac "Score: Ungraded";
        idtac "Comment: None"
    end.

End CHECK.

From SECF Require Import Preface.
Import CHECK.

Goal True.

idtac " ".
```

```
idtac "Max points - standard: 0".
idtac "Max points - advanced: 0".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "".
idtac "".
idtac "********** Summary **********".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "********** Standard **********".
idtac "".
idtac "********** Advanced **********".
Abort.
```

Chapter 18

Library SECF.NoninterferenceTest

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib Require Export String.
From SECF Require Import Noninterference.

Parameter MISSING: Type.

Module CHECK.

Ltac check_type A B :=
    match type of A with
    | context[MISSING] ⇒ idtac "Missing:" A
    | ?T ⇒ first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
")"]
    end.

Ltac print_manual_grade A :=
    match eval compute in A with
    | Some (_ ?S ?C) ⇒
        idtac "Score:" S;
        match eval compute in C with
          | ""%string ⇒ idtac "Comment: None"
          | _ ⇒ idtac "Comment:" C
        end
    | None ⇒
        idtac "Score: Ungraded";
        idtac "Comment: None"
    end.

End CHECK.

From SECF Require Import Noninterference.
Import CHECK.

Goal True.

idtac "————————- prove_or_disprove_obvious_f1 ————————".
```

```
idtac " ".
idtac "#> prove_or_disprove_obvious_f1".
idtac "Possible points: 1".
check_type @prove_or_disprove_obvious_f1 (
(or (@noninterferent nat nat nat nat obvious_f1)
    (not (@noninterferent nat nat nat nat obvious_f1)))).
idtac "Assumptions:".
Abort.
Print Assumptions prove_or_disprove_obvious_f1.
Goal True.
idtac " ".
idtac "———————- prove_or_disprove_obvious_f2 ———————".
idtac " ".
idtac "#> prove_or_disprove_obvious_f2".
idtac "Possible points: 1".
check_type @prove_or_disprove_obvious_f2 (
(or (@noninterferent nat nat nat nat obvious_f2)
    (not (@noninterferent nat nat nat nat obvious_f2)))).
idtac "Assumptions:".
Abort.
Print Assumptions prove_or_disprove_obvious_f2.
Goal True.
idtac " ".
idtac "———————- prove_or_disprove_less_obvious_f4 ———————".
idtac " ".
idtac "#> prove_or_disprove_less_obvious_f4".
idtac "Possible points: 2".
check_type @prove_or_disprove_less_obvious_f4 (
(or (@noninterferent nat nat nat nat less_obvious_f4)
    (not (@noninterferent nat nat nat nat less_obvious_f4)))).
idtac "Assumptions:".
Abort.
Print Assumptions prove_or_disprove_less_obvious_f4.
Goal True.
idtac " ".
idtac "———————- prove_or_disprove_less_obvious_f5 ———————".
idtac " ".
idtac "#> prove_or_disprove_less_obvious_f5".
idtac "Possible points: 2".
check_type @prove_or_disprove_less_obvious_f5 (
(or (@noninterferent nat nat nat nat less_obvious_f5)
```

```
    (not (@noninterferent nat nat nat nat less_obvious_f5)))).
idtac "Assumptions:".
Abort.
Print Assumptions prove_or_disprove_less_obvious_f5.
Goal True.
idtac " ".

idtac "———————- prove_or_disprove_less_obvious_f6 ———————".
idtac " ".

idtac "#> prove_or_disprove_less_obvious_f6".
idtac "Possible points: 2".
check_type @prove_or_disprove_less_obvious_f6 (
(or (@noninterferent nat nat nat nat less_obvious_f6)
    (not (@noninterferent nat nat nat nat less_obvious_f6)))).
idtac "Assumptions:".
Abort.
Print Assumptions prove_or_disprove_less_obvious_f6.
Goal True.
idtac " ".

idtac "———————- sme_another_insecure_f2 ———————".
idtac " ".

idtac "#> sme_another_insecure_f2".
idtac "Possible points: 1".
check_type @sme_another_insecure_f2 (
(∀ pi si : nat,
 @eq (prod nat nat) (@sme nat nat nat nat 0 another_insecure_f2 pi si)
    (@pair nat nat pi (PeanoNat.Nat.add pi si)))).
idtac "Assumptions:".
Abort.
Print Assumptions sme_another_insecure_f2.
Goal True.
idtac " ".

idtac "———————- sme_another_insecure_f3 ———————".
idtac " ".

idtac "#> sme_another_insecure_f3".
idtac "Possible points: 2".
check_type @sme_another_insecure_f3 (
(∀ pi si : nat,
 @eq (prod nat nat) (@sme nat nat nat nat 0 another_insecure_f3 pi si)
    (@pair nat nat pi (PeanoNat.Nat.add pi si)))).
idtac "Assumptions:".
Abort.
```

```
Print Assumptions sme_another_insecure_f3.
Goal True.
idtac " ".
idtac "———————- noninterferent_secure_ex1 ———————".
idtac " ".
idtac "#> noninterferent_secure_ex1".
idtac "Possible points: 2".
check_type @noninterferent_secure_ex1 ((noninterferent_no_while xpub secure_ex1)).
idtac "Assumptions:".
Abort.
Print Assumptions noninterferent_secure_ex1.
Goal True.
idtac " ".
idtac "———————- interferent_insecure_com_explicit ———————".
idtac " ".
idtac "#> interferent_insecure_com_explicit".
idtac "Possible points: 2".
check_type @interferent_insecure_com_explicit (
(not (noninterferent_no_while xpub insecure_com_explicit))).
idtac "Assumptions:".
Abort.
Print Assumptions interferent_insecure_com_explicit.
Goal True.
idtac " ".
idtac "———————- interferent_insecure_com_implicit ———————".
idtac " ".
idtac "#> interferent_insecure_com_implicit".
idtac "Possible points: 3".
check_type @interferent_insecure_com_implicit (
(not (noninterferent_no_while xpub insecure_com_implicit))).
idtac "Assumptions:".
Abort.
Print Assumptions interferent_insecure_com_implicit.
Goal True.
idtac " ".
idtac " ".
idtac "Max points - standard: 18".
idtac "Max points - advanced: 18".
idtac "".
idtac "Allowed Axioms:".
```

```
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "".
idtac "".
idtac "********** Summary **********".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "********** Standard **********".
idtac "---------- prove_or_disprove_obvious_f1 ----------".
Print Assumptions prove_or_disprove_obvious_f1.
idtac "---------- prove_or_disprove_obvious_f2 ----------".
Print Assumptions prove_or_disprove_obvious_f2.
idtac "---------- prove_or_disprove_less_obvious_f4 ----------".
Print Assumptions prove_or_disprove_less_obvious_f4.
idtac "---------- prove_or_disprove_less_obvious_f5 ----------".
Print Assumptions prove_or_disprove_less_obvious_f5.
idtac "---------- prove_or_disprove_less_obvious_f6 ----------".
Print Assumptions prove_or_disprove_less_obvious_f6.
idtac "---------- sme_another_insecure_f2 ----------".
Print Assumptions sme_another_insecure_f2.
idtac "---------- sme_another_insecure_f3 ----------".
Print Assumptions sme_another_insecure_f3.
idtac "---------- noninterferent_secure_ex1 ----------".
Print Assumptions noninterferent_secure_ex1.
idtac "---------- interferent_insecure_com_explicit ----------".
Print Assumptions interferent_insecure_com_explicit.
idtac "---------- interferent_insecure_com_implicit ----------".
Print Assumptions interferent_insecure_com_implicit.
idtac "".
idtac "********** Advanced **********".
Abort.
```

Chapter 19

Library SECF.StaticIFCTest

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib Require Export String.
From SECF Require Import StaticIFC.

Parameter MISSING: Type.

Module CHECK.

Ltac check_type A B :=
    match type of A with
    | context[MISSING] ⇒ idtac "Missing:" A
    | ?T ⇒ first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
")"]
    end.

Ltac print_manual_grade A :=
    match eval compute in A with
    | Some (_ ?S ?C) ⇒
        idtac "Score:" S;
        match eval compute in C with
          | ""%string ⇒ idtac "Comment: None"
          | _ ⇒ idtac "Comment:" C
        end
    | None ⇒
        idtac "Score: Ungraded";
        idtac "Comment: None"
    end.

End CHECK.

From SECF Require Import StaticIFC.
Import CHECK.

Goal True.

idtac "————————- not_cf_wt_noninterferent_com ————————".
```

```
idtac " ".
idtac "#> not_cf_wt_noninterferent_com".
idtac "Possible points: 1".
check_type @not_cf_wt_noninterferent_com (
(not
    (cf_well_typed xpub (CIf (BEq (AId Y) (ANum 0)) (CAsgn Z (ANum 0)) CSkip)))).
idtac "Assumptions:".
Abort.
Print Assumptions not_cf_wt_noninterferent_com.
Goal True.
idtac " ".
idtac "———————- ts_typechecker ———————".
idtac " ".
idtac "#> ts_typechecker".
idtac "Possible points: 2".
check_type @ts_typechecker ((∀ (_ : pub_vars) (_ : label) (_ : com), bool)).
idtac "Assumptions:".
Abort.
Print Assumptions ts_typechecker.
Goal True.
idtac " ".
idtac "———————- ts_typechecker_sound ———————".
idtac " ".
idtac "#> ts_typechecker_sound".
idtac "Possible points: 2".
check_type @ts_typechecker_sound (
(∀ (P : pub_vars) (pc : label) (c : com)
    (_ : @eq bool (ts_typechecker P pc c) true),
 ts_well_typed P pc c)).
idtac "Assumptions:".
Abort.
Print Assumptions ts_typechecker_sound.
Goal True.
idtac " ".
idtac "———————- ts_typechecker_complete ———————".
idtac " ".
idtac "#> ts_typechecker_complete".
idtac "Possible points: 2".
check_type @ts_typechecker_complete (
(∀ (P : pub_vars) (pc : label) (c : com)
    (_ : @eq bool (ts_typechecker P pc c) false),
```

```
 not (ts_well_typed P pc c))).
idtac "Assumptions:".
Abort.
Print Assumptions ts_typechecker_complete.
Goal True.
idtac " ".

idtac "———————- not_ts_non_termination_com ———————".
idtac " ".

idtac "#> not_ts_non_termination_com".
idtac "Possible points: 1".
check_type @not_ts_non_termination_com (
(not (ts_well_typed xpub public termination_leak))).
idtac "Assumptions:".
Abort.
Print Assumptions not_ts_non_termination_com.
Goal True.
idtac " ".

idtac "———————- cf_well_typed_ts_cf_secure ———————".
idtac " ".

idtac "#> cf_well_typed_ts_cf_secure".
idtac "Possible points: 6".
check_type @cf_well_typed_ts_cf_secure (
(∀ (P : pub_vars) (c : com) (_ : cf_well_typed P c), ts_cf_secure P c)).
idtac "Assumptions:".
Abort.
Print Assumptions cf_well_typed_ts_cf_secure.
Goal True.
idtac " ".

idtac "———————- public_outputs ———————".
idtac " ".

idtac "#> OUTPUT.oni_typechecker_sound".
idtac "Possible points: 0.5".
check_type @OUTPUT.oni_typechecker_sound (
(∀ (P : pub_vars) (pc : label) (c : OUTPUT.com)
    (_ : @eq bool (OUTPUT.oni_typechecker P pc c) true),
 OUTPUT.oni_well_typed P pc c)).
idtac "Assumptions:".
Abort.
Print Assumptions OUTPUT.oni_typechecker_sound.
Goal True.
idtac " ".
```

```
idtac "#> OUTPUT.oni_typechecker_complete".
idtac "Possible points: 0.5".
check_type @OUTPUT.oni_typechecker_complete (
(∀ (P : pub_vars) (pc : label) (c : OUTPUT.com)
    (_ : @eq bool (OUTPUT.oni_typechecker P pc c) false),
 not (OUTPUT.oni_well_typed P pc c))).
idtac "Assumptions:".
Abort.
Print Assumptions OUTPUT.oni_typechecker_complete.
Goal True.
idtac " ".

idtac "#> OUTPUT.not_ni_wt_output1".
idtac "Possible points: 0.5".
check_type @OUTPUT.not_ni_wt_output1 (
(not (OUTPUT.oni_well_typed xpub public OUTPUT.output_insecure_com1))).
idtac "Assumptions:".
Abort.
Print Assumptions OUTPUT.not_ni_wt_output1.
Goal True.
idtac " ".

idtac "#> OUTPUT.not_ni_wt_output2".
idtac "Possible points: 0.5".
check_type @OUTPUT.not_ni_wt_output2 (
(not (OUTPUT.oni_well_typed xpub public OUTPUT.output_insecure_com2))).
idtac "Assumptions:".
Abort.
Print Assumptions OUTPUT.not_ni_wt_output2.
Goal True.
idtac " ".

idtac "#> OUTPUT.weaken_pc".
idtac "Possible points: 1".
check_type @OUTPUT.weaken_pc (
(∀ (P : pub_vars) (pc1 pc2 : label) (c : OUTPUT.com)
    (_ : OUTPUT.oni_well_typed P pc1 c) (_ : @eq bool (can_flow pc2 pc1) true),
 OUTPUT.oni_well_typed P pc2 c)).
idtac "Assumptions:".
Abort.
Print Assumptions OUTPUT.weaken_pc.
Goal True.
idtac " ".

idtac "#> OUTPUT.secret_run_no_output".
```

```
idtac "Possible points: 2".
check_type @OUTPUT.secret_run_no_output (
(∀ (P : pub_vars) (c : OUTPUT.com) (s s' : state)
    (os : OUTPUT.outputs) (_ : OUTPUT.oni_well_typed P secret c)
    (_ : OUTPUT.oceval c s s' os),
 @eq OUTPUT.outputs os (@nil nat))).
idtac "Assumptions:".
Abort.
Print Assumptions OUTPUT.secret_run_no_output.
Goal True.
idtac " ".

idtac "#> OUTPUT.oni_well_typed_noninterferent".
idtac "Possible points: 2".
check_type @OUTPUT.oni_well_typed_noninterferent (
(∀ (P : pub_vars) (c : OUTPUT.com)
    (_ : OUTPUT.oni_well_typed P public c),
 OUTPUT.noninterferent P c)).
idtac "Assumptions:".
Abort.
Print Assumptions OUTPUT.oni_well_typed_noninterferent.
Goal True.
idtac " ".

idtac "#> OUTPUT.oni_well_typed_output_secure".
idtac "Possible points: 3".
check_type @OUTPUT.oni_well_typed_output_secure (
(∀ (P : pub_vars) (c : OUTPUT.com)
    (_ : OUTPUT.oni_well_typed P public c),
 OUTPUT.output_secure P c)).
idtac "Assumptions:".
Abort.
Print Assumptions OUTPUT.oni_well_typed_output_secure.
Goal True.
idtac " ".

idtac " ".

idtac "Max points - standard: 24".
idtac "Max points - advanced: 24".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "".
```

```
idtac "".
idtac "********** Summary **********".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "********** Standard **********".
idtac "---------- not_cf_wt_noninterferent_com ----------".
Print Assumptions not_cf_wt_noninterferent_com.
idtac "---------- ts_typechecker ----------".
Print Assumptions ts_typechecker.
idtac "---------- ts_typechecker_sound ----------".
Print Assumptions ts_typechecker_sound.
idtac "---------- ts_typechecker_complete ----------".
Print Assumptions ts_typechecker_complete.
idtac "---------- not_ts_non_termination_com ----------".
Print Assumptions not_ts_non_termination_com.
idtac "---------- cf_well_typed_ts_cf_secure ----------".
Print Assumptions cf_well_typed_ts_cf_secure.
idtac "---------- OUTPUT.oni_typechecker_sound ----------".
Print Assumptions OUTPUT.oni_typechecker_sound.
idtac "---------- OUTPUT.oni_typechecker_complete ----------".
Print Assumptions OUTPUT.oni_typechecker_complete.
idtac "---------- OUTPUT.not_ni_wt_output1 ----------".
Print Assumptions OUTPUT.not_ni_wt_output1.
idtac "---------- OUTPUT.not_ni_wt_output2 ----------".
Print Assumptions OUTPUT.not_ni_wt_output2.
idtac "---------- OUTPUT.weaken_pc ----------".
Print Assumptions OUTPUT.weaken_pc.
idtac "---------- OUTPUT.secret_run_no_output ----------".
Print Assumptions OUTPUT.secret_run_no_output.
idtac "---------- OUTPUT.oni_well_typed_noninterferent ----------".
Print Assumptions OUTPUT.oni_well_typed_noninterferent.
idtac "---------- OUTPUT.oni_well_typed_output_secure ----------".
Print Assumptions OUTPUT.oni_well_typed_output_secure.
idtac "".
idtac "********** Advanced **********".
```

```
Abort.
```

Chapter 20

Library SECF.SpecCTTest

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib Require Export String.
From SECF Require Import SpecCT.

Parameter MISSING: Type.

Module CHECK.

Ltac check_type A B :=
    match type of A with
    | context[MISSING] ⇒ idtac "Missing:" A
    | ?T ⇒ first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
")"]
    end.

Ltac print_manual_grade A :=
    match eval compute in A with
    | Some (_ ?S ?C) ⇒
        idtac "Score:" S;
        match eval compute in C with
          | ""%string ⇒ idtac "Comment: None"
          | _ ⇒ idtac "Comment:" C
        end
    | None ⇒
        idtac "Score: Ungraded";
        idtac "Comment: None"
    end.

End CHECK.

From SECF Require Import SpecCT.
Import CHECK.

Goal True.

idtac "————————- cct_insecure_prog'_is_not_cct_secure ————————".
```

```
idtac " ".

idtac "#> cct_insecure_prog'_is_not_cct_secure".
idtac "Possible points: 2".
check_type @cct_insecure_prog'_is_not_cct_secure (
(not (cct_secure XYZpub APpub cct_insecure_prog'))).
idtac "Assumptions:".
Abort.
Print Assumptions cct_insecure_prog'_is_not_cct_secure.
Goal True.
idtac " ".

idtac "————————- label_of_exp ————————".
idtac " ".

idtac "#> label_of_exp".
idtac "Possible points: 1".
check_type @label_of_exp ((∀ (_ : pub_vars) (_ : exp), label)).
idtac "Assumptions:".
Abort.
Print Assumptions label_of_exp.
Goal True.
idtac " ".

idtac "————————- label_of_exp_sound ————————".
idtac " ".

idtac "#> label_of_exp_sound".
idtac "Possible points: 1".
check_type @label_of_exp_sound (
(∀ (P : pub_vars) (e : exp), exp_has_label P e (label_of_exp P e))).
idtac "Assumptions:".
Abort.
Print Assumptions label_of_exp_sound.
Goal True.
idtac " ".

idtac "————————- label_of_exp_unique ————————".
idtac " ".

idtac "#> label_of_exp_unique".
idtac "Possible points: 1".
check_type @label_of_exp_unique (
(∀ (P : pub_vars) (e : exp) (l : label) (_ : exp_has_label P e l),
 @eq label l (label_of_exp P e))).
idtac "Assumptions:".
Abort.
Print Assumptions label_of_exp_unique.
```

```
Goal True.
idtac " ".
idtac "———————- cct_typechecker ———————".
idtac " ".
idtac "#> cct_typechecker".
idtac "Possible points: 2".
check_type @cct_typechecker ((∀ (_ : pub_vars) (_ : pub_vars) (_ : com), bool)).
idtac "Assumptions:".
Abort.
Print Assumptions cct_typechecker.
Goal True.
idtac " ".
idtac "———————- cct_typechecker_sound ———————".
idtac " ".
idtac "#> cct_typechecker_sound".
idtac "Possible points: 2".
check_type @cct_typechecker_sound (
(∀ (P PA : pub_vars) (c : com)
    (_ : @eq bool (cct_typechecker P PA c) true),
 cct_well_typed P PA c)).
idtac "Assumptions:".
Abort.
Print Assumptions cct_typechecker_sound.
Goal True.
idtac " ".
idtac "———————- cct_typechecker_complete ———————".
idtac " ".
idtac "#> cct_typechecker_complete".
idtac "Possible points: 2".
check_type @cct_typechecker_complete (
(∀ (P PA : pub_vars) (c : com)
    (_ : @eq bool (cct_typechecker P PA c) false),
 not (cct_well_typed P PA c))).
idtac "Assumptions:".
Abort.
Print Assumptions cct_typechecker_complete.
Goal True.
idtac " ".
idtac "———————- cct_insecure_prog_ill_typed ———————".
idtac " ".
idtac "#> cct_insecure_prog_ill_typed".
```

```
idtac "Possible points: 1".
check_type @cct_insecure_prog_ill_typed (
(not (cct_well_typed XYZpub APpub cct_insecure_prog))).
idtac "Assumptions:".
Abort.
Print Assumptions cct_insecure_prog_ill_typed.
Goal True.
idtac " ".

idtac "———————- cct_insecure_prog'_ill_typed ———————".
idtac " ".

idtac "#> cct_insecure_prog'_ill_typed".
idtac "Possible points: 1".
check_type @cct_insecure_prog'_ill_typed (
(not (cct_well_typed XYZpub APpub cct_insecure_prog'))).
idtac "Assumptions:".
Abort.
Print Assumptions cct_insecure_prog'_ill_typed.
Goal True.
idtac " ".

idtac "———————- cct_well_typed_div ———————".
idtac " ".

idtac "#> Manually graded: Div.cct_well_typed_div".
idtac "Possible points: 1".
print_manual_grade Div.manual_grade_for_cct_well_typed_div.
idtac " ".

idtac "———————- cct_well_typed_div_noninterferent ———————".
idtac " ".

idtac "#> Div.cct_well_typed_div_noninterferent".
idtac "Possible points: 2".
check_type @Div.cct_well_typed_div_noninterferent (
(∀ (P : pub_vars) (PA : pub_arrs) (c : Div.com)
    (st1 st2 : Maps.total_map nat) (m1 m2 : Maps.total_map (list nat))
    (st1' st2' : state) (m1' m2' : mem) (os1 os2 : Div.obs)
    (_ : Div.cct_well_typed P PA c) (_ : @pub_equiv P nat st1 st2)
    (_ : @pub_equiv PA (list nat) m1 m2)
    (_ : Div.cteval c st1 m1 st1' m1' os1)
    (_ : Div.cteval c st2 m2 st2' m2' os2),
 and (@pub_equiv P nat st1' st2') (@pub_equiv PA (list nat) m1' m2'))).
idtac "Assumptions:".
Abort.
Print Assumptions Div.cct_well_typed_div_noninterferent.
```

```
Goal True.
idtac " ".

idtac "————————————- cct_well_typed_div_secure ————————————".
idtac " ".

idtac "#> Div.cct_well_typed_div_secure".
idtac "Possible points: 2".
check_type @Div.cct_well_typed_div_secure (
(∀ (P : pub_vars) (PA : pub_arrs) (c : Div.com)
    (_ : Div.cct_well_typed P PA c),
 Div.cct_secure P PA c)).
idtac "Assumptions:".
Abort.
Print Assumptions Div.cct_well_typed_div_secure.
Goal True.
idtac " ".

idtac "————————————- speculation_bit_monotonic ————————————".
idtac " ".

idtac "#> speculation_bit_monotonic".
idtac "Possible points: 1".
check_type @speculation_bit_monotonic (
(∀ (c : com) (s : state) (a : mem) (b : bool)
    (ds : dirs) (s' : state) (a' : mem) (b' : bool)
    (os : obs) (_ : spec_eval c s a b ds s' a' b' os)
    (_ : @eq bool b true),
 @eq bool b' true)).
idtac "Assumptions:".
Abort.
Print Assumptions speculation_bit_monotonic.
Goal True.
idtac " ".

idtac "————————————- ct_well_typed_seq_spec_eval_ct_secure ————————————".
idtac " ".

idtac "#> ct_well_typed_seq_spec_eval_ct_secure".
idtac "Possible points: 1".
check_type @ct_well_typed_seq_spec_eval_ct_secure (
(∀ (P : pub_vars) (PA : pub_arrs) (c : com)
    (st1 st2 : Maps.total_map nat) (m1 m2 : Maps.total_map (list nat))
    (st1' st2' : state) (m1' m2' : mem) (os1 os2 : obs)
    (_ : cct_well_typed P PA c) (_ : @pub_equiv P nat st1 st2)
    (_ : @pub_equiv PA (list nat) m1 m2)
    (_ : seq_spec_eval c st1 m1 st1' m1' os1)
```

```
    (_ : seq_spec_eval c st2 m2 st2' m2' os2),
  @eq obs os1 os2)).
idtac "Assumptions:".
Abort.
Print Assumptions ct_well_typed_seq_spec_eval_ct_secure.
Goal True.
idtac " ".

idtac " ".

idtac "Max points - standard: 20".
idtac "Max points - advanced: 20".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "".
idtac "".
idtac "********** Summary **********".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "********** Standard **********".
idtac "---------- cct_insecure_prog'_is_not_cct_secure ----------".
Print Assumptions cct_insecure_prog'_is_not_cct_secure.
idtac "---------- label_of_exp ----------".
Print Assumptions label_of_exp.
idtac "---------- label_of_exp_sound ----------".
Print Assumptions label_of_exp_sound.
idtac "---------- label_of_exp_unique ----------".
Print Assumptions label_of_exp_unique.
idtac "---------- cct_typechecker ----------".
Print Assumptions cct_typechecker.
idtac "---------- cct_typechecker_sound ----------".
Print Assumptions cct_typechecker_sound.
idtac "---------- cct_typechecker_complete ----------".
Print Assumptions cct_typechecker_complete.
idtac "---------- cct_insecure_prog_ill_typed ----------".
```

```
Print Assumptions cct_insecure_prog_ill_typed.
idtac "------- cct_insecure_prog'_ill_typed -------".
Print Assumptions cct_insecure_prog'_ill_typed.
idtac "------- cct_well_typed_div -------".
idtac "MANUAL".
idtac "------- Div.cct_well_typed_div_noninterferent -------".
Print Assumptions Div.cct_well_typed_div_noninterferent.
idtac "------- Div.cct_well_typed_div_secure -------".
Print Assumptions Div.cct_well_typed_div_secure.
idtac "------- speculation_bit_monotonic -------".
Print Assumptions speculation_bit_monotonic.
idtac "------- ct_well_typed_seq_spec_eval_ct_secure -------".
Print Assumptions ct_well_typed_seq_spec_eval_ct_secure.
idtac "".
idtac "********** Advanced **********".
Abort.
```

Chapter 21

Library SECF.PostscriptTest

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib Require Export String.
From SECF Require Import Postscript.

Parameter MISSING: Type.

Module CHECK.

Ltac check_type A B :=
    match type of A with
    | context[MISSING] ⇒ idtac "Missing:" A
    | ?T ⇒ first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
")"]
    end.

Ltac print_manual_grade A :=
    match eval compute in A with
    | Some (_ ?S ?C) ⇒
        idtac "Score:" S;
        match eval compute in C with
          | ""%string ⇒ idtac "Comment: None"
          | _ ⇒ idtac "Comment:" C
        end
    | None ⇒
        idtac "Score: Ungraded";
        idtac "Comment: None"
    end.

End CHECK.

From SECF Require Import Postscript.
Import CHECK.

Goal True.

idtac " ".
```

```
idtac "Max points - standard: 0".
idtac "Max points - advanced: 0".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "".
idtac "".
idtac "********** Summary **********".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "********** Standard **********".
idtac "".
idtac "********** Advanced **********".
Abort.
```

Chapter 22

Library SECF.BibTest

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib Require Export String.
From SECF Require Import Bib.

Parameter MISSING: Type.

Module CHECK.

Ltac check_type A B :=
    match type of A with
    | context[MISSING] ⇒ idtac "Missing:" A
    | ?T ⇒ first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
")"]
    end.

Ltac print_manual_grade A :=
    match eval compute in A with
    | Some (_ ?S ?C) ⇒
        idtac "Score:" S;
        match eval compute in C with
          | ""%string ⇒ idtac "Comment: None"
          | _ ⇒ idtac "Comment:" C
        end
    | None ⇒
        idtac "Score: Ungraded";
        idtac "Comment: None"
    end.

End CHECK.

From SECF Require Import Bib.
Import CHECK.

Goal True.

idtac " ".
```

```
idtac "Max points - standard: 0".
idtac "Max points - advanced: 0".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "".
idtac "".
idtac "********** Summary **********".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "********** Standard **********".
idtac "".
idtac "********** Advanced **********".
Abort.
```