Contents

Chapter 1

Library QC.Preface

1.1 Preface

This volume of *Software Foundations* introduces QuickChick, a property-based random testing tool for Coq.

1.2 Setup

For working with this material, you will need to install QuickChick. Instructions can be found here:

{https://github.com/QuickChick/QuickChick}

The QuickChickInterface chapter serves as a reference manual for QuickChick. The rest of the book explains its features more gently.

1.3 Practicalities

1.3.1 Recommended Citation Format

If you want to refer to this volume in your own writing, please do so as follows:

@book {Lampropoulos:SF4, author = {Leonidas Lampropoulos and Benjamin C. Pierce}, editor = {Benjamin C. Pierce}, title = "QuickChick: Property-Based Testing in Coq", series = "Software Foundations", volume = "4", year = "2021", publisher = "Electronic textbook", note = {Version 1.2.1, \URL{http://softwarefoundations.cis.upenn.edu} }, }

1.4 Thanks

Development of the *Software Foundations* series has been supported, in part, by the National Science Foundation under the NSF Expeditions grant 1521523, *The Science of Deep Speci-*

fication. Work on this volume was also supported by NSF grant 1421243, *Random Testing for Language Design*.

Chapter 2

Library QC.Introduction

2.1 Introduction

```
Set Warnings "-extraction-opaque-accessed,-extraction".
Set Warnings "-notation-overridden,-parsing".
From QuickChick Require Import QuickChick.
Require Import List ZArith. Import ListNotations.
```

2.2 A First Taste of Testing

Consider the following definition of a function remove, which takes a natural number x and a list of nats l and removes x from the list.

```
Fixpoint remove (x : nat) (l : list nat) : list nat :=
  match l with
    | [] ⇒ []
    | h::t ⇒ if h =? x then t else h :: remove x t
  end.
```

One possible specification for remove might be this property...

```
Conjecture removeP : ∀ x l, ¬ (In x (remove x l)).
```

...which says that x never occurs in the result of remove x l for any x and l. (*Conjecture* foo... means the same as Theorem foo... *Admitted*. Formally, foo is treated as an axiom.)

Sadly, this property is false, as we would (eventually) discover if we were to try to prove it.

A different – perhaps much more efficient – way to discover the discrepancy between the definition and specification is to *test* it:

(Try uncommenting and evaluating the previous line.)

The QuickChick command takes an "executable" property (we'll see later exactly what this means) and attempts to falsify it by running it on many randomly generated inputs, resulting in output like this:

0 0, 0 Failed! After 17 tests and 12 shrinks

This means that, if we run remove with x being 0 and l being the two-element list containing two zeros, then the property *removeP* fails.

With this example in hand, we can see that the `then` branch of remove fails to make a recursive call, which means that only one occurence of x will be deleted. The last line of the output records that it took 17 tests to identify some fault-inducing input and 12 "shrinks" to reduce it to a minimal counterexample.

Exercise: 1 star, standard (insertP) Here is a somewhat mangled definition of a function for inserting a new element into a sorted list of numbers:

```
Fixpoint insert x l :=
  match l with
  | [] ⇒ [x]
  | y::t ⇒ if y <? x then insert x t else y::t
  end.
```

Write a property that says "inserting a number x into a list l always yields a list containing x." Make sure QuickChick finds a counterexample.

Exercise: 2 stars, standard (insertP2) Translate the following claim into a *Conjecture* (using In for list membership): "For all numbers x and y and lists l, if y is in l then it is also in the list that results from inserting x into l" (i.e., insert preserves all the elements already in l). Make sure QuickChick finds a counterexample.

2.3 Overview

Property-based random testing involves four basic ingredients:

- an *executable property* like *removeP*,
- *generators* for random elements of the types of the inputs to the property (here, numbers and lists of numbers),
- *printers* for converting data structures like numbers and lists to strings when reporting counterexamples, and
- *shrinkers*, which are used to minimize counterexamples.

We will delve into each of these in detail later on, but first we need to make a digression to explain Coq's support for *typeclasses*, which QuickChick uses extensively both internally and in its programmatic interface to users. This is the `Typeclasses` chapter.

In the QC chapter we'll cover the core concepts and features of QuickChick itself.

The TImp chapter develops a small case study around a typed variant of the Imp language.

The QuickChickTool chapter presents a command line tool, *quickChick*, that supports larger-scale projects and mutation testing.

The QuickChickInterface chapter is a complete reference manual for QuickChick.

Finally, the *Postscript* chapter gives some suggestions for further reading.

Chapter 3

Library QC.Typeclasses

3.1 Typeclasses: A Tutorial on Typeclasses in Coq

```
From Coq Require Import Bool.Bool.
From Coq Require Import Strings.String.
From Coq Require Import Arith.Arith.
From Coq Require Import Lia.
Require Import List. Import ListNotations.
Local Open Scope string.
```

In real-world programming, it is often necessary to convert various kinds of data structures into strings so that they can be printed out, written to files, marshalled for sending over the network, or whatever. This can be accomplished by writing string converters for each basic type

- showBool : **bool** → **string**
- showNat : **nat** → **string**
- etc.

plus combinators for structured types like **list** and pairs

- showList : {A : Type} (A → **string**) → (**list** A) → **string**
- showPair : {A B : Type} (A → **string**) → (B → **string**) → $A \times B$ → **string**

that take string converters for their element types as arguments. Once we've done this, we can build string converters for more complex structured types by assembling them from these pieces:

- *showListOfPairsOfNats* = showList (showPair showNat showNat)

While this idiom gets the job done, it feels a bit clunky in at least two ways. First, it demands that we give names to all these string converters (which must later be remembered!) whereas it seems the names could really just be generated in a uniform way from the types involved. Moreover, the definitions of converters like *showListOfPairsOfNats* are always derived in a quite mechanical way from their types, together with a small collection of primitive converters and converter combinators.

The designers of Haskell addressed this clunkiness through *typeclasses*, a mechanism by which the typechecker is instructed to automatically construct "type-driven" functions *Wadler and Blott* 1989 (in Bib.v). (Readers not already familiar with typeclasses should note that, although the word sounds a bit like "classes" from object-oriented programming, this apparent connection is rather misleading. A better analogy is actually with *interfaces* from languages like Java. But best of all is to set aside object-oriented preconceptions and try to approach typeclasses with an empty mind!)

Many other modern language designs have followed Haskell's lead, and Coq is no exception. However, because Coq's type system is so much richer than that of Haskell, and because typeclasses in Coq are used to automatically construct not only programs but also proofs, Coq's presentation of typeclasses is quite a bit less "transparent": to use typeclasses effectively, one must have a fairly detailed understanding of how they are implemented. Coq typeclasses are a power tool: they can make complex developments much more elegant and easy to manage when used properly, and they can cause a great deal of trouble when things go wrong!

This tutorial introduces the core features of Coq's typeclasses, explains how they are implemented, surveys some useful typeclasses that can be found in Coq's standard library and other contributed libraries, and collects some advice about the pragmatics of using typeclasses.

3.2 Basics

3.2.1 Classes and Instances

To automate converting various kinds of data into strings, we begin by defining a typeclass called `Show`.

```
Class Show A : Type :=
  {
    show : A → string
  }.
```

The `Show` typeclass can be thought of as "classifying" types whose values can be converted to strings – that is, types A such that we can define a function show of type $A \to$ string.

We can declare that bool is such a type by giving an `Instance` declaration that witnesses this function:

```
Instance showBool : Show bool :=
  {
    show := fun b:bool ⇒ if b then "true" else "false"
  }.
Compute (show true).
```

Other types can similarly be equipped with `Show` instances – including, of course, new types that we define.

```
Inductive primary := Red | Green | Blue.
Instance showPrimary : Show primary :=
  {
    show :=
      fun c:primary ⇒
        match c with
        | Red ⇒ "Red"
        | Green ⇒ "Green"
        | Blue ⇒ "Blue"
        end
  }.
Compute (show Green).
```

The show function is sometimes said to be *overloaded*, since it can be applied to arguments of many types, with potentially radically different behavior depending on the type of its argument.

Converting natural numbers to strings is conceptually similar, though it requires a tiny bit of programming:

```
Fixpoint string_of_nat_aux (time n : nat) (acc : string) : string :=
  let d := match n mod 10 with
           | 0 ⇒ "0" | 1 ⇒ "1" | 2 ⇒ "2" | 3 ⇒ "3" | 4 ⇒ "4" | 5 ⇒ "5"
           | 6 ⇒ "6" | 7 ⇒ "7" | 8 ⇒ "8" | _ ⇒ "9"
           end in
  let acc' := d ++ acc in
  match time with
    | 0 ⇒ acc'
    | S time' ⇒
      match n / 10 with
        | 0 ⇒ acc'
        | n' ⇒ string_of_nat_aux time' n' acc'
      end
  end.
Definition string_of_nat (n : nat) : string :=
  string_of_nat_aux n n "".
Instance showNat : Show nat :=
  {
    show := string_of_nat
  }.
Compute (show 42).
```

Exercise: 1 star, standard (showNatBool) Write a Show instance for pairs of a nat and a bool.

Next, we can define functions that use the overloaded function show like this:

```
Definition showOne {A : Type} `{Show A} (a : A) : string :=
  "The value is " ++ show a.

Compute (showOne true).
Compute (showOne 42).
```

The parameter `{Show *A*} is a *class constraint*, which states that the function showOne is expected to be applied only to types *A* that belong to the Show class.

Concretely, this constraint should be thought of as an extra parameter to showOne supplying *evidence* that *A* is an instance of Show – i.e., it is essentially just a show function for *A*, which is implicitly invoked by the expression show *a*.

More interestingly, a single function can come with multiple class constraints:

```
Definition showTwo {A B : Type}
             `{Show A} `{Show B} (a : A) (b : B) : string :=
  "First is " ++ show a ++ " and second is " ++ show b.

Compute (showTwo true 42).
Compute (showTwo Red Green).
```

In the body of showTwo, the type of the argument to each instance of show determines which of the implicitly supplied show functions (for *A* or *B*) gets invoked.

Exercise: 1 star, standard (missingConstraint) What happens if we forget the class constraints in the definitions of showOne or showTwo? Try deleting them and see what happens.

□

Of course, Show is not the only interesting typeclass. There are many other situations where it is useful to be able to choose (and construct) specific functions depending on the type of an argument that is supplied to a generic function like show. Another typical example is equality checkers.

Here is another basic example of typeclasses: a class **Eq** describing types with a (boolean) test for equality.

```
Class Eq A :=
  {
    eqb: A → A → bool;
  }.
Notation "x =? y" := (eqb x y) (at level 70).
```

And here are some basic instances:

```
Instance eqBool : Eq bool :=
  {
    eqb := fun (b c : bool) ⇒
       match b, c with
         | true, true ⇒ true
         | true, false ⇒ false
         | false, true ⇒ false
         | false, false ⇒ true
       end
  }.
Instance eqNat : Eq nat :=
  {
    eqb := Nat.eqb
  }.
```

One possible confusion should be addressed here: Why should we need to define a typeclass for boolean equality when Coq's *propositional* equality ($x = y$) is completely generic? The answer is that, while it makes sense to *claim* that two values x and y are equal no matter what their type is, it is not possible to write a decidable equality *checker* for arbitrary types. In particular, equality at types like **nat**→**nat** is undecidable.

Exercise: 3 stars, standard, optional (boolArrowBool) There are some function types, like **bool**→**bool**, for which checking equality makes perfect sense. Write an **Eq** instance for this type.

3.2.2 Parameterized Instances: New Typeclasses from Old

What about a `Show` instance for pairs?

Since we can have pairs of any types, we'll want to parameterize our `Show` instance by two types. Moreover, we'll need to constrain both of these types to be instances of `Show`.

```
Instance showPair {A B : Type} `{Show A} `{Show B} : Show (A × B) :=
  {
    show p :=
      let (a,b) := p in
        "(" ++ show a ++ "," ++ show b ++ ")"
  }.
Compute (show (true,42)).
```

Similarly, here is an **Eq** instance for pairs...

```
Instance eqPair {A B : Type} `{Eq A} `{Eq B} : Eq (A × B) :=
  {
    eqb p1 p2 :=
      let (p1a,p1b) := p1 in
      let (p2a,p2b) := p2 in
      andb (p1a =? p2a) (p1b =? p2b)
  }.
```

...and here is `Show` for lists:

```
Fixpoint showListAux {A : Type} (s : A → string) (l : list A) : string :=
  match l with
    | nil ⇒ ""
    | cons h nil ⇒ s h
    | cons h t ⇒ append (append (s h) ", ") (showListAux s t)
  end.
Instance showList {A : Type} `{Show A} : Show (list A) :=
  {
    show l := append "[" (append (showListAux show l) "]")
  }.
```

Exercise: 3 stars, standard (eqEx) Write an **Eq** instance for lists and `Show` and **Eq** instances for the **option** type constructor.

Exercise: 3 stars, standard, optional (boolArrowA) Generalize your solution to the *boolArrowBool* exercise to build an equality instance for any type of the form **bool**→*A*, where *A* itself is an **Eq** type. Show that it works for **bool**→**bool**→**nat**.

3.2.3 Class Hierarchies

We often want to organize typeclasses into hierarchies. For example, we might want a typeclass **Ord** for "ordered types" that support both equality and a less-or-equal comparison operator.

A possible (but bad) way to do this is to define a new class with two associated functions:

```
Class OrdBad A :=
  {
    eqbad : A → A → bool;
    lebad : A → A → bool
  }.
```

The reason this is bad is because we now need to use a new equality operator (eqbad) if we want to test for equality on ordered values.

```
Definition lt {A: Type} `{Eq A} `{OrdBad A} (x y : A) : bool :=
  andb (lebad x y) (negb (eqbad x y)).
```

A much better way is to parameterize the definition of **Ord** on an **Eq** class constraint:

```
Class Ord A `{Eq A} : Type :=
  {
    le : A → A → bool
  }.
Check Ord.
```

(The old class **Eq** is sometimes called a "superclass" of **Ord**, but, again, this terminology is potentially confusing: Try to avoid thinking about analogies with object-oriented programming!)

When we define instances of **Ord**, we just have to implement the le operation.

```
Instance natOrd : Ord nat :=
  {
    le := Nat.leb
  }.
```

Functions expecting to be instantiated with an instance of **Ord** now have two class constraints, one witnessing that they have an associated eqb operation, and one for le.

```
Definition max {A: Type} `{Eq A} `{Ord A} (x y : A) : A :=
  if le x y then y else x.
```

Exercise: 1 star, standard (missingConstraintAgain) What does Coq say if the **Ord** class constraint is left out of the definition of max? What about the **Eq** class constraint?
□

Exercise: 3 stars, standard (ordMisc) Define **Ord** instances for options and pairs.

Exercise: 3 stars, standard (ordList) For a little more practice, define an **Ord** instance for lists.

3.3 How It Works

Typeclasses in Coq are a powerful tool, but the expressiveness of the Coq logic makes it hard to implement sanity checks like Haskell's "overlapping instances" detector.

As a result, using Coq's typeclasses effectively – and figuring out what is wrong when things don't work – requires a clear understanding of the underlying mechanisms.

3.3.1 Implicit Generalization

The first thing to understand is exactly what the "backtick" notation means in declarations of classes, instances, and functions using typeclasses. This is actually a quite generic mechanism, called *implicit generalization*, that was added to Coq to support typeclasses but that can also be used to good effect elsewhere.

The basic idea is that unbound variables mentioned in bindings marked with ` are automatically bound in front of the binding where they occur.

To enable this behavior for a particular variable, say A, we first declare A to be implicitly generalizable:

```
Generalizable Variables A.
```

By default, Coq only implicitly generalizes variables declared in this way, to avoid puzzling behavior in case of typos. There is also a *Generalize* `Variables All` command, but it's probably not a good idea to use it!

Now, for example, we can shorten the declaration of the showOne function by omitting the binding for A at the front.

```
Definition showOne1 `{Show A} (a : A) : string :=
  "The value is " ++ show a.
```

Coq will notice that the occurrence of A inside the `{...} is unbound and automatically insert the binding that we wrote explicitly before.

```
Print showOne1.
```

The "implicit and maximally generalized" annotation on the last line means that the automatically inserted bindings are treated as if they had been written with {...}, rather than (...). The "implicit" part means that the type argument A and the `Show` witness H are usually expected to be left implicit: whenever we write showOne1, Coq will automatically insert two unification variables as the first two arguments. This automatic insertion can be disabled by writing @, so a bare occurrence of showOne1 means the same as @showOne1 _ _. The "maximally inserted" part says that these arguments should inserted automatically even when there is no following explicit argument.

In fact, even the `{`Show` $A$} form hides one bit of implicit generalization: the bound name of the `Show` constraint itself. You will sometimes see class constraints written more explicitly, like this...

```
Definition showOne2 `{_ : Show A} (a : A) : string :=
  "The value is " ++ show a.
```

... or even like this:

```
Definition showOne3 `{H : Show A} (a : A) : string :=
  "The value is " ++ show a.
```

The advantage of the latter form is that it gives a name that can be used, in the body, to explicitly refer to the supplied evidence for `Show` A. This can be useful when things get complicated and you want to make your code more explicit so you can better understand and control what's happening.

We can actually go one bit further and omit A altogether, with no change in meaning (though, again, this may be more confusing than helpful):

```
Definition showOne4 `{Show} a : string :=
  "The value is " ++ show a.
```

If we ask Coq to print the arguments that are normally implicit, we see that all these definitions are exactly the same internally.

```
Set Printing Implicit.
Print showOne.
Print showOne1.
Print showOne2.
Print showOne3.
Print showOne4.
Unset Printing Implicit.
```

The examples we've seen so far illustrate how implicit generalization works, but you may not be convinced yet that it is actually saving enough keystrokes to be worth the trouble of adding such a fancy mechanism to Coq. Where things become more convincing is when classes are organized into hierarchies. For example, here is an alternate definition of the max function:

```
Definition max1 `{Ord A} (x y : A) :=
  if le x y then y else x.
```

If we print out max1 in full detail, we can see that the implicit generalization around \`{**Ord** A} led Coq to fill in not only a binding for A but also a binding for H, which it can see must be of type **Eq** A because it appears as the second argument to **Ord**. (And why is **Ord** applied here to two arguments instead of just the one, A, that we wrote? Because **Ord**'s arguments are maximally inserted!)

```
Set Printing Implicit.
Print max1.

Check Ord.

Unset Printing Implicit.
```

For completeness, a couple of final points about implicit generalization. First, it can be used in situations where no typeclasses at all are involved. For example, we can use it to write quantified propositions mentioning free variables, following the common informal convention that these are to be quantified implicitly.

```
Generalizable Variables x y.

Lemma commutativity_property : `{x + y = y + x}.
Proof. intros. lia. Qed.

Check commutativity_property.
```

The previous examples have all shown implicit generalization being used to fill in forall binders. It will also create `fun` binders, when this makes sense:

```
Definition implicit_fun := `{x + y}.
```

Defining a function in this way is not very natural, however. In particular, the arguments are all implicit and maximally inserted (as can be seen if we print out its definition)...

```
Print implicit_fun.
```

... so we will need to use @ to actually apply the function:

```
Compute (@implicit_fun 2 3).
```

Writing ‘(...), with parentheses instead of curly braces, causes Coq to perform the same implicit generalization step, but does *not* mark the inserted binders themselves as implicit.

```
Definition implicit_fun1 := `(x + y).
Print implicit_fun1.
Compute (implicit_fun1 2 3).
```

3.3.2 Records are Products

Although records are not part of its core, Coq does provide some simple syntactic sugar for defining and using records.

Record types must be declared before they are used. For example:

```
Record Point :=
  Build_Point
    {
      px : nat;
      py : nat
    }.
```

Internally, this declaration is desugared into a single-field inductive type, roughly like this:

Inductive Point : Set := | Build_Point : nat -> nat -> Point.

Elements of this type can be built, if we like, by applying the Build_Point constructor directly.

```
Check (Build_Point 2 4).
```

Or we can use more familar record syntax, which allows us to name the fields and write them in any order:

```
Check {| px := 2; py := 4 |}.
Check {| py := 4; px := 2 |}.
```

We can also access fields of a record using conventional “dot notation” (with slightly clunky concrete syntax):

```
Definition r : Point := {| px := 2; py := 4 |}.
```

```
Compute (r.(px) + r.(py)).
```

Record declarations can also be parameterized:

```
Record LabeledPoint (A : Type) :=
  Build_LabeledPoint
    {
      lx : nat;
      ly : nat;
      label : A
    }.
```

(Note that the field names have to be different. Any given field name can belong to only one record type. This greatly simplifies type inference!)

```
Check {| lx:=2; ly:=4; label:="hello" |}.
```

Exercise: 1 star, standard (rcdParens) Note that the A parameter in the definition of **LabeledPoint** is bound with parens, not curly braces. Why is this a better choice?

□

3.3.3 Typeclasses are Records

Typeclasses and instances, in turn, are basically just syntactic sugar for record types and values (together with a bit of magic for using proof search to fill in appropriate instances during typechecking, as described below).

Internally, a typeclass declaration is elaborated into a parameterized `Record` declaration:

```
Set Printing All.
Print Show.
Unset Printing All.
```

(If you run the `Print` command yourself, you'll see that `Show` actually displays as a `Variant`; this is Coq's terminology for a single-field record.)

Analogously, `Instance` declarations become record values:

```
Print showNat.
```

Note that the syntax for record values is slightly different from `Instance` declarations. Record values are written with curly-brace-vertical-bar delimiters, while `Instance` declarations are written here with just curly braces. (To be precise, both forms of braces are actually allowed for `Instance` declarations, and either will work in most situations; however, type inference sometimes works a bit better with bare curly braces.)

Similarly, overloaded functions like show are really just record projections, which in turn are just functions that select a particular argument of a one-constructor `Inductive` type.

```
Set Printing All.
Print show.
Unset Printing All.
```

3.3.4 Inferring Instances

So far, all the mechanisms we've seen have been pretty simple syntactic sugar and binding munging. The real "special sauce" of typeclasses is the way appropriate instances are automatically inferred (and/or constructed!) during typechecking.

For example, if we write show 42, what we actually get is @show **nat** showNat 42:

```
Definition eg42 := show 42.
```

```
Set Printing Implicit.
Print eg42.
Unset Printing Implicit.
```

How does this happen?

First, since the arguments to show are marked implicit, what we typed is automatically expanded to @show _ _ 42. The first _ should obviously be replaced by **nat**. But what about the second?

By ordinary type inference, Coq knows that, to make the whole expression well typed, the second argument to @show must be a value of type `Show` **nat**. It attempts to find or construct such a value using a variant of the `eauto` proof search procedure that refers to a "hint database" called *typeclass_instances*.

Exercise: 1 star, standard (HintDb) Uncomment and execute the following command. Search for "For Show" in the output and have a look at the entries for showNat and showPair.

We can see what's happening during the instance inference process if we issue the `Set Typeclasses Debug` command.

```
Set Typeclasses Debug.
Check (show 42).
```

In this simple example, the proof search succeeded immediately because showNat was in the hint database. In more interesting cases, the proof search needs to try to assemble an *expression* of appropriate type using both functions and constants from the hint database. (This is very like what happens when proof search is used as a tactic to automatically assemble compound proofs by combining theorems from the environment.)

```
Check (show (true,42)).
```

```
Unset Typeclasses Debug.
```

In the second line, the search procedure decides to try applying showPair, from which it follows (after a bit of unification) that it needs to find an instance of `Show` *Nat* and an instance of `Show` Bool, each of which succeeds immediately as before.

In summary, here are the steps again:

show 42 ===> { Implicit arguments } @show _42 ===> { Typing } @show (?A : Type) (?Show0 : Show ?A) 42 ===> { Unification } @show nat (?Show0 : Show nat) 42 ===> { Proof search for Show Nat returns showNat } @show nat showNat 42

3.4 Typeclasses and Proofs

Since programs and proofs in Coq are fundamentally made from the same stuff, the mechanisms of typeclasses extend smoothly to situations where classes contain not only data and functions but also proofs.

This is a big topic – too big for a basic tutorial – but let's take a quick look at a few things.

3.4.1 Propositional Typeclass Members

The **Eq** typeclass defines a single overloaded function that tests for equality between two elements of some type. We can extend this to a subclass that also comes with a proof that the given equality tester is correct, in the sense that, whenever it returns true, the two values are actually equal in the propositional sense (and vice versa).

```
Class EqDec (A : Type) {H : Eq A} :=
  {
    eqb_eq : ∀ x y, x =? y = true ↔ x = y
  }.
```

To build an instance of **EqDec**, we must now supply an appropriate proof.

```
Instance eqdecNat : EqDec nat :=
  {
    eqb_eq := Nat.eqb_eq
  }.
```

If we do not happen to have an appropriate proof already in the environment, we can simply omit it. Coq will enter proof mode and ask the user to use tactics to construct inhabitants for the fields.

```
Instance eqdecBool' : EqDec bool.
Proof.
  constructor.
  intros x y. destruct x; destruct y; simpl; unfold iff; auto.
Defined.
```

Given a typeclass with propositional members, we can use these members in proving things involving this typeclass.

Here, for example, is a quick (and somewhat contrived) example of a proof of a property that holds for arbitrary values from the **EqDec** class...

```
Lemma eqb_fact `{EqDec A} : ∀ (x y z : A),
  x =? y = true → y =? z = true → x = z.
Proof.
  intros x y z Exy Eyz.
  rewrite eqb_eq in Exy.
  rewrite eqb_eq in Eyz.
  subst. reflexivity. Qed.
```

There is much more to say about how typeclasses can be used (and how they should not be used) to support large-scale proofs in Coq. See the suggested readings below.

3.4.2 Substructures

Naturally, it is also possible to have typeclass instances as members of other typeclasses: these are called *substructures*. Here is an example adapted from the Coq Reference Manual.

```
From Coq Require Import Relations.Relation_Definitions.

Class Reflexive (A : Type) (R : relation A) :=
  {
    reflexivity : ∀ x, R x x
  }.

Class Transitive (A : Type) (R : relation A) :=
  {
    transitivity : ∀ x y z, R x y → R y z → R x z
  }.

Generalizable Variables z w R.

Lemma trans3 : ∀ `{Transitive A R},
    `{R x y → R y z → R z w → R x w}.
Proof.
  intros.
  apply (transitivity x z w). apply (transitivity x y z).
  assumption. assumption. assumption. Defined.

Class PreOrder (A : Type) (R : relation A) :=
  { PreOrder_Reflexive :> Reflexive A R ;
    PreOrder_Transitive :> Transitive A R }.
```

The syntax :> indicates that each **PreOrder** can be seen as a **Reflexive** and **Transitive** relation, so that, any time a reflexive relation is needed, a preorder can be used instead.

```
Lemma trans3_pre : ∀ `{PreOrder A R},
    `{R x y → R y z → R z w → R x w}.
Proof. intros. eapply trans3; eassumption. Defined.
```

3.5 Some Useful Typeclasses

3.5.1 Dec

The ssreflect library defines what it means for a proposition P to be decidable like this...

```
Require Import ssreflect ssrbool.
```

```
Print decidable.
```

... where $\{P\} + \{\tilde{}\ P\}$ is an "informative disjunction" of P and $\neg P$.

It is easy to wrap this in a typeclass of "decidable propositions":

```
Class Dec (P : Prop) : Type :=
  {
    dec : decidable P
  }.
```

We can now create instances encoding the information that propositions of various forms are decidable. For example, the proposition $x = y$ is decidable (for any specific x and y), assuming that x and y belong to a type with an **EqDec** instance.

```
Instance EqDec__Dec {A} `{H : EqDec A} (x y : A) : Dec (x = y).
Proof.
  constructor.
  unfold decidable.
  destruct (x =? y) eqn:E.
  - left. rewrite <- eqb_eq. assumption.
  - right. intros C. rewrite <- eqb_eq in C. rewrite E in C. inversion C.
Defined.
```

Similarly, we can lift decidability through logical operators like conjunction:

```
Instance Dec_conj {P Q} {H : Dec P} {I : Dec Q} : Dec (P /\ Q).
Proof.
  constructor. unfold decidable.
  destruct H as [D]; destruct D;
    destruct I as [D]; destruct D; auto;
      right; intro; destruct H; contradiction.
Defined.
```

Exercise: 3 stars, standard (dec_neg_disj) Give instance declarations showing that, if P and Q are decidable propositions, then so are $\neg P$ and $P \vee Q$.

Exercise: 4 stars, standard (Dec_All) The following function converts a list into a proposition claiming that every element of that list satiesfies some proposition P:

```
Fixpoint All {T : Type} (P : T -> Prop) (l : list T) : Prop :=
  match l with
  | [] => True
  | x :: l' => P x /\ All P l'
  end.
```

Create an instance of **Dec** for `All` P l, given that P a is decidable for every a.

One reason for doing all this is that it makes it easy to move back and forth between the boolean and propositional worlds, whenever we know we are dealing with decidable propositions.

In particular, we can define a notation P? that converts a decidable proposition P into a boolean expression.

```
Notation "P '?'" :=
  (match (@dec P _) with
   | left _ ⇒ true
   | right _ ⇒ false
   end)
  (at level 100).
```

Now we don't need to remember that, for example, the test for equality on numbers is called eqb, because instead of this...

```
Definition silly_fun1 (x y z : nat) :=
  if andb (x =? y) (andb (y =? z) (x =? z))
  then "all equal"
  else "not all equal".
```

... we can just write this:

```
Definition silly_fun2 (x y z : nat) :=
  if (x = y ∧ y = z ∧ x = z)?
  then "all equal"
  else "not all equal".
```

3.5.2 Monad

In Haskell, one place typeclasses are used very heavily is with the **Monad** typeclass, especially in conjunction with Haskell's "do notation" for monadic actions.

Monads are an extremely powerful tool for organizing and streamlining code in a wide range of situations where computations can be thought of as yielding a result along with some kind of "effect." Examples of possible effects include

- input / output
- state mutation
- failure
- nondeterminism
- randomness
- etc.

There are many good tutorials on the web about monadic programming in Haskell. Readers who have never seen monads before may want to refer to one of these to fully understand the examples here.

In Coq, monads are also heavily used, but the fact that typeclasses are a relatively recent addition to Coq, together with the fact that Coq's `Notation` extension mechanism allows users to define their own variants of `do` notation for specific situations where a monadic style is useful, have led to a proliferation of definitions of monads: most older projects simply define their own monads and monadic notations – sometimes typeclass-based, often not – while newer projects use one of several generic libraries for monads. Our current favorite (as of Summer 2017) is the monad typeclasses in Gregory Malecha's *ext-lib* package:

{https://github.com/coq-community/coq-ext-lib/blob/master/theories/Structures/Monad.v}

Once the *ext-lib* package is installed (e.g., via OPAM), we can write:

```
Require Export ExtLib.Structures.Monads.
Export MonadNotation.
Open Scope monad_scope.
```

The main definition provided by this library is the following typeclass:

Class Monad (M : Type -> Type) : Type := { ret : forall {T : Type}, T -> M T ; bind : forall {T U : Type}, M T -> (T -> M U) -> M U }. That is, a type family *M* is an instance of the **Monad** class if we can define functions ret and bind of the appropriate types.

(If you `Print` the actual definition, you'll see something more complicated, involving *Polymorphic* `Record` *bla bla*... The *Polymorphic* part refers to Coq's "universe polymorphism," which does not concern us here.)

For example, we can define a monad instance for **option** like this:

```
Instance optionMonad : Monad option :=
  {
    ret T x :=
      Some x ;
    bind T U m f :=
      match m with
        None ⇒ None
      | Some x ⇒ f x
      end
  }.
```

The other nice thing we get from the **Monad** library is lightweight notation for bind: Instead of

bind m1 (fun x => m2),

we can write

x <- m1 ;; m2.

Or, if the result from *m1* is not needed in *m2*, then instead of

bind m1 (fun _=> m2),

we can write

m1 ;; m2.

This allows us to write functions involving "option plumbing" very compactly.

For example, suppose we have a function that looks up the nth element of a list, returning None if the list contains less than n elements.

```
Fixpoint nth_opt {A : Type} (n : nat) (l : list A) : option A :=
  match l with
     [] ⇒ None
  | h::t ⇒ if n =? 0 then Some h else nth_opt (n-1) t
  end.
```

We can write a function that sums the first three elements of a list (returning None if the list is too short) like this:

```
Definition sum3 (l : list nat) : option nat :=
  x0 ← nth_opt 0 l ;;
  x1 ← nth_opt 1 l ;;
  x2 ← nth_opt 2 l ;;
  ret (x0 + x1 + x2).
```

One final convenience for programming with monads is a collection of operators for "lifting" functions on ordinary values to functions on monadic computations. *ExtLib* defines three – one for unary functions, one for binary, and one for ternary. The definitions (slightly simplified) look like this:

```
Definition liftM
             {m : Type → Type}
             {M : Monad m}
             {T U : Type} (f : T → U)
           : m T → m U :=
  fun x ⇒ bind x (fun x ⇒ ret (f x)).

Definition liftM2
             {m : Type → Type}
             {M : Monad m}
             {T U V : Type} (f : T → U → V)
           : m T → m U → m V :=
    fun x y ⇒ bind x (fun x ⇒ liftM (f x) y).

Definition liftM3
             {m : Type → Type}
             {M : Monad m}
             {T U V W : Type} (f : T → U → V → W)
           : m T → m U → m V → m W :=
    fun x y z ⇒ bind x (fun x ⇒ liftM2 (f x) y z).
```

For example, suppose we have two **option nat**s and we would like to calculate their sum (unless one of them is None, in which case we want None). Instead of this...

```
Definition sum3opt (n1 n2 : option nat) :=
  x1 ← n1 ;;
  x2 ← n2 ;;
  ret (x1 + x2).
```

...we can use liftM2 to write it more compactly:

```
Definition sum3opt' (n1 n2 : option nat) :=
  liftM2 plus n1 n2.
```

The */examples* directory in the *ext-lib* Github repository ({https://github.com/coq-community/coq-ext-lib/blob/master/}) includes some further examples of using monads in Coq.

3.5.3 Others

Two popular typeclasses from the Coq standard library are *Equivalence* (and the associated classes **Reflexive**, **Transitive**, etc.) and *Proper*. They are described in the second half of *A Gentle Introduction to Type Classes and Relations in Coq*, by Castéran and Sozeau. {https://www.labri.fr/perso/casteran/CoqArt/TypeClassesTut/typeclassestut.pdf}.

A much larger collection of typeclasses for formalizing mathematics is described in *Type Classes for Mathematics in Type Theory*, by Bas Spitters and Eelis van der Weegen. {https://arxiv.org/pdf/

3.6 Controlling Instantiation

3.6.1 "Defaulting"

The type of the overloaded eqb operator...

```
Check @eqb.
```

... says that it works for any **Eq** type. Naturally, we can use it in a definition like this...

```
Definition foo x := if x =? x then "Of course" else "Impossible".
```

... and then we can apply foo to arguments of any **Eq** type.

Right?

```
Fail Check (foo true).
```

Huh?!

Here's what happened:

- When we defined foo, the type of x was not specified, so Coq filled in a unification variable (an "evar") $?A$.
- When typechecking the expression eqb x, the typeclass instance mechanism was asked to search for a type-correct instance of **Eq**, i.e., an expression of type **Eq** $?A$.

- This search immediately succeeded because the first thing it tried worked; this happened to be the constant *eqBoolBool* : **Eq** (**bool**→**bool**). In the process, ?A got instantiated to **bool**→**bool**.
- The type calculated for foo was therefore (**bool**→**bool**)->(**bool**→**bool**)->**bool**.

The lesson is that it matters a great deal *exactly* what problems are posed to the instance search engine.

Exercise: 1 star, standard (debugDefaulting) Do `Set Typeclasses Debug` and verify that this is what happened.
□

3.6.2 Manipulating the Hint Database

One of the ways in which Coq's typeclasses differ most from Haskell's is the lack, in Coq, of an automatic check for "overlapping instances."

That is, it is completely legal to define a given type to be an instance of a given class in two different ways.

```
Inductive baz := Baz : nat → baz.

Instance baz1 : Show baz :=
  {
    show b :=
      match b with
        Baz n ⇒ "Baz: " ++ show n
      end
  }.

Instance baz2 : Show baz :=
  {
    show b :=
      match b with
        Baz n ⇒ "[" ++ show n ++ " is a Baz]"
      end
  }.

Compute (show (Baz 42)).
```

When this happens, it is unpredictable which instance will be found first by the instance search process; here it just happened to be the second. The reason Coq doesn't do the overlapping instances check is because its type system is much more complex than Haskell's – so much so that it is very challenging in general to decide whether two given instances overlap.

The reason this is unfortunate is that, in more complex situations, it may not be obvious when there are overlapping instances.

One way to deal with overlapping instances is to "curate" the hint database by explicitly adding and removing specific instances.

To remove things, use *Remove Hints*:

```
Remove Hints baz1 baz2 : typeclass_instances.
```

To add them back (or to add arbitrary constants that have the right type to be intances – i.e., their type ends with an applied typeclass – but that were not created by `Instance` declarations), use *Existing* `Instance`:

```
Existing Instance baz1.
Compute (show (Baz 42)).
```

```
Remove Hints baz1 : typeclass_instances.
```

Another way of controlling which instances are chosen by proof search is to assign *priorities* to overlapping instances:

```
Instance baz3 : Show baz | 2 :=
  {
    show b :=
      match b with
        Baz n ⇒ "Use me first! " ++ show n
      end
  }.
Instance baz4 : Show baz | 3 :=
  {
    show b :=
      match b with
        Baz n ⇒ "Use me second! " ++ show n
      end
  }.
Compute (show (Baz 42)).
```

0 is the highest priority.

If the priority is not specified, it defaults to the number of binders of the instance. (This means that more specific – less polymorphic – instances will be chosen over less specific ones.)

Existing `Instance` declarations can also be given explicit priorities.

```
Existing Instance baz1 | 0.
Compute (show (Baz 42)).
```

3.7 Debugging

3.7.1 Instantiation Failures

One downside of using typeclasses, especially many typeclasses at the same time, is that error messages can become puzzling.

Here are some relatively easy ones.

```
Inductive bar :=
  Bar : nat → bar.
```

```
Fail Definition eqBar :=
  (Bar 42) =? (Bar 43).
```

```
Fail Definition ordBarList :=
  le [Bar 42] [Bar 43].
```

In these cases, it's pretty clear what the problem is. To fix it, we just have to define a new instance. But in more complex situations it can be trickier.

A few simple tricks can be very helpful:

- Do `Set Printing Implicit` and then use `Check` and `Print` to investigate the types of the things in the expression where the error is being reported.
- Add some @ annotations and explicitly fill in some of the arguments that should be getting instantiated automatically, to check your understanding of what they should be getting instantiated with.
- Turn on tracing of instance search with `Set Typeclasses Debug`.

The `Set Typeclasses Debug` command has a variant that causes it to print even more information: `Set Typeclasses Debug` *Verbosity* 2. Writing just `Set Typeclasses Debug` is equivalent to `Set Typeclasses Debug` *Verbosity* 1.

Another potential source of confusion with error messages comes up if you forget a ‘. For example:

```
Fail Definition max {A: Type} {Ord A} (x y : A) : A :=
  if le x y then y else x.
```

The *UNDEFINED EVARS* here is because the binders that are automatically inserted by implicit generalization are missing.

3.7.2 Nontermination

An even more annoying way that typeclass instantiation can go wrong is by simply diverging. Here is a small example of how this can happen.

Declare a typeclass involving two types parameters A and B – say, a silly typeclass that can be inhabited by arbitrary functions from A to B:

```
Class MyMap (A B : Type) : Type :=
  {
    mymap : A → B
  }.
```

Declare instances for getting from bool to nat...

```
Instance MyMap1 : MyMap bool nat :=
  {
    mymap b := if b then 0 else 42
  }.
```

... and from nat to string:

```
Instance MyMap2 : MyMap nat string :=
  {
    mymap := fun n : nat ⇒
      if le n 20 then "Pretty small" else "Pretty big"
  }.
Definition e1 := mymap true.
Compute e1.

Definition e2 := mymap 42.
Compute e2.
```

Notice that these two instances don't automatically get us from bool to string:

```
Fail Definition e3 : string := mymap false.
```

We can try to fix this by defining a generic instance that combines an instance from A to B and an instance from B to C:

```
Instance MyMap_trans {A B C : Type} `{MyMap A B} `{MyMap B C} : MyMap A C :=
  { mymap a := mymap (mymap a) }.
```

This does get us from bool to string automatically:

```
Definition e3 : string := mymap false.
Compute e3.
```

However, although this example seemed to work, we are actually in a state of great peril: If we happen to ask for an instance that doesn't exist, the search procedure will diverge.

Exercise: 1 star, standard (nonterm) Why, exactly, did the search diverge? Enable typeclass debugging, uncomment the above `Definition`, and see what gets printed. (You may want to do this from the command line rather than from inside an IDE, to make it easier to kill.)

3.8 Alternative Structuring Mechanisms

Typeclasses are just one of several mechanisms that can be used in Coq for structuring large developments. Others include:

- canonical structures
- bare dependent records
- modules and functors

An introduction to canonical structures and comparisons between canonical structures and typeclasses can be found here:

- Assia Mahboubi and Enrico Tassi. Canonical Structures for the working Coq user. In Sandrine Blazy, Christine Paulin, and David Pichardie, editors, ITP 2013, 4th Conference on Interactive Theorem Proving, volume 7998 of LNCS, pages 19–34, Rennes, France, 2013. Springer. {https://hal.inria.fr/hal-00816703v1/document}
- Gonthier et al., "How to make ad hoc proof automation less ad hoc", JFP 23 (4): 357–401, 2013. (This explains some weaknesses of typeclasses and why canonical structures are used in in the *mathcomp* libraries.)

A useful discussion of typeclasses vs. dependent records is: {https://stackoverflow.com/questions/29872
typeclasses-vs-dependent-records}

3.9 Advice from Experts

In the process of preparing this chapter, we asked people on the *coq-club* mailing list for their best tips on preventing and debugging typeclass confusions, and on best practices for choosing between typeclasses and other large-scale structuring mechanisms such as modules and canonical structures. We received a number of generous replies, which we should eventually digest and incorporate into the material above. For the moment, they are recorded here essentially as posted (lightly edited for ease of reading).

3.9.1 Matthieu Sozeau

The fact that typeclass resolution resorts to unrestricted proof search is a blessing and a curse for sure. Errors tell you only that proof search failed and the initial problem while in general you need to look at the trace of resolution steps to figure out what's missing or, in the worst case, making search diverge. If you are writing obviously recursive instances, not mixing computation with the search (e.g. Unfolding happening in the indices necessary for instances to match), and not creating dependent subgoals then you're basically writing Haskell 98-style instances and should get the same "simplicity".

I think for more elaborate use cases (terms in indices, dependencies and computation), when encountering unexpected failure the debug output (Set Typeclasses Debug) is necessary. Testing the logic program, using Checks for example is a good way to explore the proof search results. One can also debug interactively by switching to the tactic mode and looking at the *typeclasses* `eauto` behavior. We're facing the same issues as logic programming and I don't know of a silver bullet to debug these programs.

For common issues a newcomer is likely to get, there are missing instances which can be prevented/tested using some coverage tests, divergence which can be understood by looking at the trace (usually it's because of a dangerous instance like transitivity or symmetry which has to be restricted or removed, and sometimes because of a conversion which makes an instance always applicable), and ambiguity when the user does not get the instance he expected (due to overlapping mainly, priorities can help here).

One advantage of modules (over typeclasses) is that everything is explicit but the abstraction cost a bit higher as you usually have to functorize whole modules instead of individual functions, and often write down signatures separate from the implementations. One rule of thumb is that if there are potentially multiple instances of the same interface for the same type/index then a module is preferable, but adding more indexes to distinguish the instances is also possible.

3.9.2 John Wiegley

One thing that always gets me is that overlapping instances are easy to write with no warning from Coq (unlike Haskell, which ensures that resolution always pick a single instance). This requires me to often use:

Typeclasses eauto := debug.

and switch to my *coq* buffer to see which lookup did not resolve to the instance I was expecting. This is usually fixed by one of two things:

- Change the "priority" of the overlapping instance (something we cannot do in Haskell).
- Change the Instance to a Definition – which I can still use it as an explicitly passed dictionary, but this removes it from resolution.

Another scenario that often bites me is when I define an instance for a type class, and then intend to write a function using the type class and forget to provide it as an argument. In Haskell this would be an error, but in Coq it just resolves to whatever the last globally defined instance was.

For example, say I write a function that uses a functor, but forget to mention the functor:

Definition foo (C D : Category) (x y : C) (f : x ˜> y) : fobj x ˜> fobj y := fmap f.

In Haskell this gives an error stating that no Functor is available. In Coq, it type checks using the highest priority $C -> D$ functor instance in scope. I typically discover that this has happened when I try to use `foo` and find the Functor to be too specific, or by turning on Printing All and looking at the definition of ‘foo‘. However, there are times when ‘foo‘ is deep down in an expression, and then it's not obvious *at all* why it's failing.

The other way to solve this is to manually ensure there are no Instance definitions that might overlap, such as a generic `Instance` for *C* –> *D*, but only instances from specific categories to specific categories (though again, I might define several functors of that same type). It would be nice if I could make this situation into an error.

Finally, there is the dreaded "diamond problem", when referring to type classes as record members rather than type indices:

Class Foo := { method : Type -> Type }.

Class Bar := { bar_is_foo :> Foo }.

Class Baz := { baz_is_foo :> Foo }.

Class Oops := { oops_is_bar :> Bar oops_is_baz :> Baz }.

Oops refers to two *Foos*, and I need explicit evidence to know when they are the same *Foo*. I work around this using indices:

Class Foo := { method : Type -> Type }.

Class Bar (F : Foo) := { }.

Class Baz (F : Foo) := { }.

Class Oops (F : Foo) := { oops_is_bar :> Bar F oops_is_baz :> Baz F }.

Only those classes which might be multiply-inherited need to be lifted like this. It forces me to use Sections to avoid repetition, but allows Coq to see that base classes sharing is plainly evident.

The main gotcha here for the newcomer is that it is not obvious at all when the diamond problem is what you're facing, since when it hits the parameters are hidden indices, and you end up with goals like:

A, B : Type F : Foo O : Oops H : @method (@bar_is_foo (@oops_is_bar O)) A = @method F B

@method F A = @method F B

You can't apply here without simplying in H. However, what you see at first is:

A, B : Type F : Foo O : Oops H : method A = method B

---

method A = method B

As a newcomer, knowing to turn on Printing All is just not obvious here, but it quickly becomes something you learn to do whenever what looks obvious is not.

Other than these things, I use type classes heavily in my own libraries, and very much enjoy the facility they provide. I have a category-theory library that is nearly all type classes, and I ran into every one of the problems described above, before learning how to "work with Coq" to make things happy.

3.9.3 Michael Soegtrop

What I had most problems with initially is that some type classes have implicit parameters. This works quite well when the nesting level of these parameters is small, but when the nesting of parameters gets deeper one can have the issue that unification takes long, that error messages are hard to understand and that later in a proof one might require certain relations between different parameters of a type class which are not explicit when the initial resolution was done and that an instance is chosen which is not compatible with these requirements, although there is one which would be compatible. The solution is typically to explicitly specify some of the implicit parameters, especially their relation to each other. Another advantage of stating certain things explicitly is that it is easier to understand what actually happens.

3.9.4 Abhishek Anand

Typeclasses are merely about inferring some implicit arguments using proof search. The underlying modularity mechanism, which is the ability to define “existential types” using induction, was *always* there in Coq: typeclasses merely cuts down on verbosity because more arguments can now be implicit because they can be automatically inferred. Relying on proof search often brings predictability concerns. So, guidance on taming proof search would be very useful: Chapter 13 of Chipala’s Certified Programming with Dependent Types (CPDT) might be a good background for using typeclasses. Also, it is good to keep in mind that if typeclass-resolution fails to instantiate an implicit argument, some/all of those arguments can be provided manually. Often, just providing one such implicit argument gives enough clues to the inference engine to infer all of them. I think it is important to emphasize that typeclass arguments are just implicit arguments.

Also, another design decision in typeclasses/records is whether to unbundle. The following paper makes a great case for unbundling: Spitters, Bas, and Eelis Van Der Weegen. “Type Classes for Mathematics in Type Theory.” MSCS 21, no. Special Issue 04 (2011): 795–825. doi:10.1017/S0960129511000119. {https://arxiv.org/pdf/1102.1323v1.pdf}.

I think the above paper is missing one argument for unbundling: I’ve seen many libraries that begin by making an interface (say I) that bundles *all* the operations (and their correctness properties) they will *ever* need and then *all* items in the library (say L) are parametrized by over I. A problem with this bundled approach is impossible to use *any* part of D if you are missing *any* operation (or proof of a logical property of the operation) in the interface I, even if parts of D don’t actually need that operation: I’ve run into situations where it is impossible to cook up an operation that 90 percent of L doesn’t use anyway. When using the unbundled approach, one can use Coq’s Section mechanism to ensure that definitions/proofs only depend on items of the interface they actually use, and not on a big bundle.

3.10 Further Reading

On the origins of typeclasses in Haskell:

- How to make ad-hoc polymorphism less ad hoc Philip Wadler and Stephen Blott. 16’th Symposium on Principles of Programming Languages, ACM Press, Austin, Texas, January 1989. {https://homepages.inf.ed.ac.uk/wadler/topics/type-classes.html}

The original paper on typeclasses In Coq:

- Matthieu Sozeau and Nicolas Oury. First-Class Type Classes. TPHOLs 2008. {https://link.springer.c
  3-540-71067-7_23}

Sources for this tutorial:

- Coq Reference Manual: {https://coq.inria.fr/refman/}
- Casteran and Sozeau's "Gentle Introduction": {https://www.labri.fr/perso/casteran/CoqArt/TypeCl
- Sozeau's slides from a talk at Penn: {https://www.cis.upenn.edu/~bcpierce/courses/670Fall12/slides.

Some of the many tutorials on typeclasses in Haskell:

- {https://en.wikibooks.org/wiki/Haskell/Classes_and_types}
- {http://learnyouahaskell.com/types-and-typeclasses} and {http://learnyouahaskell.com/making-our-own-types-and-typeclasses}

Chapter 4

Library QC.QC

4.1 QC: Core QuickChick

```
Require Import Arith Bool List ZArith. Import ListNotations.
From QuickChick Require Import QuickChick. Import QcNotation.
Set Warnings "-extraction-opaque-accessed,-extraction".
Require Export ExtLib.Structures.Monads.
Export MonadNotation. Open Scope monad_scope.
Require Import String. Local Open Scope string.
```

4.2 Generators

The heart of property-based random testing is generation of random inputs.

4.2.1 The *G* Monad

In QuickChick, a generator for elements of some type A belongs to the type *G* A. Intuitively, this type describes functions that take a random seed to an element of A. We will see below that *G* is actually a bit more than this, but this intuition will do for now.

QuickChick provides a number of primitives for building generators. First, *returnGen* takes a constant value and yields a generator that always returns this value.

```
Check returnGen.
```

===> returnGen : ?A -> G ?A

We can see how it behaves by using the *Sample* command, which supplies its generator argument with several different random seeds:

Next, given a random generator for A and a *function* f taking an A and yielding a random generator for B, we can plumb the two together into a generator for B that works by internally generating a random A and applying f to it.

```
Check bindGen.
```

===> bindGen : G ?A -> (?A -> G ?B) -> G ?B

Naturally, the implementation of *bindGen* must take care to pass different random seeds to the two sub-generators!

With these two primitives in hand, we can make *G* an instance of the **Monad** typeclass.

```
Section GMonadDef.

Instance GMonad : `{Monad G} | 3 :=
  {
    ret := @returnGen;
    bind := @bindGen
  }.

End GMonadDef.
```

4.2.2 Primitive generators

QuickChick provides several primitive generators for "ordered types," accessed via the *choose* combinator.

```
Check @choose.
```

===> @choose : forall A : Type, ChoosableFromInterval A -> A * A -> G A

The **ChoosableFromInterval** typeclass describes primitive types A, like natural numbers and integers (**Z**), for which it makes sense to randomly generate elements from a given interval.

```
Print ChoosableFromInterval.
```

===> Record ChoosableFromInterval (A : Type) : Type := Build_ChoosableFromInterval { super : Ord A; randomR : A * A -> RandomSeed -> A * RandomSeed; ... }.

Exercise: 1 star, standard, optional (cfi) Print out the full definition of **Choosable-FromInterval**. Can you understand what it means?

☐

4.2.3 Lists

Since they are a very commonly used compound datatype, lists have special combinators in QuickChick: *listOf* and *vectorOf*.

The *listOf* combinator takes as input a generator for elements of type A and returns a generator for lists of As.

```
Check listOf.
```

===> listOf : G ?A -> G (list ?A)

The second combinator, *vectorOf*, receives an additional numeric argument n, the length of the list to be generated.

```
Check vectorOf.
    ===> vectorOf : nat -> G ?A -> G (list ?A)
```

This raises a question. It's clear how *vectorOf* decides how big to make its lists (we tell it!), but how does *listOf* do it? The answer is hidden inside *G*.

In addition to handling random-seed plumbing, the *G* monad also maintains a "current maximum size" (in the style of a "reader monad", if you like that terminology): a natural number that can be used as an upper bound on the depth of generated objects.

Internally, *G* A is just a synonym for **nat** → *RandomSeed* → A.

```
Module DefineG.
Inductive G (A:Type) : Type :=
| MkG : (nat → RandomSeed → A) → G A.
Arguments MkG {A}.
End DefineG.
```

When it is searching for counterexamples, QuickChick progressively tries larger and larger values for the size bound n, to gradually explore deeper into the search space.

Each generator can choose to interpret the size bound however it wants, and there is no *enforced* guarantee that generators pay any attention to it at all; however, it is good practice to respect this bound when programming generators.

4.2.4 Custom Generators

Naturally, we also need generators for user-defined datatypes. Here's a simple one to play with:

```
Inductive color := Red | Green | Blue | Yellow.
```

In order for commands like *Sample* to display colors, we should make **color** an instance of the `Show` typeclass:

```
Instance show_color : Show color :=
  {| show c :=
       match c with
         | Red ⇒ "Red"
         | Green ⇒ "Green"
         | Blue ⇒ "Blue"
         | Yellow ⇒ "Yellow"
       end
  |}.
```

To generate a random color, we just need to pick one of the constructors Red, Green, Blue, or Yellow. This is done using *elems_*.

```
Check elems_.
    ===> elems : ?A -> list ?A -> G ?A
```

```
Definition genColor' : G color :=
  elems_ Red [ Red ; Green ; Blue ; Yellow ].
```

The first argument to *elems_* serves as a default result. If its list argument is not empty, *elems_* returns a generator that always picks an element of that list; otherwise the generator always returns the default object. This makes Coq's totality checker happy, but makes *elem_* a little awkward to use, since typically its second argument will be a non-empty constant list.

To make the common case smoother, QuickChick provides convenient notations that automatically extract the default element.

" 'elems' x " := elems x (cons x nil) " 'elems' x ; y " := elems x (cons x (cons y nil)) " 'elems' x ; y ; .. ; z " := elems x (cons x (cons y (.. (cons z nil)))) " 'elems' (x ;; l) " := elems x (cons x l)

Armed with *elems*, we can write a **color** generator the way we'd hope.

```
Definition genColor : G color :=
  elems [ Red ; Green ; Blue ; Yellow ].
```

For more complicated ADTs, QuickChick provides more combinators.

We showcase these using everyone's favorite datatype: trees!

Our trees are standard polymorphic binary trees; either Leafs or Nodes containing some payload of type A and two subtrees.

```
Inductive Tree A :=
| Leaf : Tree A
| Node : A → Tree A → Tree A → Tree A.

Arguments Leaf {A}.
Arguments Node {A} _ _ _.
```

Before getting to generators for trees, we again give a straightforward `Show` instance. (The need for a local `let fix` declaration stems from the fact that Coq's typeclasses, unlike Haskell's, are not automatically recursive. We could alternatively define *aux* with a separate top-level `Fixpoint`.)

```
Instance showTree {A} `{_ : Show A} : Show (Tree A) :=
  {| show := let fix aux t :=
       match t with
         | Leaf ⇒ "Leaf"
         | Node x l r ⇒
             "Node (" ++ show x ++ ") ("
                      ++ aux l ++ ") ("
                      ++ aux r ++ ")"
       end
     in aux
  |}.
```

This brings us to our first generator *combinator*, called *oneOf_*.

```
Check oneOf_.
```

===> oneOf : G ?A -> list (G ?A) -> G ?A

This combinator takes a default generator and a list of generators, and it picks one of the generators from the list uniformly at random (unless the list is empty, in which case it picks from the default generator). As with *elems_*, QuickChick introduces a more convenient notation *oneOf* to hide this default element.

Next, Coq's termination checker will save us from shooting ourselves in the foot!

The "obvious" first attempt at a generator is the following function genTree, which generates either a Leaf or else a Node whose subtrees are generated recursively (and whose payload is produced by a generator g for elements of type A).

Fixpoint genTree {A} (g : G A) : G (Tree A) := oneOf ret Leaf ;; liftM3 Node g (genTree g) (genTree g) .

Of course, this fixpoint will not pass Coq's termination checker. Attempting to justify this fixpoint informally, one might first say that at some point the random generation will pick a Leaf so it will eventually terminate. But the termination checker cannot understand this kind of probabilistic reasoning. Moreover, even informally, the reasoning is wrong: Every time we choose to generate a Node, we create two separate branches that must both be terminated with leaves. It is not hard to show that the *expected* size of the generated trees is actually infinite!

The solution is to use the standard "fuel" idiom that Coq users are familiar with. We add a natural number *sz* as a parameter. We decrease this size in each recursive call, and when it reaches O, we always generate Leaf. Thus, the initial *sz* parameter serves as a bound on the depth of the tree.

```
Fixpoint genTreeSized {A} (sz : nat) (g : G A) : G (Tree A) :=
  match sz with
    | O ⇒ ret Leaf
    | S sz' ⇒
        oneOf [
          ret Leaf ;
          liftM3 Node g (genTreeSized sz' g) (genTreeSized sz' g)
        ]
  end.
```

While this generator succeeds in avoiding nontermination, we can see just by observing the result of *Sample* that there is a problem: it produces way too many Leafs! This is actually not surprising, since half the time we generate a Leaf right at the outset.

We can obtain bigger trees more often if we skew the distribution towards Nodes using a more expressive QuickChick combinator, *freq_*.

```
Check freq_.
```

===> freq : G ?A -> seq (nat * G ?A) -> G ?A

As with *oneOf*, we usually use a convenient derived notation, called *freq*, that takes a list of generators, each tagged with a natural number that serves as the weight of that generator.

For example, in the following generator, a Leaf will be generated 1 / (sz + 1) of the time and a Node the remaining sz / (sz + 1) of the time.

```
Fixpoint genTreeSized' {A} (sz : nat) (g : G A) : G (Tree A) :=
  match sz with
    | O ⇒ ret Leaf
    | S sz' ⇒
        freq [ (1, ret Leaf) ;
               (sz, liftM3 Node g (genTreeSized' sz' g)
                                  (genTreeSized' sz' g))
             ]
  end.
```

This looks better.

Exercise: 2 stars, standard (genListSized) Write a sized generator for lists, following genTreeSized'.

Exercise: 3 stars, standard (genColorOption) Write a custom generator for values of type **option color**. Make it generate None about 1/10th of the time, and make it generate Some Red three times as often as the other colors.

4.2.5 Checkers

To showcase how such generators can be used to find counterexamples, suppose we define a function for "mirroring" a tree – swapping its left and right subtrees recursively.

```
Fixpoint mirror {A : Type} (t : Tree A) : Tree A :=
  match t with
    | Leaf ⇒ Leaf
    | Node x l r ⇒ Node x (mirror r) (mirror l)
  end.
```

To formulate a property about mirror, we need a structural equality test on trees. We can obtain that with minimal effort using the **Dec** typeclass and the *dec_eq* tactic.

```
Instance eq_dec_tree (t1 t2 : Tree nat) : Dec (t1 = t2).
Proof. constructor; dec_eq. Defined.
```

We expect that mirroring a tree twice should yield the original tree.

```
Definition mirrorP (t : Tree nat) := (mirror (mirror t)) = t?.
```

Now we want to use our generator to create a lot of random trees and, for each one, check whether mirrorP returns true or false.

Another way to say this is that we want to use mirrorP to build a *generator for test results*.

Let's see how this works.

(Let's open a playground module so we can show simplified versions of the actual QuickChick definitions.)

```
Module CheckerPlayground1.
```

First, we need a type of test results – let's call it **Result**. For the moment, think of **Result** as just an enumerated type with constructors Success and Failure.

```
Inductive Result := Success | Failure.
Instance showResult : Show Result :=
  {
    show r := match r with Success ⇒ "Success" | Failure ⇒ "Failure" end
  }.
```

Then we can define the type *Checker* to be *G* **Result**.

```
Definition Checker := G Result.
```

That is, a *Checker* embodies some way of performing a randomized test of the truth of some proposition, which, when applied to a random seed, will yield either Success (meaning that the proposition survived this test) or Failure (meaning that this test demonstrates that the proposition was false).

Sampling a *Checker* many times with different seeds will cause many different tests to be performed.

To check mirrorP, we'll need a way to build a *Checker* out of a function from trees to booleans.

Actually, we will be wanting to build *Checker*s based on many different types. So let's begin by defining a typeclass **Checkable**, where an instance for **Checkable** A will provide a way of converting an A into a *Checker*.

```
Class Checkable A :=
  {
    checker : A → Checker
  }.
```

It is easy to give a **bool** instance for **Checkable**.

```
Instance checkableBool : Checkable bool :=
  {
    checker b := if b then ret Success else ret Failure
  }.
```

That is, the boolean value true passes every test we might subject it to (i.e., the result of checker is a *G* that returns true for every seed), while false fails all tests.

Let's see what happens if we sample checkers for our favorite booleans, true and false.

(We need to exit the playground so that we can do *Sample*: because *Sample* is implemented internally via extraction to OCaml, which usually does not work from inside a `Module`.)

```
End CheckerPlayground1.
```

What we've done so far may look a a bit strange, since these checkers always generate the same results. We'll make things more interesting in a bit, but first let's pause to define one more basic instance of **Checkable**.

A decidable Prop is not too different from a boolean, so we should be able to build a checker from that too.

```
Module CheckerPlayground2.
Export CheckerPlayground1.

Instance checkableDec `{P : Prop} `{Dec P} : Checkable P :=
  {
    checker p := if P? then ret Success else ret Failure
  }.
```

(The definition looks a bit strange since it doesn't use its argument p. The intuition is that all the information in p is already encoded in P!)

Now suppose we pose a couple of (decidable) conjectures:

```
Conjecture c1 : 0 = 42.
Conjecture c2 : 41 + 1 = 42.

End CheckerPlayground2.
```

The somewhat astononishing thing about the **Checkable** instance for decidable Props is that, even though these are *conjectures* (we haven't proved them, so the "evidence" that Coq has for them internally is just an uninstantiated "evar"), we can still build checkers from them and sample from these checkers! (Why? Technically, it is because the **Checkable** instance for decidable properties does not look at its argument.)

Again, the intuition is that, although we didn't present proofs (and could not have, in the first case!), Coq already "knows" either a proof or a disproof of each of these conjectures because they are decidable.

```
Module CheckerPlayground3.
Import CheckerPlayground2.
```

Now let's go back to mirrorP.

We have seen that the result of mirrorP is **Checkable**. What we need is a way to take a function returning a checkable thing and make the function itself checkable.

We can easily do this, as long as the argument type of the function is something we know how to generate!

```
Definition forAll {A B : Type} `{Checkable B}
                  (g : G A) (f : A → B)
                : Checker :=
  a ← g ;;
  checker (f a).
```

```
End CheckerPlayground3.
```

Let's try this out!

First, let's revisit our color example to execute a simple check. We can define a boolean test that returns true for Red and false for other colors.

```
Definition isRed c :=
  match c with
    Red ⇒ true
  | _ ⇒ false
  end.
```

Since we can generate elements of **color** and we have a **Checkable** instance for **bool**, we can apply *forAll* to isRed and sample from the resulting *Checker* to run some tests.

Looks like not all colors are Red.

Good to know.

Now back to a more realistic example. What about mirrorP?

===> Success, Success, Success, Success, Success, Success, Success, Success, Success, Success, Success

Excellent: It looks like many tests are succeeding – maybe the property is true.

Let's instead try defining a bad property and see if we can detect that it's bad...

```
Definition faultyMirrorP (t : Tree nat) := (mirror t) = t ?.
```

===> Failure, Success, Failure, Success, Success, Success, Failure, Success, Failure, Failure, Success

Great – looks like a good number of tests are failing now, as expected.

There's only one little issue: What *are* the tests that are failing? We can tell by looking at the samples that the property is bad, but we can't see the counterexamples!

We can fix this by going back to the beginning and enriching the **Result** type to keep track of failing counterexamples.

```
Module CheckerPlayground4.

Inductive Result :=
  | Success : Result
  | Failure : ∀ {A} `{Show A}, A → Result.

Instance showResult : Show Result :=
  {
    show r := match r with
                    Success ⇒ "Success"
                  | Failure a ⇒ "Failure: " ++ show a
                  end
  }.

Definition Checker := G Result.

Class Checkable A :=
  {
    checker : A → Checker
  }.
Instance showUnit : Show unit :=
  {
    show u := "tt"
  }.
```

The failure cases in the **bool** and **Dec** checkers don't need to record anything except the Failure, so we put tt (the sole value of type **unit**) as the "failure reason."

```
Instance checkableBool : Checkable bool :=
  {
    checker b := if b then ret Success else ret (Failure tt)
  }.
Instance checkableDec `{P : Prop} `{Dec P} : Checkable P :=
  {
    checker p := if P? then ret Success else ret (Failure tt)
  }.
```

The interesting case is the *forAll* combinator. Here, we *do* have some useful information to record in the failure case – namely, the argument that caused the failure.

```
Definition forAll {A B : Type} `{Show A} `{Checkable B}
                  (g : G A) (f : A → B)
                : Checker :=
  a ← g ;;
  r ← checker (f a) ;;
  match r with
    Success ⇒ ret Success
  | Failure b ⇒ ret (Failure (a,b))
  end.
```

Note that, rather than just returning Failure a, we package up a together with b, which explains the reason for the failure of $f\ a$. This allows us to write several *forAll*s in sequence and capture all of their results in a nested tuple.

End CheckerPlayground4.

```
===> Failure: (Node (2) (Node (3) (Node (2) (Leaf) (Leaf)) (Leaf)) (Node (0) (Node
(2) (Leaf) (Leaf)) (Leaf)), tt), Success, Failure: (Node (2) (Node (3) (Node (3) (Leaf) (Leaf))
(Leaf)) (Leaf), tt), Success, Success, Success, Failure: (Node (1) (Node (2) (Leaf) (Node (3)
(Leaf) (Leaf))) (Node (0) (Leaf) (Node (1) (Leaf) (Leaf))), tt), Success, Failure: (Node (3)
(Node (0) (Node (0) (Leaf) (Leaf)) (Leaf)) (Node (0) (Leaf) (Node (2) (Leaf) (Leaf))), tt),
Failure: (Node (2) (Node (2) (Node (0) (Leaf) (Leaf)) (Leaf)) (Node (1) (Leaf) (Node (2)
(Leaf) (Leaf))), tt), Success
```

The bug is found several times and actual counterexamples are reported: nice!

Sampling repeatedly from a generator is just what the QuickChick command does, except that, instead of running a fixed number of tests and returning their results in a list, it runs tests only until the first counterexample is found.

```
===> QuickChecking (forAll (genTreeSized' 3 (choose (0, 3))) faultyMirrorP)
Node (0) (Node (0) (Node (2) (Leaf) (Leaf)) (Node (1) (Leaf) (Leaf))) (Node (1) (Node (0) (Leaf) (Leaf)) (Leaf))
```

Failed after 1 tests and 0 shrinks. (0 discards)

However, these counterexamples themselves still leave something to be desired: they are all *much* larger than is really needed to illustrate the bad behavior of faultyMirrorP.

This is where *shrinking* comes in.

4.3 Shrinking

Shrinking (sometimes known as "delta debugging") is a process that, given a counterexample to some property, searches (greedily) for smaller counterexamples.

Given a shrinking function s of type $A \to$ **list** A and a value x of type A that is known to falsify some property P, QuickChick tries P on all members of $s\ x$ until it finds another counterexample. It repeats this process, starting from the new counterexample, until it reaches a point where x fails property P but every element of $s\ x$ succeeds. This x is a "locally minimal" counterexample.

Clearly, this greedy algorithm only work if all elements of $s\ x$ are strictly "smaller" than x for all x – that is, there should be some total order on the type of x such that s is strictly decreasing in this order. This is the basic correctness requirement for a shrinker. It is guaranteed by all the shrinkers that QuickChick derives automatically, but ifyou roll your own shrinkers you need to be careful to maintain it.

Here is a shrinker for **color**s.

```
Instance shrinkColor : Shrink color :=
  {
   shrink c :=
     match c with
     | Red ⇒ [ ]
     | Green ⇒ [ Red ]
     | Blue ⇒ [ Red; Green ]
     | Yellow ⇒ [ Red; Green; Blue ]
     end
  }.
```

Most of the time, shrinking functions should try to return elements that are "just one step" smaller than the one they are given. For example, consider the default shrinking function for lists provided by QuickChick.

Print shrinkList.

===> shrinkList = fun (A : Type) (H : Shrink A) => {| shrink := shrinkListAux shrink |} : forall A : Type, Shrink A -> Shrink (list A)

Print shrinkListAux.

===> shrinkListAux = fix shrinkListAux (A : Type) (shr : A -> list A) (l : list A) : list (list A) := match l with | □ => □ | x :: xs => ((xs :: map (fun xs' : list A => x :: xs') (shrinkListAux A shr xs)) ++ map (fun x' : A => x' :: xs) (shr x)) end : forall A : Type, (A -> list A) -> list A -> list (list A)

An empty list cannot be shrunk, as there is no smaller list. A cons cell can be shrunk in three ways: by returning the tail of the list, by shrinking the tail of the list and keeping the same head, or by shrinking the head and keeping the same tail. By induction, this process can generate all smaller lists.

(Pro tip: One refinement to the advice that shrinkers should return "one step smaller" values is that, for potentially large data structures, it can sometimes greatly improve performance if a shrinker returns not only all the "slightly smaller" structures but also one or two "much smaller" structures. E.g. for a number x greater than 2, we might shrink to 0; x *div* 2; *pred* x.)

Writing a shrinking instance for trees is equally straightforward: we don't shrink Leafs, while for Nodes we can either return the left or right subtree, or shrink the payload, or shrink one of the subtrees.

```
Open Scope list.
Fixpoint shrinkTreeAux {A}
                 (s : A → list A) (t : Tree A)
               : list (Tree A) :=
  match t with
    | Leaf ⇒ []
    | Node x l r ⇒ [l] ++ [r] ++
                        map (fun x' ⇒ Node x' l r) (s x) ++
                        map (fun l' ⇒ Node x l' r) (shrinkTreeAux s l) ++
                        map (fun r' ⇒ Node x l r') (shrinkTreeAux s r)
  end.
Instance shrinkTree {A} `{Shrink A} : Shrink (Tree A) :=
  {| shrink x := shrinkTreeAux shrink x |}.
```

With shrinkTree in hand, we can use the *forAllShrink* property combinator, a variant of *forAll* that takes a shrinker as an additional argument, to test properties like faultyMirrorP.

===> Node (0) (Leaf) (Node (0) (Leaf) (Leaf))

Failed! After 1 tests and 8 shrinks

We now get a quite simple counterexample (in fact, one of two truly minimal ones), from which it is easy to see that the bad behavior occurs when the subtrees of a Node are different.

4.3.1 Exercise: Ternary Trees

For a comprehensive but still fairly simple exercise involving most of what we have seen, let's consider ternary trees, whose nodes have three childen instead of two.

```
Inductive TernaryTree A :=
| TLeaf : TernaryTree A
| TNode :
    A → TernaryTree A → TernaryTree A → TernaryTree A → TernaryTree A.
Arguments TLeaf {A}.
Arguments TNode {A} _ _ _ _.
```

Also consider the following (faulty?) mirror function. We'll want to do some testing to see what we can find about this function.

```
Fixpoint tern_mirror {A : Type} (t : TernaryTree A) : TernaryTree A :=
  match t with
    | TLeaf ⇒ TLeaf
    | TNode x l m r ⇒ TNode x (tern_mirror r) m (tern_mirror l)
  end.
```

Exercise: 1 star, standard (show_tern_tree) Write a `Show` instance for Ternary Trees.

Exercise: 2 stars, standard (gen_tern_tree) Write a generator for ternary trees.

The following line should generate a bunch of nat ternary trees.

Exercise: 2 stars, standard (shrink_tern_tree) Write a shrinker for ternary trees.

From the root node, we can define a path in a ternary tree by first defining a direction corresponding to a choice of child tree...

```
Inductive direction := Left | Middle | Right.
```

A direction tells us which child node we wish to visit. The traverse_node function uses a **direction** to visit the corresponding child.

```
Definition traverse_node {A} (t:direction) (l m r: TernaryTree A) :=
  match t with
  | Left ⇒ l
  | Middle ⇒ m
  | Right ⇒ r
  end.
```

A path in a ternary tree is a list of **direction**s. `Definition path := list direction.`

We can traverse a path by iterating over the **direction**s in the path and traversing the corresponding nodes.

```
Fixpoint traverse_path {A} (p:path) (t: TernaryTree A) :=
  match t with
  | TLeaf ⇒
    TLeaf
  | TNode x l m r ⇒
    match p with
    | h :: tl ⇒ traverse_path tl (traverse_node h l m r)
    | [] ⇒ t
    end
  end.
```

And we can mirror a path by simply swapping left and right throughout.

```
Definition path_mirror := map (fun t ⇒ match t with
                                        | Left ⇒ Right
                                        | Right ⇒ Left
                                        | _ ⇒ t
                                        end).
```

Before we can state some properties about paths in ternary trees, it's useful to have a decidable equality for ternary trees of **nat**s.

```
Instance eq_dec_tern_tree (t1 t2 : TernaryTree nat) : Dec (t1 = t2).
Proof. constructor; dec_eq. Defined.
```

Traversing a path in a tree should be the same as traversing the mirror of the path in the mirror of the tree, just with a mirrored tree result.

```
Definition tern_mirror_path_flip (t:TernaryTree nat) (p:path) :=
  ( traverse_path p t
   = tern_mirror (traverse_path (path_mirror p) (tern_mirror t))) ?.
```

Before using this property to test tern_mirror, we'll need to be able to generate, show, and shrink paths. To start, we'll derive those For *directions*. The Show and G definitions for **direction** are simple to write; the shrink will by default go left.

```
Instance showDirection : Show direction :=
  {| show x :=
       match x with
       | Left ⇒ "Left"
       | Middle ⇒ "Middle"
       | Right ⇒ "Right"
       end |}.

Definition genDirection : G direction :=
  elems [Left; Middle; Right].

Instance shrinkDirection : Shrink direction :=
  {| shrink d :=
       match d with
```

```
    | Left ⇒ []
    | Right | Middle ⇒ [Left]
    end |}.
```

For generating a list of paths, we'll use the built-in function *listOf*, which takes a generator for a type and generates lists of elements of that type. We do the same for show and shrink with showList and shrinkList respectively.

```
Instance showPath : Show path := showList.
```

```
Definition genPath : G path :=
  listOf genDirection.
```

```
Instance shrinkPath : Shrink path := shrinkList.
```

Exercise: 4 stars, standard (bug_finding_tern_tree) Using *genTernTreeSized* and *shrinkTernTree* find (and fix!) any bugs in tern_mirror.

4.4 Putting it all Together

Now we've got pretty much all the basic machinery we need, but the way we write properties – using *forAllShrink* and explicitly providing generators and shrinkers – is still heavier than it needs to be. We can use a bit more typeclass magic to further lighten things.

First, we introduce a typeclass **Gen** A with a single operator arbitrary of type $G\ A$.

```
Module DefineGen.
```

```
Class Gen (A : Type) :=
  {
    arbitrary : G A
  }.
```

```
End DefineGen.
```

Intuitively, **Gen** is a way of uniformly packaging up the generators for various types so that we do not need to remember their individual names – we can just call them all arbitrary.

```
Instance gen_color : Gen color :=
  {
    arbitrary := genColor
  }.
```

Next, for convenience we package **Gen** and **Shrink** together into an **Arbitrary** typeclass that is a subclass of both.

```
Module DefineArbitrary.
Import DefineGen.
```

```
Class Arbitrary (A : Type) `{Gen A} `{Shrink A}.
```

(The empty "body" of **Arbitrary** is elided.)

```
End DefineArbitrary.
```

We can use the top-level QuickChick command on quantified propositions with generatable and decidable conclusions, stating just the property and letting the typeclass machinery figure out the rest.

For example, suppose we want to test this:

```
Conjecture every_color_is_red : ∀ c, c = Red.
```

Since we have already defined **Gen** and **Shrink** instances for **color**, we automatically get an **Arbitrary** instance. The **Gen** part is used by the checker instance for "forall" propositions to generate random **color** arguments.

To show that the conclusion is decidable, we need to define a **Dec** instance for equality on colors.

```
Instance eq_dec_color (x y : color) : Dec (x = y).
Proof. dec_eq. Defined.
```

Putting it all together:

4.5 Sized Generators

Suppose we want to build an **Arbitrary** instance for trees.

We would like to use genTreeSized'; however, that generator takes an additional size argument.

Fortunately, since the *G* monad itself includes a size argument, we can "plumb" this argument into generators like genTreeSized'.

In other words, we can define an operator *sized* that takes a] sized generator and produces an unsized one.

```
Module DefineSized.
Import DefineG.
Definition sized {A : Type} (f : nat → G A) : G A :=
  MkG
     (fun n r ⇒
        match f n with
          MkG g ⇒ g n r
        end).
End DefineSized.
```

To streamline assembling generators, it is convenient to introduce one more typeclass, **GenSized**, whose instances are sized generators.

```
Module DefineGenSized.
Class GenSized (A : Type) :=
  {
    arbitrarySized : nat → G A
  }.
```

We can then define a generic **Gen** instance for types that have a **GenSized** instance, using *sized*:

```
Instance GenOfGenSized {A} `{GenSized A} : Gen A :=
  {
    arbitrary := sized arbitrarySized
  }.
End DefineGenSized.
```

Now we can make a **Gen** instance for trees by providing just an implementation of arbitrarySized.

```
Instance genTree {A} `{Gen A} : GenSized (Tree A) :=
  {| arbitrarySized n := genTreeSized' n arbitrary |}.
```

Finally, with the **Arbitrary** instance for trees, we can supply just faultyMirrorP to the QuickChick command.

Exercise: 2 stars, standard (tern_tree_typeclasses) Add typeclass instances for **GenSized** and **Shrink** so that you can QuickChick tern_mirror_path_flip directly.

4.6 Automation

Writing `Show` and **Arbitrary** instances is usually not hard, but it can get tedious when we are testing code that involves many new `Inductive` type declarations. To streamline this process, QuickChick provides some automation for deriving such instances for "plain datatypes" automatically!

```
Derive Arbitrary for Tree.
  ===> GenSizedree is defined ===> ShrinkTree is defined
Print GenSizedTree.
Print ShrinkTree.
```

```
Derive Show for Tree.
  ===> ShowTree is defined
Print ShowTree.
```

4.7 Collecting Statistics

Earlier in this tutorial we observed that our first definition of genTreeSized seemed to be producing too many Leaf constructors.

In that case, just eyeballing at a few results from *Sample* gave us an idea that something was wrong with the distribution of test cases, but it's often useful to collect more extensive statistics from larger sets of samples.

This is where *collect*, another property combinator, comes in.

```
Check @collect.
```

===> @collect : forall A prop : Type, Show A -> Checkable prop -> A -> prop -> Checker

That is, *collect* takes a checkable proposition and returns a new *Checker* (for the same proposition).

On the side, it takes a value from some `Showable` type A, which it remembers internally (in an enriched variant of the **Result** structure that we saw above) so that it can be collated and displayed at the end.

For example, suppose we measure the size of **Tree**s like this:

```
Fixpoint size {A} (t : Tree A) : nat :=
  match t with
    | Leaf ⇒ O
    | Node _ l r ⇒ 1 + size l + size r
  end.
```

We can write a dummy property treeProp to collect the sizes of the trees we are generating.

```
Definition treeProp (g : nat → G nat → G (Tree nat)) n :=
  forAll (g n (choose (0,n))) (fun t ⇒ collect (size t) true).
```

We see that 62.5% of the tests (4947 + 1258 / 10000) are either Leafs or empty Nodes, while rather few tests have larger sizes.

Compare this with genTreeSized'.

This generates far fewer tiny examples, likely leading to more efficient testing of interesting properties. However, there are still a lot of empty trees being generated. Can we do something about that?

Exercise: 2 stars, standard (genTreeSized'') Write a generator *genTreeSized''* that generates fewer empty trees. Check your results using *collect*.

4.8 Dealing with Preconditions

A large class of properties that are commonly encountered in property-based testing are *properties with preconditions*. The default QuickChick approach of generating inputs based on type information can be inefficient for such properties, especially for those with sparse preconditions (i.e. ones that are satisfied rarely with respect to their input domain).

```
Open Scope bool.
```

Consider a function that inserts a natural number into a sorted list.

```
Fixpoint sorted (l : list nat) :=
  match l with
  | [] ⇒ true
  | x::xs ⇒ match xs with
              | [] ⇒ true
              | y :: ys ⇒ (x <=? y) && (sorted xs)
              end
  end.
```

```
Fixpoint insert (x : nat) (l : list nat) :=
  match l with
  | [] ⇒ [x]
  | y::ys ⇒ if x <=? y then x :: l
              else y :: insert x ys
  end.
```

We could test insert using the following *conditional* property:

```
Definition insert_spec (x : nat) (l : list nat) :=
  sorted l ==> sorted (insert x l).
```

```
Definition test_insert_spec :=
  forAll (choose (0,10)) (fun x ⇒
  forAll (listOf (choose (0,10))) (fun l ⇒
  insert_spec x l)).
```

To test this property, QuickChick will try to generate random integers x and lists l, *check* whether the generated l is sorted, and, if it is, proceed to check the conclusion. If it is not, it will discard the generated inputs and try again.

As we can see, this can lead to many discarded tests (in this case, about twice as many as successful ones), which wastes a lot of CPU and leads to inefficient testing.

But the wasted effort is the least of our problems! Let's take a peek at the distribution of the lengths of generated lists using *collect*.

```
Definition insert_spec' (x : nat) (l : list nat) :=
  collect (List.length l) (insert_spec x l).
```

```
Definition test_insert_spec' :=
  forAll (choose (0,10)) (fun x ⇒
  forAll (listOf (choose (0,10))) (fun l ⇒
  insert_spec' x l)).
```

The vast majority of inputs have length 2 or less, while the larger lists are almost always discarded!

(This explains something you might have found suspicious in the previous statistics: that 1/3 of the randomly generated lists were already sorted!)

When dealing with properties with preconditions, it is common practice to write custom generators for well-distributed random data that satisfy the property.

For example, we can generate sorted lists with elements between *low* and *high* like this...

```
Fixpoint genSortedList (low high : nat) (size : nat)
              : G (list nat) :=
  match size with
  | O ⇒ ret []
  | S size' ⇒
    if high <? low then
      ret []
    else
      freq [ (1, ret []) ;
             (size, x ← choose (low, high);;
                    xs ← genSortedList x high size';;
                    ret (x :: xs)) ] end.
```

We use a size parameter as usual to control the length of generated lists.

If size is zero, we can only return the empty list which is always sorted. If size is nonzero, we need to perform an additional check whether *high* is less than *low* (in which case we also return the empty list). If it is not, we can proceed to choose to generate a cons cell, with its head generated between *low* and *high* and its tail generated recursively.

Finally, we can use *forAllShrink* to define a property using the new generator:

```
Definition insert_spec_sorted (x : nat) :=
  forAllShrink
    (genSortedList 0 10 10)
    shrink
    (fun l ⇒ insert_spec' x l).
```

Now the distribution of lengths looks much better, and we don't discard any tests!

QuickChick insert_spec_sorted.

===> QuickChecking insert_spec_sorted 947 : 0 946 : 4 946 : 10 938 : 6 922 : 9 916 : 2 900 : 7 899 : 3 885 : 8 854 : 5 847 : 1 +++ Passed 10000 tests (0 discards) Does this mean we are happy?

Exercise: 5 stars, standard, optional (uniform_sorted) Using "collect", find out whether generating a sorted list of numbers between 0 and 5 is uniform in the frequencies with which different *numbers* are found in the generated lists.

If not, figure out why. Then write a different generator that achieves a more uniform distribution (preserving uniformity in the lengths).

4.8.1 Another Precondition: Binary Search Trees

To conclude this chapter, let's look at *binary search trees*.

The isBST predicate characterizes trees with elements between *low* and *high*.

```
Fixpoint isBST (low high: nat) (t : Tree nat) :=
  match t with
  | Leaf ⇒ true
  | Node x l r ⇒ (low <? x) && (x <? high)
                   && (isBST low x l) && (isBST x high r)
  end.
```

Here is a (faulty?) insertion function for binary search trees.

```
Fixpoint insertBST (x : nat) (t : Tree nat) :=
  match t with
  | Leaf ⇒ Node x Leaf Leaf
  | Node x' l r ⇒ if x <? x' then Node x' (insertBST x l) r
                    else Node x' l (insertBST x r)
  end.
```

We would expect that if we insert an element that is within the bounds *low* and *high* into a binary search tree, then the result is also a binary search tree.

```
Definition insertBST_spec (low high : nat) (x : nat) (t : Tree nat) :=
  (low <? x) ==>
  (x <? high) ==>
  (isBST low high t) ==>
  isBST low high (insertBST x t).
```

We can see that a bug exists when inserting an element into a Node with the same payload: if the element already exists in the binary search tree, we should not change it.

However we are wasting too much testing effort. Indeed, if we fix the bug ...

```
Fixpoint insertBST' (x : nat) (t : Tree nat) :=
  match t with
  | Leaf ⇒ Node x Leaf Leaf
  | Node x' l r ⇒ if x <? x' then Node x' (insertBST' x l) r
                    else if x' <? x then Node x' l (insertBST' x r)
                    else t
  end.
```

```
Definition insertBST_spec' (low high : nat) (x : nat) (t : Tree nat) :=
  (low <? x) ==> (x <? high) ==> (isBST low high t) ==>
  isBST low high (insertBST' x t).
```

... and try again...

... we see that 90% of tests are being discarded.

Exercise: 4 stars, standard (gen_bst) Write a generator that produces binary search trees directly, so that you run 10000 tests with 0 discards.

Chapter 5

Library QC.TImp

5.1 TImp: Case Study: a Typed Imperative Language

```
Set Warnings "-notation-overridden,-parsing".
Set Warnings "-extraction-opaque-accessed,-extraction".
Require Import Bool Arith EqNat List. Import ListNotations.
From QuickChick Require Import QuickChick Tactics.
Import QcNotation QcDefaultNotation. Open Scope qc_scope.
Set Bullet Behavior "Strict Subproofs".
From QC Require Import QC.
```

Having covered the basics of QuickChick in the previous chapter, we are ready to dive into a more realistic case study: a typed variant of Imp, the simple imperative language introduced in *Logical Foundations*.

The version of Imp presented there enforces a syntactic separation between boolean and arithmetic expressions: *bexp* just ranges over boolean expressions, while *aexp* ranges over arithmetic ones. Moreover, variables are only allowed in *aexp* and hence only take numeric values. By contrast, in *Typed Imp* (TImp) we collapse the expression syntax and allow variables to range over both numbers and booleans. With the unified syntax, we introduce the notion of *well-typed* Imp expressions and programs (where every variable only ranges over values of a single type throughout the whole program). We then give an operational semantics to TImp in the form of a (partial) evaluation function – partial since, in the unified syntax, we can write nonsensical expressions such as $0 +$ True.

A common mantra in functional programming is "well-typed programs cannot go wrong," and TImp is no exception. The *soundness* property for TImp will state that evaluating well-typed expressions and programs always succeeds.

From the point of view of testing, soundness is interesting because it is a *conditional* property. As we saw in the previous chapter, testing such properties effectively requires custom generators. In this chapter, we show how to scale the techniques for writing generators explained in QC to more realistic generators for well-typed expressions and programs. In addition, we dicuss the need for custom shrinkers preserving invariants, a problem dual to that of custom generators.

Acknowledgement: We are grateful to Nicolas Koh for important contributions to an early version of this chapter.

5.2 Identifiers, Types and Contexts

5.2.1 Identifiers

For the type of identifiers of TImp we will use a wrapper around plain natural numbers.

```
Inductive id :=
| Id : nat → id.
```

We will need one identifier-specific operation, `fresh`: given any finite set of identifiers, we can produce one that is distinct from all other identifiers in the set.

To compute a fresh **id** given a list of **id**s we can just produce the number that is 1 larger than the maximum element:

```
Fixpoint max_elt (al:list id) : nat :=
  match al with
  | nil ⇒ 0
  | (Id n')::al' ⇒ max n' (max_elt al')
  end.
Definition fresh (al:list id) : id :=
  Id (S (max_elt al)).
```

We will also need a way of testing for equality, which we can derive with the standard *dec_eq* tactic. `Instance eq_id_dec (x1 x2 : id) : Dec (x1 = x2).`
`Proof. dec_eq. Defined.`

One advantage of using natural numbers as identifiers is that we can take advantage of the `Show` instance of **nat** to print them.

```
Instance show_id : Show id :=
  { show x := let '(Id n) := x in show n }.
```

To generate identifiers for TImp, we will not just generate arbitrary natural numbers. More often than not, we will need to generate a set of identifiers, or pick an identifier from such a set. If we represent a set as a list, we can do the former with a recursive function that generates n fresh **nat**s starting from the empty list. For the latter, we have QuickChick's *elems_* combinator.

```
Fixpoint get_fresh_ids n l :=
  match n with
  | 0 ⇒ l
  | S n' ⇒ get_fresh_ids n' ((fresh l) :: l)
  end.
```

Exercise: 2 stars, standard (genId) Write a **Gen** instance for **id** using the *elems_* combinator and get_fresh_ids.

□

There remains the question of how to shrink **id**s. We will answer that question when **id**s are used later in the chapter. For now, let's leave the **Shrink** instance as a no-op.

```
Instance shrinkId : Shrink id :=
  { shrink x := [] }.
```

5.2.2 Types

Here is the type of TImp types:

```
Inductive ty := TBool | TNat.
```

That is, TImp has two kinds of values: booleans and natural numbers.

To use **ty** in testing, we will need **Arbitrary**, Show, and **Dec** instances.

In QC.v, we saw how to write such generators by hand. We also saw, however, that this process can largely be automated for simple inductive types (like **ty**, **nat**, **list**, *tree*, etc.). QuickChick provides a top-level vernacular command to derive such instances.

```
Derive (Arbitrary, Show) for ty.
```

```
Check GenSizedty.
Check Shrinkty.
Check Showty.
```

Decidable equality instances are not yet derived fully automatically by QuickChick. However, the boilerplate we have to write is largely straightforward. As we saw in the previous chapters, **Dec** is a typeclass wrapper around ssreflect's decidable and we can use the *dec_eq* tactic to automate the process.

```
Instance eq_dec_ty (x y : ty) : Dec (x = y).
Proof. dec_eq. Defined.
```

5.2.3 List-Based Maps

To encode typing environments (and, later on, states), we will need maps from identifiers to values. However, the function-based representation in the *Software Foundations* version of Imp is not well suited for testing: we need to be able to access the domain of the map, fold over it, and test for equality; these are all awkward to define for Coq functions. Therefore, we introduce a simple list-based map representation that uses **id**s as the keys.

The operations we need are:

- *empty* : To create the empty map.
- *get* : To look up the binding of an element, if any.
- set : To update the binding of an element.

- *dom* : To get the list of keys in the map.

The implementation of a map is a simple association list. If a list contains multiple tuples with the same key, then the binding of the key in the map is the one that appears first in the list; that is, later bindings can be shadowed.

```
Definition Map A := list (id × A).
```

The *empty* map is the empty list.

```
Definition map_empty {A} : Map A := [].
```

To *get* the binding of an identifier x, we just need to walk through the list and find the first cons cell where the key is equal to x, if any.

```
Fixpoint map_get {A} (m : Map A) x : option A :=
  match m with
  | [] ⇒ None
  | (k, v) :: m' ⇒ if x = k ? then Some v else map_get m' x
  end.
```

To set the binding of an identifier, we just need to cons it at the front of the list.

```
Definition map_set {A} (m:Map A) (x:id) (v:A) : Map A := (x, v) :: m.
```

Finally, the domain of a map is just the set of its keys. `Fixpoint map_dom {A} (m:Map A) : list id :=`

```
  match m with
  | [] ⇒ []
  | (k', v) :: m' ⇒ k' :: map_dom m'
  end.
```

We next introduce a simple inductive relation, **bound_to** m x a, that holds precisely when the binding of some identifier x is equal to a in m

```
Inductive bound_to {A} : Map A → id → A → Prop :=
  | Bind : ∀ x m a, map_get m x = Some a → bound_to m x a.
```

5.2.4 Deciding bound_to (optional)

We can now decide whether **bound_to** m x a holds for a given arrangement of m, x and a. On a first reading, you may prefer to skip the next few paragraphs (until the start of the `Context` subsection), which deal with partially automating the proofs for such instances.

```
Instance dec_bound_to {A : Type} Gamma x (T : A)
            `{D : ∀ (x y : A), Dec (x = y)}
             : Dec (bound_to Gamma x T).
Proof.
  constructor. unfold ssrbool.decidable.
  destruct (map_get Gamma x) eqn:Get.
```

After unfolding decidable and destructing map_get *Gamma* *x*, we are left with two subgoals. In the first, we know that map_get *Gamma* *x* = Some *a* and effectively want to decide whether map_get *Gamma* *x* = Some *T* or not. Which means we need to decide whether *a* = *T*. Thankfully, we can decide that using our hypothesis *D*.

```
- destruct (D a T) as [[Eq | NEq]]; subst.
```

At this point, the first goal can be immediately decided positevely using constructor.

```
  + left. constructor. auto.
```

In the second subgoal, we can show that **bound_to** doesn't hold.

```
  + right; intro Contra; inversion Contra; subst; clear Contra.
    congruence.
```

Both of these tactic patterns are very common in non-trival **Dec** instances. It is worth we automating them a bit using LTac.

```
Abort.
```

A lot of the time, we can immediately decide a property positivly using constructor applications. That is captured in *solve_left*.

```
Ltac solve_left := try solve [left; econstructor; eauto].
```

Much of the time, we can also immediately decide that a property *doesn't* hold by assuming it, doing inversion, and using congruence.

```
Ltac solve_right :=
  let Contra := fresh "Contra" in
  try solve [right; intro Contra; inversion Contra; subst;
             clear Contra; eauto; congruence].
```

We group both in a single tactic, which does nothing at all if it fails to solve a goal, thanks to try solve.

```
Ltac solve_sum := solve_left; solve_right.
```

We can now prove the **Dec** instance quite concisely.

```
Instance dec_bound_to {A : Type} Gamma x (T : A)
         `{D : ∀ (x y : A), Dec (x = y)}
  : Dec (bound_to Gamma x T).
Proof.
  constructor. unfold ssrbool.decidable.
  destruct (map_get Gamma x) eqn:Get; solve_sum.
  destruct (D a T) as [[Eq | NEq]]; subst; solve_sum.
Defined.
```

5.2.5 Contexts

Typing contexts in TImp are just maps from identifiers to types.

```
Definition context := Map ty.
```

Given a context *Gamma* and a type *T*, we can try to generate a random *identifier* whose binding in *Gamma* is *T*.

We use *List.filter* to extract all of the elements in *Gamma* whose type is equal to *T* and then, for each (*a*,*T'*) that remains, return the name *a*. We use the *oneOf_* combinator to pick an generator from this list at random.

Since the filtered list might be empty we return an **option**, we use ret None as the default element for *oneOf_*.

```
Definition gen_typed_id_from_context (Gamma : context) (T : ty)
                 : G (option id) :=
  oneOf_ (ret None)
           (List.map (fun '(x,T') ⇒ ret (Some x))
                       (List.filter (fun '(x,T') ⇒ T = T'?) Gamma)).
```

We also need to generate typing contexts.

Given some natural number *n* to serve as the size of the context, we first create its domain, *n* fresh identifiers. We then create *n* arbitrary types with *vectorOf*, using the **Gen** instance for **ty** we derived earlier. Finally, we zip (*List.combine*) the domain with the ranges which creates the (list-based) map.

```
Definition gen_context (n : nat) : G context :=
  let domain := get_fresh_ids n [] in
  range ← vectorOf n arbitrary ;;
  ret (List.combine domain range).
```

5.3 Expressions

We are now ready to introduce the syntax of expressions in TImp. The original Imp had two distinct types of expressions, arithmetic and boolean expressions; variables were only allowed to range over natural numbers. In TImp, we extend variables to range over boolean values as well, and we collapse expressions into a single type **exp**.

```
Inductive exp : Type :=
  | EVar : id → exp
  | ENum : nat → exp
  | EPlus : exp → exp → exp
  | EMinus : exp → exp → exp
  | EMult : exp → exp → exp
  | ETrue : exp
  | EFalse : exp
  | EEq : exp → exp → exp
  | ELe : exp → exp → exp
  | ENot : exp → exp
  | EAnd : exp → exp → exp.
```

To print expressions we derive a Show Instance.

Derive Show for *exp*.

5.3.1 Typed Expressions

The following inductive relation characterizes well-typed expressions of a particular type. It is straightforward, using **bound_to** to access the typing context in the variable case

```
Reserved Notation "Gamma '||-' e '\IN' T" (at level 40).
Inductive has_type : context → exp → ty → Prop :=
| Ty_Var : ∀ x T Gamma,
    bound_to Gamma x T → Gamma ||- EVar x \IN T
| Ty_Num : ∀ Gamma n,
    Gamma ||- ENum n \IN TNat
| Ty_Plus : ∀ Gamma e1 e2,
    Gamma ||- e1 \IN TNat → Gamma ||- e2 \IN TNat →
    Gamma ||- EPlus e1 e2 \IN TNat
| Ty_Minus : ∀ Gamma e1 e2,
    Gamma ||- e1 \IN TNat → Gamma ||- e2 \IN TNat →
    Gamma ||- EMinus e1 e2 \IN TNat
| Ty_Mult : ∀ Gamma e1 e2,
    Gamma ||- e1 \IN TNat → Gamma ||- e2 \IN TNat →
    Gamma ||- EMult e1 e2 \IN TNat
| Ty_True : ∀ Gamma, Gamma ||- ETrue \IN TBool
| Ty_False : ∀ Gamma, Gamma ||- EFalse \IN TBool
| Ty_Eq : ∀ Gamma e1 e2,
    Gamma ||- e1 \IN TNat → Gamma ||- e2 \IN TNat →
    Gamma ||- EEq e1 e2 \IN TBool
| Ty_Le : ∀ Gamma e1 e2,
    Gamma ||- e1 \IN TNat → Gamma ||- e2 \IN TNat →
    Gamma ||- ELe e1 e2 \IN TBool
| Ty_Not : ∀ Gamma e,
    Gamma ||- e \IN TBool → Gamma ||- ENot e \IN TBool
| Ty_And : ∀ Gamma e1 e2,
    Gamma ||- e1 \IN TBool → Gamma ||- e2 \IN TBool →
    Gamma ||- EAnd e1 e2 \IN TBool

where "Gamma '||-' e '\IN' T" := (has_type Gamma e T).
```

While the typing relation is almost entirely standard, there is a choice to make about the Ty_Eq rule. The Ty_Eq constructor above requires that the arguments to an equality check are both arithmetic expressions (just like it was in Imp), which simplifies some of the discussion in the remainder of the chapter. We could have allowed for equality checks between booleans as well - that will become an exercise at the end of this chapter.

Once again, we need a decidable instance for the typing relation of TImp. You can skip to the next exercise if you are not interested in specific proof details.

We will need a bit more automation for this proof. We will have a lot of hypotheses of the form:

IH : forall (T : ty) (Gamma : context), ssrbool.decidable (Gamma ||- e1 \IN T)

Using a brute-force approach, we instantiate such *IH* with both TNat and TBool, destruct them and then call *solve_sum*.

The `pose` *proof* tactic introduces a new hypothesis in our context, while `clear` *IH* removes it so that we don't try the same instantiations again and again.

```
Ltac solve_inductives Gamma :=
  repeat (match goal with
      [ IH : ∀ _ _, _ ⊢ _ ] ⇒
      let H1 := fresh "H1" in
      pose proof (IH TNat Gamma) as H1;
      let H2 := fresh "H2" in
      pose proof (IH TBool Gamma) as H2;
      clear IH;
      destruct H1; destruct H2; solve_sum
    end).
```

Typing in TImp is decidable: given an expression *e*, a context *Gamma* and a type *T*, we can decide whether **has_type** *Gamma e T* holds.

```
Instance dec_has_type (e : exp) (Gamma : context) (T : ty)
  : Dec (Gamma ||- e \IN T).
Proof with solve_sum.
  constructor.
  generalize dependent Gamma.
  generalize dependent T.
  induction e; intros T Gamma; unfold ssrbool.decidable;
    try solve [destruct T; solve_sum];
    try solve [destruct T; solve_inductives Gamma].
  destruct (dec_bound_to Gamma i T); destruct dec; solve_sum.
Defined.
```

Exercise: 3 stars, standard (arbitraryExp) Derive **Arbitrary** for expressions. To see how good it is at generating *well-typed* expressions, write a conditional property *cond_prop* that is (trivially) always true, with the precondition that some expression is well-typed. Try to check that property like this:

QuickChickWith (updMaxSize stdArgs 3) cond_prop.

This idiom sets the maximum-size parameter for all generators to 3, rather the default, which is something larger like 10. When generating examples, QuickChick will start with size 0, gradually increase the size until the maximum size is reached, and then start over. What happens when you vary the size bound?

5.3.2 Generating Typed Expressions

Instead of generating expressions and filtering them using **has_type**, we can be smarter and generate *well-typed* expressions for a given context directly.

It is common for conditional generators to return **option**s, allowing the possibility of failure if a wrong choice is made internally. For example, if we wanted to generate an expression of type TNat and chose to try to do so by generating a variable, then we might not be able to finish (if the context is empty or only binds booleans).

To chain together two generators with types of the form G (**option** ...), we need to execute the first generator, match on its result, and, when it is a Some, apply the second generator.

```
Definition bindGenOpt {A B : Type}
                (gma : G (option A)) (k : A → G (option B))
              : G (option B) :=
  ma ← gma ;;
  match ma with
  | Some a ⇒ k a
  | None ⇒ ret None
  end.
```

This pattern is common enough that QuickChick introduces explicit monadic notations.

```
Print GOpt.
    GOpt = fun A : Type => G (option A) : Type -> Type
Check Monad_GOpt.
    Monad_GOpt : Monad GOpt
```

This brings us to our first interesting generator – one for typed expressions. We assume that *Gamma* and *T* are inputs to the generation process. We also use a size parameter to control the depth of generated expressions (i.e., we'll define a sized generator).

Let's start with a much smaller relation: *has_type_1* (which consists of just the first constructor of **has_type**), to demonstrate how to build up complex generators for typed expressions from smaller parts.

```
Module TypePlayground1.

Inductive has_type : context → exp → ty → Prop :=
  | Ty_Var : ∀ x T Gamma,
      bound_to Gamma x T → has_type Gamma (EVar x) T.

End TypePlayground1.
```

To generate *e* such that **has_type** *Gamma e T* holds, we need to pick one of its constructors (there is only one choice, here) and then try to satisfy its preconditions by generating more things. To satisfy Ty_Var (given *Gamma* and *T*), we need to generate *x* such that

bound_to *Gamma x T*. But we already have such a generator! We just need to wrap it in an EVar.

```
Definition gen_typed_evar (Gamma : context) (T : ty) : GOpt exp :=
  x ← gen_typed_id_from_context Gamma T;;
  ret (EVar x).
```

(Note that this is the ret of the GOpt monad.)

Now let's consider an extended typing relation, extending the previous one with all of the constructors of **has_type** that do not recursively require **has_type** as a side-condition. These will be the *base cases* for our final generator.

```
Module TypePlayground2.

Inductive has_type : context → exp → ty → Prop :=
| Ty_Var : ∀ x T Gamma,
     bound_to Gamma x T → has_type Gamma (EVar x) T
| Ty_Num : ∀ Gamma n,
     has_type Gamma (ENum n) TNat
| Ty_True : ∀ Gamma, has_type Gamma ETrue TBool
| Ty_False : ∀ Gamma, has_type Gamma EFalse TBool.

End TypePlayground2.
```

We can already generate values satisfying Ty_Var using gen_typed_evar. For the rest of the rules, we will need to pattern match on the input T, since Ty_Num can only be used if T = TNat, while Ty_True and Ty_False can only be used if T = TBool.

```
Definition base' Gamma T : list (GOpt exp) :=
        gen_typed_evar Gamma T ::
        match T with
        | TNat ⇒ [ n ← arbitrary;; ret (Some (ENum n))]
        | TBool ⇒ [ ret ETrue ; ret EFalse ]
        end.
```

We now need to go from a list of (optional) generators to a single generator. We could do that using the *oneOf* combinator (which chooses uniformly), or the *freq* combinator (by adding weights).

Instead, we introduce a new one, called *backtrack*:

backtrack : list (nat * GOpt ?A) -> GOpt ?A

Just like *freq*, *backtrack* selects one of the generators according to the input weights. Unlike *freq*, if the chosen generator fails (i.e. produces None), *backtrack* will discard it, choose another, and keep going until one succeeds or all possibilities are exhausted. Our base-case generator could then be like this:

```
Definition base Gamma T :=
        (2, gen_typed_evar Gamma T) ::
        match T with
        | TNat ⇒ [ (2, n ← arbitrary;; ret (Some (ENum n)))]
```

```
    | TBool ⇒ [ (1, ret ETrue)
              ; (1, ret EFalse) ]
    end.
```

```
Definition gen_has_type_2 Gamma T := backtrack (base Gamma T).
```

To see how we handle recursive rules, let's consider a third sub-relation, *has_type_3*, with just variables and addition:

```
Module TypePlayground3.

Inductive has_type : context → exp → ty → Prop :=
 | Ty_Var : ∀ x T Gamma,
     bound_to Gamma x T → has_type Gamma (EVar x) T
 | Ty_Plus : ∀ Gamma e1 e2,
     has_type Gamma e1 TNat → has_type Gamma e2 TNat →
     has_type Gamma (EPlus e1 e2) TNat.

End TypePlayground3.
```

Typing derivations involving EPlus nodes are binary trees, so we need to add a size parameter to enforce termination. The base case (*Ty_Var3*) is handled using gen_typed_evar just like before. The non-base case can choose between trying to generate *Ty_Var3* and trying to generate *Ty_Plus3*. For the latter, the input type *T* must be TNat, otherwise it is not applicable. Once again, this leads to a match on *T*:

```
Fixpoint gen_has_type_3 size Gamma T : GOpt exp :=
  match size with
  | O ⇒ gen_typed_evar Gamma T
  | S size' ⇒
    backtrack
      ([ (1, gen_typed_evar Gamma T) ]
      ++ match T with
         | TNat ⇒
           [ (size, e1 ← gen_has_type_3 size' Gamma TNat;;
                    e2 ← gen_has_type_3 size' Gamma TNat;;
                    ret (EPlus e1 e2)) ]
         | _ ⇒ []
         end)
  end.
```

Putting all this together, we get the full generator for well-typed expressions.

```
Fixpoint gen_exp_typed_sized
            (size : nat) (Gamma : context) (T : ty)
     : GOpt exp :=
  let base := base Gamma T in
  let recs size' :=
    match T with
  | TNat ⇒
    [ (size, e1 ← gen_exp_typed_sized size' Gamma TNat ;;
            e2 ← gen_exp_typed_sized size' Gamma TNat ;;
            ret (EPlus e1 e2))
    ; (size, e1 ← gen_exp_typed_sized size' Gamma TNat ;;
            e2 ← gen_exp_typed_sized size' Gamma TNat ;;
            ret (EMinus e1 e2))
    ; (size, e1 ← gen_exp_typed_sized size' Gamma TNat ;;
            e2 ← gen_exp_typed_sized size' Gamma TNat ;;
            ret (EMult e1 e2)) ]
  | TBool ⇒
  [ (size, e1 ← gen_exp_typed_sized size' Gamma TNat ;;
             e2 ← gen_exp_typed_sized size' Gamma TNat ;;
             ret (EEq e1 e2))
     ; (size, e1 ← gen_exp_typed_sized size' Gamma TNat ;;
             e2 ← gen_exp_typed_sized size' Gamma TNat ;;
             ret (ELe e1 e2))
     ; (size, e1 ← gen_exp_typed_sized size' Gamma TBool ;;
             ret (ENot e1))
     ; (size, e1 ← gen_exp_typed_sized size' Gamma TBool ;;
             e2 ← gen_exp_typed_sized size' Gamma TBool ;;
             ret (EAnd e1 e2)) ]
    end in
  match size with
  | O ⇒
    backtrack base
  | S size' ⇒
    backtrack (base ++ recs size')
  end.
```

When writing such complex generators, it's good to have some tests to verify that we are generating what we expect. For example, here we would expect gen_exp_typed_sized to always return expressions that are well typed.

We can use *forAll* to encode such a property.

```
Definition gen_typed_has_type :=
  let num_vars := 4 in
  let top_level_size := 3 in
  forAll (gen_context num_vars) (fun Gamma ⇒
  forAll arbitrary (fun T ⇒
  forAll (gen_exp_typed_sized top_level_size Gamma T) (fun me ⇒
  match me with
  | Some e ⇒ (has_type Gamma e T)?
  | None ⇒ false
  end))).
```

5.4 Values and States

5.4.1 Values

In the original Imp language from *Logical Foundations*, variables ranged over natural numbers, so states were just maps from identifiers to **nat**. Since we now want to extend this to also include booleans, we need a type of **value**s that includes both.

```
Inductive value := VNat : nat → value | VBool : bool → value.
```

Derive Show for *value*.

We can also quickly define a typing relation for values, a Dec instance for it, and a generator for values of a given type.

```
Inductive has_type_value : value → ty → Prop :=
  | TyVNat : ∀ n, has_type_value (VNat n) TNat
  | TyVBool : ∀ b, has_type_value (VBool b) TBool.
```

```
Instance dec_has_type_value v T : Dec (has_type_value v T).
Proof. constructor; unfold ssrbool.decidable.
destruct v; destruct T; solve_sum.
Defined.
```

```
Definition gen_typed_value (T : ty) : G value :=
  match T with
  | TNat ⇒ n ← arbitrary;; ret (VNat n)
  | TBool ⇒ b ← arbitrary;; ret (VBool b)
  end.
```

5.4.2 States

States in TImp are just maps from identifiers to values

```
Definition state := Map value.
```

We introduce an inductive relation that specifies when a state is well typed in a context (that is, when all of its variables are mapped to values of appropriate types).

We encode this in an element-by-element style inductive relation: empty states are only well typed with respect to an empty context, while non-empty states need to map their head identifier to a value of the appropriate type (and their tail must similarly be well typed).

```
Inductive well_typed_state : context → state → Prop :=
| TS_Empty : well_typed_state map_empty map_empty
| TS_Elem : ∀ x v T st Gamma,
    has_type_value v T → well_typed_state Gamma st →
    well_typed_state ((x, T) :: Gamma) ((x, v) :: st).
Instance dec_well_typed_state Gamma st : Dec (well_typed_state Gamma st).
Proof.
constructor; unfold ssrbool.decidable.
generalize dependent Gamma.
induction st; intros; destruct Gamma; solve_sum.
destruct a as [a v]; destruct p as [a' T].
destruct (@dec (a = a') _ ); solve_sum.
subst; specialize (IHst Gamma); destruct IHst; solve_sum.
destruct (dec_has_type_value v T); destruct dec; solve_sum.
Defined.

Definition gen_well_typed_state (Gamma : context) : G state :=
  sequenceGen (List.map (fun '(x, T) ⇒
                                    v ← gen_typed_value T;;
                                    ret (x, v)) Gamma).
```

5.5 Evaluation

The evaluation function takes a state and an expression and returns an optional value, which can be None if the expression encounters a dynamic type error like trying to perform addition on a boolean.

```
Fixpoint eval (st : state) (e : exp) : option value :=
  match e with
  | EVar x ⇒ map_get st x
  | ENum n ⇒ Some (VNat n)
  | EPlus e1 e2 ⇒
    match eval st e1, eval st e2 with
    | Some (VNat n1), Some (VNat n2) ⇒ Some (VNat (n1 + n2))
    | _, _ ⇒ None
    end
  | EMinus e1 e2 ⇒
    match eval st e1, eval st e2 with
    | Some (VNat n1), Some (VNat n2) ⇒ Some (VNat (n1 - n2))
    | _, _ ⇒ None
    end
  | EMult e1 e2 ⇒
    match eval st e1, eval st e2 with
    | Some (VNat n1), Some (VNat n2) ⇒ Some (VNat (n1 × n2))
    | _, _ ⇒ None
    end
  | ETrue ⇒ Some (VBool true )
  | EFalse ⇒ Some (VBool false )
  | EEq e1 e2 ⇒
    match eval st e1, eval st e2 with
    | Some (VNat n1), Some (VNat n2) ⇒ Some (VBool (n1 =? n2))
    | _, _ ⇒ None
    end
  | ELe e1 e2 ⇒
    match eval st e1, eval st e2 with
    | Some (VNat n1), Some (VNat n2) ⇒ Some (VBool (n1 <? n2))
    | _, _ ⇒ None
    end
  | ENot e ⇒
    match eval st e with
    | Some (VBool b) ⇒ Some (VBool (negb b))
    | _ ⇒ None
    end
  | EAnd e1 e2 ⇒
    match eval st e1, eval st e2 with

    | Some (VBool b), Some (VNat n2) ⇒ Some (VBool (negb b))

    | _, _ ⇒ None
    end
  end.
```

We will see in a later chapter (QuickChickTool) how we can use QuickChick to introduce such *mutations* and have them automatically checked.

Type soundness states that, if we have an expression *e* of a given type *T* as well as a well-typed state *st*, then evaluating *e* in *st* will never fail.

```
Definition isNone {A : Type} (m : option A) :=
  match m with
  | None ⇒ true
  | Some _ ⇒ false
  end.
```

```
Conjecture expression_soundness : ∀ Gamma st e T,
    well_typed_state Gamma st → Gamma ||- e \IN T →
    isNone (eval st e) = false.
```

To test this property, we construct an appropriate checker:

```
Definition expression_soundness_exec :=
  let num_vars := 4 in
  let top_level_size := 3 in
  forAll (gen_context num_vars) (fun Gamma ⇒
  forAll (gen_well_typed_state Gamma) (fun st ⇒
  forAll arbitrary (fun T ⇒
  forAll (gen_exp_typed_sized 3 Gamma T) (fun me ⇒
  match me with
  | Some e ⇒ negb (isNone (eval st e))
  | _ ⇒ true
  end)))).
```

Where is the bug?? Looks like we need some shrinking!

5.5.1 Shrinking for Expressions

Let's see what happens if we use the default shrinker for expressions carelessly.

Derive Shrink for *exp*.

```
Definition expression_soundness_exec_firstshrink :=
  let num_vars := 4 in
  let top_level_size := 3 in
  forAll (gen_context num_vars) (fun Gamma ⇒
  forAll (gen_well_typed_state Gamma) (fun st ⇒
  forAll arbitrary (fun T ⇒
  forAllShrink (gen_exp_typed_sized 3 Gamma T) shrink (fun me ⇒
  match me with
  | Some e ⇒ negb (isNone (eval st e))
  | _ ⇒ true
  end)))).
```

The expression shrank to something ill-typed! Since it causes the checker to fail, QuickChick views this as a succesfull shrink, even though this could not actually be produced by our generator and doesn't satisfy our preconditions! One solution would be to check the preconditions in the Checker, filtering out shrinks. But that would be inefficient.

We not only need to shrink expressions, we need to shrink them so that their type is preserved! To accomplish this, we need to intuitively follow the opposite of the procedure we did for generators: look at a typing derivation and see what parts of it we can shrink to while maintaining their types so that the type of the entire thing is preserved.

As in the case of *gen_exp_typed*, we are going to build up the full shrinker in steps. Let's begin with shrinking constants.

- If e = ENum x for some x, all we can do is try to shrink x.
- If e = ETrue or e = EFalse, we could shrink it to the other. But remember, we don't want to do both, as this would lead to an infinite loop in shrinking! We choose to shrink EFalse to ETrue.

```
Definition shrink_base (e : exp) : list exp :=
  match e with
  | ENum n ⇒ map ENum (shrink n)
  | ETrue ⇒ []
  | EFalse ⇒ [ETrue]
  | _ ⇒ []
  end.
```

The next case, EVar, must take the type T to be preserved into account. To shrink an EVar we could try shrinking the inner identifier, but shrinking an identifier by shrinking its natural number representation makes little sense. Better, we can try to shrink the EVar to a constant of the appropriate type.

```
Definition shrink_evar (T : ty) (e : exp) : list exp :=
  match e with
  | EVar x ⇒
    match T with
    | TNat ⇒ [ENum 0]
    | TBool ⇒ [ETrue ; EFalse]
    end
  | _ ⇒ []
  end.
```

Finally, we need to be able to shrink the recursive cases. Consider EPlus e1 e2:

- We could try (recursively) shrinking e1 or e2 preserving their TNat type.
- We could try to shrink directly to e1 or e2 since their type is the same as EPlus e1 e2.

On the other hand, consider EEq e1 e2:

- Again, we could recursively shrink e1 or e2.
- But we can't shrink *to* e1 or e2 since they are of a different type.
- For faster shrinking, we can also try to shrink such expressions to boolean constants directly.

```
Fixpoint shrink_rec (T : ty) (e : exp) : list exp :=
  match e with
  | EPlus e1 e2 ⇒
    e1 :: e2
       :: (List.map (fun e1' ⇒ EPlus e1' e2) (shrink_rec T e1))
       ++ (List.map (fun e2' ⇒ EPlus e1 e2') (shrink_rec T e2))
  | EEq e1 e2 ⇒
    ETrue :: EFalse
       :: (List.map (fun e1' ⇒ EEq e1' e2) (shrink_rec TNat e1))
```

```
          ++ (List.map (fun e2' ⇒ EEq e1 e2') (shrink_rec TNat e2))
  | _ ⇒ []
  end.
```

Putting it all together yields the following smart shrinker:

```
Fixpoint shrink_exp_typed (T : ty) (e : exp) : list exp :=
  match e with
  | EVar _ ⇒
    match T with
    | TNat ⇒ [ENum 0]
    | TBool ⇒ [ETrue ; EFalse]
    end
  | ENum _ ⇒ []
  | ETrue ⇒ []
  | EFalse ⇒ [ETrue]
  | EPlus e1 e2 ⇒
    e1 :: e2
       :: (List.map (fun e1' ⇒ EPlus e1' e2) (shrink_exp_typed T e1))
       ++ (List.map (fun e2' ⇒ EPlus e1 e2') (shrink_exp_typed T e2))
  | EMinus e1 e2 ⇒
    e1 :: e2 :: (EPlus e1 e2)
       :: (List.map (fun e1' ⇒ EMinus e1' e2) (shrink_exp_typed T e1))
       ++ (List.map (fun e2' ⇒ EMinus e1 e2') (shrink_exp_typed T e2))
  | EMult e1 e2 ⇒
    e1 :: e2 :: (EPlus e1 e2)
       :: (List.map (fun e1' ⇒ EMult e1' e2) (shrink_exp_typed T e1))
       ++ (List.map (fun e2' ⇒ EMult e1 e2') (shrink_exp_typed T e2))
  | EEq e1 e2 ⇒
    ETrue :: EFalse
       :: (List.map (fun e1' ⇒ EEq e1' e2) (shrink_exp_typed TNat e1))
       ++ (List.map (fun e2' ⇒ EEq e1 e2') (shrink_exp_typed TNat e2))
  | ELe e1 e2 ⇒
    ETrue :: EFalse :: (EEq e1 e2)
       :: (List.map (fun e1' ⇒ ELe e1' e2) (shrink_exp_typed TNat e1))
       ++ (List.map (fun e2' ⇒ ELe e1 e2') (shrink_exp_typed TNat e2))
  | ENot e ⇒
    ETrue :: EFalse :: e :: (List.map ENot (shrink_exp_typed T e))
  | EAnd e1 e2 ⇒
    ETrue :: EFalse :: e1 :: e2
       :: (List.map (fun e1' ⇒ EAnd e1' e2) (shrink_exp_typed TBool e1))
       ++ (List.map (fun e2' ⇒ EAnd e1 e2') (shrink_exp_typed TBool e2))
  end.
```

As we saw for generators, we can also perform sanity checks on our shrinkers. Here, when the shrinker is applied to an expression of a given type, all of its results should have the same type.

```
Definition shrink_typed_has_type :=
  let num_vars := 4 in
  let top_level_size := 3 in
  forAll (gen_context num_vars) (fun Gamma ⇒
  forAll arbitrary (fun T ⇒
  forAll (gen_exp_typed_sized top_level_size Gamma T) (fun me ⇒
  match me with
  | Some e ⇒
    List.forallb (fun e' ⇒ (has_type Gamma e' T)?) (shrink_exp_typed T e)
  | _ ⇒ false
  end))).
```

5.5.2 Back to Soundness

To lift the shrinker to optional expressions, QuickChick provides the following function.

```
Definition lift_shrink {A}
                  (shr : A → list A) (m : option A)
              : list (option A) :=
  match m with
  | Some x ⇒ List.map Some (shr x)
  | _ ⇒ []
  end.
```

Armed with shrinking, we can pinpoint the bug in the EAnd branch of the evaluator.

```
Definition expression_soundness_exec' :=
  let num_vars := 4 in
  let top_level_size := 3 in
  forAll (gen_context num_vars) (fun Gamma ⇒
  forAll (gen_well_typed_state Gamma) (fun st ⇒
  forAll arbitrary (fun T ⇒
  forAllShrink (gen_exp_typed_sized 3 Gamma T)
                    (lift_shrink (shrink_exp_typed T))
                    (fun me ⇒
  match me with
  | Some e ⇒ negb (isNone (eval st e))
  | _ ⇒ true
  end)))).
```

5.6 Well-Typed Programs

Now we're ready to introduce TImp commands; they are just like the ones in Imp.

```
Inductive com : Type :=
  | CSkip : com
  | CAsgn : id → exp → com
  | CSeq : com → com → com
  | CIf : exp → com → com → com
  | CWhile : exp → com → com.

Notation "'SKIP'" :=
  CSkip.
Notation "x '::=' a" :=
  (CAsgn x a) (at level 60).
Notation "c1 ;;; c2" :=
  (CSeq c1 c2) (at level 80, right associativity).
Notation "'WHILE' b 'DO' c 'END'" :=
  (CWhile b c) (at level 80, right associativity).
Notation "'TEST' c1 'THEN' c2 'ELSE' c3 'FI'" :=
  (CIf c1 c2 c3) (at level 80, right associativity).

Derive Show for com.
```

(Of course, the derived Show instance is not going to use these notations!)

We can now define what it means for a command to be well typed for a given context. The interesting cases are TAsgn and TIf/TWhile. The first one, ensures that the type of the variable we are assigning to is the same as that of the expression. The latter, requires that the conditional is indeed a boolean expression.

```
Inductive well_typed_com : context → com → Prop :=
  | TSkip : ∀ Gamma, well_typed_com Gamma CSkip
  | TAsgn : ∀ Gamma x e T,
      bound_to Gamma x T →
      Gamma ||- e \IN T →
      well_typed_com Gamma (CAsgn x e)
  | TSeq : ∀ Gamma c1 c2,
      well_typed_com Gamma c1 → well_typed_com Gamma c2 →
      well_typed_com Gamma (CSeq c1 c2)
  | TIf : ∀ Gamma b c1 c2,
      Gamma ||- b \IN TBool →
      well_typed_com Gamma c1 → well_typed_com Gamma c2 →
      well_typed_com Gamma (CIf b c1 c2)
  | TWhile : ∀ Gamma b c,
      Gamma ||- b \IN TBool → well_typed_com Gamma c →
      well_typed_com Gamma (CWhile b c).
```

5.6.1 Decidable instance for well-typed.

A couple of lemmas and a custom tactic will help the decidability proof...

```
Lemma bind_deterministic Gamma x (T1 T2 : ty) :
  bound_to Gamma x T1 → bound_to Gamma x T2 →
  T1 = T2.
Proof.
  destruct T1; destruct T2; intros H1 H2; eauto;
    inversion H1; inversion H2; congruence.
Qed.

Lemma has_type_deterministic Gamma e (T1 T2 : ty) :
  has_type e Gamma T1 → has_type e Gamma T2 →
  T1 = T2.
Proof.
  destruct T1; destruct T2; intros H1 H2; eauto;
    inversion H1; inversion H2; subst; eauto; try congruence;
  inversion H7; subst;
    eapply bind_deterministic; eauto.
Qed.

Ltac solve_det :=
  match goal with
  | [ H1 : bound_to _ _ ?T1 ,
        H2 : bound_to _ _ ?T2 ⊢ _ ] ⇒
    assert (T1 = T2) by (eapply bind_deterministic; eauto)
  | [ H1 : has_type _ _ ?T1 ,
        H2 : has_type _ _ ?T2 ⊢ _ ] ⇒
    assert (T1 = T2) by (eapply bind_deterministic; eauto)
  end.
```

Now, here is a brute-force decision procedure for the typing relation (which amounts to a simple typechecker).

```
Instance dec_well_typed_com (Gamma : context) (c : com)
  : Dec (well_typed_com Gamma c).
Proof with eauto.
  constructor. unfold ssrbool.decidable.
  induction c; solve_sum.
  - destruct (dec_bound_to Gamma i TNat); destruct dec;
    destruct (dec_has_type e Gamma TNat); destruct dec;
    destruct (dec_bound_to Gamma i TBool); destruct dec;
    destruct (dec_has_type e Gamma TBool); destruct dec; solve_sum;
    try solve_det; try congruence;
    right; intro Contra; inversion Contra; subst; clear Contra;
    try solve_det; try congruence;
    destruct T; eauto.
  - destruct IHc1; destruct IHc2; subst; eauto; solve_sum.
  - destruct IHc1; destruct IHc2; subst; eauto; solve_sum.
    destruct (dec_has_type e Gamma TBool); destruct dec; solve_sum.
  - destruct IHc;
    destruct (dec_has_type e Gamma TBool); destruct dec; solve_sum.
Qed.
```

Exercise: 4 stars, standard (arbitrary_well_typed_com) Write a generator and a shrinker for well_typed programs given some context *Gamma*. Write some appropriate sanity checks and make sure they give expected results.

To complete the tour of testing for TImp, here is a (buggy??) evaluation function for commands given a state. To ensure termination, we've included a "fuel" parameter: if it gets to zero we return OutOfGas, signifying that we're not sure if evaluation would have succeeded, failed, or diverged if we'd gone on evaluating.

```
Inductive result :=
| Success : state → result
| Fail : result
| OutOfGas : result.

Fixpoint ceval (fuel : nat) (st : state) (c : com) : result :=
  match fuel with
  | O ⇒ OutOfGas
  | S fuel' ⇒
    match c with
    | SKIP ⇒
        Success st
    | x ::= e ⇒
        match eval st e with
        | Some v ⇒ Success (map_set st x v)
        | _ ⇒ Fail
        end
    | c1 ;;; c2 ⇒
        match ceval fuel' st c1 with
        | Success st' ⇒ ceval fuel' st' c2
        | _ ⇒ Fail
        end
    | TEST b THEN c1 ELSE c2 FI ⇒
      match eval st b with
      | Some (VBool b) ⇒
        ceval fuel' st (if b then c1 else c2)
      | _ ⇒ Fail
      end
    | WHILE b DO c END ⇒
      match eval st b with
      | Some (VBool b') ⇒
        if b'
          then ceval fuel' st (c ;;; WHILE b DO c END)
          else Success st
      | _ ⇒ Fail
      end
    end
  end.

Definition isFail r :=
  match r with
  | Fail ⇒ true
  | _ ⇒ false
  end.
```

Type soundness: well-typed commands never fail.

```
Conjecture well_typed_state_never_stuck :
  ∀ Gamma st, well_typed_state Gamma st →
  ∀ c, well_typed_com Gamma c →
  ∀ fuel, isFail (ceval fuel st c) = false.
```

Exercise: 4 stars, standard (well_typed_state_never_stuck) Write a checker for the above property, find any bugs, and fix them.

Exercise: 4 stars, standard (ty_eq_polymorphic) In the **has_type** relation we allowed equality checks between only arithmetic expressions. Introduce an additional typing rule that allows for equality checks between booleans.

| Ty_Eq : forall Gamma e1 e2, Gamma ||- e1 \IN TBool -> Gamma ||- e2 \IN TBool -> Gamma ||- EEq e1 e2 \IN TBool

Make sure you also update the evaluation relation to compare boolean values. Update the generators and shrinkers accordingly to find counterexamples to the buggy properties above.

HINT: When updating the shrinker, you will need to come up with the type of the equated expressions. The **Dec** instance of **has_type** will come in handy.

5.7 Automation (Revisited)

QuickChick is under very active development. Our vision is that it should automate most of the tedious parts of testing, while retaining full customizability.

We close this case study with a brief demo of some things it can do now.

Recall the **has_type_value** property and its corresponding generator:

Inductive has_type_value : value -> ty -> Prop := | TyVNat : forall n, has_type_value (VNat n) TNat | TyVBool : forall b, has_type_value (VBool b) TBool.

Definition gen_typed_value (T : ty) : G value := match T with | TNat => n <- arbitrary;; ret (VNat n) | TBool => b <- arbitrary;; ret (VBool b) end.

QuickChick includes a derivation mechanism that can *automatically* produce such generators – i.e., generators for data structures satisfying inductively defined properties!

Derive ArbitrarySizedSuchThat for (fun *v* ⇒ *has_type_value v T*).

===> GenSizedSuchThathas_type_value is defined.

Let's take a closer look at what is being generated (after doing some renaming and reformatting).

Print GenSizedSuchThathas_type_value.

===> GenSizedSuchThathas_type_value = fun T : ty => {| arbitrarySizeST := let fix aux_arb (size0 : nat) (T : ty) {struct size0} : G (option value) := match size0 with | 0 => backtrack (1, match *T* with | TBool ⇒ ret None | TNat ⇒ *n* ← arbitrary;; ret (Some (VNat *n*)) end) ;(1, match *T* with | TBool ⇒ *b* ← arbitrary;; ret (Some (VBool *b*))) | TNat ⇒ ret None end) | S _=> backtrack (1, match *T* with | TBool ⇒ ret None | TNat ⇒ *n* ← arbitrary;; ret (Some (VNat *n*)) end) ;(1, match *T* with | TBool ⇒ *b* ← arbitrary;; ret (Some (VBool *b*))) | TNat ⇒ ret None end) end in fun size0 : nat => aux_arb size0 T |}

: forall T : ty, GenSizedSuchThat value (fun v => has_type_value v T)

This is a rather more verbose version of the gen_typed_value generator, but the end result is actually exactly the same distribution!

5.7.1 (More) Typeclasses for Generation

QuickChick provides typeclasses for automating the generation for data satisfying predicates.

Module GenSTPlayground.

A variant that takes a size,...

Class **GenSizedSuchThat** (A : Type) (P : A → Prop) :=
  { arbitrarySizeST : **nat** → *G* (**option** A) }.

...an unsized variant,...

Class **GenSuchThat** (A : Type) (P : A → Prop) :=
  { arbitraryST : *G* (**option** A) }.

...convenient notation,...

Notation "'genST' x" := (@arbitraryST _ x _) (at level 70).

...and a coercion between the two:

Instance GenSuchThatOfBounded (A : Type) (P : A → Prop)

```
        (H : GenSizedSuchThat A P)
  : GenSuchThat A P :=
  { arbitraryST := sized arbitrarySizeST }.
End GenSTPlayground.
```

5.7.2 Using "SuchThat" Typeclasses

QuickChick can now (ab)use the typeclass resolution mechanism to perform a bit of black magic:

```
Conjecture conditional_prop_example :
  ∀ (x y : nat), x = y → x = y.
```

Notice the "0 discards": that means that quickchick is using generators that produce x and y such that $x = y$!

5.8 Acknowledgements

The first version of this material was developed in collaboration with Nicolas Koh.

Chapter 6

Library QC.QuickChickTool

6.1 QuickChickTool: The QuickChick Command-Line Tool

```
Set Warnings "-notation-overridden,-parsing".
Set Warnings "-extraction-opaque-accessed,-extraction".
From QuickChick Require Import QuickChick Tactics.
Import QcDefaultNotation. Open Scope qc_scope.
Require Import Arith List. Import ListNotations.
Set Bullet Behavior "Strict Subproofs".
```

6.2 Overview

In this chapter we will introduce the QuickChick command-line tool, which supports testing of larger-scale projects than the examples we have seen so far. The command-line tool offers several useful features:

- Batch processing, compilation and execution of tests
- Mutation testing
- Sectioning of tests and mutants

This chapter reads a bit differently than most SF chapters, as it deals with a command-line tool. The code that it discusses, a simple compiler from a high-level expression language to a low-level stack machine, can be found in the directory *stack-compiler*, broken up into two files: *Exp.v* containing the high-level expression languages and *Stack.v* containing the low-level stack machine and the compiler. The chapter's text will tell you what to type on the command line or where to look in the subdirectory as needed.

To get started, let's try the tool out and see it's output! Go to the *stack-compiler* subdirectory and run the following command:

```
quickChick -color -top Stack
```

The -color flag colors certain lines in the output for easier reading. The -*top* flag controls the namespace for the compilation and should be the same as the -*R* or -*Q* command in your _*CoqProject* file. Running this command should produce quite a bit of output in your terminal.

The output consists of four parts, delimited by (colored) headers such as

```
Testing base...
```

or

```
Testing mutant 2 (./Exp.v: line 20): Plus-copy-paste-error
```

Let's take a closer look at the first one.

```
Testing base...
make -f Makefile.coq
make[1]: Entering directory '/home/lemonidas/sfdev/qc/_qc_stack-compiler.tmp'
COQDEP VFILES
COQC Exp.v
COQC Stack.v
COQC QuickChickTop.v
make[1]: Leaving directory '/home/lemonidas/sfdev/qc/_qc_stack-compiler.tmp'
Checking Exp.optimize_correct_prop...
+++ Passed 10000 tests (0 discards)
Checking Stack.compiles_correctly...
+++ Passed 10000 tests (0 discards)
```

As we will see later in this chapter, the QuickChick command-line tool gathers QuickChick tests from the sources and runs them together in one single, efficient extraction. To do that, it copies all the files (here *Exp.v* and *Stack.v*) in a new subdirectory that is a "sibling" of the current one (that is both the directory where you ran *quickChick* and the new directory are subdirectories of the same parent). This new directory is always named _*qc*_<*DIRNAME*>.*tmp*. QuickChick also produces a new file *QuickChickTop.v* that contains all the tests that will be run and more extraction directives.

Following the output of the QuickChick command-line tool, all it does is compile everything in _*qc*_*stack-compiler.tmp*, using the *Makefile* of the original development as is and a _*CoqProject* modified to include the new *QuickChickTop.v* file. This compilation leads to extracting all necessary files in *separate* OCaml modules, which are in turn compiled using *ocamlbuild*, and then run. The separate extraction into distinct ocaml modules allows us to reuse compilation effort across different mutants as well as different calls to *quickChick*, as we can identify whether newly extracted modules are actually different and recompile them or not accordingly!

The rest of the 3 parts of the output are similar, with the main difference being that instead of running all the tests on the unaltered, base development, they run the same tests on *mutated* code. We will see exactly what mutation testing is later in this chapter.

If all is well, the last line should be a reassuring success report:

```
All tests produced the expected results
```

6.3 Arithmetic Expressions

The code in the *stack-compiler* subdirectory consists of two modules, *Exp* and *Stack*, each containing a number of definitions and properties. After some `Imports` at the top, the *Exp* module begins with a *section declaration*:

```
(*! Section arithmetic_expressions *)
```

We will explain what quickChick sections are and how to use them later in this chapter.

It then defines a little arithmetic language, consisting of natural literals, addition, subtraction and multiplication.

```
Inductive exp : Type :=
  | ANum : nat → exp
  | APlus : exp → exp → exp
  | AMinus : exp → exp → exp
  | AMult : exp → exp → exp.
```

Since **exp** is a simple datatype, QuickChick can derive a generator, a shrinker, and a printer automatically.

Derive (*Arbitrary*, `Show`) `for` *exp*.

The `eval` function evaluates an expression to a number.

```
Fixpoint eval (e : exp) : nat :=
  match e with
  | ANum n ⇒ n
  | APlus e1 e2 ⇒ (eval e1) + (eval e2)
  | AMinus e1 e2 ⇒ (eval e1) - (eval e2)
  | AMult e1 e2 ⇒ (eval e1) × (eval e2)
  end.
```

(The actual definition in the file *Exp.v* contains a few more annotations in comments, defining a *mutant*. We will discuss these annotations later in this chapter.

Now let's write a simple optimization: whenever we see an unnecessary operation (adding/subtracting 0) we optimize it away.

```
Fixpoint optimize (e : exp) : exp :=
  match e with
  | ANum n ⇒ ANum n
  | APlus e (ANum 0) ⇒ optimize e
  | APlus (ANum 0) e ⇒ optimize e
```

| APlus *e1* *e2* ⇒ APlus (optimize *e1*) (optimize *e2*)
| AMinus *e* (ANum 0) ⇒ optimize *e*
| AMinus *e1* *e2* ⇒ AMinus (optimize *e1*) (optimize *e2*)
| AMult *e1* *e2* ⇒ AMult (optimize *e1*) (optimize *e2*)
end.

Again, the actual definition in *Exp.v* contains again a few more annotations in comments (another section annotation and another mutant).

We can now write a simple correctness property for the optimizer, namely that evaluating an optimized expression yields the same number as evaluating the original one.

Definition optimize_correct_prop (*e* : **exp**) := eval (optimize *e*) = eval *e*?.

```
(*! QuickChick optimize_correct_prop. *)

QuickChecking optimize_correct_prop
+++ Passed 10000 tests (0 discards)
```

6.4 QuickChick Test Annotations

In earlier chapters, we have included QuickChick commands in comments, with an invitation to the reader to uncomment and execute them. This has been done to avoid executing each and every test when compiling the volume as a whole. If we were to leave the QuickChick commands uncommented, then for each test we would extract the entire volume up to that point, compile the extracted OCaml, execute the test (up to 10000 tests by default for successes), and report the outcome. While this process is often adequate for small developments, it quickly becomes intractable for large Coq files, multi-file developments, or large numbers of properties that need to be tested.

One main feature of the command line tool is to gather all QuickChick commands, perform a *single* extraction and compilation pass, and report the results for all tests. This is achieved with special QuickChick annotations.

Notice that this QuickChick comment, just like all QuickChick-specific annotations in the file, begin with an exclamation mark. *Comments that begin with an exclamation mark are special to the QuickChick command-line tool parser and signify a test, a section, or a mutant.*

The annotation above defines a test of the property optimize_correct_prop. For simplicity, each test annotation requires a *named* property, like optimize_correct_prop. That is, while inline one could successfully execute a command like the one below, the command-line tool requires a defined constant in the test annotation.

```
QuickChick (fun e => eval (optimize e) = eval e?).
```

6.5 Sections

When in the middle of a large development, it is useful to be able to concentrate your tests in the parts of the development you are actively changing. For example, if you are playing with the optimizer for your high-level language it is not ideal to spend time re-running the (successful) tests of the code generator. This is where QuickChick's sections come in.

Sections are contiguous blocks of code within modules, and are allowed to depend on earlier ones. They contain sets of tests (and later on mutants) that correspond to a single aspect of the development and that are meant to be run together.

Thare are two kinds of section declarations in QuickChick. The first section declaration in the *Exp* module simply defines the start of a new block that can be identified by the name "arithmetic_expressions".

```
(*! Section arithmetic_expressions *)
```

The second also includes an *extends* clause.

```
(*! Section optimizations *)(*! extends arithmetic_expressions *)
```

This signifies that this new block (until the end of the file, in this case, since there are no further section headers), also contains all tests and mutants from the *arithmetic_expressions* section as well.

To see sectioning in action, execute the following command from the *stack-compiler* directory:

```
quickChick -color -top Stack -s optimizations
```

```
Testing base...
make -f Makefile.coq
make[1]: Entering directory '/home/lemonidas/sfdev/qc/_qc_stack-compiler.tmp'
COQDEP VFILES
COQC Exp.v
COQC Stack.v
COQC QuickChickTop.v
make[1]: Leaving directory '/home/lemonidas/sfdev/qc/_qc_stack-compiler.tmp'
Checking Exp.optimize_correct_prop...
+++ Passed 10000 tests (0 discards)
... etc ...
```

In addition to the standard arguments (-**color**, -*top Stack*) we also specified that we only care about the *optimizations* section with the -*s* flag. Therefore this time, when testing the base development, only the single test in *optimizations* was executed.

6.6 Mutation Testing

The last major feature of the QuickChick command-line tool is *mutation testing.*

A question that naturally arises when random testing a software artifact is whether the testing is actually good. Does our property succeed for 10000 tests because everything is correct, or because we are not covering enough of the input space? How much testing is enough? Mutation testing can be used to answer such questions, by intentionally introducing bugs in either the artifacts or the properties we are testing and then checking that the tests can detect them.

Here is an excerpt of the `eval` function with a simple mutant annotation:

```
(*! *)
  | APlus e1 e2 => (eval e1) + (eval e2)
(*!! Plus-copy-paste-error *)
(*! | APlus e1 e2 => (eval e1) + (eval e1) *)
```

Let's break it down into its parts.

QuickChick mutants come in three parts. First, an annotation that signifies the beginning of a mutant. That is always the same:

```
(*! *)
```

This is followed by the correct code. In our example, by the evaluation of APlus e1 e2.

Afterwards, we can include an optional (but recommended) name for the mutant, which begins with two exclamation marks to help the parser.

```
(*!! Plus-copy-paste-error *)
```

Finally, we include mutations of the original code, each in a QuickChick single-exclamation-mark annotation. The code of the mutation is meant to be able to verbatim replace the original code. Here, a copy-paste error has been made to evaluate `e1` twice as both operands in an addition.

Similarly, in the optimize function we encounter the following mutant.

```
(*! *)
  | AMinus e (ANum 0) => optimize e
(*!! Minus-Reverse *)
(*!
  | AMinus (ANum 0) e => optimize e
*)
```

This bug allows the optimization of 0-e to e instead of e-0 to e.

To see these mutants in action, let's look at the rest of the output of the last *quickChick* command we ran:

```
quickChick -color -top Stack -s optimizations

Testing mutant 0 (./Exp.v: line 35): Minus-Reverse
make -f Makefile.coq
make[1]: Entering directory '/home/lemonidas/sfdev/qc/_qc_stack-compiler.tmp'
COQDEP VFILES
COQC Exp.v
COQC Stack.v
COQC QuickChickTop.v
make[1]: Leaving directory '/home/lemonidas/sfdev/qc/_qc_stack-compiler.tmp'
AMinus (ANum 0) (ANum 1)
Checking Exp.optimize_correct_prop...
*** Failed after 13 tests and 4 shrinks. (0 discards)

Testing mutant 1 (./Exp.v: line 20): Plus-copy-paste-error
make -f Makefile.coq
make[1]: Entering directory '/home/lemonidas/sfdev/qc/_qc_stack-compiler.tmp'
COQDEP VFILES
COQC Exp.v
COQC Stack.v
COQC QuickChickTop.v
make[1]: Leaving directory '/home/lemonidas/sfdev/qc/_qc_stack-compiler.tmp'
APlus (ANum 1) (ANum 0)
Checking Exp.optimize_correct_prop...
*** Failed after 5 tests and 3 shrinks. (0 discards)
All tests produced the expected results
```

After running all the tests for base (the unmutated artifact), the *quickChick* tool proceeds to run the single test in the *optimizations* section for each of the mutants it finds. Since the *optimizations* section *extends* the *arithmetic_expressions* section, the mutants from both sections will be included. As expected, the optimize property fails in both cases.

6.7 A Low-Level Stack Machine

The second module of our development is *Stack.v*, which describes a low-level stack machine for arithmetic expressions. It contains a single section, *stack_instructions* which extends *arithmetic_expressions* but not *optimizations*. This allows us to independently run tests for the compiler or the optimzer without worrying about extra tests or mutants.

We begin by defining a low-level stack machine instruction set, which closely corresponds to the high-level expression language. Inductive **sinstr** : Type :=
| SPush : nat → sinstr
| SPlus : sinstr

```
| SMinus : sinstr
| SMult : sinstr.
```

Then we describe how to execute the stack machine.

```
Fixpoint execute (stack : list nat) (prog : list sinstr) : list nat :=
  match (prog, stack) with
  | (nil, _ ) ⇒ stack
  | (SPush n::prog', _ ) ⇒ execute (n::stack) prog'
  | (SPlus::prog', m::n::stack') ⇒ execute ((m+n)::stack') prog'
  | (SMinus::prog', m::n::stack') ⇒ execute ((m-n)::stack') prog'
  | (SMult::prog', m::n::stack') ⇒ execute ((m×n)::stack') prog'
  | (_::prog', _ ) ⇒ execute stack prog'
  end.
```

Given the current *stack* (a list of natural numbers) and the *prog*ram to be executed, we will produce a resulting stack.

- If *prog* is empty, we return the current *stack*.
- To *Push* an integer, we cons it in the front of the list and execute the remainder of the program.
- To perform an arithmetic operation, we expect two integers at the top of the stack, operate, push the result, and execute the remainder of the program.
- Finally, if such a thing is not possible (i.e. we tried to perform a binary operation with less than 2 elements on the stack), we ignore the instruction and proceed to the rest.

Now we can compile expressions to their corresponding stack-instruction sequences.

```
Fixpoint compile (e : exp) : list sinstr :=
  match e with
  | ANum n ⇒ [SPush n]
  | APlus e1 e2 ⇒ compile e1 ++ compile e2 ++ [SPlus]
  | AMinus e1 e2 ⇒ compile e1 ++ compile e2 ++ [SMinus]
  | AMult e1 e2 ⇒ compile e1 ++ compile e2 ++ [SMult]
  end.
```

In the compilation above we have made a rookie compiler-writer mistake. (Can you spot it without running QuickChick?)

The property we would expect to hold is that executing the compiled instruction sequence of a given expression *e*, would result in a single element stack with `eval` *e* as its only element.

```
Definition compiles_correctly (e : exp) := (execute [] (compile e)) = [eval e]?.
```

```
===>
    QuickChecking compiles_correctly
    AMinus (ANum 0) (ANum 1)
    *** Failed after 3 tests and 2 shrinks. (0 discards)
```

The problem is that subtraction is not associative and we have compiled the two operands in the wrong order! We can now log that mutant in our development as shown in the *Stack* module.

```
Fixpoint compile (e : exp) : list sinstr :=
  match e with
  | ANum n => [SPush n]
(*! *)
| APlus  e1 e2 => compile e2 ++ compile e1 ++ [SPlus]
  | AMinus e1 e2 => compile e2 ++ compile e1 ++ [SMinus]
  | AMult  e1 e2 => compile e2 ++ compile e1 ++ [SMult]
(*!! Wrong associativity *)
(*!
  | APlus  e1 e2 => compile e1 ++ compile e2 ++ [SPlus]
  | AMinus e1 e2 => compile e1 ++ compile e2 ++ [SMinus]
  | AMult  e1 e2 => compile e1 ++ compile e2 ++ [SMult]
*)
  end.
```

We can now run a different command to test compiles_correctly, using -*s stack* to only check the *stack* section.

```
quickChick -color -top Stack -s stack

Testing base...
make -f Makefile.coq
make[1]: Entering directory '/home/lemonidas/sfdev/qc/_qc_stack-compiler.tmp'
COQDEP VFILES
COQC Exp.v
COQC Stack.v
COQC QuickChickTop.v
make[1]: Leaving directory '/home/lemonidas/sfdev/qc/_qc_stack-compiler.tmp'
Checking Stack.compiles_correctly...
+++ Passed 10000 tests (0 discards)

Testing mutant 0 (./Stack.v: line 33): Wrong associativity
make -f Makefile.coq
make[1]: Entering directory '/home/lemonidas/sfdev/qc/_qc_stack-compiler.tmp'
COQDEP VFILES
COQC Stack.v
COQC QuickChickTop.v
make[1]: Leaving directory '/home/lemonidas/sfdev/qc/_qc_stack-compiler.tmp'
AMinus (ANum 0) (ANum 1)
```

```
Checking Stack.compiles_correctly...
*** Failed after 5 tests and 6 shrinks. (0 discards)

Testing mutant 1 (./Exp.v: line 20): Plus-copy-paste-error
make -f Makefile.coq
make[1]: Entering directory '/home/lemonidas/sfdev/qc/_qc_stack-compiler.tmp'
COQDEP VFILES
COQC Exp.v
COQC Stack.v
COQC QuickChickTop.v
make[1]: Leaving directory '/home/lemonidas/sfdev/qc/_qc_stack-compiler.tmp'
APlus (ANum 0) (ANum 1)
Checking Stack.compiles_correctly...
*** Failed after 5 tests and 2 shrinks. (0 discards)
All tests produced the expected results
```

We can see that the main property succeeds, while the two mutants (the one in *arithmetic_expressions* that was included because of the extension and the one in *stack*) both fail.

6.8 QuickChick Command-Line Tool Flags

For more information on the tool's flags, look at the reference manual in QuickChickInterface.

Chapter 7

Library QC.QuickChickInterface

7.1 QuickChickInterface: QuickChick Reference Manual

From *QuickChick* Require Import QuickChick.
Require Import ZArith Strings.Ascii Strings.String.

From *ExtLib.Structures* Require Import Functor Applicative.

QuickChick provides a large collection of combinators and notations for writing property-based random tests. This file documents the entire public interface (the module type QuickChickSig).

Module Type QuickChickSig.

7.2 The Show Typeclass

Show typeclass allows the test case to be printed as a string.

Class Show (A : Type) : Type := { show : A -> string }.

Here are some Show instances for some basic types: Declare Instance showNat : **Show** **nat**.
Declare Instance showBool : **Show** **bool**.
Declare Instance showZ : **Show** **Z**.
Declare Instance showString : **Show** **string**.

Declare Instance showList :
  $\forall$ {A : Type} ‘{*Show* A}, **Show** (list A).
Declare Instance showPair :
  $\forall$ {A B : Type} ‘{*Show* A} ‘{*Show* B}, **Show** ($A \times B$).
Declare Instance showOpt :
  $\forall$ {A : Type} ‘{*Show* A}, **Show** (option A).
Declare Instance showEx :
  $\forall$ {A} ‘{*Show* A} P, **Show** ({x : A | P x}).

When defining `Show` instance for your own datatypes, you sometimes need to start a new line for better printing. nl is a shorthand for it. `Definition` nl : **string** := String (ascii_of_nat 10) EmptyString.

7.3 Generators

7.3.1 Fundamental Types

A *RandomSeed* represents a particular starting point in a pseudo-random sequence. `Parameter` *RandomSeed* : `Type`.

G A is the type of random generators for type A. `Parameter` *G* : `Type` → `Type`.

Run a generator with a size parameter (a natural number denoting the maximum depth of the generated A) and a random seed. `Parameter` *run* : ∀ {A : `Type`}, *G* A → **nat** → *RandomSeed* → A.

The semantics of a generator is its set of possible outcomes. `Parameter` *semGen* : ∀ {A : `Type`} (g : *G* A), set A.
`Parameter` *semGenSize* : ∀ {A : `Type`} (g : *G* A) ($size$: **nat**), set A.

7.3.2 Structural Combinators

Generators are also instances of several generic typeclasses. Many handy generator combinators can be found in the **Monad**, Functor, Applicative, *Foldable*, and *Traversable* modules in the *ExtLib.Structures* library from *coq-ext-lib*.

`Declare Instance` Monad_G : **Monad** *G*.
`Declare Instance` Functor_G : **Functor** *G*.
`Declare Instance` Applicative_G : **Applicative** *G*.

A variant of monadic bind where the continuation also takes a *proof* that the value received is within the set of outcomes of the first generator. `Parameter` *bindGen'* : ∀ {A B : `Type`} (g : *G* A),
(∀ (a : A), (a \in *semGen* g) → *G* B) → *G* B.

A variant of bind for the (*G* (**option** –)) monad. Useful for chaining generators that can fail / backtrack. `Parameter` *bindGenOpt* : ∀ {A B : `Type`},
G (**option** A) → (A → *G* (**option** B)) → *G* (**option** B).

7.3.3 Basic Generator Combinators

The *listOf* and *vectorOf* combinators construct generators for **list** A, provided a generator g for type A: *listOf* g yields an arbitrary-sized list (which might be empty), while *vectorOf* n g yields a list of fixed size n. `Parameter` *listOf* : ∀ {A : `Type`}, *G* A → *G* (**list** A).
`Parameter` *vectorOf* : ∀ {A : `Type`}, **nat** → *G* A → *G* (**list** A).

*elems_* a l constructs a generator from a list l and a default element a. If l is non-empty, the generator picks an element from l uniformly; otherwise it always yields a. Parameter *elems_* : ∀ {A : Type}, A → list A → G A.

Similar to *elems_*, instead of choosing from a list of As, *oneOf_* g l returns g if l is empty; otherwise it uniformly picks a generator for A in l. Parameter *oneOf_* : ∀ {A : Type}, G A → list (G A) → G A.

We can also choose generators with distributions other than the uniform one. *freq_* g l returns g if l is empty; otherwise it chooses a generator from l, where the first field indicates the chance that the second field is chosen. For example, *freq_* z [(2, x); (3, y)] has 40% probability of choosing x and 60% probability of choosing y. Parameter *freq_* :
  ∀ {A : Type}, G A → list (nat × G A) → G A.

Try all generators until one returns a Some value or all failed once with None. The generators are picked at random according to their weights (like *frequency*), and each one is run at most once. Parameter *backtrack* :
  ∀ {A : Type}, list (nat × G (option A)) → G (option A).

Internally, the G monad hides a size parameter that can be accessed by generators. The *sized* combinator provides such access. The *resize* combinator sets it. Parameter *sized* : ∀ {A: Type}, (nat → G A) → G A.
Parameter *resize* : ∀ {A: Type}, nat → G A → G A.

Generate-and-test approach to generate data with preconditions. Parameter *suchThatMaybe* :
  ∀ {A : Type}, G A → (A → bool) → G (option A).
Parameter *suchThatMaybeOpt* :
  ∀ {A : Type}, G (option A) → (A → bool) → G (option A).

The *elems_*, *oneOf_*, and *freq_* combinators all take default values; these are only used if their list arguments are empty, which should not normally happen. The *QcDefaultNotation* sub-module exposes notation (without the underscores) to hide this default.

Module QcDefaultNotation.

elems is a shorthand for *elems_* without a default argument. Notation " 'elems' [x] " :=
    (*elems_* x (cons x nil)) : *qc_scope*.
  Notation " 'elems' [x ; y] " :=
    (*elems_* x (cons x (cons y nil))) : *qc_scope*.
  Notation " 'elems' [x ; y ; .. ; z] " :=
    (*elems_* x (cons x (cons y .. (cons z nil) ..))) : *qc_scope*.
  Notation " 'elems' (x ;; l) " :=
    (*elems_* x (cons x l)) (at level 1, no associativity) : *qc_scope*.

oneOf is a shorthand for *oneOf_* without a default argument. Notation " 'oneOf' [x] " :=
    (*oneOf_* x (cons x nil)) : *qc_scope*.

Notation " 'oneOf' [x ; y] " :=
(*oneOf_* x (cons x (cons y nil))) : *qc_scope*.
Notation " 'oneOf' [x ; y ; .. ; z] " :=
(*oneOf_* x (cons x (cons y .. (cons z nil) ..))) : *qc_scope*.
Notation " 'oneOf' (x ;; l) " :=
(*oneOf_* x (cons x l)) (at level 1, no associativity) : *qc_scope*.

freq is a shorthand for *freq_* without a default argument. Notation " 'freq' [x] " :=
(*freq_* x (cons x nil)) : *qc_scope*.
Notation " 'freq' [(n , x) ; y] " :=
(*freq_* x (cons (n, x) (cons y nil))) : *qc_scope*.
Notation " 'freq' [(n , x) ; y ; .. ; z] " :=
(*freq_* x (cons (n, x) (cons y .. (cons z nil) ..))) : *qc_scope*.
Notation " 'freq' ((n , x) ;; l) " :=
(*freq_* x (cons (n, x) l)) (at level 1, no associativity) : *qc_scope*.

End QcDefaultNotation.

The original version of QuickChick used *elements*, *oneof* and *frequency* as the default-argument versions of the corresponding combinators. These have since been deprecated in favor of a more consistent naming scheme.

7.3.4 Choosing from Intervals

The combinators above allow us to generate elements by enumeration and lifting. However, for numeric data types, we sometimes hope to choose from an interval without writing down all the possible values.

Such intervals can be defined on ordered data types, instances of **OrdType**, whose ordering *leq* satisfies reflexive, transitive, and antisymmetric predicates.

Existing Class *OrdType*.

Declare Instance OrdBool : **OrdType** **bool**.
Declare Instance OrdNat : **OrdType** **nat**.
Declare Instance OrdZ : **OrdType** **Z**.

We also expect the random function to be able to pick every element in any given interval.

Existing Class *ChoosableFromInterval*.

QuickChick has provided some instances for ordered data types that are choosable from intervals, including **bool**, **nat**, and **Z**. Declare Instance ChooseBool : **ChoosableFromInterval** **bool**.
Declare Instance ChooseNat : **ChoosableFromInterval** **nat**.
Declare Instance ChooseZ : **ChoosableFromInterval** **Z**.

choose l r generates a value between l and r, inclusive the two extremes. It causes a runtime error if r $< l$. Parameter *choose* :
$\forall$ {A : Type} '{*ChoosableFromInterval* A}, ($A \times A$) $\rightarrow$ *G* A.

7.3.5 The Gen and GenSized Typeclasses

GenSized and **Gen** are typeclasses whose instances can be generated randomly. More specifically, **GenSized** depends on a generator for any given natural number that indicate the size of output.

Class GenSized (A : Type) := { arbitrarySized : nat -> G A }. Class Gen (A : Type) := { arbitrary : G A }.

Given an instance of **GenSized**, we can convert it to **Gen** automatically, using *sized* function. `Declare Instance` GenOfGenSized $\{A\}$ ‘$\{GenSized\ A\}$: **Gen** A.

Here are some basic instances for generators: `Declare Instance` genBoolSized : **GenSized** bool.
`Declare Instance` genNatSized : **GenSized** nat.
`Declare Instance` genZSized : **GenSized** Z.

`Declare Instance` genListSized :
  ∀ $\{A$: Type$\}$ ‘$\{GenSized\ A\}$, **GenSized** (list A).
`Declare Instance` genList :
  ∀ $\{A$: Type$\}$ ‘$\{Gen\ A\}$, **Gen** (list A).
`Declare Instance` genOption :
  ∀ $\{A$: Type$\}$ ‘$\{Gen\ A\}$, **Gen** (option A).
`Declare Instance` genPairSized :
  ∀ $\{A\ B$: Type$\}$ ‘$\{GenSized\ A\}$ ‘$\{GenSized\ B\}$, **GenSized** $(A \times B)$.
`Declare Instance` genPair :
  ∀ $\{A\ B$: Type$\}$ ‘$\{Gen\ A\}$ ‘$\{Gen\ B\}$, **Gen** $(A \times B)$.

7.3.6 Generators for Data Satisfying Inductive Predicates

Just as QuickChick provides the **GenSized** and **Gen** typeclasses for generators of type A, it provides constrained variants for generators of type A such that $P : A \rightarrow$ `Prop` holds of all generated values. Since it is not guaranteed that any such A exist, these generators are partial.

Class GenSizedSuchThat (A : Type) (P : A -> Prop) := { arbitrarySizeST : nat -> G (option A) }.

Class GenSuchThat (A : Type) (P : A -> Prop) := { arbitraryST : G (option A) }.

So, for example, if you have a typing relation **has_type** : **exp** → **type** → `Prop` for some language, you could, given some type T as input, write (or derive as we will see later on) an instance of **GenSizedSuchThat** (`fun` e ⇒ **has_type** e T), that produces an expression of with type T.

Calling arbitraryST through such an instance would require making an explicit application to @arbitraryST as follows:

@arbitraryST _(fun e => has_type e T) _]] where the first placeholder is the type of expressions **exp** and the second placeholder is the actual instance to be inferred.

To avoid this, QuickChick also provides convenient notation to call by providing only the predicate P that constraints the generation. The typeclass constraint is inferred.

Notation "'genST' x" := (@arbitraryST _ x _) (at level 70).

7.4 Shrinking

7.4.1 The Shrink Typeclass

Shrink is a typeclass whose instances have an operation for shrinking larger elements to smaller ones, allowing QuickChick to search for a minimal counter example when errors occur.

Class Shrink (A : Type) := { shrink : A -> list A }.

Default shrinkers for some basic datatypes: Declare Instance shrinkBool : **Shrink** bool.

Declare Instance shrinkNat : **Shrink** nat.
Declare Instance shrinkZ : **Shrink** Z.

Declare Instance shrinkList $\{A : \text{Type}\}$ `$\{\mathit{Shrink}\ A\}$: **Shrink** (list A).
Declare Instance shrinkPair $\{A\ B\}$ `$\{\mathit{Shrink}\ A\}$ `$\{\mathit{Shrink}\ B\}$: **Shrink** ($A \times B$).
Declare Instance shrinkOption $\{A : \text{Type}\}$ `$\{\mathit{Shrink}\ A\}$: **Shrink** (option A).

7.4.2 The Arbitrary Typeclass

The **Arbitrary** typeclass combines generation and shrinking.

Class Arbitrary (A : Type) `{Gen A} `{Shrink A}.

7.4.3 The Generator Typeclass Hierarchy

GenSized | | Gen Shrink \ / \ / Arbitrary

If a type has a **Gen** and a **Shrink** instance, it automatically gets an **Arbitrary** one.
Declare Instance ArbitraryOfGenShrink :
$\forall\ \{A\}$ `$\{\mathit{Gen}\ A\}$ `$\{\mathit{Shrink}\ A\}$, **Arbitrary** A.

7.5 Checkers

7.5.1 Basic Definitions

Checker is the opaque type of QuickChick properties. Parameter *Checker* : Type.

The **Checkable** class indicates we can check a type A.

Class Checkable (A : Type) : Type := { checker : A -> Checker }.

Boolean checkers always pass or always fail. Declare Instance testBool : **Checkable** bool.

The unit checker is always discarded (that is, it represents a useless test). It is used, for example, in the implementation of the "implication *Checker*" combinator ==>. `Declare Instance` testUnit : **Checkable** **unit**.

Given a generator for showable As, construct a *Checker*. `Parameter` *forAll* :
∀ {A $prop$: `Type`} ‘{$Checkable\ prop$} ‘{$Show\ A$}
  (gen : *G* A) (pf : $A \to prop$), *Checker*.

A variant of *forAll* that provides evidence that the generated values are members of the semantics of the generator. (Such evidence can be useful when constructing dependently typed data, such as bounded integers.) `Parameter` *forAllProof* :
∀ {A $prop$: `Type`} ‘{$Checkable\ prop$} ‘{$Show\ A$}
  (gen : *G* A) (pf : ∀ (x : A), *semGen* gen $x \to prop$), *Checker*.

Given a generator and a shrinker for showable As, construct a *Checker*. `Parameter` *forAllShrink* :
∀ {A $prop$: `Type`} ‘{$Checkable\ prop$} ‘{$Show\ A$}
  (gen : *G* A) ($shrinker$: $A \to$ **list** A) (pf : $A \to prop$), *Checker*.

Lift (`Show`, **Gen**, **Shrink**) instances for A to a *Checker* for functions A -> prop. This is what makes it possible to write (for some example property foo := `fun` $x \Rightarrow x$ >? 0, say) QuickChick foo instead of QuickChick (*forAllShrink* arbitrary shrink foo). `Declare Instance` testFun :
∀ {A $prop$: `Type`} ‘{$Show\ A$} ‘{$Arbitrary\ A$} ‘{$Checkable\ prop$},
  Checkable ($A \to prop$).

Lift products similarly. `Declare Instance` testProd :
∀ {A : `Type`} {$prop$: $A \to$ `Type`} ‘{$Show\ A$} ‘{$Arbitrary\ A$}
  ‘{∀ x : A, $Checkable\ (prop\ x)$},
  Checkable (∀ (x : A), $prop\ x$).

Lift polymorphic functions by instantiating to 'nat'. :-) `Declare Instance` testPolyFun :
∀ {$prop$: `Type` → `Type`} ‘{$Checkable\ (prop\ nat)$},
  Checkable (∀ T, $prop\ T$).

7.5.2 Checker Combinators

Print a specific string if the property fails. `Parameter` *whenFail* :
∀ {$prop$: `Type`} ‘{$Checkable\ prop$} (str : **string**), $prop \to$ *Checker*.

Record an expectation that a property should fail, i.e. the property will fail if all the tests succeed. `Parameter` *expectFailure* :
∀ {$prop$: `Type`} ‘{$Checkable\ prop$} (p: $prop$), *Checker*.

Collect statistics across all tests. `Parameter` *collect* :
∀ {A $prop$: `Type`} ‘{$Show\ A$} ‘{$Checkable\ prop$} (x : A),
  $prop \to$ *Checker*.

Set the reason for failure. Will only count shrinks as valid if they preserve the tag.
`Parameter` *tag* :
  ∀ {*prop* : `Type`} ‘{*Checkable prop*} (*t* : **string**), *prop* → *Checker*.

Form the conjunction / disjunction of a list of checkers. `Parameter` *conjoin* : ∀ (*l* : **list** *Checker*), *Checker*.
`Parameter` *disjoin* : ∀ (*l* : **list** *Checker*), *Checker*.

Define a checker for a conditional property. Invalid generated inputs (ones for which the antecedent fails) are discarded. `Parameter` *implication* :
  ∀ {*prop* : `Type`} ‘{*Checkable prop*} (*b* : **bool**) (*p* : *prop*), *Checker*.

Notation for implication. Clashes with many other notations in other libraries, so it lives in its own module. Note that this includes the notations for the generator combinators above to avoid needing to import two modules.

```
Module QcNotation.
  Export QcDefaultNotation.

  Notation "x ==> y" :=
    (implication x y) (at level 55, right associativity)
     : Checker_scope.
End QcNotation.
```

7.6 Decidability

7.6.1 The Dec Typeclass

Decidability typeclass using ssreflect’s ’decidable’.
  Class Dec (P : Prop) : Type := { dec : decidable P }.
  Decidable properties are Checkable. `Declare Instance` testDec {*P*} ‘{*H* : **Dec** *P*} : **Checkable** *P*.

Logic Combinator instances. `Declare Instance` Dec_neg {*P*} {*H* : **Dec** *P*} : **Dec** (¬ *P*).
`Declare Instance` Dec_conj {*P Q*} {*H* : **Dec** *P*} {*I* : **Dec** *Q*} : **Dec** (*P* ∧ *Q*).
`Declare Instance` Dec_disj {*P Q*} {*H* : **Dec** *P*} {*I* : **Dec** *Q*} : **Dec** (*P* ∨ *Q*).

A convenient notation for coercing a decidable proposition to a **bool**. `Notation` "P ’?’"

```
:= (match (@dec P _) with
                              | left _ ⇒ true
                              | right _ ⇒ false
                              end) (at level 100).
```

7.6.2 The Dec_Eq Typeclass

Class Dec_Eq (A : Type) := { dec_eq : forall (x y : A), decidable (x = y) }.

Automation and conversions for Dec. `Declare Instance` Eq__Dec $\{A\}$ ‘$\{H$: **Dec_Eq** $A\}$ $(x\ y : A)$: **Dec** $(x = y)$.

Since deciding equalities is a very common requirement in testing, QuickChick provides a tactic that can define instances of the form **Dec** $(x = y)$.

Ltac dec_eq.

QuickChick also lifts common decidable instances to the **Dec** typeclass. `Declare Instance` Dec_eq_bool $(x\ y$: **bool**$)$: **Dec** $(x = y)$.
`Declare Instance` Dec_eq_nat $(m\ n$: **nat**$)$: **Dec** $(m = n)$.
`Declare Instance` Dec_eq_opt $(A$: Type$)$ $(m\ n$: **option** $A)$
‘$\{\_ : \forall\ (x\ y : A)$, **Dec** $(x = y)\}$: **Dec** $(m = n)$.
`Declare Instance` Dec_eq_prod $(A\ B$: Type$)$ $(m\ n : A \times B)$
‘$\{\_ : \forall\ (x\ y : A)$, **Dec** $(x = y)\}$
‘$\{\_ : \forall\ (x\ y : B)$, **Dec** $(x = y)\}$
: **Dec** $(m = n)$.
`Declare Instance` Dec_eq_list $(A$: Type$)$ $(m\ n$: **list** $A)$
‘$\{\_ : \forall\ (x\ y : A)$, **Dec** $(x = y)\}$: **Dec** $(m = n)$.

`Declare Instance` Dec_ascii $(m\ n$: **Ascii.ascii**$)$: **Dec** $(m = n)$.
`Declare Instance` Dec_string $(m\ n$: **string**$)$: **Dec** $(m = n)$.

7.7 Automatic Instance Derivation

QuickChick allows the automatic derivation of typeclass instances for simple types:

Derive <class> for T.

Here <*class*> must be one of **GenSized**, **Shrink**, **Arbitrary**, or `Show`, and T must be an inductive defined datatype (think Haskell/OCaml).

To derive multiple classes at once, write:

Derive (<class>,...,<class>) for T.

QuickChick also allows for the automatic derivation of generators satisfying preconditions in the form of inductive relations:

Derive ArbitrarySizedSuchThat for (fun x => P x1 ... x .... xn).

<P> must be an inductively defined relation. <x> is the function to be generated. <x1...xn> are (implicitly universally quantified) variable names.

QuickChick also allows automatic derivations of proofs of correctness of its derived generators! For more, look at:

- A paper on deriving QuickChick generators for a large class of inductive relations. {https://lemonidas.github.io/pdf/GeneratingGoodGenerators.pdf}
- Leo’s PhD dissertation. {https://lemonidas.github.io/pdf/Leo-PhD-Thesis.pdf}
- examples/DependentTest.v

7.8 Top-level Commands and Settings

QuickChick provides a series of toplevel commands to sample generators, test properties, and derive useful typeclass instances.

The *Sample* command samples a generator. The argument g needs to have type $\mathsf{G}\ A$ for some showable type A.

Sample g.

The main testing command, QuickChick, runs a test. The argument *prop* must belong to a type that is an instance of **Checkable**.

QuickChick prop.

QuickChick uses arguments to customize execution.

```
Record Args :=
  MkArgs
    {
      replay : option (RandomSeed × nat);
      maxSuccess : nat;
      maxDiscard : nat;
      maxShrinks : nat;
      maxSize : nat;
      chatty : bool
    }.
```

Instead of record updates, you should overwrite extraction constants.

Extract Constant defNumTests => "10000". Extract Constant defNumDiscards => "(2 * defNumTests)". Extract Constant defNumShrinks => "1000". Extract Constant defSize => "7".

7.9 The *quickChick* Command-Line Tool

QuickChick comes with a command-line tool that supports:

- Batch processing, compilation and execution of tests

- Mutation testing
- Sectioning of tests and mutants

Comments that begin with an exclamation mark are special to the QuickChick command-line tool parser and signify a test, a section, or a mutant.

7.9.1 Test Annotations

A test annotation is just a QuickChick command wrapped inside a comment with an exclamation mark.

Only tests that are annotated this way will be processed. Only property names are allowed.

7.9.2 Mutant Annotations

A mutant annotation consists of 4 components. First an anottation that signifies the beginning of the mutant . That is followed by the actual code. Then, we can include an optional annotation (in a comment with double exclamation marks) that corresponds to the mutant names. Finally, we can add a list of mutations inside normal annotated comments. Each mutant should be able to be syntactically substituted in for the normal code.

Normal code ... etc ...

7.9.3 Section Annotations

To organize larger developments better, we can group together different tests and mutants in sections. A section annotation is a single annotation that defines the beginning of the section (which lasts until the next section or the end of the file).

Optionally, one can include an extends clause

This signifies that the section being defined also contains all tests and mutants from *other-section-name*.

7.9.4 Command-Line Tool Flags

The QuickChick command line tool can be passed the following options:

- *-s <section>*: Specify which sections properties and mutants to test
- *-v*: Verbose mode for debugging
- *-failfast*: Stop as soon as a problem is detected
- **-color**: Use colors on an ANSI-compatible terminal

- -*cmd* <*command*>: What compile command is used to compile the current directory if it is not *make*
- -*top* <*name*>: Specify the name of the top-level logical module. That should be the same as the -*Q* or -*R* directive in *_CoqProject* or *Top* which is the default
- -*ocamlbuild* <*args*>: Any arguments necessary to pass to ocamlbuild when compiling the extracted code (e.g. linked libraries)
- -*nobase*: Pass this option to not test the base mutant
- -*m* <*number*>: Pass this to only test a mutant with a specific id number
- -*tag* <*name*>: Pass this to only test a mutant with a specific tag
- -*include* <*name*>: Specify a to include in the compilation
- -*exclude* <*names*>: Specify files to be excluded from compilation. Must be the last argument passed.

7.10 Deprecated Features

The following features are retained for backward compatibility, but their use is deprecated.

Use the monad notations from *coq-ext-lib* instead of the *QcDoNotation* sub-module:

```
Module QcDoNotation.
  Notation "'do!' X <- A ; B" :=
    (bindGen A (fun X ⇒ B))
    (at level 200, X ident, A at level 100, B at level 200).
  Notation "'do\'' X <- A ; B" :=
    (bindGen' A (fun X H ⇒ B))
    (at level 200, X ident, A at level 100, B at level 200).
  Notation "'doM!' X <- A ; B" :=
    (bindGenOpt A (fun X ⇒ B))
    (at level 200, X ident, A at level 100, B at level 200).
End QcDoNotation.

End QuickChickSig.
```

Chapter 8

Library QC.Postscript

8.1 Postscript

8.2 Future Directions

We have lots of plans for future directions:

- Automatic derivation of generators and shrinkers for data satisfying Inductive relations
- Vellum2 testing
- DeepSpec Web Server
- Testing-only variant of *Software Foundations*?

8.3 Recommended Reading

The material presented in this short course serves as an introduction to property based random testing using QuickChick. For the interested reader, we provide a few more references for additional reading:

- The original QuickCheck paper by Koen Claessen and John Hughes from ICFP 2000. {https://www.cs.tufts.edu/~nr/cs257/archive/john-hughes/quick.pdf}
- The original QuickChick paper that focuses on a framework for proving the correctness of QuickChick generators. {https://hal.inria.fr/hal-01162898/document}
- A case study that uses QuickCheck to test non-interference for information-flow-control abstract machines. {https://arxiv.org/abs/1409.0393v2}

- Code for that case study exists under the QuickChick organization of github ({https://github.com/Qu
for both Haskell ("Testing Noninterference") and Coq ("IFC").
- A paper on deriving QuickChick generators for a large class of inductive relations. {https://lemonidas.github.io/pdf/GeneratingGoodGenerators.pdf}
- Leo's PhD dissertation. {https://lemonidas.github.io/pdf/Leo-PhD-Thesis.pdf}

Chapter 9

Library QC.Bib

9.1 Bib: Bibliography

Wadler and Blott 1989 Philip Wadler, Stephen Blott, *How to Make ad-hoc Polymorphism Less ad-hoc*. POPL 1989. {https://dl.acm.org/citation.cfm?doid=75277.75283}