SOFTWARE FOUNDATIONS

VOLUME 4: QUICKCHICK: PROPERTY-BASED TESTING IN COQ

TABLE OF CONTENTS INDEX

TIMP

CASE STUDY: A TYPED IMPERATIVE LANGUAGE

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.

Identifiers, Types and Contexts

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.

```
(* BCP: Print them as "A", "B", etc.  Or maybe "X1", "X2", etc. *)
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 (genId)

Write a `Gen` instance for `id` using the `elems_` combinator and `get_fresh_ids`.

```
(* FILL IN HERE *)
```

☐

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 := [] }.
```

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.
(* ==>
    GenSizedty is defined
    Shrinkty is defined
    Showty is defined
*)

Check GenSizedty.
(* ==> GenSizedty : GenSized ty *)
Check Shrinkty.
(* ==> Shrinkty  : Shrink ty   *)
Check Showty.
(* ==> Showty : Show ty     *)
```

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.
```

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.
```

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).
+
```

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).
```

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.
```

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 : ∀ (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).
+
```

Exercise: 3 stars (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?

```
(* FILL IN HERE *)
```

☐

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 `options`, 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.

```
Inductive has_type_1 : context → exp → ty → Prop :=
  | Ty_Var1 : ∀ x T Gamma,
      bound_to Gamma x T → has_type_1 Gamma (EVar x) T.
```

To generate `e` such that `has_type_1 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_Var1` (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 a typing relation `has_type_2`, extending `has_type_1` 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.

```
Inductive has_type_2 : context → exp → ty → Prop :=
| Ty_Var2 : ∀ x T Gamma,
    bound_to Gamma x T → has_type_2 Gamma (EVar x) T
| Ty_Num2 : ∀ Gamma n,
    has_type_2 Gamma (ENum n) TNat
```

```
  | Ty_True2 : ∀ Gamma, has_type_2 Gamma ETrue TBool
  | Ty_False2 : ∀ Gamma, has_type_2 Gamma EFalse TBool.
```

We can already generate values satisfying `Ty_Var2` 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:

```
Inductive has_type_3 : context → exp → ty → Prop :=
 | Ty_Var3 : ∀ x T Gamma,
     bound_to Gamma x T → has_type_3 Gamma (EVar x) T
 | Ty_Plus3 : ∀ Gamma e1 e2,
    has_type_3 Gamma e1 TNat → has_type_3 Gamma e2 TNat →
    has_type_3 Gamma (EPlus e1 e2) TNat.
```

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))).

(* QuickChick gen_typed_has_type. *)
```

Values and States

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 `values` 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).
+

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

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).
+

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

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
(* Let's include a silly bug here! *)
      | Some (VBool b), Some (VNat n₂) ⇒ Some (VBool (negb b))
(*    | Some (VBool b₁), Some (VBool b₂) => Some (VBool (andb b₁ b₂)) *)
      | _, _ ⇒ 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)))).

(* QuickChick expression_soundness_exec. *)
```

```
===>
  QuickChecking expression_soundness_exec
  [(1,TNat), (2,TNat), (3,TBool), (4,TNat)]
  [(1,VNat 0), (2,VNat 0), (3,VBool true), (4,VNat 0)]
  TBool
  Some EAnd (EAnd (EEq (EVar 4) (EVar 1)) (EEq (ENum 0) (EVar 4))) EFalse
  *** Failed after 8 tests and 0 shrinks. (0 discards)
```

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

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)))).

(* QuickChick expression_soundness_exec_firstshrink. *)
```

```
===>
  QuickChecking expression_soundness_exec_firsttry
  [(1,TBool), (2,TNat), (3,TBool), (4,TBool)]
  [(1,VBool false), (2,VNat 0), (3,VBool true), (4,VBool false)]
  TBool
  Some EAnd (ENum 0) ETrue
  *** Failed after 28 tests and 7 shrinks. (0 discards)
```

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` e_1 e_2:

- We could try (recursively) shrinking e_1 or e_2 preserving their `TNat` type.
- We could try to shrink directly to e_1 or e_2 since their type is the same as `EPlus` e_1 e_2.

On the other hand, consider `EEq` e_1 e_2:

- Again, we could recursively shrink e_1 or e_2.
- But we can't shrink *to* e_1 or e_2 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))).

(* QuickChick shrink_typed_has_type. *)
```

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)))).

(* QuickChick expression_soundness_exec'. *)


   ===>
      QuickChecking expression_soundness_exec'
      [(1,TNat), (2,TNat), (3,TNat), (4,TBool)]
```

```
[(1,VNat 0), (2,VNat 0), (3,VNat 0), (4,VBool false)]
TBool
Some EAnd ETrue ETrue
*** Failed after 8 tests and 1 shrinks. (0 discards)
```

Well-Typed Programs

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

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

Notation "'SKIP'" :=
  CSkip.
Notation "x '::=' a" :=
  (CAss 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 "'IFB' 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 `TAss` 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
  | TAss : ∀ Gamma x e T,
      bound_to Gamma x T →
      Gamma ||- e \IN T →
      well_typed_com Gamma (CAss 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).
```

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.
+

Lemma has_type_deterministic Gamma e (T1 T2 : ty) :
  has_type e Gamma T1 → has_type e Gamma T2 →
  T1 = T2.
+

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).
+
```

Exercise: 4 stars (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.

```
(* FILL IN HERE *)
```

☐

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
      | IFB 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 (well_typed_state_never_stuck)

Write a checker for the above property, find any bugs, and fix them.

```
(* FILL IN HERE *)
```

Exercise: 4 stars (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 : ∀ 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.

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  : ∀ n, has_type_value (VNat  n) TNat
  | TyVBool : ∀ 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!

(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.
```

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.

(* QuickChick conditional_prop_example. *)
```

```
==>
   QuickChecking conditional_prop_example
   +++ Passed 10000 tests (0 discards)
```

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

Acknowledgements

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