Implementing the Representation Invariant

When implementing a complex abstract data type, it is often helpful to write an internal function that can be used to check that the rep invariant holds of a given data item. By convention we will call this function `rep_ok`. If the module accepts values of the abstract type that are created outside the module, say by exposing the implementation of the type in the signature, then `rep_ok` should be applied to these to ensure the representation invariant is satisfied. In addition, if the implementation creates any new values of the abstract type, `rep_ok` can be applied to them as a sanity check. With this approach, bugs are caught early, and a bug in one function is less likely to create the appearance of a bug in another.

A convenient way to write `rep_ok` is to make it an identity function that just returns the input value if the rep invariant holds and raises an exception if it fails.

```
(* Checks whether x satisfies the representation invariant. *)
let rep_ok (x : int list) : int list = ...
```

Here is an implementation of `Set` that uses the same data representation as `ListSetNoDups`, but includes copious `rep_ok` checks. Note that `rep_ok` is applied to all input sets and to any set that is ever created. This ensures that if a bad set representation is created, it will be detected immediately. In case we somehow missed a check on creation, we also apply `rep_ok` to incoming set arguments. If there is a bug, these checks will help us quickly figure out where the rep invariant is being broken.

```
(* Implementation of sets as lists without duplicates.
 * Includes rep_ok checks. *)
module ListSetNoDupsRepOk : Set = struct
  (* Abstraction function:  the list [a1; ...; an] represents the
   * set {a1, ..., an}.  [] represents the empty set {}.
   *
   * Representation invariant: the list contains no duplicates.
   *)
  type 'a set = 'a list

  let rep_ok (l : 'a set) : 'a set =
    List.fold_right
      (fun x t -> assert (not (List.mem x t)); x :: t)
      l []

  let empty = []
  let mem x l = List.mem x (rep_ok l)
  let add x l = rep_ok (if mem x (rep_ok l) then l else x :: l)
  let rem x l = rep_ok (List.filter ((<>) x) (rep_ok l))
  let size l = List.length (rep_ok l)
  let union l1 l2 =
    rep_ok (List.fold_left
          (fun a x -> if mem x l2 then a else x :: a)
          (rep_ok l2) (rep_ok l1))
  let inter l1 l2 = rep_ok (List.filter (fun h -> mem h l2) (rep_ok l1))
end
```

Calling `rep_ok` on every argument can be too expensive for the production version of a program. The `rep_ok` above is quite expensive (though it could be implemented more cheaply). For production code, it may be more appropriate to use a version of `rep_ok` that only checks the parts of the rep invariant that are cheap to check. When there is a requirement that there be no run-time cost, `rep_ok` can be changed to an identity function (or macro) so the compiler optimizes away the calls to it. However, it is a good idea to keep around the full code of `rep_ok` (perhaps in a comment) so it can be easily reinstated during future debugging.

Some languages provide support for conditional compilation. These constructs are ideal for checking representation invariants and other types of sanity checks. There is a compiler option that allows such assertions to be turned on during development and turned off for the final production version. For example, the ocaml compiler supports a flag `noassert` that disables assertion checking.