Interfaces and Specifications

One reason why software design is hard is because you need to repeatedly cross various abstraction barriers. Think of it this way. Every piece of software has, or should have, an abstraction barrier that splits the world in two: the clients and the implementors.

- The clients are those who use the software. They do not need to know how the software works, they just trust it.
- The implementors are those who build the software. They *do* need to know how it works, and their job is to distrust it. As an implementor, you should always assume that there are bugs in the software you are implementing, and you should search for them using testing, walkthroughs, and formal verification.

Unfortunately, it is not always straightforward to determine whether you are a client or an implementor. When you implement software, you are also using other software, for instance, libraries, so you are an implementor of some software but a client of other software. Things get even more funky when you consider that programmers often end up writing different parts of a big piece of software at the same time, meaning that they often switch back and forth between different parts of the program, and therefore have to switch from being a client to being an implementor and back to the being a client in shorts periods of time. This sort of context switching is quite difficult, psychologically speaking.

The abstraction barrier separates the implementors from the clients using their software. It means to hide the details of the implementation. In order for the code hidden behind the abstraction barrier to be useful to a client, though, the client needs to assume *something* about how the code behaves. This is what a *specification* means to capture:

- what the client can assume about the code, and
- what an implementor needs to guarantee his implementation provides.

From an implementor's perspective, then, he needs to establish, via testing, formal proof, or other arguments, that his implementation satisfies the specification. As you can imagine, we are now in a good position to use formal proofs.

Contracts and Type Systems

Most programming languages provide at least a very thin layer of specification that they make available to programmers. Most programming languages provide a way to assign types (or contracts) to functions in an interface, and can check that these types are respected.

In Java, for instance, you annotate methods with the type of arguments that they expect, and the return type of the method. That's a specification: it says that the method, no matter how it is implemented, is guaranteed to return a value of the specified return type, provided it is called with values of the specified types. Java in fact goes one step further, and does *static type checking*: it checks, before you can run your program, that all the types agree at all places where a method is called. (Static type checking is not completely trivial when method can hide inside objects that can be passed around to other functions, and when you toss in generics in the mix, it's even harder.)

In a sense that can be made precise, a static type checker of the kind implemented by the Java compiler is actually an automated theorem prover for proving formulas in a very restricted class, namely formulas that correspond to formulas such as the following in ACL2:

$$\texttt{(true-listp x)} \wedge \texttt{(true-listp y)} \Longrightarrow \texttt{(true-listp (app x y))}$$

Of course, a static type checker can prove *only* only those kinds of formulas, as opposed to a full theorem prover, which can prove much more general properties.

Example: Specifying Sort

While type (or contract) specifications are useful, they do not go very far capturing what a piece of code does.

Suppose that Alice is interested in a library that includes a sorting function. For the sake of simplicity, let us suppose that she is interested only in sorting integers, and only in increasing order. Bob is happy to provide one, and uses a specification that Alice gives him to narrow down what his function should do.

Clearly, a specification that says that sort should take in a list of integers and return a list of integers is insufficient. There are many functions that satisfy this specification that have nothing to do with sorting. (For instance, the identity function that just returns its argument untouched.)

So if Alice is interesting in asking Bob for a sorting function, she better figure out how to specify the behavior she wants. We'll do this in ACL2. The contract information can be specified as:

$$\texttt{(int-true-listp x)} \Longrightarrow \texttt{(int-true-listp (sort x))}$$

where

```
(defun int-true-listp (x)
```

```
  (if (endp x)
      (= x NIL)
    (and (integerp (car x))
         (int-true-listp (cdr x)))))
```

To a first approximation, she will want that the result of calling `sort` is ordered in increasing order. This can be written:

```
(ordered->= (sort x))
```

where

```
(defun ordered->= (x)
  (if (endp x)
      T
    (if (endp (cdr x))
        T
      (and (>= (car (cdr x)) (car x))
           (ordered->= (cdr x))))))
```

But this is also insufficient. Consider the following function:

```
(defun foo (x)
  (if (endp x)
      NIL
    (list (car x))))
```

It is easy to check that the result of calling `foo` on a list of integers is a list of integers that is always ordered. So `foo` satisfies the specification. Yet one would be hard pressed to call this a sorting function.

What we want is an additional property that the result of sorting contains all the elements in the input list. But making this precise is somewhat tricky. You don't want to return more elements that were in the input list: so you not only want that all the elements in the original list are in the result, but also that every element in the result is in the input list too. And what if there are repeated elements in the input list? You want to make sure that they occur in the same number in the result.

Really, what you want is to *match up* all the elements in the result with all the elements in the input list. Two lists that you can match up in that way are called permutations of each other. Thus, we want a function that checks whether two lists are pernutations of each other.

```
(defun rem1 (a L)    ; remove first occurrence of a in L
  (if (endp L)
```

```
      L
    (if (= (car L) a)
        (cdr L)
      (cons (car L) (rem1 a (cdr L))))))

(defun perm (L M)    ; check if two lists are permutations
  (if (endp L)
      (endp M)
    (if (endp M)
        (endp L)
      (and (member (car L) M)
           (perm (cdr L) (rem1 (car L) M))))))
```

(Exercise: can you think of another way of specifying that the elements in the input list match up with all the elements in the result of sorting?)

The complete specification for `sort` can be written:

```
(and (implies (int-true-listp x) (int-true-listp (sort x)))
     (ordered->= (sort x))
     (perm x (sort x)))
```