Can the user define genuinely new types with the same status as the built-in types?

Ideally, to mimic the behavior of built-in types, user-defined types should have an associated set of operators, and it should only be possible to manipulate types via their operators (and maybe a few generic operators such as assignment or equality testing).

In particular, when new types are given a representation in terms of existing types, it shouldn’t be possible for programs to inspect or change the fields of the representation.

Such a type is called an abstract data type (ADT), because to clients (users) of the type, its implementation is hidden.

We can implement an ADT by combining a type definition together with a set of function operating on the type into a module (or package, cluster, class, etc.) Additional hiding features are needed to make the type’s representation more-or-less invisible outside the module.

Purely functional operators yield simpler and more elegant ADTs.

**Example: Environments in SML**

```sml
signature ENV =
sig
  type env
  val empty : env
  val extend : (env * string * int) -> env
  val lookup : (env * string) -> int option
end

structure Env :> ENV =
struct
  type env = (string * int) list
  val empty = nil
  fun extend (env,k,v) = (k,v)::env
  fun lookup ((k0,v0)::rest,k) =
    if k = k0 then SOME v0 else lookup (rest,k)
  | lookup (nil,k) = NONE
end (* Env *)
```

**Example: Environments in Java**

```java
class Env {
  private Link contents = null;
  private Env (Link c) { contents = c; }

  static final Env empty = new Env(null);

  Env extend(String k, int v) {
    return new Env(new Link(k,v,contents));
  }

  int lookup(String k) {
    Link c = contents;
    while (c != null) {
      if (c.key.equals(k))
        return c.value;
      else
        c = c.next;
    }
    return -1;
  }
}
```
If clients are to be able to use an ADT without knowing anything about
the implementation, they need a full specification of the operations’
behavior.

Type signatures give only a partial specification.

A standard approach is to add axioms describing the behavior of different
combinations of axioms. Example:

ADT env
Signatures:
empty : env
extend : env * key * value -> env
lookup : env * key -> value option

Axioms:
lookup(empty, k_0) = NONE
lookup(extend(e, k, v), k_0) =
  if k = k_0 then SOME v else lookup(e,k_0)

How many axioms are enough?
We can identify two important subsets of operations:

• constructors return new instances of the ADT.
• observers (or inspectors) take one or more instances of the ADT as
  arguments and return some other type(s) as result.

Example: for the Env ADT, the constructors are empty and extend; the
sole observer is lookup.

The only way to create an ADT value is to call a constructor. So every
ADT value can be built up inductively by applying constructors.

The only aspect of an ADT value that matters is how it behaves when
passed to an observer. (We can’t tell anything else about the value!)

So, it suffices if we give enough axioms to define the behavior of every
observer on every possible combination of constructors.
structure Env :> ENV =
 struct
datatype env =
   EMPTY
   | EXTEND of env * string * int
val empty = EMPTY
fun extend (e,k,v) = EXTEND(e,k,v)
fun lookup (EMPTY,k0) = NONE
   | lookup (EXTEND(e,k,v),k0) =
     if k = k0 then
       SOME v
     else
       lookup (e,k0)
end (* Env *)

We can use the axioms to prove the observational equivalence of two ADT values, even in cases where the representations of the values are different!

Example: suppose we have

\[ e_1 = \text{extend}(\text{extend}(\text{empty}, "a", 1), "b", 2) \]
\[ e_2 = \text{extend}(\text{extend}(\text{empty}, "b", 2), "a", 1) \]

Using the axioms, we can prove that, for any key \( k \),

\[ \text{lookup}(e_1, k) = \text{lookup}(e_2, k) \]

Hence \( e_1 \) and \( e_2 \) are observationally equivalent, even though they may have different representations (e.g. in the implementations we gave).

In conventional languages, axioms only have the status of comments. So reasoning using observational equivalence is dangerous unless we have proved that the actual implementation obeys the axioms; we can imagine systems that checked (or helped us check) this.

Ideally, the client of an ADT is not supposed to know or care about its internal implementation details – only about its exported interface. Thus, it makes sense to separate the textual description of the interface from that of the implementation, e.g., into separate files.

For example, ML distinguishes signatures (module specifications) from structures (module bodies), and encourages them to be in separate files. Specifications give the names of types, and the names and types of functions in the package. Bodies give the definitions of the types and functions mentioned in the specification, and possibly additional private definitions.

One advantage of this separation is that clients of module \( X \) can be compiled on the basis of the information in the specification of \( X \), without needing access to the the body of \( X \) (which might not even exist yet!)

Many languages, particularly in the C/C++ tradition, don’t make this separation very cleanly. Java doesn’t support it cleanly either, even given the notion of interfaces (constructors are one sticking point).

Although the idea of defining explicitly all the operators for a type makes good logical sense, it can get quite inconvenient.

Programmers are used to assigning values or passing them as arguments without providing type-specific code for doing so. They may also expect to be able to compare them, at least for equality, without writing type-specific code.

So most languages that support ADT’s have built-in support for these basic operations, defined in a uniform way across all types. They also usually have facilities for overriding the built-in definitions with type-specific versions. (Some of the complexity of C++ derives from this.)

Unfortunately, it is impossible to generate code for operations that move or compare data without knowing things like the size and layout of the data. But these are characteristics of the type’s implementation, not its interface. So these “universal” operations break the abstraction barrier around types, and conflicts with separate compilation.