ADTs

An Abstract Data Type is a poorly-defined and old concept in SE. A standard way to formalize it in the modern world is with an algebraic description.

An algebra consists of sets (sorts, types) of objects (carrier sets), a set of operations (functions) closed over those objects (all operations take arguments of carrier set type and return a result of carrier set type), and a collection of laws (equations) that constrain how the operations work.

For example:

\[ \text{Counter} = \{ \mathbb{Z}; \text{incr}, \text{decr} \} \]

\[ \text{new} : \text{Counter} \]
\[ \text{val} : \text{Counter} \rightarrow \mathbb{Z} \]
\[ \text{incr} : \text{Counter} \rightarrow \text{Counter} \]
\[ \text{decr} : \text{Counter} \rightarrow \text{Counter} \]

\[ \text{val}(\text{new}) = 0 \]
\[ \forall x : \text{Counter} \cdot \text{incr}(\text{decr}(x)) = \text{decr}(\text{incr}(x)) = x \]
\[ \forall x : \text{Counter} \cdot \text{val}(\text{incr}(x)) > \text{val}(x) \]
\[ \forall x : \text{Counter} \cdot \text{val}(\text{decr}(x)) < \text{val}(x) \]

Z ADTs

An ADT specification looks an awful lot like a Z specification. The most problematic is the actual definition of Counter, which looks like it should just be a schema.
Unfortunately, this won’t typecheck, since we cannot use Counter until it is defined. Probably better, if a little odd, is to make Counter just be a type and use a generic schema to capture its operations and laws.

Note the key ADT property: we cannot tell anything about the structure of Counter except what is implied by the laws.

**ADT Implementation**

Let us start with the obvious implementation of Counter ADT.

The key question is whether this implementation obeys the laws of counters:
\[\text{val(new)} = \text{val(0)} = 0\]
\[\forall x: \text{Counter} \quad \begin{align*}
\text{incr}(\text{decr}(x)) &= (x - 1) + 1 = x \\
\text{decr}(\text{incr}(x)) &= (x + 1) - 1 = x
\end{align*}\]
\[\forall x: \text{Counter} \quad \begin{align*}
\text{val}(	ext{incr}(x)) &= \text{val}(x + 1) = x + 1 > \\
\text{val}(x) &= x \\
\forall x: \text{Counter} \quad \begin{align*}
\text{val}(\text{decr}(x)) &= \text{val}(x - 1) = x - 1 < \\
\text{val}(x) &= x
\end{align*}\]

So...yes.

**Partial Operations Are Awkward**

One nice property that \textit{Counter} has as specified is that every operation is a total function. It is not uncommon, though, that we would prefer a “natural” counter such that

\[\forall x: \text{NatCounter} \quad \text{val}(x) \geq 0\]

We could try just doing the obvious thing and adding this law to the counter laws.

\[
\text{NatCounterADTUnsound}[\text{NatCounter}]
\]

\[
\begin{align*}
\text{new} &: \text{NatCounter} \\
\text{val} &: \text{NatCounter} \rightarrow \mathbb{Z} \\
\text{incr} &: \text{NatCounter} \rightarrow \text{NatCounter} \\
\text{decr} &: \text{NatCounter} \rightarrow \text{NatCounter}
\end{align*}
\]

\[\forall x: \text{NatCounter} \quad \text{val}(x) \geq 0\]

\[\text{val}(\text{new}) = 0\]
\[\forall x: \text{NatCounter} \quad \text{val}(\text{decr}(x)) = \text{decr}(\text{incr}(x)) = x\]
\[\forall x: \text{NatCounter} \quad \text{val}(\text{incr}(x)) > \text{val}(x)\]
\[\forall x: \text{NatCounter} \quad \text{val}(\text{decr}(x)) < \text{val}(x)\]

Unfortunately, we get into trouble immediately: \textit{decr} can no longer be a total function.

\[\text{val}(\text{new}) = 0\] [1: given]
\[\forall x: \text{NatCounter} \quad \text{val}(\text{decr}(x)) < \text{val}(x)\] [2: given]
\[\forall x: \text{NatCounter} \quad \text{val}(x) \geq 0\] [3: given]
\[\text{val}(\text{decr}(\text{new})) < 0\] [4: (1), \forall-inst (2)]
\[\text{val}(\text{decr}(\text{new})) \geq 0\] [5: \forall-inst (2)]
\[\neg (\text{val}(\text{decr}(\text{new})) < 0)\] [6: math (5)]
\[\Box\] [\Box\text{-intro (4), (6)}]
In a way, the fact that we can prove our specification unsound is good news. This keeps us from building a program that will have runtime errors. However, we have to figure out what to do about it. There are three standard approaches.

**Restrict Operation Domains**

The easiest thing to do is simply to restrict the domain of \textit{decr}.

\[
\begin{array}{l}
\text{NatCounterADTRestrict[Counter]} \\
\text{CounterADT[Counter]} \\
\text{ran}(\text{val}) = \mathbb{N} \\
\text{dom}(\text{decr}) = \text{Counter} \setminus \{\text{new}\}
\end{array}
\]

This isn’t quite right: we must also relax the laws of \textit{CounterADT} a bit so that \textit{incr}(\textit{decr}(\textit{new})) is undefined.

Note that we now have a proof obligation every time we use \textit{decr}: we must prove that it is not being passed \textit{new}. This is probably good practice and the right way to go, but it can significantly complicate proofs.

**Force Operations To Be Total**

We could certainly insist that the \textit{decr} function always return a result with non-negative \textit{val}. The obvious way to do this is to modify the laws so that decrementing from zero just returns zero again.

\[
\begin{array}{l}
\text{NatCounterADTTotal[NatCounter]} \\
\text{new : NatCounter} \\
\text{val : NatCounter} \rightarrow \mathbb{Z} \\
\text{incr : NatCounter} \rightarrow \text{NatCounter} \\
\text{decr : NatCounter} \rightarrow \text{NatCounter}
\end{array}
\]

\[
\begin{array}{l}
\forall x : \text{NatCounter} \bullet \text{val}(x) \geq 0 \\
\text{val}(\text{new}) = 0 \\
\text{decr}(\text{new}) = \text{new} \\
\forall x : \text{NatCounter} \bullet \text{decr}(\text{incr}(x)) = x \\
\forall x : \text{NatCounter} \mid x \neq \text{new} \bullet \text{incr}(\text{decr}(x)) = x \\
\forall x : \text{NatCounter} \bullet \text{val}(\text{incr}(x)) > \text{val}(x) \\
\forall x : \text{NatCounter} \mid x \neq \text{new} \bullet \text{val}(\text{decr}(x)) < \text{val}(x)
\end{array}
\]
Unfortunately, this revised counter “acts weird”. Some of the laws of the unsound counter were things we wanted to hold, and now they don’t. The behavior that calling `decr` may not actually decrease the counter, in particular, is surprising and will probably lead to bugs in the code that uses counters.

**Lift To An Error Value**

Let us define a generic type for values that either indicate an error or a non-error value.

\[
\text{[GenericCounter]}
\]

\[
\text{RESULT ::= nope | ok\langle\text{GenericCounter}\rangle}
\]

\[
\text{RESULTN ::= nopen | okn\langle\text{N}\rangle}
\]

We can now rewrite the laws to have an explicit `nope` when decrementing too far.

\[
\text{\underline{NatCounterADTLifted}}
\]

| \text{new : RESULT} |
| \text{val : RESULT \rightarrow RESULTN} |
| \text{incr : RESULT \rightarrow RESULT} |
| \text{decr : RESULT \rightarrow RESULT} |

| \text{val(new) = okn(0)} |
| \text{decr(new) = nope} |
| \forall x : \text{RESULT} \bullet x = \text{decr(incr(x))} |
| \forall x : \text{RESULT} \mid x \neq \text{new} \bullet \text{incr(decr(x))} = x |
| \text{val(nope) = nopen} |
| \forall x : \text{RESULT}; y, z : \text{N} \mid |
| \quad x \neq \text{nope} \land \text{okn(y)} = \text{val(incr(x))} \land \text{okn(z)} = \text{val(x)} \bullet |
| \quad y > z |
| \text{val(nope) = nopen} |
| \forall x : \text{RESULT}; y, z : \text{N} \mid |
| \quad x \notin \{\text{nope, new}\} \land \text{okn(y)} = \text{val(decr(x))} \land \text{okn(z)} = \text{val(x)} \bullet |
| \quad y < z |
| \text{val(decr(new)) = nopen} |

Notice that this is a huge mess, making proofs tough. It also punts all errors to runtime. Not a great choice either.