Specifying Exceptional (or Error) Behavior

The philosophical orientation we adopt is that exceptional behavior is pervasive in software, and is a vital component of a specification Furthermore, the behavior in exceptional cases has the least intuitive basis and is the most in need of precise description. Adding the specification of exceptional behavior after the "normal" cases have been specified is highly prone to oversight and likely to lead to the introduction of mistakes. Hence we should think through all the exceptional circumstances at the outset, and systematically incorporate their points of presence and effect as integral to a specification.

As we will see, the orientation indicated above is only workable if all the pre-defined types have fully specified exception behavior too, and if the strategy they chose is compatible with that desired for the TOI. Hence we will outline a strategy for error specification that we assume is to be adopted at *all* levels. Even though there may be no exceptional behavior inherent in a pre-defined type, Definitions dependent on the type may later extend behavior in a way that leads to errors (e.g., extracting the top of an empty stack). It is assumed that every ADT will provide a means to identify objects of the TOI that may be troublesome.

General Principles

- Exceptional (i.e., error) behavior is generally associated with unusual circumstances and one is ordinarily least confident about what to expect. Since the purpose of an ADT specification is to let us know *exactly* what to expect in all circumstances, exception behavior should be an integral part of the ADT.

- Describing "normal" behavior first, and then adding treatment for exceptions has a high probability of introducing oversights and mistakes. Exceptions are best identified at the outset, and integrated into the specification at every step by explicit tests in all sensitive equations.

- In order to treat exceptions in the TOI, we invariably require that exceptions be provided for in all the pre-defined types. We assume that an exceptional element of each sort is provided in each pre-defined type, and that its occurrence in that type creates no inconsistency. Operations must include a function OK: T → Boolean for each sort T so that 'OK' determines if an element of the sort is an exceptional element or not (several such categories can be handled by adding further tests) — these operations are required to always yield a proper Boolean value.

- Whenever conditional equations are used, it must be assured that the test expression evaluates to either True or False (i.e., not the error value in the Boolean pre-defined type) so that an equality is effectively determined.

- We lose flexibility by making implicit assumptions about exceptional behavior (e.g., "errors propagate"). It's sometimes more laborious to provide explicit equations to describe *all* behavior, but doing so avoids the possibility that exceptional behavior has been neglected, and moreover the default (implicit) error behavior may inadvertently corrupt the specification of normal behavior.

  However, the "errors propagate" methodology is frequently the approach of choice, so we want to make fully precise what is meant by this, and require that this assumption be *explicitly* written into a specification when it is chosen. The "errors propagate" behavior means that whenever one or more arguments is an error value then so is the result, and this is synonymous with the behavior obtained by including an equation yielding the error value of the result sort for each ADT operation when each of its arguments is the error value of the argument sort.

Example — Queues of Integers

In this example we provide a consistent, sufficiently complete initial algebra specification that provides fully for error behavior. The pre-defined types are Boolean and Int with the familiar operations, plus an error element and 'OK' test functions for exceptional elements. We use the same 'OK' function name for all types, so context must be used to distinguish them (i.e., OK is polymorphic). Of course, while OK is pre-defined for Int and Boolean, we must provide its specification for the TOI.

• Signature
  New: → Queue
  $Error_{Que}$: → Queue
  Add: Queue × Int → Queue
  Del: Queue → Queue
  Frt: Queue → Int
  IsNew: Queue → Boolean
  OK: Queue → Boolean

The constructors for this ADT are New, $Error_{Que}$ and Add. The canonical representatives for the initial algebra equivalence classes are New, $Error_{Que}$, and Add(Add(… Add(New,i_1), i_2), … , i_n) for $n \geq 1$, and OK(i_j) = True for $1 \leq j \leq n$.

• OK specification
  OK(New) = True
  OK($Error_{Que}$) = False
  OK(Add(q,i)) = OK(q) ∧ OK(i)

• Error-equations (this is “errors propagate” plus two additional equations)
  Add($Error_{Que}$,i) = $Error_{Que}$
  Add(q,$Error_{Int}$) = $Error_{Que}$
  Del(New) = $Error_{Que}$
  Del($Error_{Que}$) = $Error_{Que}$
  Frt(New) = $Error_{Int}$
  Frt($Error_{Que}$) = $Error_{Int}$
  IsNew($Error_{Que}$) = $Error_{Bool}$

• OK-equations
  IsNew(New) = True
  IsNew(Add(q,i)) = **if** OK(q) ∧ OK(i) **then** False **else** $Error_{Bool}$
  Del(Add(q,i)) = **if** OK(q) ∧ OK(i)
  then if IsNew(q) **then** New **else** Add(Del(q),i)
  else $Error_{Que}$
  Frt(Add(q,i)) = **if** OK(q) ∧ OK(i)
  then if IsNew(q) **then** i **else** Frt(q)
  else $Error_{Int}$

Claim 1: New, $Error_{Que}$, and Add constitute generators for this ADT.

Claim 2: This ADT is sufficiently complete.

Claim 3: This ADT is consistent.