Advanced Generic Definitions

[*BGA*]

Example 1 : Generic Three - Way Product

Generic function for three-component tuples:

$$[X, Y, Z]$$

$$\begin{aligned}
&triple : X \times Y \times \mathbb{Z} \longrightarrow X \times Y \times \mathbb{Z} \\
&getFirst : X \times Y \times \mathbb{Z} \longrightarrow X \\
&getSecond : X \times Y \times \mathbb{Z} \longrightarrow Y \\
&getThird : X \times Y \times \mathbb{Z} \longrightarrow \mathbb{Z}
\end{aligned}$$

$$\begin{aligned}
\forall x : X \bullet \forall y : Y \bullet \forall z : \mathbb{Z} \bullet\ & triple(x, y, z) = (x, y, z) \land \\
& getFirst(x, y, z) = x \land \\
& getSecond(x, y, z) = y \land \\
& getThird(x, y, z) = z
\end{aligned}$$

Projection functions for triples.

Example 2 : Generic Binary Tree (Conceptual)

Binary trees are typically defined for specific types in Z notation rather than as fully polymorphic structures. For a concrete type like N, you would *define* : *Tree* ::= leaf | node⟨N cross Tree cross Tree⟩. Generic trees require complex workarounds beyond standard gendef blocks, as free types in fuzz are not easily parameterized.

Example 6 : Generic State Schema

A generic container with capacity:

Note: Fuzz supports generic schemas using the schema[X] syntax (not gendef):

$$Container[X]$$

$$\begin{aligned}
&contents : \text{seq } X \\
&capacity : \mathbb{N}
\end{aligned}$$

$$\#contents \leq capacity$$

This schema is parameterized by the element type X.

It can be instantiated as Container[N], Container[Char], etc.

Example 7 : Generic Relation Operations

Relational composition for any types:

$$[X, Y, Z]$$

$$compose : (X \leftrightarrow Y) \times (Y \leftrightarrow \mathbb{Z}) \longrightarrow (X \leftrightarrow \mathbb{Z})$$

$$\forall R : X \leftrightarrow Y \bullet \forall S : Y \leftrightarrow \mathbb{Z} \bullet compose(R, S) = R \fatsemi S$$

Composes two relations using forward composition.

Example 8 : Generic Option Type (Conceptual)

Optional values (like Maybe or Option in other languages) are challenging to express generically in Z notation due to limitations with parameterized free types. For a specific type, you would *define* : *Option* ::= none | some⟨T⟩. Fully generic option types with operations require advanced patterns beyond standard gendef blocks.

Example 9 : Generic Stack Schema

Note: Define generic schema separately, then declare operations in gendef:

$$
\begin{array}{|l}
\hline
Stack[X] \\
items : \mathrm{seq}\ X \\
maxSize : \mathbb{N} \\
\hline
\#items \leq maxSize \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
[X] \\
emptyStack : Stack[X] \\
stackSize : Stack[X] \longrightarrow \mathbb{N} \\
\hline
emptyStack.items = \langle\rangle \land emptyStack.maxSize = 100 \\
\forall\, s : Stack[X] \bullet stackSize(s) = \#s.items \\
\hline
\end{array}
$$

Generic stack with query operations.

The schema is defined separately using schema[X] syntax, then operations use gendef.

Example 10 : Generic Pair Schema

Use schema[X, Y] syntax for generic schemas:

$$
\begin{array}{|l}
\hline
OrderedPair[X, Y] \\
first : X \\
second : Y \\
\hline
\end{array}
$$

A generic pair type.

The schema has two type parameters and can be instantiated as OrderedPair[N, Char], etc.

Example 11 : Generic Zip Function

Combine two sequences into a sequence of pairs:

$$
\begin{array}{|l}
\hline
[X, Y] \\
zip : \mathrm{seq}\ X \times \mathrm{seq}\ Y \longrightarrow \mathrm{seq}\,(X \times Y) \\
\hline
zip(\langle\rangle, \langle\rangle) = \langle\rangle \\
\forall\, x : X \bullet \forall\, y : Y \bullet \forall\, xs : \mathrm{seq}\ X \bullet \forall\, ys : \mathrm{seq}\ Y \bullet zip(\langle x\rangle \frown xs, \langle y\rangle \frown ys) = \\
\qquad \langle (x, y)\rangle \frown zip(xs, ys) \\
\forall\, xs : \mathrm{seq}\ X \bullet zip(xs, \langle\rangle) = \langle\rangle \\
\forall\, ys : \mathrm{seq}\ Y \bullet zip(\langle\rangle, ys) = \langle\rangle \\
\hline
\end{array}
$$

Pairs up corresponding elements from two sequences.

Example 12 : Practical Example - Generic Database TableGA

Note: Define generic schema separately, then query operations in gendef:

$[Key, Value]$

$$
\begin{array}{|l}
\hline
TableGA[Key, Value] \\
entries : Key \pfun Value \\
size : \mathbb{N} \\
\hline
size = \#(\dom entries) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
[Key, Value] \\
emptyTable : TableGA[Key, Value] \\
tableSize : TableGA[Key, Value] \fun \mathbb{N} \\
allKeys : TableGA[Key, Value] \fun \mathbb{P}\, Key \\
\hline
emptyTable.entries = \{\} \land emptyTable.size = 0 \\
\forall\, t : TableGA[Key, Value] \bullet tableSize(t) = t.size \\
\forall\, t : TableGA[Key, Value] \bullet allKeys(t) = \dom t.entries \\
\hline
\end{array}
$$

A generic key-value table with query operations. This demonstrates how generic definitions enable reusable, type-safe data structures.

The TableGA schema is defined with schema[Key, Value] syntax, allowing proper fuzz typechecking.