## 1 Recursive definitions and mathematical induction

Consider the following set of values: $\{0, 3, 7\}$. Now consider the following rule for enlarging sets: "find any member of the set and add 6 or 16 and add the result to the set". Applying this rule to the set once gives the set $\{0, 3, 6, 7, 9, 13, 16, 19, 23\}$. Applying the rule again gives $\{0, 3, 6, 7, 9, 12, 13, 15, 16, 19, 22, 23, 25, 29, 32, 35, 39\}$. Repeated application can add any value from the natural numbers $\mathcal{N}$ except those in $\{1, 2, 4, 5, 8, 10, 11, 14, 17, 20, 26\}$. Thus, the set $\{0, 3, 7\}$ together with the rule can be viewed as a way to specify the set $\mathcal{N} - \{1, 2, 4, 5, 8, 10, 11, 14, 17, 20, 26\}$. The set $\mathcal{N}$ itself can be specified in this way using the starting set $\{0\}$ and "closing under" the rule "add one to any member to get a new member".

More complex rules can be used that refer to multiple set members to determine new members. For instance the starting set $\{1, 3, 5\}$ and the rule "multiply any two members to get a new member" describes the set of numbers with all factors among 3 and 5, in other words $\{3^m 5^n \mid m \in \mathcal{N}, n \in \mathcal{N}\}$.

Because the rules used in these definitions refer to members of the set being defined, these are "recursive" definitions. The set being defined is the smallest set containing the starting members for which the rule adds no new members. In fact, the notion of starting members can be eliminated by folding them into the rule: "put in 0, 3, or 7 or add 6 or 16 to any member". Then applying the rule repeatedly to the empty set of values constructs the desired set; more declaratively, the defined set is the smallest set that is already closed under the rule.

The defined set is called the "least fixed point" of the rule. It should be noted that some rules cannot be used to define a set in this manner, as they have no least fixed point or the least fixed-point would not be unique. For instance, the rule "put 0 in the set or add any number bigger than all set members." The set constructed will depend on the order of the choices made, so there is no unique least fixed point. (The even numbers and the odd numbers plus zero both provide example sets to which this rule cannot add any new values.) Fortunately, simple syntactic restrictions on the language of the rule enable a wide class of recursive definitions that do have unambiguous meaning and these restrictions are enforced in Ontic.

All lists of numbers can be described in this manner using the rule "put the empty list in the set or add any number at the front of a list already in the set."

Mathematical induction can be used to prove that a desired property $\phi(x)$ holds of all objects $x$ in a recursively defined class. The essential proof step is showing that the rule preserves the property: A rule preserves a property $\phi()$, if when the rule adds an element $x$ to the set, based on elements $y$ and $z$ already in the set, then the added element $x$ can be proven to have the property, $\phi(x)$, based on the assumption that the already present elements $y$ and $z$ have the property, $\phi(y)$ and $\phi(z)$. More generally:

*A rule preserves a property $\phi()$ if any set member specified for addition by the rule must have the property whenever the rule uses only set members previously added that have the property $\phi()$.*

Using this proof approach, we are essentially showing by induction on $i \in \mathcal{N}$ that a set member constructed by $i$ rule applications must have the property. Since we start with the empty set, the base case $i = 0$ is trivial, as there are no set members formed by zero rule applications (so they all have the property). So, for the key step, we imagine a member $x$

added by the rule based on members $y_1, \ldots, y_m$ for which we have $\phi(y_1), \ldots, \phi(y_m)$. If we can show that $\phi(x)$ holds, the proof is complete.

For example, with the rule defining $\{3^m 5^n \mid m \in \mathcal{N}, n \in \mathcal{N}\}$ above, we can show that every set member is odd by observing that the rule can only add odd members directly ($\{1, 3, 5\}$) and any member $x$ built from members $y$ and $z$ by multiplying must be multiplying two odd members and thus will add an odd member.

## 2 Ontic representation of induction proofs

In Ontic definitions, the recursive rule for constructing a class of values[1] is described by giving an Ontic expression with all the new values to be added as its values, where this expression can refer to values already added by recursively invoking the type being defined. The examples described above are shown next. Here, we allow naturally written integers as Ontic object expressions, though the current implementation only allows 0 and 1.

```
(define (an example1)
  (either 0 3 7
          (+ 6 (an example1))
          (+ 16 (an example1))))

(define (a natural-number)
  (either 0 (+ 1 (a natural-number))))

(define (a three-five-product)
  (either 1 3 5 (* (a three-five-product)
                   (a three-five-product))))

(define (a number-list)
  (either 'nil (the list-extended-by (a number) (a number-list))))
```

We conduct induction on such a definition by writing a `show-by-induction-on` expression where the last binding in the binding list binds a variable to a call to a recursively defined type such as those just shown, where every argument to the call must be a variable. So, we might show that `(is (a three-five-product) (an odd-number))` by writing the following proof form:

```
(show-by-induction-on ((z (a three-five-product)))
    (is z (an odd-integer))
  ...proof body...)
```

This proof translates to natural mathematics as "Supposing that $z$ is an output of the rule for the type three-five-product, where the rule inputs used are all themselves produced previously by the same rule and are all odd integers. We show that this $z$ is itself an odd

[1]Here, we refer not to a "set" of values but to a "class" of values, i.e. a non-deterministic choice between values, to avoid confusion with the Ontic set object that is a single value but has the many values as members.

integer as well, using proof ‘proof-body’, so we can conclude that this rule preserves the property that type members are odd integers. As a result, any number of applications of this rule preserve this property, and so by induction on the number of such applications `(is (a three-five-product) (an odd-number))`, as desired.

During the evaluation of ‘proof-body’, the induction hypothesis (that the type members used by the rule are previous products of the rule that satisfy the goal formula) is represented by the (implicit) presence of the following inductive assumptions:

```
(axiom (is z (either 1 3 5 (* (a (the wishful-version three-five-product)))
                           (* (a (the wishful-version three-five-product))))))

(forall ((x (a (the wishful-version three-five-product))))
  (is x (an odd-number)))
```

Note that these implicit assumptions are constructed automatically from the definition of the function on which we conduct induction (here, three-five-product). The proof body can refer directly to the new function `(the wishful-version three-five-product)` if it is helpful, representing previous function outputs guaranteed to satisfy the goal formula (in this case, being an odd-number). This new function has the same usage as the original function `three-five-product`, hence the use of `(a ...)` around each when used. They each refer to an Ontic ‘type’, a function expecting no arguments.

For practice, construct the Ontic proof forms that would be used to show facts by induction on the other example definitions shown above, and write the implicit axioms that would be provided in each case.