C Pizza, Recursion, and induction

In this short appendix, written in tutorial style, we illustrate some of the key ideas of algebraic datatypes, recursive function definitions, term evaluation, automated testing (model checking), and structural induction proofs, in a simple setting of universal familiarity: food. Some more subtle points regarding identity criteria for inductive datatypes are made toward the end of the first section, and interspersed in the discussion are also a few remarks concerning automated proof. The subject matter (pizza) is inspired by books like *The Little MLer* [36] and *The Little Schemer* [41], which also use food (especially pizza!) to introduce functional programming ideas.

C.1 Representing and reasoning about pizzas

A pizza can be thought of as an inductive construction. We start with a given basis (the crust), and then we add a number of ingredients in a finite number of steps. That's precisely the kind of thing that algebraic datatypes are good at capturing. In Athena we can say:

```
datatype Pizza := crust
                | (cheese Pizza)
                | (pepper Pizza)
                | (mushroom Pizza)
                | (sausage Pizza)
```

We can now write terms like this one, which represents a cheese pizza:

```
(cheese crust);
```

and the following, which represents a pizza with mushrooms and cheese:

```
(mushroom (cheese crust)).
```

Since these are unary constructors, we need not separate their applications by parentheses. Thus, for example, we can also express the second term simply as:

```
(mushroom cheese crust).
```

Athena will parse the expression properly:

```
> (mushroom cheese crust)

Term: (mushroom (cheese crust))
```

Let us define some variables of sort `Pizza`:

```
define [p p' p1 p2] := [?p:Pizza ?p':Pizza ?p1:Pizza ?p2:Pizza]
```

and let us go on to introduce a predicate that determines whether a pizza is vegetarian:

```
declare vegetarian: [Pizza] -> Boolean

assert* veg-def :=
        [(vegetarian crust)
         (vegetarian cheese _)
         (vegetarian pepper _)
         (vegetarian mushroom _)
         (~ vegetarian sausage _)]
```

We can immediately put our definition to the test with `eval`:

```
> (eval vegetarian cheese mushroom pepper crust)

Term: true

> (eval vegetarian sausage crust)

Term: false

> (eval vegetarian pepper sausage crust)

Term: true
```

The last test, according to which the pizza

`(pepper sausage crust)`

is vegetarian, shows that our definition is incorrect. A pizza is vegetarian only if *all* its ingredients are vegetarian, not just the top one. To examine all of the ingredients rather than just the top one, the definition has to be recursive:

```
retract veg-def

assert* veg-def :=
        [(vegetarian crust)
         (vegetarian cheese p = vegetarian p)
         (vegetarian pepper p = vegetarian p)
         (vegetarian mushroom p = vegetarian p)
         (vegetarian sausage _ = false)]
```

We now get the expected results:

```
> (eval vegetarian pepper sausage crust)

Term: false
```

Note that we could have also defined `vegetarian` in logical (relational) rather than functional style, as follows:

```
assert* veg-def :=
        [(vegetarian crust)
         (vegetarian cheese p <==> vegetarian p)
         (vegetarian pepper p <==> vegetarian p)
         (vegetarian mushroom p <==> vegetarian p)
         (~ vegetarian sausage _)]
```

Athena can work with either style equally comfortably. In fact, the two can be freely mixed, and the former definition is actually an example of such a mix because of the first clause, `(vegetarian crust)`. A purely functional definition would be written as follows:

```
assert* veg-def :=
        [(vegetarian crust = true)
         (vegetarian cheese p = vegetarian p)
         (vegetarian pepper p = vegetarian p)
         (vegetarian mushroom p = vegetarian p)
         (vegetarian sausage _ = false)]
```

Again, we can opt for any of these three style choices (purely functional, purely relational, or a mix).

Next, we define a function that (recursively) removes meat from a pizza:

```
declare remove-meat: [Pizza] -> Pizza

assert* remove-meat-def :=
   [(remove-meat crust = crust)
    (remove-meat sausage p = remove-meat p)
    (remove-meat cheese p = cheese remove-meat p)
    (remove-meat pepper p = pepper remove-meat p)
    (remove-meat mushroom p = mushroom remove-meat p)]
```

Evaluation seems to yield reasonable results, suggesting that the definition is correct:

```
> (eval remove-meat cheese sausage mushroom crust)

Term: (cheese (mushroom crust))
```

The following now sounds like a reasonable conjecture: If we remove all meat from a pizza, the result should be a vegetarian pizza:

```
define conjecture-1 := (forall p . vegetarian remove-meat p)
```

Let's put this conjecture to the test:

```
> (falsify conjecture-1 1000)

Term: 'failure
```

There seem to be no obvious counterexamples—so can we prove the conjecture? Yes, by structural induction:

```
> by-induction conjecture-1 {
    crust =>
      (!chain<- [(vegetarian remove-meat crust)
             <== (vegetarian crust)                  [remove-meat-def]
             <== true                                [veg-def]])

  | (cheese p) =>
      (!chain<- [(vegetarian remove-meat cheese p)
             <== (vegetarian cheese remove-meat p)   [remove-meat-def]
             <== (vegetarian remove-meat p)          [veg-def]])

  | (pepper p) =>
      (!chain-> [(vegetarian remove-meat p)
             ==> (vegetarian pepper remove-meat p)   [veg-def]
             ==> (vegetarian remove-meat pepper p)   [remove-meat-def]])

  | (mushroom p) =>
      (!chain-> [(vegetarian remove-meat p)
             ==> (vegetarian mushroom remove-meat p) [veg-def]
             ==> (vegetarian remove-meat mushroom p) [remove-meat-def]])

  | (sausage p) =>
      (!chain-> [(vegetarian remove-meat p)
             ==> (vegetarian remove-meat sausage p)  [remove-meat-def]])
  }

Theorem: (forall ?p:Pizza
           (vegetarian (remove-meat ?p:Pizza)))
```

Note that in the inductive cases (i.e., for all constructors *c* except `crust`), the starting point is the inductive hypothesis, `(vegetarian remove-meat p)`, where `p` is the smaller pizza to which *c* is applied. Athena automatically inserts that inductive hypothesis into the relevant assumption base for each such case. Also, here we have written some chains in a forward direction and some backward in order to illustrate both styles, but in practice most proofs use only one style.

Also note that the reasoning in most of the above cases is identical, which always suggests an opportunity for abstraction. Such abstraction will result not only in a shorter and cleaner proof, but one that will also be easier to extend if the set of pizza ingredients is ever changed. Here we abstract over the pizza term that we are inducting on, as determined by the top ingredient constructor *c*. The only precondition is that *c* is a vegetarian constructor, meaning that `remove-meat` and `vegetarian` behave accordingly on pizzas constructed with *c*:

```
define (rm-veg-case pizza) :=
  match pizza {
    ((some-symbol c) p) =>
      (!chain<- [(vegetarian remove-meat pizza)
             <== (vegetarian (c (remove-meat p))) [remove-meat-def]
             <== (vegetarian remove-meat p)       [veg-def]])
  }
```

The proof can now be expressed much more succinctly as follows:

```
by-induction conjecture-1 {
   crust => (!chain<- [(vegetarian remove-meat crust)
                   <== (vegetarian crust)
                   <== true])
 | (pizza as (cheese _)) =>   (!rm-veg-case pizza)
 | (pizza as (pepper _)) =>   (!rm-veg-case pizza)
 | (pizza as (mushroom _)) => (!rm-veg-case pizza)
 | (sausage p) =>
      (!chain-> [(vegetarian remove-meat p)
             ==> (vegetarian remove-meat sausage p)])
}
```

Note that in the last case, for the `sausage` constructor, we did not cite any justification for the `chain` step. We started with the inductive hypothesis

$$\texttt{(vegetarian remove-meat p)}, \tag{1}$$

which is guaranteed to be in the assumption base when that `chain` is applied (by virtue of Athena's evaluation semantics), and we inferred from that hypothesis the conclusion

$$\texttt{(vegetarian remove-meat sausage p)}. \tag{2}$$

Now (2) follows from (1) simply because, by `remove-meat-def`, we have

```
(remove-meat sausage p = remove-meat p).
```

We didn't explicitly cite `remove-meat-def` as our justification for deriving (2) from (1), but the implementation of `chain` figured it out on its own quickly enough.

Let us now define a function that tells us whether or not a pizza has a given ingredient, where ingredients are represented here simply as meta-identifiers:

```
declare has-ingredient: [Pizza Ide] -> Boolean

define [i i' i1 i2] := [?i:Ide ?i':Ide ?i1:Ide ?i2:Ide]

assert* has-ingredient-def :=
  [(~ crust has-ingredient i)
   (cheese p has-ingredient i <==> i = 'cheese | p has-ingredient i)
   (pepper p has-ingredient i <==> i = 'pepper | p has-ingredient i)
   (mushroom p has-ingredient i <==> i = 'mushroom | p has-ingredient i)
   (sausage p has-ingredient i <==> i = 'sausage | p has-ingredient i)]

define cp := (cheese pepper crust)

> (eval cp has-ingredient 'pepper & cp has-ingredient 'cheese)

Term: true

> (eval cp has-ingredient 'sausage | cp has-ingredient 'mushroom)

Term: false
```

Now here is another conjecture: Removing meat from a pizza which has no sausage has no effect—we get back the original pizza.

```
define conjecture-2 :=
  (forall p . ~ p has-ingredient 'sausage ==> remove-meat p = p)
```

We can test the conjecture as usual using `falsify`:

```
> (falsify conjecture-2 1000)

Term: 'failure
```

So we see that the conjecture is intuitively sound and that no obvious counterexamples to it can be mechanically (and thus possibly non-intuitively) generated. So let us go ahead and prove it. We will use induction for this conjecture as well. If we were working with ATPs and the Athena method `induction*` that automates structural induction (see Section D.1.3), we could actually prove this property automatically in a few seconds:

```
> (!induction* conjecture-2)

Theorem: (forall ?p:Pizza
           (if (not (has-ingredient ?p:Pizza 'sausage))
               (= (remove-meat ?p:Pizza)
                  ?p:Pizza)))
```

But let us see how we can prove it directly with **`by-induction`** and chaining instead. First, we define a procedure that constructs the property of interest for any given pizza `p`:

```
define (property p) :=
  (~ p has-ingredient 'sausage ==> remove-meat p = p)

conclude conjecture-2
  by-induction (forall p . property p) {
    crust => (!chain [(~ crust has-ingredient 'sausage)
                  ==> (remove-meat crust = crust)     [remove-meat-def]])

```

```
 9   | (pizza as (cheese p)) =>
10       (!chain [(~ pizza has-ingredient 'sausage)
11            ==> (~ p has-ingredient 'sausage)         [has-ingredient-def]
12            ==> (remove-meat p = p)                   [(property p)]
13            ==> (cheese remove-meat p = cheese p)
14            ==> (remove-meat cheese p = cheese p)     [remove-meat-def]])
15
16   | (pizza as (pepper p)) =>
17       (!chain [(~ pizza has-ingredient 'sausage)
18            ==> (~ p has-ingredient 'sausage)         [has-ingredient-def]
19            ==> (remove-meat p = p)                   [(property p)]
20            ==> (pepper remove-meat p = pepper p)
21            ==> (remove-meat pepper p = pepper p)     [remove-meat-def]])
22
23   | (pizza as (mushroom p)) =>
24       (!chain [(~ pizza has-ingredient 'sausage)
25            ==> (~ p has-ingredient 'sausage)         [has-ingredient-def]
26            ==> (remove-meat p = p)                   [(property p)]
27            ==> (mushroom remove-meat p = mushroom p)
28            ==> (remove-meat mushroom p = mushroom p) [remove-meat-def]])
29
30    | (pizza as (sausage p)) =>
31       assume (~ pizza has-ingredient 'sausage)
32         (!chain-> [('sausage = 'sausage)
33                ==> (pizza has-ingredient 'sausage)   [has-ingredient-def]
34                ==> (pizza has-ingredient 'sausage &
35                     ~ pizza has-ingredient 'sausage) [augment]
36                ==> false                             [prop-taut]
37                ==> (remove-meat pizza = pizza)       [prop-taut]])
38 }
```

Observe that there are no justifications for the steps on lines 13, 20, and 27. Each of these steps proceeds from a premise of the form $(s = t)$ to a conclusion of the form $(f\ s = f\ t)$, and therefore follows from simple identity congruence. We could explicitly justify them if we choose to do so, e.g., the step on line 13 could be written as

```
==> (cheese remove-meat p = cheese p) [(method (_ q) (!fcong q))],
```

but `chain` does not require explicit annotations for such steps. In fact we could remove all of the given justifications and the proof would still go through.

Why do we start the `chain->` application on line 32 with the identity

```
('sausage = 'sausage)?
```

That has to do with the given justification, `has-ingredient-def`, and specifically with the relevant clause of `has-ingredient-def`, namely the clause for `sausage`:

```
(sausage p has-ingredient i <==> i = 'sausage | p has-ingredient i).
```

The chain step from line 32 to line 33 essentially uses this defining biconditional clause in a right-to-left direction: It instantiates the universally quantified variable `i` with `'sausage`, and then from

```
('sausage = 'sausage),
```

which is a special sort of tautology that can always start off a `chain->`, it infers

```
('sausage = 'sausage | p has-ingredient 'sausage)
```

by disjunction introduction, and then it finally applies biconditional elimination and modus ponens to get the conclusion on line 33, namely

```
(sausage p has-ingredient 'sausage),
```

or equivalently, `(pizza has-ingredient 'sausage)`, since `pizza` in that context is just an alias for `(sausage p)` owing to the pattern on line 30.

As before, there is much repetitious reasoning here, specifically in the cases of the vegetarian constructors. We abstract that reasoning in a separate method:

```
1  define (conjecture-2-method pizza) :=
2    match pizza {
3     ((some-symbol c:(OP 1)) p) =>
4       (!chain
5         [(~ pizza has-ingredient 'sausage)
6      ==> (~ p has-ingredient 'sausage) [has-ingredient-def]
7      ==> (remove-meat p = p)           [(property p)]  # ind. hypothesis
8      ==> (c remove-meat p = c p)       # identity congruence
9      ==> (remove-meat c p = c p)       [remove-meat-def]])
10   }
```

We inserted the `(OP 1)` annotation on line 3 to allow for the usual infix conventions in using `c`, as in lines 8 and 9. Without it, Athena would not know that `c` is a unary function symbol and thus an expression such as `(c remove-meat p = c p)` would not parse; it would have to be written in full prefix form as `(= (c (remove-meat p)) (c p))`.[1]

We can now express the proof in a clearer form:

```
conclude conjecture-2
  by-induction (forall p . property p) {
    crust => (!chain [(~ crust has-ingredient 'sausage)
                  ==> (remove-meat crust = crust) [remove-meat-def]])

  | (pizza as (cheese _))   => (!conjecture-2-method pizza)
  | (pizza as (pepper _))   => (!conjecture-2-method pizza)
  | (pizza as (mushroom _)) => (!conjecture-2-method pizza)
```

1 In this particular case we could infer from the structure of the pattern on line 3 that `c` is a unary function symbol, which means that the `(OP 1)` declaration is redundant here, but that is not always possible.

```
  | (pizza as (sausage _)) =>
      assume h := (~ pizza has-ingredient 'sausage)
        (!chain-> [('sausage = 'sausage)
               ==> (pizza has-ingredient 'sausage)     [has-ingredient-def]
               ==> (pizza has-ingredient 'sausage & h) [augment]
               ==> false                               [prop-taut]
               ==> (remove-meat pizza = pizza)         [prop-taut]])
}
```

There are other conjectures relating the various operations and predicates we have defined so far, and which we could proceed to investigate, such as:

```
define conjecture-3 :=
  (forall p i . (remove-meat p) has-ingredient i <==>
                p has-ingredient i & i =/= 'sausage)
```

In other words, removing meat from a pizza removes the meat (thereby resulting in a vegetarian pizza, as we have already proved), but it *does not remove anything else*.

However, there are two issues with the general approach we have taken so far. First, the ingredients are hardwired into the definition of `Pizza`. As we will show in the next section, it will be much better to somehow pull that factor out of the definition of `Pizza`.

Second, and more important, we have defined `Pizza` as a *datatype*, which means that terms such as

`(pepper cheese crust)`

and

`(cheese pepper crust)`

represent different pizzas (distinct elements of `Pizza`):

```
define pizza1 := (pepper cheese crust)

define pizza2 := (cheese pepper crust)

> (eval pizza1 = pizza2)

Term: false

> (eval pizza1 =/= pizza2)

Term: true
```

But that's not quite right in this case. For our purposes, two pizzas should be the same iff they have the same ingredients; the order in which these ingredients are added makes no difference. On the other hand, we want to retain the intuition that a pizza is built in stages, in a finite number of steps, and that *any* pizza (over the given set of ingredients) can be built that way. In other words, we want to retain the notion that `Pizza` is *inductively generated*. We simply need to adjust the identity relation to make it a little less fine-grained. We can do that by defining `Pizza` as a **`structure`** rather than a **`datatype`**, and by explicitly defining the identity relation for that structure.

One way to do that is to first define a function `ingredients` that takes a pizza *p* and produces the *set* of ingredients in *p*, and then define two pizzas to be identical iff they have the same set of ingredients. Using the module `Set` (see Section 10.3), we would have:

```
declare ingredients: [Pizza] -> (Set.Set Ide)

define [null ++] := [Set.null Set.++]

assert* ingredients-def :=
   [(ingredients crust = null)
    (ingredients pepper p = 'pepper ++ ingredients p)
    (ingredients cheese p = 'cheese ++ ingredients p)
    (ingredients mushroom p = 'mushroom ++ ingredients p)
    (ingredients sausage p = 'sausage ++ ingredients p)]

assert* pizza-identity := (p1 = p2 <==> ingredients p1 = ingredients p2)
```

With this new constructively defined notion of pizza identity in the assumption base, the algorithmic evaluation of equality on terms of sort `Pizza` is automatically adjusted:

```
define pizza1 := (cheese pepper mushroom crust)

define pizza2 := (mushroom cheese pepper crust)

> (eval pizza1 = pizza2)

Term: true
```

C.2 Polymorphic pizzas

Let us now define `Pizza` as a polymorphic structure over an arbitrary ingredient sort `I`:

```
structure (Pizza I) := crust | (topping I (Pizza I))

define [p p' p1 p2] := [?p:(Pizza 'S1) ··· ?p2:(Pizza 'S4)]
```

To get the previous kind of pizzas, we could define, e.g.:

```
datatype Main-Ingredient := cheese | pepper | mushroom | sausage
```

and then:

```
define-sort Standard-Pizza := (Pizza Main-Ingredient)
```

To get pizzas topped with fish we only need to define, e.g.:

```
datatype Fish := tuna | lox | anchovy | salmon
```

and then `(Pizza Fish)` gives us all kinds of fish-topped pizzas. Likewise for, e.g.,

```
datatype Meat := sausage | pepperoni | chicken | ham | bacon
```

etc. Or we could get even fancier and make the ingredients themselves into a polymorphic datatype:

```
datatype (Ingredient T) := (basic Main-Ingredient) | (fancy T)
```

and then consider pizzas of sorts like `(Pizza (Ingredient Fish))`.

For greater notational flexibility, we introduce `++` as an infix-friendly alias for `topping`, so that (x `++` p) represents pizza p with topping x added to it:

```
define ++ := topping

> (tuna ++ lox ++ crust)

Term: (topping tuna
               (topping lox
                        crust:(Pizza Fish)))
```

Note that Hindley-Milner sort inference has deduced that this is a `(Pizza Fish)`.

Let us now define a function that removes a specific type of topping from a pizza. The function is polymorphic and the definition is recursive. We introduce an infix alternative `-` for it, so that (p `-` x) represents the removal of ingredient x from pizza p, and we make it bind looser than `++`:

```
declare remove: (T) [(Pizza T) T] -> (Pizza T)

overload - remove

set-precedence ++ (plus 10 (get-precedence -))

> (lox ++ crust - lox)

Term: (remove (topping lox
                       crust:(Pizza Fish))
              lox)
```

We can now define the function as follows:

```
assert* remove-def :=
        [(crust - _ = crust)
         (x ++ p - x = p - x)
         (x =/= y ==> y ++ p - x = y ++ (p - x))]
```

Note that this definition is not painstakingly expressed in terms of individual constructors, as was the case with the earlier definition of the `remove-meat` function in the previous section, back when we had baked all possible ingredients into the very definition of `Pizza`. By making pizzas polymorphic over an arbitrary sort of ingredients, we now only ever have to deal with two constructors, `crust` and `topping`. Let's evaluate the operation to make sure it behaves properly:

```
> define fp := (lox ++ tuna ++ salmon ++ crust - tuna)

Term: (remove (topping lox
                       (topping tuna
                                (topping salmon
                                         crust:(Pizza Fish))))
              tuna)

> (eval fp)

Term: (topping lox
               (topping salmon
                        crust:(Pizza Fish)))
```

We also define a much more general and at the same time more succinct ingredients function as follows:

```
declare ingredients-of: (I) [(Pizza I)] -> (Set.Set I)

define [null in] := [Set.null Set.in]

overload ++ Set.++

assert* ingredients-def :=
  [(ingredients-of crust = null)
   (ingredients-of x ++ p = x ++ ingredients-of p)]

transform-output eval [Set.set->lst]

> (eval ingredients-of fp)

List: [lox salmon]
```

Note that because we have overloaded ++, the two occurrences of it on the second clause of the preceding definition:

```
(ingredients-of x ++ p = x ++ ingredients-of p)
```

have different meanings. The left occurrence represents `topping` while the right occurrence represents `Set.++`.

We can now prove much more general results, e.g.:

```
# If we remove a topping x from a pizza, the result
# doesn't have x among its ingredients:

define conjecture-1 := (forall p  x . ~ x in ingredients-of p - x)

> (falsify conjecture-1 100)

Term: 'failure
```

A structured proof can be given as follows:

```
by-induction conjecture-1 {
    crust => pick-any x
               (!chain<- [(~ x in ingredients-of crust - x)
                      <== (~ x in ingredients-of crust)
                      <== (~ x in null)
                      <== true])
  | (pizza as (topping t:'T rest:(Pizza 'T))) =>
      pick-any x:'T
       (!two-cases
         assume case-1 := (x = t)
           (!chain<-
             [(~ x in ingredients-of pizza - x)
          <== (~ x in ingredients-of pizza - t)          [(x = t)]
          <== (~ x in ingredients-of rest - t)           [remove-def]
          <== (~ x in ingredients-of rest - x)           [(x = t)]
          <== true])
         assume case-2 := (x =/= t)
           (!chain<-
              [(~ x in ingredients-of (t ++ rest) - x)
          <== (~ x in ingredients-of t ++ (rest - x))    [remove-def]
          <== (~ x in t ++ ingredients-of rest - x)      [ingredients-def]
          <== (~ (x = t | x in ingredients-of rest - x)) [Set.in-def]
          <== (case-2 & ~ x in ingredients-of rest - x)  [prop-taut]
          <== (~ x in ingredients-of rest - x)           [augment]
          <== true]))
}
```

Note that while we provide justifications for the chain steps of the inductive case, we do not do so for the basis steps, as those are deemed fairly obvious.

An interesting observation regarding automated proof can be made here. Because this new pizza formalization uses the `Set` module, which is a fairly large theory, the resulting assumption base ends up having more than 300 sentences, most of them set-theoretic results. If we thus tried to prove this conjecture completely mechanically and with default options, using `induction*`, we would not succeed in a reasonable amount of time (at least not after several minutes on a moderately powered laptop). This is an occasion where even a mildly effective premise selection procedure can make a big difference. If we use Athena's implementation of the SINE selection algorithm (see Section D.1 for details), we can prove this result completely automatically in a few seconds. The main reason is that the size of the relevant sentences goes down from 368 when no selection filter is used to 40:

```
load "sine"

> (length (ab))

Term: 368

> (length SINE.all-relevant-sentences conjecture-1)

Term: 40

> retract conjecture-1

The sentence
(forall ?p:(Pizza 'S)
  (forall ?x:'S
    (not (Set.in ?x:'S
                 (ingredients-of (remove ?p:(Pizza 'S)
                                         ?x:'S))))))
has been removed from the assumption base.

> (!induction*-with conjecture-1
    method (g)
      (!derive-from g (SINE.all-relevant-sentences g)
                      |{'atp := 'vampire, 'max-time := 100}|))

Theorem: (forall ?p:(Pizza 'S)
           (forall ?x:'S
             (not (Set.in ?x:'S
                          (ingredients-of (remove ?p:(Pizza 'S)
                                                  ?x:'S))))))
```