```
> clear-assumption-base

Assumption base cleared.

> assert (1 = 2)

The sentence
(= 1 2)
has been added to the assumption base.

> retract (1 = 2)

The sentence
(= 1 2)
has been removed from the assumption base.
```

All three of these (`assert`, `clear-assumption-base`, and `retract`) are top-level directives; they cannot appear inside other code.

The last two constructs are powerful and should be used sparingly. A directive

`retract` *p*

in particular, simply removes *p* from the assumption base without checking to see what other sentences might have been derived from *p* in the interim (between its assertion and its retraction), so a careless removal may well leave the assumption base in an incorrect state. This tool is meant to be used only when we assert a sentence *p* and then right away realize that something went wrong, for instance, that *p* contains certain free variables that it should not contain, in which case we can promptly remove it from the assumption base with `retract`.

2.7 Datatypes

A `datatype` is a special kind of domain. It is special in that it is *inductively generated*, meaning that every element of the domain can be built up in a finite number of steps by applying certain operations known as the *constructors* of the datatype. A datatype D is specified by giving its name, possibly followed by some sort parameters (if D is polymorphic; we discuss that in Section 2.8), and then a nonempty sequence of constructor profiles separated by the symbol `|`. A constructor profile without selectors[13] is of the form

$$(c \;\; S_1 \cdots S_n), \tag{2.3}$$

consisting of the name of the constructor, c, along with n sorts $S_1 \cdots S_n$, where S_i is the sort of the i^{th} argument of c. The range of c is not explicitly mentioned—it is tacitly understood

13 Selectors are discussed in Section A.5.

to be the datatype D. That is why, after all, c is called a “constructor” of D; because it builds or *constructs* elements of the datatype. Thus, every application of c to n arguments of the appropriate sorts produces an element of D. In terms of the **declare** directive that we have already seen, a constructor c of a datatype D with profile (2.3) can be thought of as a function symbol with the following signature:

$$\texttt{declare}\ c\texttt{: [}S_1 \cdots S_n\texttt{] -> }D.$$

A nullary constructor (one that takes no arguments, so that $n = 0$) is called a *constant constructor* of D, or simply a constant of D. Such a constructor represents an individual element of D. The outer parentheses in (2.3) may be (and usually are) omitted from the profile of a constant constructor.

Here is an example:

```
datatype Boolean := true | false
```

This defines a datatype by the name of `Boolean` that has two constant constructors, `true` and `false`. (This particular datatype is predefined in Athena.) Thus, this definition says that the datatype `Boolean` has two elements, `true` and `false`. The definition conveys more information than that, however. It also licenses the conclusion that `true` and `false` are *distinct* elements, and, moreover, that `true` and `false` are the *only* elements of `Boolean`.

The intended effect of this datatype definition could be approximated in terms of mechanisms with which we are already familiar as follows:

```
domain Boolean

declare true, false: Boolean

assert (true =/= false)

assert (forall ?b:Boolean . ?b = true | ?b = false)
```

Here we have made two assertions that ensure that (a) `true` and `false` refer to distinct objects; and (b) the domain `Boolean` does not contain any other elements besides those denoted by `true` and `false`. For readers who are familiar with universal algebra, these axioms ensure that the datatype is freely generated, or that it is a *free algebra*. Given a datatype D, we will collectively refer to these axioms as the *free-generation axioms* for D. Roughly, the axioms express the following propositions: (a) different constructor applications create different elements of D; and (b) every element of D is represented by some constructor application. Axioms of the first kind are usually called *no-confusion* axioms, while axioms of the second kind are called *no-junk* axioms.

Thus, to a certain extent, a datatype definition can be viewed as syntax sugar for a domain declaration (plus appropriate symbol declarations for the constructors), along with the relevant free-generation axioms. We say “to a certain extent” because an additional and important effect of a datatype definition is the introduction of an inductive principle for performing structural induction on the datatype (we discuss this in Section 3.8). For infinite datatypes such as the natural numbers, to which we will turn next, structural induction is an ingredient that goes above and beyond the free-generation axioms, meaning that the induction principle allows us to prove results that would not be derivable from the free-generation axioms alone.

Here is a datatype for the natural numbers that we will use extensively in this book:

```
datatype N := zero | (S N)
```

This defines a datatype `N` that has one constant constructor `zero` and one unary constructor `S`. Because the argument of `S` is of the same sort as its result, `N`, we say that `S` is a *reflexive* constructor. Thus, `S` requires an element of `N` as input in order to construct another such element as output. By contrast, `zero`, as well as `true` and `false` in the `Boolean` example, are *irreflexive* constructors—trivially, since they take no arguments.

Unlike domains, which can be interpreted by arbitrary sets, the interpretation of a datatype is fixed: a datatype definition always picks out a unique set (up to isomorphism). Roughly, in the case of `N`, that set can be understood as given by the following rules:

1. `zero` is an element of `N`.
2. For all *n*, if *n* is an element of `N`, then `(S` *n*`)` is an element of `N`.
3. Nothing else is an element of `N`.

The last clause, in particular, ensures the minimality of the defined set, whereby the *only* elements of `N` are those that can be obtained by a finite number of constructor applications.

Essentially, every datatype definition can be understood as a recursive set definition of the preceding form. How do we know that a recursive definition of this form always succeeds in determining a unique set, and how do we know that a datatype definition can always be thus understood? We will not get into the details here, but briefly, the answer to the first question is that we can prove the existence of a unique set satisfying the recursive definition using fixed-point theory; and the answer to the second question is that there are simple syntactic constraints on datatype definitions that ensure that every datatype gives rise to a recursive set definition of the proper form.[14]

The following are the free-generation axioms for `N`:

```
(forall ?n . zero =/= S ?n)

(forall ?n ?m . S ?n = S ?m ==> ?n = ?m)

(forall ?n . ?n = zero | exists ?m . ?n = S ?m)
```

14 For instance, definitions such as `datatype D := (c D)` are automatically rejected.

You might recognize these if you have ever seen Peano's axioms. The first two are no-confusion axioms: (1) `zero` is different from every application of `S` (i.e., `zero` is not the successor of any natural number), and (2) applications of `S` to different arguments produce different results (i.e., `S` is injective). The third axiom says that the constructors `zero` and `S` span the domain `N`: Every natural number is either `zero` or else the successor of some natural number.[15]

The free-generation axioms of a datatype can be obtained automatically by the unary procedure `datatype-axioms`, which takes the name of the datatype as an input string and returns a list of the free-generation axioms for that datatype:

```
> (datatype-axioms "N")

List: [
(forall ?y1:N
  (not (= zero
          (S ?y1:N))))

(forall ?x1:N
  (forall ?y1:N
    (iff (= (S ?x1:N)
            (S ?y1:N))
         (= ?x1:N ?y1:N))))

(forall ?v:N
  (or (= ?v:N zero)
      (exists ?x1:N
        (= ?v:N
           (S ?x1:N)))))
]

> (datatype-axioms "Boolean")

List: [
(not (= true false))

(forall ?v:Boolean
  (or (= ?v:Boolean true)
      (= ?v:Boolean false)))
]
```

Whether or not these axioms are added to the assumption base automatically is determined by a global flag `auto-assert-dt-axioms`, off by default. If turned on (see Section 2.12), then every time a new datatype T is defined, its axioms will be automatically added to the assumption base. In addition, the identifier T`-axioms` will be automatically bound in the global environment to the list of these axioms.

15 Strictly speaking, the third axiom (and other axioms of a similar form, in the case of other datatypes) can be proved by induction, but it is useful to have it directly available.

Once a datatype D has been defined, it can be used as a bona fide Athena sort; for example, we can declare functions that take elements of D as arguments or return elements of D as results. We can introduce binary addition on natural numbers, for instance, as follows:

```
> declare Plus: [N N] -> N

New symbol Plus declared.
```

A number of mutually recursive datatypes can be defined with the **datatypes** keyword, separating the component datatypes with &&. For example:

```
> datatypes Even := zero | (s1 Odd) &&
            Odd  := (s2 Even)

New datatypes Even and Odd defined.
```

The constructors of a top-level datatype (or a set of mutually recursive datatypes) must have distinct names from one another, as well as from the constructors of every other top-level datatype and from every other function symbol declared at the top level. However, the same sort, constructor, or function symbol name can be used inside two different modules.

Not all inductively defined sets are freely generated. For example, in some cases it is possible for two distinct constructor terms to denote the same element. Consider, for instance, a hypothetical datatype for finite sets of integers:

```
datatype Set := null | (insert Int Set)
```

This definition says that a finite set of integers is either `null` (the empty set) or else of the form `(insert` i s`)`, obtained by inserting an integer i into the set s. Now, two sets are identical iff they have the same members, e.g., $\{1,3\} = \{3,1\}$. Hence,

```
(insert 3 (insert 1 null))
```

and

```
(insert 1 (insert 3 null))
```

ought to be identical. That is, we must be able to prove

```
((insert 3 (insert 1 null)) = (insert 1 (insert 3 null))).        (2.4)
```

But one of the no-confusion axioms would have us conclude that `insert` is injective:

```
(forall ?i1 ?s1 ?i2 ?s2 . (insert ?i1 ?s1) = (insert ?i2 ?s2)
                      ==> ?i1 = ?i2 & ?s1 = ?s2)
```

This is inconsistent with (2.4), as it would allow us to conclude, for example, that `(1 = 3)`.

The preferred approach here is to define Set as a **structure** rather than a **datatype**:

```
structure Set := null | (insert Int Set)
```

A *structure* is a datatype with a coarser identity relation. Just like regular datatypes, a structure is inductively generated by its constructors, meaning that every element of the structure is obtainable by a finite number of constructor applications. This means that structural induction (via **by-induction**) is available for structures. The only difference is that there may be some “confusion,” for instance, the constructors might not be injective (the usual case), so that one and the same constructor applied to two distinct sequences of arguments might result in the same value. More rarely, we might even obtain the same value by applying two distinct constructors. It is the user’s responsibility to assert a proper identity relation for a structure. In the set example, we would presumably assert something along the following lines:

```
(forall ?s1 ?s2 . ?s1 = ?s2 <==> ?s1 subset ?s2 & ?s2 subset ?s1),
```

where `subset` has the obvious definition in terms of membership.

The unary procedure `structure-axioms` will return a list of all the inductive axioms that are usually appropriate for a structure, assuming that the only difference is that constructors are not injective. The input argument is the name of the structure:

```
> (structure-axioms "Set")

List: [
(forall ?y1:Int
  (forall ?y2:Set
    (not (= null
            (insert ?y1:Int ?y2:Set)))))

(forall ?v:Set
  (or (= ?v:Set null)
      (exists ?x1:Int
        (exists ?x2:Set
          (= ?v:Set
             (insert ?x1:Int ?x2:Set))))))
]
```

If the structure has other differences (most notably, if applying two distinct constructors might result in the same value), then it is the user’s responsibility to assert the relevant axioms as needed.

2.8 Polymorphism

2.8.1 Polymorphic domains and sort identity

A domain can be polymorphic. As an example, consider sets over an arbitrary universe, call it `S`:

```
> domain (Set S)

New domain Set introduced.
```

The syntax for introducing polymorphic domains is the same as before, except that now the domain name is flanked by an opening parenthesis to its left and a list of identifiers to its right, followed by a closing parenthesis; say, `(Set S)`, or in the general case, $(I\ I_1 \cdots I_n)$, where I is the domain name. The identifiers $I_1, \ldots, I_n$ serve as *sort variables*, indicating that I is a *sort constructor* that takes any n sorts $S_1, \ldots, S_n$ as arguments and produces a new sort as a result, namely $(I\ S_1 \cdots S_n)$. For instance, `Set` is a unary sort constructor that can be applied to an arbitrary sort, say the domain `Int`, to produce the sort `(Set Int)`. For uniformity, monomorphic sorts such as `Person` and `N` can be regarded as nullary sort constructors. Polymorphic datatypes and structures, discussed in Section 2.8.3, can also serve as sort constructors.

Equipped with the notion of a sort constructor, we can define Athena sorts more precisely. Suppose we have a collection of sort constructors $\boldsymbol{SC}$, with each $sc \in \boldsymbol{SC}$ having a unique arity $n \geq 0$, and a disjoint collection of sort variables $\boldsymbol{SV}$. We then define a *sort over* $\boldsymbol{SC}$ *and* $\boldsymbol{SV}$ as follows:

- Every sort variable ψ in $\boldsymbol{SV}$ is a sort over $\boldsymbol{SC}$ and $\boldsymbol{SV}$.
- Every nullary sort constructor $sc \in \boldsymbol{SC}$ is a sort over $\boldsymbol{SC}$ and $\boldsymbol{SV}$.
- If $S_1, \ldots, S_n$ are sorts over $\boldsymbol{SC}$ and $\boldsymbol{SV}$, $n > 0$, and $sc \in \boldsymbol{SC}$ is a sort constructor of arity n, then $(sc\ S_1 \cdots S_n)$ is a sort over $\boldsymbol{SC}$ and $\boldsymbol{SV}$.
- Nothing else is a sort over $\boldsymbol{SC}$ and $\boldsymbol{SV}$.

We write $Sorts(\boldsymbol{SC}, \boldsymbol{SV})$ for the set of all sorts over $\boldsymbol{SC}$ and $\boldsymbol{SV}$.

For example, suppose that $\boldsymbol{SC} = \{$`Int`, `Boolean`, `Set`$\}$ and $\boldsymbol{SV} = \{$`S1`, `S2`$\}$, where `Int` and `Boolean` are nullary sort constructors, while `Set` is unary. Then the following are all sorts over $\boldsymbol{SC}$ and $\boldsymbol{SV}$:

```
Boolean
Int
(Set Boolean)
```

```
S1
(Set Int)
(Set S2)
(Set (Set Int))
(Set (Set S1))
```

There are infinitely many sorts over these three sort constructors and two sort variables.

Sorts of the form $(sc\ S_1 \cdots S_n)$ for $n > 0$ are called *compound*, or complex. A *ground* (or *monomorphic*) sort is one that contains no sort variables. All of the above examples are ground except `S1`, `(Set S2)`, and `(Set (Set S1))`. A sort that is not ground is said to be polymorphic.

A polymorphic domain is really a domain *template*. Substituting ground sorts for its sort variables gives rise to a specific domain, which is an *instance* of the template. Intuitively, you can think of a polymorphic domain as the collection of all its ground instances.

Formally, let us define a *sort valuation* τ as a finite function from sort variables to sorts. Any such τ can be extended to a function

$$\widehat{\tau} : Sorts(\boldsymbol{SC}, \boldsymbol{SV}) \rightarrow Sorts(\boldsymbol{SC}, \boldsymbol{SV})$$

(i.e., to a function $\widehat{\tau}$ from sorts over $\boldsymbol{SC}$ and $\boldsymbol{SV}$ to sorts over $\boldsymbol{SC}$ and $\boldsymbol{SV}$) as follows:

$$\widehat{\tau}(\psi) = \tau(\psi)$$
$$\widehat{\tau}((sc\ S_1 \cdots S_n)) = (sc\ \widehat{\tau}(S_1) \cdots \widehat{\tau}(S_n))$$

for any sort variable $\psi \in \boldsymbol{SV}$ and sort constructor $sc \in \boldsymbol{SC}$ of arity n. We say that a sort S_1 is an *instance of* (or *matches*) a sort S_2 iff there exists a sort valuation τ such that $\widehat{\tau}(S_2) = S_1$. And we say that two sorts S_1 and S_2 are *unifiable* iff there exists a sort valuation τ such that $\widehat{\tau}(S_1) = \widehat{\tau}(S_2)$. There are algorithms for determining whether one sort matches another, or whether two sorts are unifiable, and these algorithms are widely used in Athena's operational semantics.

Two sorts are considered identical iff they differ only in their variable names, that is, iff each can be obtained from the other by (consistently) renaming sort variables. It follows that a ground sort (such as `N`) is identical only to itself, since it does not contain any sort variables.

2.8.2 Polymorphic function symbols

Polymorphic sorts pave the way for polymorphic function symbols. The syntax for declaring a polymorphic function symbol f is the same as before, except that the relevant sort variables must appear listed within parentheses and separated by commas before the list of input sorts. Thus, the general syntax form is

$$\texttt{declare}\ f\texttt{:}\ (I_1, \ldots, I_n)\ \texttt{[}S_1 \cdots S_n\texttt{]}\ \texttt{->}\ S \qquad (2.5)$$

where $I_1, \ldots, I_n$ are distinct identifiers that will serve as sort variables, and as before, S_i is the sort of the i^{th} argument and S is the sort of the result. Presumably, some of these sorts will involve the sort variables. For example:

```
declare in: (S) [S (Set S)] -> Boolean

declare union: (S) [(Set S) (Set S)] -> (Set S)

declare =: (S) [S S] -> Boolean
```

The first declaration introduces a polymorphic membership predicate that takes an element of an arbitrary sort S and a set over S and "returns" either `true` or `false`. The second declaration introduces a polymorphic union function that takes two sets over an arbitrary sort S and produces another set over the same sort. Finally, the last declaration introduces a polymorphic equality predicate; that particular symbol is built-in.

Polymorphic constants can be declared as well, say:

```
declare empty-set: (T) [] -> (Set T)
```

A function symbol declaration of the form (2.5) is admissible iff (a) f is distinct from every function symbol (including datatype constructors) introduced before it; (b) every S_i, as well as S, is a sort over $\boldsymbol{SC}$ and $\{I_1, \ldots, I_n\}$, where $\boldsymbol{SC}$ is the set of sort constructors available prior to the declaration of f; and (c) every sort variable that appears in the output sort S appears in some input sort S_i. Hence, the following are all inadmissible:

```
> declare g: (S) [(Set Int S)] -> Boolean

standard input:1:18: Error: Ill-formed sort: (Set Int S).

> declare g: (S) [(Set T) S] -> Int

standard input:1:18: Error: Ill-formed sort: (Set T).

> declare g: (S) [Int] -> S

standard input:1:25: Error: The sort variable S appears in the
resulting sort but not in any argument sort.
```

The first declaration is rejected because `(Set Int S)` is not a legal sort (we introduced `Set` as a unary sort constructor, but here we tried to apply it to *two* sorts). The second attempt is rejected because `T` is neither a previously introduced sort nor one of the sort variables listed before the input sorts. The third error message explains why the last attempt is also rejected.

Intuitively, a polymorphic function symbol f can be thought of as a collection of monomorphic function symbols, each of which can be viewed as an instance of f. The declaration of each of those instances is obtainable from the declaration of f by consistently replacing sort variables by ground sorts. For example, the foregoing declaration of the polymorphic predicate symbol `in` might be regarded as syntax sugar for infinitely many monomorphic function symbol declarations:

```
declare in_Int: [Int (Set Int)] -> Boolean

declare in_Real: [Real (Set Real)] -> Boolean

declare in_Boolean: [Boolean (Set Boolean)] -> Boolean

declare in_(Set Int): [(Set Int) (Set (Set Int))] -> Boolean
```

and so on for infinitely more ground sorts. This is elaborated further in Section 5.6.

Polymorphic function symbols are harnessed by Athena's polymorphic terms and sentences. Try typing a variable such as `?x` into Athena without any sort annotations:

```
> ?x

Term: ?x:'T175
```

Note the sort that Athena has assigned to the input variable, namely, `'T175`. This is a sort variable. Athena generally prints out sort variables in the format `'T`n or `'S`n, where n is some integer index. This is the most general sort that Athena could assign to the variable `?x` in this context. So this is a polymorphic variable, and hence a polymorphic term. Users can also enter explicitly polymorphic variables, i.e., variables annotated with polymorphic sorts, writing sort variables in the form `'`I, for example:

```
> ?y:'T3

Term: ?y:'T177
```

Note that the sort that Athena assigned to `?y` is `'T177`, which is identical to `'T3`, since each can be obtained from the other by renaming sort variables (recall our discussion of sort identity in the previous subsection). Here are some more examples of polymorphic terms:

```
> (?x in ?y)

Term: (in ?x:'T203
              ?y:(Set 'T203))

> (?a = ?b)

Term: (= ?a:'T206 ?b:'T206)

> (?x:(Set 'T) in ?y:(Set (Set 'T)))

Term: (in ?x:(Set 'T209)
              ?y:(Set (Set 'T209)))
```

In the first two examples, Athena automatically infers the most general possible polymorphic sorts for every variable occurrence. Also note that the common occurrence of `'T203` in the first example indicates that Athena has inferred a constraint on the sorts of `?x` and `?y`, namely, that whatever sort `?x` is, `?y` must be a set of *that* sort. Likewise for the second example: The sorts of `?a` and `?b` can be arbitrary but must be identical. In the third example we have explicitly provided specific polymorphic sorts for `?x` and `?y`: `(Set 'T)` for `?x` and `(Set (Set 'T))` for `?y`. Athena verified that these were consistent with the signature of `in` and thus accepted them (modulo the sort-variable permutation `'T` $\leftrightarrow$ `'T209`). Observe, however, that it is not necessary to provide both of these sorts. We can just annotate one of them and have the sort of the other be inferred automatically, for example:

```
> (?x:(Set 'T) in ?y)

Term: (in ?x:(Set 'T213)
              ?y:(Set (Set 'T213)))
```

We can write sort annotations not just for variables but also for constant terms:

```
> (in ?x empty-set:(Set Real))

Term: (in ?x:Real
              empty-set:(Set Real))

> empty-set:(Set (Set 'S))

Term: empty-set:(Set (Set 'T4395))
```

Any constant term can be annotated in Athena, including monomorphic ones. Athena will just ignore such annotations, as long as they are correct:

```
> joe:Person

Term: joe

> (S zero:N)

Term: (S zero)
```

There is an easy way to check whether a term *t* is polymorphic: Give it as input to the Athena prompt and then scan Athena's output for sort variables (prefixed by `'`). If Athena annotates at least one variable or constant symbol in *t* with a polymorphic sort (i.e., one containing sort variables), then *t* is polymorphic; otherwise it is monomorphic. But there is also a primitive unary procedure `poly?` that will take any term *t* and return `true` or `false` depending on whether or not *t* is polymorphic.

Informally, it is helpful to think of a polymorphic term as a schema or template that represents infinitely many monomorphic terms. For instance, you can think of the polymorphic term `empty-set` as representing infinitely many monomorphic terms, such as

```
empty-set:(Set Int)
empty-set:(Set Boolean)
empty-set:(Set (Set Int))
```

and so on. Keep in mind that the mere presence of a polymorphic symbol in a term does not make that term polymorphic. For instance, the term `(?x:Int = 3)` contains the polymorphic symbol =, but it is not itself polymorphic. No variable or constant symbol in it has a nonground sort, as you can verify by typing the term into Athena:

```
> (?x = 3)

Term: (= ?x:Int 3)
```

or by supplying it as an argument to `poly?`:

```
> (poly? (?x = 3))

Term: false
```

The presence of polymorphic function symbols is a necessary condition for a term to be polymorphic, but it is not sufficient.

A polymorphic sentence is one that contains at least one polymorphic term, or a quantified variable with a nonground sort. Here are some examples:

```
> (forall ?x . ?x = ?x)

Sentence: (forall ?x:'S
            (= ?x:'S ?x:'S))

> (forall ?x ?y . ?x union ?y = ?y union ?x)

Sentence: (forall ?x:(Set 'S)
            (forall ?y:(Set 'S)
              (= (union ?x:(Set 'S)
                        ?y:(Set 'S))
                 (union ?y:(Set 'S)
                        ?x:(Set 'S)))))

> (~ exists ?x . ?x in empty-set)

Sentence: (not (exists ?x:'S
                 (in ?x:'S
                         empty-set:(Set 'S))))
```

Note that quantified variables can be explicitly annotated with polymorphic sorts:

```
> (exists ?x:(Set (Set 'T)) . ?x =/= empty-set)

Sentence: (exists ?x:(Set (Set 'S))
            (not (= ?x:(Set (Set 'S))
                    empty-set:(Set (Set 'S)))))
```

As with terms, a simple way to test whether an unannotated sentence is polymorphic is to give it as input to the Athena prompt and then scan the output for sort variables. If you see any, the sentence is polymorphic, otherwise it is monomorphic. But `poly?` can also be used on sentences:

```
> (poly? (forall ?x . ?x = ?x))

Term: true
```

Also as with terms, a polymorphic sentence such as `(forall ?x . ?x = ?x)` can be seen as a collection of infinitely many monomorphic sentences, namely:

```
  (forall ?x:Int . ?x = ?x)
 (forall ?x:Boolean . ?x = ?x)
(forall ?x:(Set Int) . ?x = ?x)
```

and so on. This expressivity is the power of parametric polymorphism. A single polymorphic sentence manages to express infinitely many propositions about infinitely many sets of objects.

2.8.3 Polymorphic datatypes

Since datatypes are just special kinds of domains, they too can be polymorphic, and likewise for structures. Athena's polymorphic datatypes resemble polymorphic algebraic datatypes found in languages such as ML and Haskell. Here are two examples for polymorphic lists and ordered pairs:

```
datatype (List S) := nil | (:: S (List S))

datatype (Pair S T) := (pair S T)
```

The syntax is the same as for monomorphic datatypes, except that the datatype name is now flanked by an opening parenthesis to its left and a list of distinct identifiers to its right, followed by a closing parenthesis; e.g., `(List S)` or `(Pair S T)`, or, in general, $(I \; I_1 \cdots I_n)$, where I is the name of the datatype. Just as for polymorphic domains, the identifiers $I_1, \ldots, I_n$ serve as sort variables, indicating that I is a sort constructor that takes n sorts $S_1, \ldots, S_n$ as arguments and produces a new sort as a result, $(I \; S_1 \cdots S_n)$. For instance, `List` is a unary sort constructor that can be applied to an arbitrary sort, say the domain `Int`, to produce the sort `(List Int)`; while `Pair` is a binary sort constructor and can thus be applied to any two sorts to produce a new one, e.g., `(Pair Boolean (List Int))`.

The profile of each constructor is the same as before: (c $S_1 \cdots S_k$), where c is the name of the constructor.[16] Here, each S_i must be a sort over all previously introduced sort constructors *plus* the datatype that is being defined (thus allowing recursion), and the sort variables $I_1, \ldots, I_n$. Recursive datatype definitions are quite useful and common, the definition of lists given above being a typical example.

The notion of a reflexive constructor remains unchanged: If an argument of a constructor c is of a sort that involves the name of the datatype of which c is a constructor, then c is reflexive; otherwise it is irreflexive. Every datatype must have at least one irreflexive constructor. More specifically, we say that a definition of a datatype D is admissible if (1) the name D is distinct from all previously introduced sorts (domains or datatypes); (2) the constructors of D are distinct from one another, as well as from every function symbol introduced prior to the definition of D; (3) every argument sort of every constructor of D is a sort over $\boldsymbol{SC} \cup \{D\}$ and $\{I_1, \ldots, I_n\}$, where $\boldsymbol{SC}$ is the set of previously available sort constructors and $I_1, \ldots, I_n$ are the sort variables (if any) listed in the definition of D; and finally, (4) D has at least one irreflexive constructor.

The procedures `datatype-axioms` and `structure-axioms` work just as well with polymorphic datatypes, for example:

```
> (datatype-axioms "List")

List: [
(forall ?y1:'S
  (forall ?y2:(List 'S)
    (not (= nil:(List 'S)
            (:: ?y1:'S
                ?y2:(List 'S))))))

(forall ?x1:'S
  (forall ?x2:(List 'S)
    (forall ?y1:'S
      (forall ?y2:(List 'S)
        (if (= (:: ?x1:'S
                   ?x2:(List 'S))
               (:: ?y1:'S
                   ?y2:(List 'S)))
            (and (= ?x1:'S ?y1:'S)
                 (= ?x2:(List 'S)
                    ?y2:(List 'S))))))))

(forall ?v:(List 'S)
  (or (= ?v:(List 'S)
         nil:(List 'S))
```

16 Also as before, the outer parentheses may be dropped when $k = 0$.

```
      (exists ?x1:'S
        (exists ?x2:(List 'S)
          (= ?v:(List 'S)
             (:: ?x1:'S
                 ?x2:(List 'S)))))))
]
```

When D is a polymorphic datatype of arity n and $S_1, \ldots, S_n$ are ground sorts, then the sort $(D\ S_1 \cdots S_n)$ may be regarded as a monomorphic datatype. That datatype's definition can be retrieved from the definition of D by consistently replacing the sort variables $I_1, \ldots, I_n$ by the sorts $S_1, \ldots, S_n$, respectively. For example, `(Pair Int Boolean)` may be seen as a monomorphic datatype $\texttt{Pair}_{\texttt{Int}\times\texttt{Boolean}}$ with one binary constructor

$$\texttt{pair}_{\texttt{Int}\times\texttt{Boolean}}$$

whose first argument is `Int` and whose second argument is `Boolean`, namely, as the datatype

$$\textbf{datatype}\ \texttt{Pair}_{\texttt{Int}\times\texttt{Boolean}} := (\texttt{pair}_{\texttt{Int}\times\texttt{Boolean}}\ \texttt{Int Boolean}).$$

Likewise, `(List Int)` can be understood as a monomorphic datatype $\texttt{List}_{\texttt{Int}}$, definable as

$$\textbf{datatype}\ \texttt{List}_{\texttt{Int}} := \texttt{nil}_{\texttt{Int}}\ |\ (\texttt{::}_{\texttt{Int}}\ \texttt{Int}\ \texttt{List}_{\texttt{Int}}).$$

Thus, just as with other polymorphic domains, a polymorphic datatype may be viewed as the collection of all its ground instances.

2.8.4 Integers and reals

Athena comes with two predefined numeric domains, `Int` for integers and `Real` for real numbers. The domain `Int` has infinitely many constant symbols associated with it, namely, all integer numerals, positive, negative, and zero:

```
> (?x = 47)

Term: (= ?x:Int 47)
```

Note that Athena automatically recognized the sort of `?x` as `Int`. Negative integer numerals are written as `(-` n`)`:

```
> (exists ?x . ?x = (- 5))

Sentence: (exists ?x:Int
            (= ?x:Int
               (- 5)))
```