2 Built-In Types

2.1 Simple and Structured Types

Virtually every programming language has implementations of several ADTs built into it. We distinguish two sorts of built-in types:

Simple types: The values of the carrier set are atomic, that is, they cannot be divided into parts. Common examples of simple types are integer, Boolean, character, floating point, and enumerations. Some languages also provide string as a built-in simple type.

Structured types: The values of the carrier set are not atomic, consisting instead of several atomic values arranged in some way. Common examples of structured types are arrays, records, classes, and sets. Some languages treat strings as a built-in structured types.

Note that both simple and structured types are implementations of ADTs, it is simply a question of how the programming language treats the values of the carrier set of the ADT in its implementation. The remainder of this chapter considers some Ruby simple and structured types to illustrate these ideas.

2.2 Types in Ruby

Ruby is a pure object-oriented language, meaning that all types in Ruby are classes, and every value in a Ruby program is an instance of a class. This has several consequences for the way values can be manipulated that may seem odd to programmers familiar with languages that have values that are not objects. For example, values in Ruby respond to method calls: The expressions `142.even?` and `“Hello”.empty?` are perfectly legitimate (the first expression is `true` and the second is `false`).

Ruby has many built-in types because it has many built-in classes. Here we only consider a few Ruby types to illustrate how they realize ADTs.

2.3 Symbol: A Simple Type in Ruby

Ruby has many simple types, including numeric classes such as `Integer`, `Fixnum`, `Bignum`, `Float`, `BigDecimal`, `Rational`, and `Complex`, textual classes such as `String`, `Symbol`, and `Regexp`, and many more. One unusual and interesting simple type is `Symbol`, which we consider in more detail to illustrate how a type in a programming language realizes an ADT.

Ruby has a `String` class whose instances are mutable sequences of Unicode characters. `Symbol` class instances are character sequences that are not mutable, and consequently the `Symbol` class has far fewer operations than the `String` class. Ruby in effect has implementations of two String ADTs—we consider the simpler one, calling it the *Symbol* ADT for purposes of this discussion.

The carrier set of the Symbol ADT is the set of all finite sequences of characters over the Unicode characters set (Unicode is a standard character set of over 110,000 characters from 93 scripts). Hence this carrier set includes the string of zero characters (the empty string), all strings of one character, all strings of two characters, and so forth. This carrier set is infinite.

The operations of the Symbol ADT are the following.

a==*b*—returns true if and only if symbols *a* and *b* are identical.

a<=b—returns true if and only if either symbols *a* and *b* are identical, or symbol *a* precedes symbol *b* in Unicode collating sequence order.

a<*b*—returns true if and only if symbol *a* precedes symbol *b* in Unicode collating sequence order.

empty?(*a*)—returns true if and only if symbol *a* is the empty symbol.

a=~*b*—returns the index of the first character of the first portion of symbol *a* that matches the regular expression *b*. If there is no match, the result is undefined.

caseCompare(*a*,*b*)—compares symbols *a* and *b*, ignoring case, and returns -1 if *a*<*b*, 0 if *a*==*b*, and 1 otherwise.

length(*a*)—returns the number of characters in symbol *a*.

capitalize(*a*)—returns the symbol generated from *a* by making its first character uppercase and making its remaining characters lowercase.

downcase(*a*)—returns the symbol generated from *a* by making all characters in *a* lowercase.

upcase(*a*)—returns the symbol generated from *a* by making all characters in *a* uppercase.

swapcase(*a*)—returns the symbol generated from *a* by making all lowercase characters in *a* uppercase and all uppercase characters in *a* lowercase.

charAt(*a*,*i*)—returns the one character symbol consisting of the character of symbol *a* at index *i* (counting from 0); the result is undefined if *i* is less than 0 or greater than or equal to the length of *a*.

charAt(*a*,*i*,*c*)—returns the substring of symbol *a* beginning at index *i* (counting from 0), and continuing for *c* characters; the result is undefined if *i* is less than 0 or greater than or equal to the length of *a*, or if *c* is negative. If *i*+*c* is greater than the length of *a*, the result is the suffix of symbol *a* beginning at index *i*.

succ(*a*)—returns the symbol that is the successor of symbol *a*. If *a* contains characters or letters, the successor of *a* is found by incrementing the right-most letter or digit according to the Unicode collating sequence, carrying leftward if necessary when the last digit or letter in the collating sequence is encountered. If *a* has no letters or digits, then the right-most character of *a* is incremented, with carries to the left as necessary.

toString(*a*)—returns a string whose characters correspond to the characters of symbol *a*.

toSymbol(*a*)—returns a symbol whose characters correspond to the characters of string *a*.

The Symbol ADT has no concatenation operations, but assuming we have a full-featured String ADT, symbols can be concatenated by converting them to strings, concatenating the strings, then converting the result back to a symbol. Similarly, String ADT operations can be used to do other manipulations. This explains why the Symbol ADT has a rather odd mix of operations: The Symbol ADT models the `Symbol` class in Ruby, and this class only has operations often used for Symbols, with most string operations appearing in the `String` class.

The Ruby implementation of the Symbol ADT, as mentioned, hinges on making `Symbol` class instances immutable, which corresponds to the relative lack of operations in the Symbol ADT. `Symbol` values are stored in the Ruby interpreter's symbol table, which guarantees that they cannot be changed. This also guarantees that only a single `Symbol` instance will exist corresponding to any sequence of characters, which is an important characteristic of the Ruby `Symbol` class that is not required by the Symbol ADT, and distinguishes it from the `String` class.

All Symbol ADT operations listed above are implemented in the `Symbol` class, except *toSymbol*(), which is implemented in classes (such as `String`), that can generate a `Symbol` instance. When a result is undefined in the ADT, the result of the corresponding `Symbol` class method is `nil`. The names are sometimes different, following Ruby conventions; for example, *toString*() in the ADT becomes `to_s()` in Ruby, and *charAt*() in the ADT is `[]()` in Ruby.

Ruby is written in C, so carrier set members (that is, individual symbols) are implemented as fixed-size arrays of characters (which is how C represents strings) inside the `Symbol` class. The empty symbol is an array of length 0, symbols of length one are arrays with a single element, symbols of length two are arrays with two elements, and so forth. `Symbol` class operations are either written to use these arrays directly, or to generate a `String` instance, do the operation on the string, and convert the result back into a `Symbol` instance.

Excellent Economics and Business programmes at:
university of groningen
AACSB ACCREDITED
"The perfect start of a successful, international career."
CLICK HERE to discover why both socially and academically the University of Groningen is one of the best places for a student to be
www.rug.nl/feb/education

2.4 Range: A Structured Type in Ruby

Ruby has a several structured types, including arrays, hashes, sets, classes, streams, and ranges. In this section we will only discuss ranges briefly as an example of a structured type.

The *Range of T* ADT represents a set of values of type *T* (called the *base type*) between two extremes. The *start value* is a value of type *T* that sets the lower bound of a range, and the *end value* is a value of type *T* that sets the upper bound of a range. The range itself is the set of values of type *T* between the lower and upper bounds. For example, the Range of Integers from 1 to 10 inclusive is the set of values {1, 2, 3, …, 10}.

A range can be *inclusive*, meaning that it includes the end value, or *exclusive*, meaning that it does not include the end value. Inclusive ranges are written with two dots between the extremes, and exclusive ranges with three dots. Hence the Range of Integers from 1 to 10 inclusive is written 1…10, and the Range of Integers from 1 to 10 exclusive (the set {1, 2, 3, …, 9}), is written 1…10.

A type can be a range base type only if it supports order comparisons. For example, the Integer, Real, and String types support order comparisons and so may be range base types, but Sets and Arrays do not, so they cannot be range base types.

The carrier set of a Range of *T* is the set of all sets of values $v \in T$ such that for some start value $s \in T$ and end value $e \in T$, either $s \leq v$ and $v \leq e$ (the inclusive ranges), or $s \leq v$ and

$v < s$ (the exclusive ranges), plus the empty set. For example, the carrier set of the Range of Integer is the set of all sequences of contiguous integers. The carrier set of the Range of Real is the set of all sets of real number greater than or equal to a given number, and either less than or equal to another, or less than another. These sets are called *intervals* in mathematics.

The operations of the Range of *T* ADT includes the following, where $a, b \in T$ and *r* and *s* are values of Range of *T*:

a…*b*—returns a range value (an element of the carrier set) consisting of all $v \in T$ such that $a \leq v$ and $v \leq b$.

a…*b*—returns a range value (an element of the carrier set) consisting of all.

$v \in T$ such that $a \leq v$ and $v < b$.

r==*s*—returns true if and only if *r* and *s* have the same base type, start and end values, and are either both inclusive or both exclusive ranges.

min(*r*)—returns the smallest value in *r*. The result is undefined if *r* is the empty range.

max(*r*)—returns the largest value in *r*. The result is undefined if *r* has no largest value (for example, the Range of Real 0…3 has no largest value because there is no largest Real number less than 3).

cover?(*r*, *x*)—returns true if and only if $x \in r$.

The Range of *T* ADT is a structured type because the values in its carrier set are composed of values of some other type, in this case, sets of value of the base type *T*.

Ruby implements the Range of *T* ADT in its `Range` class. Elements of the carrier set are represented in `Range` instances by recording internally the type, start, and end values of the range, along with an indication of whether the range is inclusive or exclusive. Ruby implements all the operations above, returning `nil` when the ADT operations are undefined. It is quite easy to see how to implement these operations given the representation elements of the carrier set. In addition, the `Range` class provides operations for accessing the begin and end values defining the range, which are easily accessible because they are recorded. Finally, the Range class has an `include?()` operation that tests range membership by stepping through the values of the range from start value to end value when the range is non-numeric. This gives slightly different results from `cover?()` in some cases (such as with `String` instances).

2.5 Review Questions

1. What is the difference between a simple and a structured type?
2. What is a pure object-oriented language?
3. Name two ways that `Symbol` instances differ from `String` instances in Ruby.
4. Is `String` a simple or structured type in Ruby? Explain.
5. List the carrier set of Range of {1, 2, 3} (inclusive). In this type, what values are 1..1, 2..1, and 1…3? What is max(1…3)?

2.6 Exercises

1. Choose a language that you know well and list its simple and structures types.
2. Choose a language that you know well and compare its simple and structured types to those of Ruby. Does one language have a type that is simple while the corresponding type in the other language is structured? Which language has more simple types or more structured types?
3. Every Ruby type is a class, and every Ruby value is an instance of a class. What advantage and disadvantages do you see with this approach?
4. Write pseudocode to implement the `cover?()` operation for the Range class in Ruby.
5. Give an example of a Ruby `String` range `r` and `String` instance `v` such that `r.cover?(v)` and `r.include?(v)` differ.

2.7 Review Question Answers

1. The values of a simple type cannot be divided into parts, while the values of a structured type can be. For example, the values of the `Integer` type in Ruby cannot be broken into parts, while the values of a `Range` in Ruby can be (the parts are the individual elements of the ranges).
2. A pure object-oriented language is one whose types are all classes. Java and C++, for example, are not pure object-oriented languages because they include primitive data types, such as `int`, `float`, and `char`, that are not classes. Smalltalk and Ruby are pure object-oriented languages because they have no such types.
3. `Symbol` instances in Ruby are immutable while `String` instances are mutable. `Symbol` instances consisting of a particular sequence of characters are unique, while there may be arbitrarily many `String` instances with the same sequence of characters.
4. `String` is a simple type in Ruby because strings are not composed of other values—in Ruby there is no character type, so a `String` value cannot be broken down into parts composed of characters. If `s` is a `String` instance, then `s[0]` is not a character, but another `String` instance.
5. The carrier set of Range of {1, 2, 3} is { {}, {1}, {2}, {3}, {1, 2}, {2, 3}, {1, 2, 3} }. The value 1..1 is {1}, the value 2..1 is {}, and the value 1…3 is {1, 2}, and max(1…3) is 2.