Types in Dafny

Manuscript KRML 243
27 February 2015

K. Rustan M. Leino
leino@microsoft.com

Abstract. This part of the Dafny language reference describes the types in the Dafny programming language. What is described is what is implemented in version 1.9.3.20107 of Dafny, with the exception of async-task types which have not yet made it into the main branch.

0. Basic types

Dafny offers three basic types, `bool` for booleans, `int` for integers, and `real` for reals.

0.0. Booleans

There are two boolean values and each has a corresponding literal in the language: `false` and `true`.

In addition to equality (`==`) and disequality (`!=`), which are defined on all types, type `bool` supports the following operations:

operator	description
`<==>`	equivalence (if and only if)

`==>`
`<==`
	implication (implies)
	reverse implication (follows from)

`&&` `||`	conjunction (and) disjunction (or)

!	negation (not)

Negation is unary; the others are binary. The table shows the operators in groups of increasing binding power, with equality binding stronger than conjunction and disjunction, and weaker than negation. Within each group, different operators do not associate, so parentheses need to be used. For example,

```
A && B || C    // error
```

would be ambiguous and instead has to be written as either

```
(A && B) || C
```

or

```
A && (B || C)
```

depending on the intended meaning.

The expressions `A <==> B` and `A == B` give the same value, but note that `<==>` is *associative* whereas `==` is *chaining*. So,

```
A <==> B <==> C
```

is the same as

```
A <==> (B <==> C)
```

and

```
(A <==> B) <==> C
```

whereas

```
A == B == C
```

is simply a shorthand for

```
A == B && B == C
```

Conjunction is associative and so is disjunction. These operators are are *short circuiting (from left to right)*, meaning that their second argument is evaluated only if the evaluation of the first operand does not determine the value of the expression. Logically speaking, the expression `A && B` is defined when `A` is defined and either `A` evaluates to `false` or `B` is defined. When `A && B` is defined, its meaning is the same as the ordinary, symmetric mathematical conjunction ∧. The same holds for `||` and ∨.

Implication is *right associative* and is short-circuiting from left to right. Reverse implication `B <== A` is exactly the same as `A ==> B`, but gives the ability to write the operands in the opposite order. Consequently, reverse implication is *left associative* and is short-circuiting from *right to left*. To illustrate the associativity rules, each of the following four lines expresses the same property, for any `A`, `B`, and `C` of type `bool`:

```
A ==> B ==> C
A ==> (B ==> C)  // parentheses redundant, since ==> is right associative
C <== B <== A
(C <== B) <== A  // parentheses redundant, since <== is left associative
```

To illustrate the short-circuiting rules, note that the expression `a.Length` is defined for an array `a` only if `a` is not `null` (see Section 5), which means the following two expressions are well-formed:

```
a != null ==> 0 <= a.Length
0 <= a.Length <== a != null
```

The contrapositive of these two expressions would be:

```
a.Length < 0 ==> a == null  // not well-formed
a == null <== a.Length < 0  // not well-formed
```

but these expressions are not well-formed, since well-formedness requires the left (and right, respectively) operand, `a.Length < 0`, to be well-formed by itself.

Implication `A ==> B` is equivalent to the disjunction `!A || B`, but is sometimes (especially in specifications) clearer to read. Since, `||` is short-circuiting from left to right, note that

```
a == null || 0 <= a.Length
```

is well-formed, whereas

```
0 <= a.Length || a == null  // not well-formed
```

is not.

In addition, booleans support *logical quantifiers* (forall and exists), described in a different part of the Dafny language reference.

0.1. Numeric types

Dafny supports *numeric types* of two kinds, *integer-based*, which includes the basic type `int` of all integers, and *real-based*, which includes the basic type `real` of all real numbers. User-defined numeric types based on `int` and `real`, called *newtypes*, are described in Section 7. Also, the *subset type* `nat`, representing the non-negative subrange of `int`, is described in Section 8.

The language includes a literal for each non-negative integer, like `0`, `13`, and `1985`. Integers can also be written in hexadecimal using the prefix “`0x`”, as in `0x0`, `0xD`, and `0x7c1` (always with a lower case `x`, but the hexadecimal digits themselves are case insensitive). Leading zeros are allowed. To form negative integers, use the unary minus operator.

There are also literals for some of the non-negative reals. These are written as a decimal point with a nonempty sequence of decimal digits on both sides. For example, `1.0`, `1609.344`, and `0.5772156649`.

For integers (in both decimal and hexidecimal form) and reals, any two digits in a literal may be separated by an underscore in order to improve human readability of the literals. For example:

```
1_000_000        // easier to read than 1000000
0_12_345_6789    // strange but legal formatting of 123456789
0x8000_0000      // same as 0x80000000 -- hex digits are often placed in groups of 4
0.000_000_000_1  // same as 0.0000000001 -- 1 Ångström
```

In addition to equality and disequality, numeric types support the following relational operations:

operator	description
`<`	less than
`<=`	at most
`>=`	at least
`>`	greater than

Like equality and disequality, these operators are chaining, as long as they are chained in the “same direction”. That is,

```
A <= B < C == D <= E
```

is simply a shorthand for

```
A <= B && B < C && C == D && D <= E
```

whereas

```
A < B > C
```

is not allowed.

There are also operators on each numeric type:

operator	description
`+`	addition (plus)
`-`	subtraction (minus)
`*`	multiplication (times)
`/`	division (divided by)
`%`	modulus (mod)
`-`	negation (unary minus)

The binary operators are left associative, and they associate with each other in the two groups. The groups are listed in order of increasing binding power, with equality binding more strongly than the multiplicative operators and weaker than the unary operator. Modulus is supported only for integer-based numeric types. Integer division and modulus are the *Euclidean division and modulus*. This means that modulus always returns a non-negative, regardless of the signs of the two operands. More precisely, for any integer `a` and non-zero integer `b`,

```
a == a / b * b + a % b
0 <= a % b < B
```

where `B` denotes the absolute value of `b`.

Real-based numeric types have a member `Trunc` that returns the *floor* of the real value, that is, the largest integer not exceeding the real value. For example, the following properties hold, for any `r` and `r'` of type `real`:

```
3.14.Trunc == 3
(-2.5).Trunc == -3
-2.5.Trunc == -2
real(r.Trunc) <= r
r <= r' ==> r.Trunc <= r'.Trunc
```

Note in the third line that member access (like `.Trunc`) binds stronger than unary minus. The fourth line uses the conversion function `real` from `int` to `real`, as described in Section 7.0.

0.2. Characters

Dafny supports a type `char` of *characters*. Character literals are enclosed in single quotes, as in `'D'`. To write a single quote as a character literal, it is necessary to use an *escape sequence*. Escape sequences can also be used to write other characters. The supported escape sequences are as follows:

escape sequence	meaning
`\'`	the character `'`
`\"`	the character `"`
`\\`	the character `\`
`\0`	the null character, same as `\u0000`
`\n`	line feed
`\r`	carriage return
`\t`	horizontal tab
`\u`*xxxx*	universal character whose hexadecimal code is *xxxx*

The escape sequence for a double quote is redundant, because `'"'` and `'\"'` denote the same character—both forms are provided in order to support the same escape sequences as for string literals (Section 2.2.0). In the form `\u`*xxxx*, the `u` is always lower case, but the four hexadecimal digits are case insensitive.

Character values are ordered and can be compared using the standard relational operators:

operator	description
`<`	less than
`<=`	at most
`>=`	at least
`>`	greater than

Sequences of characters represent *strings*, as described in Section 2.2.0.

1. Type parameters

Many of the types (as well as functions and methods) in Dafny can be parameterized by types. These *type parameters* are typically declared inside angle brackets and can stand for any type. It is sometimes necessary to restrict these type parameters so that they can only be instantiated by certain families of types. As such, Dafny distinguishes types that support the equality operation not only in ghost contexts but also in compiled contexts. To indicate that a type parameter is restricted to such *equality supporting* types, the name of the type parameter takes the suffix "`(==)`".[0] For example,

```
method Compare<T(==)>(a: T, b: T) returns (eq: bool)
```

```
{
  if a == b { eq := true; } else { eq := false; }
}
```

is a method whose type parameter is restricted to equality-supporting types. Again, note that *all* types support equality in *ghost* contexts; the difference is only for non-ghost (that is, compiled) code. Co-inductive datatypes, function types, as well as inductive datatypes with ghost parameters are examples of types that are not equality supporting.

Dafny has some inference support that makes certain signatures less cluttered (described in a different part of the Dafny language reference). In some cases, this support will infer that a type parameter must be restricted to equality-supporting types, in which case Dafny adds the "(==)" automatically.

2. Collection types

Dafny offers several built-in collection types.

2.0. Sets

For any type `T`, each value of type `set<T>` is a finite set of `T` values. Set membership is determined by equality in the type `T`, so `set<T>` can be used in a non-ghost context only if `T` is equality supporting.

A set can be formed using a *set display* expression, which is a possibly empty, unordered, duplicate-insensitive list of expressions enclosed in curly braces. To illustrate,

```
{}        {2, 7, 5, 3}        {4+2, 1+5, a*b}
```

are three examples of set displays. There is also a *set comprehension* expression (with a binder, like in logical quantifications), described in a different part of the Dafny language reference.

In addition to equality and disequality, set types support the following relational operations:

operator	description
`<`	proper subset
`<=`	subset
`>=`	superset
`>`	proper superset

Like the arithmetic relational operators, these operators are chaining.

Sets support the following binary operators, listed in order of increasing binding power:

operator	description
`!!`	disjointness
`+`	set union
`-`	set difference
`*`	set intersection

The associaivity rules of `+`, `-`, and `*` are like those of the arithmetic operators with the same names. The expression `A !! B`, whose binding power is the same as equality (but which neither associates nor chains with equality), says that sets `A` and `B` have no elements in common, that is, it is equivalent to

```
A * B == {}
```

However, the disjointness operator is chaining, so `A !! B !! C !! D` means:

```
A * B == {} && (A + B) * C == {} && (A + B + C) * D == {}
```

In addition, for any set `s` of type `set<T>` and any expression `e` of type `T`, sets support the following operations:

expression	description

`|s|`	set cardinality
`e in s`	set membership
`e !in s`	set non-membership

The expression `e !in s` is a syntactic shorthand for `!(e in s)`.

2.1. Multisets

A *multiset* is similar to a set, but keeps track of the multiplicity of each element, not just its presence or absence. For any type `T`, each value of type `multiset⟨T⟩` is a map from `T` values to natural numbers denoting each element's multiplicity. Multisets in Dafny are finite, that is, they contain a finite number of each of a finite set of elements. Stated differently, a multiset maps only a finite number of elements to non-zero (finite) multiplicities.

Like sets, multiset membership is determined by equality in the type `T`, so `multiset⟨T⟩` can be used in a non-ghost context only if `T` is equality supporting.

A multiset can be formed using a *multiset display* expression, which is a possibly empty, unordered list of expressions enclosed in curly braces after the keyword `multiset`. To illustrate,

```
multiset{}    multiset{0, 1, 1, 2, 3, 5}    multiset{4+2, 1+5, a*b}
```

are three examples of multiset displays. There is no multiset comprehension expression.

In addition to equality and disequality, multiset types support the following relational operations:

operator	description
`<`	proper multiset subset
`<=`	multiset subset
`>=`	multiset superset
`>`	proper multiset superset

Like the arithmetic relational operators, these operators are chaining.

Multisets support the following binary operators, listed in order of increasing binding power:

operator	description
`!!`	multiset disjointness
`+`	multiset union
`-`	multiset difference
`*`	multiset intersection

The associaivity rules of `+`, `-`, and `*` are like those of the arithmetic operators with the same names. The expression `A !! B` says that multisets `A` and `B` have no elements in common, that is, it is equivalent to

```
A * B == multiset{}
```

Like the analogous set operator, `!!` is chaining.

In addition, for any multiset `s` of type `multiset⟨T⟩`, expression `e` of type `T`, and non-negative integer-based numeric `n`, multisets support the following operations:

expression	description
`|s|`	multiset cardinality
`e in s`	multiset membership
`e !in s`	multiset non-membership
`s[e]`	multiplicity of `e` in `s`
`s[e := n]`	multiset update (change of multiplicity)

The expression `e in s` returns `true` if and only if `s[e] != 0`. The expression `e !in s` is a syntactic shorthand for `!(e in s)`. The expression `s[e := n]` denotes a multiset like `s`, but where the multiplicity of element `e` is `n`. Note that the multiset update `s[e := 0]` results in a multiset like `s` but without any occurrences of `e` (whether or not `s` has occurrences of `e` in the first place). As another example, note that `s - multiset{e}` is equivalent to:

```
if e in s then s[e := s[e] - 1] else s
```

2.2. Sequences

For any type `T`, a value of type `seq⟨T⟩` denotes a *sequence* of `T` elements, that is, a mapping from a finite set of consecutive natural numbers (called *indicies*) to `T` values. (Thinking of it as a map, a sequence is therefore something of a dual of a multiset.)

A sequence can be formed using a *sequence display* expression, which is a possibly empty, ordered list of expressions enclosed in square brackets. To illustrate,

```
[]        [3, 1, 4, 1, 5, 9, 3]        [4+2, 1+5, a*b]
```

are three examples of sequence displays. There is no sequence comprehension expression.

In addition to equality and disequality, sequence types support the following relational operations:

operator	description
`<`	proper prefix
`<=`	prefix

Like the arithmetic relational operators, these operators are chaining. Note the absence of `>` and `>=`.

Sequences support the following binary operator:

operator	description
`+`	concatenation

Operator `+` is associative, like the arithmetic operator with the same name.

In addition, for any sequence `s` of type `seq⟨T⟩`, expression `e` of type `T`, integer-based numeric `i` satisfying `0 <= i < |s|`, and integer-based numerics `lo` and `hi` satisfying `0 <= lo <= hi <= |s|`, sequences support the following operations:

expression	description
`|s|`	sequence length
`s[i]`	sequence selection
`s[i := e]`	sequence update
`e in s`	sequence membership
`e !in s`	sequence non-membership
`s[lo..hi]`	subsequence
`s[lo..]`	drop
`s[..hi]`	take
`s[`*`slices`*`]`	slice
`multiset(s)`	sequence conversion to a `multiset⟨T⟩`

Expression `s[i := e]` returns a sequence like `s`, except that the element at index `i` is `e`. The expression `e in s` says there exists an index `i` such that `s[i] == e`. It is allowed in non-ghost contexts only if the element type `T` is equality supporting. The expression `e !in s` is a syntactic shorthand for `!(e in s)`.

Expression `s[lo..hi]` yields a sequence formed by taking the first `hi` elements and then dropping the first `lo` elements. The resulting sequence thus has length `hi - lo`. Note that `s[0..|s|]` equals `s`. If the upper bound is omitted, it defaults to `|s|`, so `s[lo..]` yields the sequence formed by dropping the first `lo` elements of `s`. If the lower bound is omitted, it defaults to `0`, so `s[..hi]` yields the sequence formed by taking the first `hi` elements of `s`.

In the sequence slice operation, *`slices`* is a nonempty list of length designators separated and optionally terminated by a colon, and there is at least one colon. Each length designator is a non-negative integer-based numeric, whose sum is no greater than `|s|`. If there are *k* colons, the operation produces *k* consecutive subsequences from `s`, each of the length indicated by the corresponding length designator, and returns these as a sequence of sequences.[1] If *slices* is terminated by a colon, then the length of the last slice extends until the end of s, that is, its length is |s| minus the sum of the given length designators. For example, the following equalities hold, for any sequence s of length at least 10:

```
var t := [3.14, 2.7, 1.41, 1985.44, 100.0, 37.2][1:0:3];
assert |t| == 3 && t[0] == [3.14] && t[1] == [];
assert t[2] == [2.7, 1.41, 1985.44];
var u := [true, false, false, true][1:1:];
assert |u| == 3 && u[0][0] && !u[1][0] && u[2] == [false, true];
assert s[10:][0] == s[..10];
assert s[10:][1] == s[10..];
```

The operation multiset(s) yields the multiset of elements of sequence s. It is allowed in non-ghost contexts only if the element type T is equality supporting.

2.2.0. Strings

A special case of a sequence type is seq⟨char⟩, for which Dafny provides a synonym: string. Strings are like other sequences, but provide additional syntax for sequence display expressions, namely *string literals*. There are two forms of the syntax for string literals: the *standard form* and the *verbatim form*.

String literals of the standard form are enclosed in double quotes, as in "Dafny". To include a double quote in such a string literal, it is necessary to use an escape sequence. Escape sequences can also be used to include other characters. The supported escape sequences are the same as those for character literals, see Section 0.2. For example, the Dafny expression "say \"yes\"" represents the string say "yes". The escape sequence for a single quote is redundant, because "'" and "\'" denote the same string —both forms are provided in order to support the same escape sequences as for character literals.

String literals of the verbatim form are bracketed by @" and ", as in @"Dafny". To include a double quote in such a string literal, it is necessary to use the escape sequence "", that is, to write the character twice. In the verbatim form, there are no other escape sequences. Even characters like newline can be written inside the string literal (hence spanning more than one line in the program text).

For example, the following three expressions denote the same string:

```
"C:\\tmp.txt"
@"C:\tmp.txt"
['C', ':', '\\', 't', 'm', 'p', '.', 't', 'x', 't']
```

Since strings are sequences, the relational operators < and <= are defined on them. Note, however, that these operators still denote proper prefix and prefix, respectively, not some kind of alphabetic comparison as might be desireable, for example, when sorting strings.

2.3. Finite and Infinite Maps

For any types T and U, a value of type map⟨T,U⟩ denotes a *(finite) map* from T to U. In other words, it is a look-up table indexed by T. The *domain* of the map is a finite set of T values that have associated U values. Since the keys in the domain are compared using equality in the type T, type map⟨T,U⟩ can be used in a non-ghost context only if T is equality supporting.

Similarly, for any types T and U, a value of type imap⟨T,U⟩ denotes a *(possibly) infinite map*. In most regards, imap⟨T,U⟩ is like map⟨T,U⟩, but a map of type imap⟨T,U⟩ is allowed to have an infinite domain.

A map can be formed using a *map display* expression, which is a possibly empty, ordered list of *maplets*, each maplet having the form t := u where t is an expression of type T and u is an expression of type U, enclosed in square brackets after the keyword map. To illustrate,

```
map[]    map[20 := true, 3 := false, 20 := false]    map[a+b := c+d]
```

are three examples of map displays. By using the keyword imap instead of `map`, the map produced will be of type `imap⟨T,U⟩` instead of `map⟨T,U⟩`. Note that an infinite map (`imap`) is allowed to have a finite domain, whereas a finite map (`map`) is not allowed to have an infinite domain. If the same key occurs more than once, only the last occurrence appears in the resulting map.[2] There is also a *map comprehension expression*, explained in a different part of the Dafny language reference.

For any map `fm` of type `map⟨T,U⟩`, any map `m` of type `map⟨T,U⟩` or `imap⟨T,U⟩`, any expression `t` of type `T`, any expression `u` of type `U`, and any `d` in the domain of `m` (that is, satisfying `d in m`), maps support the following operations:

expression	description
`|fm|`	map cardinality
`m[d]`	map selection
`m[t := u]`	map update
`t in m`	map domain membership
`t !in m`	map domain non-membership

`|fm|` denotes the number of mappings in `fm`, that is, the cardinality of the domain of `fm`. Note that the cardinality operator is not supported for infinite maps. Expression `m[d]` returns the `U` value that `m` associates with `d`. Expression `m[t := u]` is a map like `m`, except that the element at key `t` is `u`. The expression `t in m` says `t` is in the domain of `m` and `t !in m` is a syntactic shorthand for `!(t in m)`.[3]

Here is a small example, where a map `cache` of type `map⟨int,real⟩` is used to cache computed values of Joule-Thomson coefficients for some fixed gas at a given temperature:

```
if K in cache {  // check if temperature is in domain of cache
  coeff := cache[K];  // read result in cache
} else {
  coeff := ComputeJouleThomsonCoefficient(K);  // do expensive computation
  cache := cache[K := coeff];  // update the cache
}
```

3. Types that stand for other types

It is sometimes useful to know a type by several names or to treat a type abstractly.

3.0. Type synonyms

A *type synonym* declaration:

```
type Y⟨T⟩ = G
```

declares `Y⟨T⟩` to be a synonym for the type `G`. Here, `T` is a nonempty list of type parameters (each of which is optionally designated with the suffix "`(==)`"), which can be used as free type variables in `G`. If the synonym has no type parameters, the "`⟨T⟩`" is dropped. In all cases, a type synonym is just a synonym. That is, there is never a difference, other than possibly in error messages produced, between `Y⟨T⟩` and `G`.

For example, the names of the following type synonyms may improve the readability of a program:

```
type Replacements⟨T⟩ = map⟨T,T⟩
type Vertex = int
```

As already described in Section 2.2.0, `string` is a built-in type synonym for `seq⟨char⟩`, as if it would have been declared as follows:

```
type string = seq⟨char⟩
```

3.1. Opaque types

A special case of a type synonym is one that is underspecified. Such a type is declared simply by:

```
type Y⟨T⟩
```

It is a known as an *opaque type*. Its definition can be revealed in a refining module. To indicate that `Y` designates an equality-supporting type, "`(==)`" can be written immediately following the name "`Y`".

For example, the declarations

```
type T
function F(t: T): T
```

can be used to model an uninterpreted function `F` on some arbitrary type `T`. As another example,

```
type Monad⟨T⟩
```

can be used abstractly to represent an arbitrary parameterized monad.

4. Datatypes

Dafny offers two kinds of algebraic datatypes, those defined inductively and those defined co-inductively. The salient property of every datatype is that each value of the type uniquely identifies one of the datatype's constructors and each constructor is injective in its parameters.

4.0. Inductive datatypes

The values of inductive datatypes can be seen as finite trees where the leaves are values of basic types, numeric types, reference types, co-inductive datatypes, or function types. Indeed, values of inductive datatypes can be compared using Dafny's well-founded `<` ordering.

An inductive datatype is declared as follows:

```
datatype D⟨T⟩ = Ctors
```

where *`Ctors`* is a nonempty `|`-separated list of *(datatype) constructors* for the datatype. Each constructor has the form:

```
C(params)
```

where *`params`* is a comma-delimited list of types, optionally preceded by a name for the parameter and a colon, and optionally preceded by the keyword `ghost`. If a constructor has no parameters, the parentheses after the constructor name can be omitted. If no constructor takes a parameter, the type is usually called an *enumeration*; for example:

```
datatype Friends = Agnes | Agatha | Jermaine | Jack
```

For every constructor `C`, Dafny defines a *discriminator* `C?`, which is a member that returns `true` if and only if the datatype value has been constructed using `C`. For every named parameter `p` of a constructor `C`, Dafny defines a *destructor* `p`, which is a member that returns the `p` parameter from the `C` call used to construct the datatype value; its use requires that `C?` holds. For example, for the standard `List` type

```
datatype List⟨T⟩ = Nil | Cons(head: T, tail: List⟨T⟩)
```

the following holds:

```
Cons(5, Nil).Cons? && Cons(5, Nil).head == 5
```

Note that the expression

```
Cons(5, Nil).tail.head
```

is not well-formed, since `Cons(5, Nil).tail` does not satisfy `Cons?`.

The names of the destructors must be unique across all the constructors of the datatype. A constructor can have the same name as the enclosing datatype; this is especially useful for single-constructor datatypes, which are often called *record types*. For example, a record type for black-and-white pixels might be represented as follows:

```
datatype Pixel = Pixel(x: int, y: int, on: bool)
```

To call a constructor, it is usually necessary only to mention the name of the constructor, but if this is ambiguous, it is always possible to qualify the name of constructor by the name of the datatype. For example, `Cons(5,` `Nil`) above can be written

```
List.Cons(5, List.Nil)
```

As an alternative to calling a datatype constructor explicitly, a datatype value can be constructed as a change in one parameter from a given datatype value using the *datatype update* expression. For any `d` whose type is a datatype that includes a constructor `C` that has a parameter (destructor) named `f` of type `T`, and any expression `t` of type `T`,

```
d[f := t]
```

constructs a value like `d` but whose `f` parameter is `t`. The operation requires that `d` satisfies `C?`. For example, the following equality holds:

```
Cons(4, Nil)[tail := Cons(3, Nil)] == Cons(4, Cons(3, Nil))
```

4.1. Tuple types

Dafny builds in record types that correspond to tuples and gives these a convenient special syntax, namely parentheses. For example, what might have been declared as:

```
datatype Pair⟨T,U⟩ = Pair(0: T, 1: U)
```

Dafny provides as the type `(T, U)` and the constructor `(t, u)`, as if the datatype's name were “” and its type arguments are given in round parentheses, and as if the constructor name were “”. Note that the destructor names are `0` and `1`, which are legal identifier names for members. For example, showing the use of a tuple destructor, here is a property that holds of 2-tuples (that is, *pairs*):

```
(5, true).1 == true
```

Dafny declares *n*-tuples where *n* is 0 or 2 or up. There are no 1-tuples, since parentheses around a single type or a single value have no semantic meaning. The 0-tuple type, `()`, is often known as the *unit type* and its single value, also written `()`, is known as *unit*.

4.2. Co-inductive datatypes

Whereas Dafny insists that there is a way to construct every inductive datatype value from the ground up, Dafny also supports *co-inductive datatypes*, whose constructors are evaluated lazily and hence allows infinite structures. A co-inductive datatype is declared using the keyword `codatatype`; other than that, it is declared and used like an inductive datatype.

For example,

```
codatatype IList⟨T⟩ = Nil | Cons(head: T, tail: IList⟨T⟩)
codatatype Stream⟨T⟩ = More(head: T, tail: Stream⟨T⟩)
codatatype Tree⟨T⟩ = Node(left: Tree⟨T⟩, value: T, right: Tree⟨T⟩)
```

declare possibly infinite lists (that is, lists that can be either finite or infinite), infinite streams (that is, lists that are always infinite), and infinite binary trees (that is, trees where every branch goes on forever), respectively.

5. Reference types

Dafny offers a host of *reference types*. These represent *references* to objects allocated dynamically in the program heap. To access the members of an object, a reference to (that is, a *pointer* to or *object identity* of) the object is *dereferenced*.

The special value `null` is part of every reference type.[4]

5.0. Classes

A *class* `C` is a reference type declared as follows:

```
class C⟨T⟩ extends J
```

```
{
  members
}
```

where the list of type parameters `T` is optional and so is “`extends J`”, which says that the class extends a trait `J`. The members of a class are *fields*, *functions*, and *methods*. These are accessed or invoked by dereferencing a reference to a `C` instance. A function or method is invoked on an *instance* of `C`, unless the function or method is declared `static`. Mechanically, this just means the method takes an implicit *receiver* parameter, namely, the instance used to access the member. In the specification and body of an instance function or method, the receiver parameter can be referred to explicitly by the keyword `this`. However, in such places, members of `this` can also be mentioned without any qualification. To illustrate, the qualified `this.f` and the unqualified `f` refer to the same field of the same object in the following example:

```
class C {
  var f: int;
  method Example() returns (b: bool)
  {
    b := f == this.f;
  }
}
```

so the method body always assigns `true` to the out-parameter `b`. There is no semantic difference between qualified and unqualified accesses to the same receiver and member.

A `C` instance is created using `new`, for example:

```
c := new C;
```

Note that `new` simply allocates a `C` object and returns a reference to it; the initial values of its fields are arbitrary values of their respective types. Therefore, it is common to invoke a method, known as an *initialization method*, immediately after creation, for example:

```
c := new C;
c.InitFromList(xs, 3);
```

When an initialization method has no out-parameters and modifies no more than `this`, then the two statements above can be combined into one:

```
c := new C.InitFromList(xs, 3);
```

Note that a class can contain several initialization methods, that these methods can be invoked at any time, not just as part of a `new`, and that `new` does not require that an initialization method be invoked at creation.

To write structured object-oriented programs, one often relies on that objects are constructed only in certain ways. For this purpose, Dafny provides *constructor (method)s*, which are a restricted form of initialization methods. A constructor is declared with the keyword `constructor` instead of `method`. When a class contains a constructor, every call to `new` for that class must be accompanied with a call to one of the constructors. Moreover, a constructor cannot be called at other times, only during object creation. Other than these restrictions, there is no semantic difference between using ordinary initialization methods and using constructors.

The Dafny design allows the constructors to be named, which promotes using names like `InitFromList` above. Still, many classes have just one constructor or have a typical constructor. Therefore, Dafny allows one *anonymous constructor*, that is, a constructor whose name is essentially “”. For example:

```
class Item {
  constructor (x: int, y: int)
  // ...
}
```

When invoking this constructor, the “`.`” is dropped, as in:

```
m := new Item(45, 29);
```

Note that an anonymous constructor is just one way to name a constructor; there can be other constructors as well.

5.1. Arrays

Dafny supports mutable fixed-length *array types* of any positive dimension. Array types are reference types.

5.1.0. One-dimensional arrays

A one-dimensional array of `n` `T` elements is created as follows:

```
a := new T[n];
```

The initial values of the array elements are arbitrary values of type `T`. The length of an array is retrieved using the immutable `Length` member. For example, the array allocated above satisfies:

```
a.Length == n
```

For any integer-based numeric `i` in the range `0 <= i < a.Length`, the *array selection* expression `a[i]` retrieves element `i` (that is, the element preceded by `i` elements in the array). The element stored at `i` can be changed to a value `t` using the array update statement:

```
a[i] := t;
```

Caveat: The type of the array created by `new T[n]` is `array⟨T⟩`. A mistake that is simple to make and that can lead to befuddlement is to write `array⟨T⟩` instead of `T` after `new`. For example, consider the following:

```
var a := new array⟨T⟩;
var b := new array⟨T⟩[n];
var c := new array⟨T⟩(n);  // resolution error
var d := new array(n);  // resolution error
```

The first statement allocates an array of type `array⟨T⟩`, but of unknown length. The second allocates an array of type `array⟨array⟨T⟩⟩` of length `n`, that is, an array that holds `n` values of type `array⟨T⟩`. The third statement allocates an array of type `array⟨T⟩` and then attempts to invoke an anonymous constructor on this array, passing argument `n`. Since `array` has no constructors, let alone an anonymous constructor, this statement gives rise to an error. If the type-parameter list is omitted for a type that expects type parameters, Dafny will attempt to fill these in, so as long as the `array` type parameter can be inferred, it is okay to leave off the "`⟨T⟩`" in the fourth statement above. However, as with the third statement, `array` has no anonymous constructor, so an error message is generated.

One-dimensional arrays support operations that convert a stretch of consecutive elements into a sequence. For any array `a` of type `array⟨T⟩`, integer-based numerics `lo` and `hi` satisfying `0 <= lo <= hi <= a.Length`, the following operations each yields a `seq⟨T⟩`:

expression	description
`a[lo..hi]`	subarray conversion to sequence
`a[lo..]`	drop
`a[..hi]`	take
`a[..]`	array conversion to sequence

The expression `a[lo..hi]` takes the first `hi` elements of the array, then drops the first `lo` elements thereof and returns what remains as a sequence. The resulting sequence thus has length `hi - lo`. The other operations are special instances of the first. If `lo` is omitted, it defaults to `0` and if `hi` is omitted, it defaults to `a.Length`. In the last operation, both `lo` and `hi` have been omitted, thus `a[..]` returns the sequence consisting of all the array elements of `a`.

The subarray operations are especially useful in specifications. For example, the loop invariant of a binary search algorithm that uses variables `lo` and `hi` to delimit the subarray where the search `key` may be still found can be expressed as follows:

```
key !in a[..lo] && key !in a[hi..]
```

Another use is to say that a certain range of array elements have not been changed since the beginning of a method:

```
a[lo..hi] == old(a[lo..hi])
```

or since the beginning of a loop:

```
ghost var prevElements := a[..];
while // ...
  invariant a[lo..hi] == prevElements[lo..hi];
{
  // ...
}
```

Note that the type of `prevElements` in this example is `seq⟨T⟩`, if `a` has type `array⟨T⟩`.

A final example of the subarray operation lies in expressing that an array's elements are a permutation of the array's elements at the beginning of a method, as would be done in most sorting algorithms. Here, the subarray operation is combined with the sequence-to-multiset conversion:

```
multiset(a[..]) == multiset(old(a[..]))
```

5.1.1. Multi-dimensional arrays

An array of 2 or more dimensions is mostly like a one-dimensional array, except that `new` takes more length arguments (one for each dimension), and the array selection expression and the array update statement take more indices. For example:

```
matrix := new T[m, n];
matrix[i, j], matrix[x, y] := matrix[x, y], matrix[i, j];
```

create a 2-dimensional array whose dimensions have lengths `m` and `n`, respectively, and then swaps the elements at `i,j` and `x,y`. The type of `matrix` is `array2⟨T⟩`, and similarly for higher-dimensional arrays (`array3⟨T⟩`, `array4⟨T⟩`, etc.). Note, however, that there is no type `array0⟨T⟩`, and what could have been `array1⟨T⟩` is actually named just `array⟨T⟩`.

The `new` operation above requires `m` and `n` to be non-negative integer-based numerics. These lengths can be retrieved using the immutable fields `Length0` and `Length1`. For example, the following holds of the array created above:

```
matrix.Length0 == m && matrix.Length1 == n
```

Higher-dimensional arrays are similar (`Length0`, `Length1`, `Length2`, ...). The array selection expression and array update statement require that the indices are in bounds. For example, the swap statement above is well-formed only if:

```
0 <= i < matrix.Length0 && 0 <= j < matrix.Length1 &&
0 <= x < matrix.Length0 && 0 <= y < matrix.Length1
```

In contrast to one-dimensional arrays, there is no operation to convert stretches of elements from a multi-dimensional array to a sequence.

5.2. Traits

A *trait* is an "abstract superclass", or call it an "interface" or "mixin". Traits are new to Dafny and are likely to evolve for a while.

The declaration of a trait is much like that of a class:

```
trait J
{
  members
}
```

where *`members`* can include fields, functions, and methods, but no constructor methods. The functions and methods are allowed to be declared `static`.

A reference type `C` that extends a trait `J` is assignable to `J`, but not the other way around. The members of `J` are available as members of `C`. A member in `J` is not allowed to be redeclared in `C`, except if the member is a non-`static` function or method without a body in `J`. By doing so, type `C` can supply a stronger specification and a body for the member.

`new` is not allowed to be used with traits. Therefore, there is no object whose allocated type is a trait. But there can of course be objects of a class `C` that implements a trait `J`, and a reference to such a `C` object can be used as a value of type `J`.

As an example, the following trait represents movable geometric shapes:

```
trait Shape
{
  function method Width(): real
    reads this
  method Move(dx: real, dy: real)
    modifies this
  method MoveH(dx: real)
    modifies this
  {
    Move(dx, 0.0);
  }
}
```

Members `Width` and `Move` are *abstract* (that is, body less) and can be implemented differently by different classes that extend the trait. The implementation of method `MoveH` is given in the trait and thus gets used by all classes that extend `Shape`. Here are two classes that each extends `Shape`:

```
class UnitSquare extends Shape
{
  var x: real, y: real;
  function method Width(): real {  // note the empty reads clause
    1.0
  }
  method Move(dx: real, dy: real)
    modifies this
  {
    x, y := x + dx, y + dy;
  }
}
class LowerRightTriangle extends Shape
{
  var xNW: real, yNW: real, xSE: real, ySE: real;
  function method Width(): real
    reads this
  {
    xSE - xNW
  }
  method Move(dx: real, dy: real)
    modifies this
  {
    xNW, yNW, xSE, ySE := xNW + dx, yNW + dy, xSE + dx, ySE + dy;
  }
}
```

Note that the classes can declare additional members, that they supply implementations for the abstract members of the trait, that they repeat the member signatures, and that they are responsible for providing their own member specifications that both strengthen the corresponding specification in the trait and are satisfied by the provided body. Finally, here is some code that creates two class instances and uses them together as shapes:

```
var myShapes: seq⟨Shape⟩;
var A := new UnitSquare;
myShapes := [A];
var tri := new LowerRightTriangle;
myShapes := myShapes + [tri];  // myShapes contains two Shape values, of different classes
myShapes[1].MoveH(myShapes[0].Width());  // move shape 1 to the right by the width of shap
```

5.3. Type `object`

There is a built-in reference type `object` that is like a supertype of all reference types.[5] The purpose of type `object` is to enable a uniform treatment of *dynamic frames*. In particular, it is useful to keep a ghost field (typically named `Repr` for "representation") of type `set⟨object⟩`.

5.4. Iterator types

An *iterator* provides a programming abstraction for writing code that iteratively returns elements. These CLU-style iterators are *co-routines* in the sense that they keep track of their own program counter and control can be transferred into and out of the iterator body.

An iterator is declared as follows:

```
iterator Iter⟨T⟩(in-params) yields (yield-params)
  specification
{
  body
}
```

where `T` is a list of type parameters (as usual, if there are no type parameters, "`⟨T⟩`" is omitted). This declaration gives rise to a reference type with the same name, `Iter⟨T⟩`. In the signature, in-parameters and yield-parameters are the iterator's analog of a method's in-parameters and out-parameters. The difference is that the out-parameters of a method are returned to a caller just once, whereas the yield-parameters of an iterator are returned each time the iterator body performs a `yield`. The details of the specification are described in a different part of the Dafny language reference. The body consists of statements, like in a method body, but with the availability also of `yield` statements.

From the perspective of an iterator client, the `iterator` declaration can be understood as generating a class `Iter⟨T⟩` with various members, a simplified version of which is described next.

The `Iter⟨T⟩` class contains an anonymous constructor whose parameters are the iterator's in-parameters:

```
predicate Valid()
constructor (in-params)
  modifies this
  ensures Valid()
```

An iterator is created using `new` and this anonymous constructor. For example, an iterator willing to return ten consecutive integers from `start` can be declared as follows:

```
iterator Gen(start: int) yields (x: int)
{
  var i := 0;
  while i < 10 {
    x := start + i;
    yield;
    i := i + 1;
  }
}
```

An instance of this iterator is created using:

```
iter := new Gen(30);
```

The predicate `Valid()` says when the iterator is in a state where one can attempt to compute more elements. It is a postcondition of the constructor and occurs in the specification of the `MoveNext` member:

```
method MoveNext() returns (more: bool)
  requires Valid()
  modifies this
  ensures more ==> Valid()
```

Note that the iterator remains valid as long as `MoveNext` returns `true`. Once `MoveNext` returns `false`, the `MoveNext` method can no longer be called. Note, the client is under no obligation to keep calling `MoveNext` until it returns `false`, and the body of the iterator is allowed to keep returning elements forever.

The in-parameters of the iterator are stored in immutable fields of the iterator class. To illustrate in terms of the example above, the iterator class `Gen` contains the following field:

```
var start: int;
```

The yield-parameters also result in members of the iterator class:

```
var x: int;
```

These fields are set by the `MoveNext` method. If `MoveNext` returns `true`, the latest yield values are available in these fields and the client can read them from there.

To aid in writing specifications, the iterator class also contains ghost members that keep the history of values returned by `MoveNext`. The names of these ghost fields follow the names of the yield-parameters with an “s” appended to the name (to suggest plural). Name checking rules make sure these names do not give rise to ambiguities. The iterator class for `Gen` above thus contains:

```
ghost var xs: seq⟨int⟩;
```

These history fields are changed automatically by `MoveNext`, but are not assignable by user code.

Finally, the iterator class contains some special fields for use in specifications. In particular, the iterator specification gets recorded in the following immutable fields:

```
ghost var _reads: set⟨object⟩;
ghost var _modifies: set⟨object⟩;
ghost var _decreases0: T0;
ghost var _decreases1: T1;
// ...
```

where there is a `_decreases`*i*`: T`*i* field for each component of the iterator's `decreases` clause.[6] In addition, there is a field:

```
ghost var _new: set⟨object⟩;
```

to which any objects allocated on behalf of the iterator body get added. The iterator body is allowed to remove elements from the `_new` set, but cannot by assignment to `_new` add any elements.

Note, in the precondition of the iterator, which is to hold upon construction of the iterator, the in-parameters are indeed in-parameters, not fields of `this`.

5.5. Async-task types

Another experimental feature in Dafny that is likely to undergo some evolution is *asynchronous methods*. When an asynchronous method is called, it does not return values for the out-parameters, but instead returns an instance of an *async-task type*. An asynchronous method declared in a class `C` with the following signature:

```
async method AM⟨T⟩(in-params) returns (out-params)
```

also gives rise to an async-task type `AM⟨T⟩` (outside the enclosing class, the name of the type needs the qualification `C.AM⟨T⟩`). The async-task type is a reference type and can be understood as a class with various members, a simplified version of which is described next.

Each in-parameter `x` of type `X` of the asynchronous method gives rise to a immutable ghost field of the async-task type:

```
ghost var x: X;
```

Each out-parameter `y` of type `Y` gives rise to a field

```
var y: Y;
```

These fields are changed automatically by the time the asynchronous method is successfully awaited, but are not assignable by user code.

The async-task type also gets a number of special fields that are used to keep track of dependencies, outstanding tasks, newly allocated objects, etc. These fields will be described in more detail as the design of asynchronous methods evolves.

6. Function types

Functions are first-class values in Dafny. Function types have the form `(T) -> U` where `T` is a comma-delimited list of types and `U` is a type. `T` is called the function's *domain type(s)* and `U` is its *range type*. For example, the type of a function

```
function F(x: int, b: bool): real
```

is `(int, bool) -> real`. Parameters are not allowed to be ghost.

To simplify the appearance of the basic case where a function's domain consist of a list of exactly one type, the parentheses around the domain type can be dropped in this case, as in `T -> U`. This innocent simplification requires additional explanation in the case where that one type is a tuple type, since tuple types are also written with enclosing parentheses. If the function takes a single argument that is a tuple, an additional set of parentheses is needed. For example, the function

```
function G(pair: (int, bool)): real
```

has type `((int, bool)) -> real`. Note the necessary double parentheses. Similarly, a function that takes no arguments is different from one that takes a 0-tuple as an argument. For instance, the functions

```
function NoArgs(): real
function Z(unit: ()): real
```

have types `() -> real` and `(()) -> real`, respectively.

The function arrow, `->`, is right associative, so `A -> B -> C` means `A -> (B -> C)`. The other association requires explicit parentheses: `(A -> B) -> C`.

Note that the receiver parameter of a named function is not part of the type. Rather, it is used when looking up the function and can then be thought of as being captured into the function definition. For example, suppose function `F` above is declared in a class `C` and that `c` references an object of type `C`; then, the following is type correct:

```
var f: (int, bool) -> real := c.F;
```

whereas it would have been incorrect to have written something like:

```
var f': (C, int, bool) -> real := F;  // not correct
```

Outside its type signature, each function value has three properties, described next.

Every function implicitly takes the heap as an argument. No function ever depends on the *entire* heap, however. A property of the function is its declared upper bound on the set of heap locations it depends on for a given input. This lets the verifier figure out that certain heap modifications have no effect on the value returned by a certain function. For a function `f: T -> U` and a value `t` of type `T`, the dependency set is denoted `f.reads(t)` and has type `set⟨object⟩`.

The second property of functions stems from that every function is potentially *partial*. In other words, a property of a function is its *precondition*. For a function `f: T -> U`, the precondition of `f` for a parameter value `t` of type `T` is denoted `f.requires(t)` and has type `bool`.

The third property of a function is more obvious—the function's body. For a function `f: T -> U`, the value that the function yields for an input `t` of type `T` is denoted `f(t)` and has type `U`.

Note that `f.reads` and `f.requires` are themselves functions. Suppose `f` has type `T -> U` and `t` has type `T`. Then, `f.reads` is a function of type `T -> set⟨object⟩` whose `reads` and `requires` properties are:

```
f.reads.reads(t) == f.reads(t)
f.reads.requires(t) == true
```

`f.requires` is a function of type `T -> bool` whose `reads` and `requires` properties are:

```
f.requires.reads(t) == f.reads(t)
f.requires.requires(t) == true
```

6.0. Lambda expressions

In addition to named functions, Dafny supports expressions that define functions. These are called *lambda (expression)s* (some languages know them as *anonymous functions*). A lambda expression has the form:

```
(params) specification => body
```

where *params* is a comma-delimited list of parameter declarations, each of which has the form `x` or `x: T`. The type `T` of a parameter can be omitted when it can be inferred. If the identifier `x` is not needed, it can be replaced by "_". If *params* consists of a single parameter `x` (or _) without an explicit type, then the parentheses can be dropped; for example, the function that returns the successor of a given integer can be written as the following lambda expression:

```
x => x + 1
```

The *specification* is a list of clauses `requires E` or `reads W`, where `E` is a boolean expression and `W` is a frame expression.

body is an expression that defines the function's return value. The body must be well-formed for all possible values of the parameters that satisfy the precondition (just like the bodies of named functions and methods). In some cases, this means it is necessary to write explicit `requires` and `reads` clauses. For example, the lambda expression

```
x requires x != 0 => 100 / x
```

would not be well-formed if the `requires` clause were omitted, because of the possibility of division-by-zero.

In settings where functions cannot be partial and there are no restrictions on reading the heap, the *eta expansion* of a function `F: T -> U` (that is, the wrapping of `F` inside a lambda expression in such a way that the lambda expression is equivalent to `F`) would be written `x => F(x)`. In Dafny, eta expansion must also account for the precondition and reads set of the function, so the eta expansion of `F` looks like:

```
x requires F.requires(x) reads F.reads(x) => F(x)
```

7. Newtypes

A new numeric type can be declared with the *newtype* declaration[7]

```
newtype N = x: M | Q
```

where `M` is a numeric type and `Q` is a boolean expression that can use `x` as a free variable. If `M` is an integer-based numeric type, then so is `N`; if `M` is real-based, then so is `N`. If the type `M` can be inferred from `Q`, the ": `M`" can be omitted. If `Q` is just `true`, then the declaration can be given simply as:

```
newtype N = M
```

Type `M` is known as the *base type* of `N`.

A newtype is a numeric type that supports the same operations as its base type. The newtype is distinct from and incompatible with other numeric types; in particular, it is not assignable to its base type without an explicit conversion. An important difference between the operations on a newtype and the operations on its base type is that the newtype operations are defined only if the result satisfies the predicate `Q`, and likewise for the literals of the newtype.[8]

For example, suppose `lo` and `hi` are integer-based numerics that satisfy `0 <= lo <= hi` and consider the following code fragment:

```
var mid := (lo + hi) / 2;
```

If `lo` and `hi` have type `int`, then the code fragment is legal; in particular, it never overflows, since `int` has no upper bound. In contrast, if `lo` and `hi` are variables of a newtype `int32` declared as follows:

```
newtype int32 = x | -0x80000000 <= x < 0x80000000
```

then the code fragment is erroneous, since the result of the addition may fail to satisfy the predicate in the definition of `int32`. The code fragment can be rewritten as

```
var mid := lo + (hi - lo) / 2;
```

in which case it is legal for both `int` and `int32`.

Since a newtype is incompatible with its base type and since all results of the newtype's operations are members of the newtype, a compiler for Dafny is free to specialize the run-time representation of the newtype. For example, by scrutinizing the definition of `int32` above, a compiler may decide to store `int32` values using signed 32-bit integers in the target hardware.

Note that the bound variable `x` in `Q` has type `M`, not `N`. nConsequently, it may not be possible to state `Q` about the `N` value. For example, consider the following type of 8-bit 2's complement integers:

```
newtype int8 = x: int | -128 <= x < 128
```

and consider a variable `c` of type `int8`. The expression

```
-128 <= c < 128
```

is not well-defined, because the comparisons require each operand to have type `int8`, which means the literal `128` is checked to be of type `int8`, which it is not. A proper way to write this expression would be to use a conversion operation, described next, on `c` to convert it to the base type:

```
-128 <= int(c) < 128
```

There is a restriction that the value `0` must be part of every newtype.[9]

7.0. Numeric conversion operations

For every numeric type `N`, there is a conversion function with the same name. It is a partial identity function. It is defined when the given value, which can be of any numeric type, is a member of the type converted to. When the conversion is from a real-based numeric type to an integer-based numeric type, the operation requires that the real-based argument has no fractional part. (To round a real-based numeric value down to the nearest integer, use the `.Trunc` member, see Section 0.1.)

To illustrate using the example from above, if `lo` and `hi` have type `int32`, then the code fragment can legally be written as follows:

```
var mid := (int(lo) + int(hi)) / 2;
```

where the type of `mid` is inferred to be `int`. Since the result value of the division is a member of type `int32`, one can introduce yet another conversion operation to make the type of `mid` be `int32`:

```
var mid := int32((int(lo) + int(hi)) / 2);
```

If the compiler does specialize the run-time representation for `int32`, then these statements come at the expense of two, respectively three, run-time conversions.

8. Subset types

A *subset type* is a restricted use of an existing type, called the *base type* of the subset type. A subset type is like a combined use of the base type and a predicate on the base type.

An assignment from a subset type to its base type is always allowed. An assignment in the other direction, from the base type to a subset type, is allowed provided the value assigned does indeed satisfy the predicate of the subset type. (Note, in contrast, assignments between a newtype and its base type are never allowed, even if the value assigned is a value of the target type. For such assignments, an explicit conversion must be used, see Section 7.0.)

Dafny supports one subset type, namely the built-in type `nat`, whose base type is `int`.[10] Type `nat` designates the non-negative subrange of `int`. A simple example that puts subset type `nat` to good use is the standard

Fibonacci function:

```
function Fib(n: nat): nat
{
  if n < 2 then n else Fib(n-2) + Fib(n-1)
}
```

An equivalent, but clumsy, formulation of this function (modulo the wording of any error messages produced at call sites) would be to use type `int` and to write the restricting predicate in pre- and postconditions:

```
function Fib(n: int): int
  requires 0 <= n  // the function argument must be non-negative
  ensures 0 <= Fib(n)  // the function result is non-negative
{
  if n < 2 then n else Fib(n-2) + Fib(n-1)
}
```

Type inference will never infer the type of a variable to be a subset type. It will instead infer the type to be the base type of the subset type. For example, the type of `x` in

```
forall x :: P(x)
```

will be `int`, even if predicate `P` declares its argument to have type `nat`.

Acknowledgments

This document has been improved as a result of helpful comments from Nadia Polikarpova and Paqui Lucio.

---

0. Being equality-supporting is just one of many *modes* that one can imagine types in a rich type system to have. For example, other modes could include having a total order, being zero-initializable, and possibly being uninhabited. If Dafny were to support more modes in the future, the “`( )`”-suffix syntax may be extended. For now, the suffix can only indicate the equality-supporting mode. ↩

1. Now that Dafny supports built-in tuples, the plan is to change the sequence slice operation to return not a sequence of subsequences, but a tuple of subsequences. ↩

2. This is likely to change in the future to disallow multiple occurrences of the same key. ↩

3. This is likely to change in the future as follows: The `in` and `!in` operations will no longer be supported on maps. Instead, for any map `m`, `m.Domain` will return its domain as a set and `m.Range` will return, also as a set, the image of `m` under its domain. ↩

4. This will change in a future version of Dafny that will support both nullable and (by default) non-null reference types. ↩

5. Soon, `object` will be made into a built-in trait rather than being a built-in special class. When this happens, it will no longer be possible to do `new object`. The current compiler restriction that `object` cannot be used as a type parameter will then also go away. ↩

6. It would make sense to rename the special fields `_reads` and `_modifies` to have the same names as the corresponding keywords, `reads` and `modifies`, as is done for function values. Also, the various `_decreases`*i* fields can combined into one field named `decreases` whose type is a *n*-tuple.

   ↩

7. Should `newtype` perhaps be renamed to `numtype`? ↩

8. Would it be useful to also automatically define `predicate N?(m: M) { Q }`? ↩

9. The restriction is due to a current limitation in the compiler. This will change in the future and will also open up the possibility for subset types and non-null reference types. ↩

10. A future version of Dafny will support user-defined subset types. ↩