Microsoft Research

Search Microsoft Research

Videos Projects Publications People Downloads

Home Our Research Connections Careers Hub

Worldwide Labs | Research Areas | Research Groups

Be an insider
The Inside Microsoft Research blog

> Projects > Dafny: A language and program verifier for functional correctness > Dafny Quick Reference

Dafny Quick Reference

This page illustrates many of the most common language features in Dafny.

Programs

At the top level, a Dafny program (stored as a file with extension .dfy) is a set of declarations. The declarations introduce fields, methods, and functions, as well as classes and inductive datatypes, where the order of introduction is irrelevant. A class also contains a set of declarations, introducing fields, methods, and functions. Fields, methods, and functions declared outside a class go into an implicit class called _default, giving the appearance of the program having global variables, procedures, and functions. If the program contains a unique parameter-less method called Main, then program execution starts there, but it is not necessary to have a main method to do verification.

Comments start with // and go to the end of the line, or start with /* and end with */ and can be nested.

Fields

A field x of some type T is declared as:

```
var x: T;
```

Unlike for local variables and bound variables, the type is required and will not be inferred. The field can be declared to be a ghost field by preceding the declaration with the keyword ghost. Dafny's types include bool for booleans, int for mathematical (that is, unbounded) integers, user-defined classes and inductive datatypes, set<T> for a finite mathematical (immutable) set of T values (where T is any type), and seq<T> for a mathematical (immutable) sequence of T values. In addition, there are array types (which are like predefined "class" types) of one and more dimensions, written array<T>, array2<T>, array3<T>, …. The type object is a supertype of all class types, that is, an object denotes any reference, including null. Finally, the type nat denotes a subrange of int, namely the non-negative integers.

Methods

A method declaration has the form:

```
method M(a: A, b: B, c: C) returns (x: X, y: Y, z: Y)
  requires Pre;
  modifies Frame;
  ensures Post;
  decreases Rank;
{
  Body
}
```

where a, b, c are the method's in-parameters, x, y, z are the method's out-parameters, Pre is a boolean expression denoting the method's precondition, Frame denotes a set of objects whose fields may be updated by the method, Post is a boolean expression denoting the method's postcondition, Rank is the method's variant function, and Body is a statement that implements the method. Frame can be a list of expressions, each of which is a set of objects or a single object, the latter standing for the singleton set consisting of that one object. The method's frame is the union of these sets, plus the set of objects allocated by the method body. For example, if c and d are parameters of a class type C, then

```
modifies {c, d};
```

```
modifies {c} + {d};

modifies c, {d};

modifies c, d;
```

all mean the same thing.

If omitted, the pre- and postconditions default to true and the frame defaults to the empty set. The variant function is a list of expressions, denoting the unending lexicographic tuple consisting of the given expressions followed implicitly by "top" elements. If omitted, Dafny will guess a variant function for the method, namely the lexicographic tuple that starts with the list of the method's in-parameters.

A method can be declared as ghost by preceding the declaration with the keyword ghost. By default, a method has an implicit receiver parameter, this. This parameter can be removed by preceding the method declaration with the keyword static. A static method M in a class C can be invoked by C.M(…).

In a class, a method can be declared to be a constructor method by replacing the keyword method with the keyword constructor. A constructor can only be called at the time an object is allocated (see object-creation examples below), and for a class that contains one or more constructors, object creation must be done in conjunction with a call to a constructor.

Functions

A function declaration has the form:

```
function F(a: A, b: B, c: C): T
  requires Pre;
  reads Frame;
  ensures Post;
  decreases Rank;
{
  Body
}
```

where a, b, c are the method's parameters, T is the type of the function's result, Pre is a boolean expression denoting the function's precondition, Frame denotes a set of objects whose fields the function body may depend on, Post is a boolean expression denoting the function's postcondition, Rank is the function's variant function, and Body is an expression that defines the function. The precondition allows a function to be partial, that is, the precondition says when the function is defined (and Dafny will verify that every use of the function meets the precondition). The postcondition is usually not needed, since the body of the function gives the full definition. However, the postcondition can be a convenient place to declare properties of the function that may require an inductive proof to establish. For example:

```
function Factorial(n: int): int
  requires 0 <= n;
  ensures 1 <= Factorial(n);
{
  if n == 0 then 1 else Factorial(n-1) * n
}
```

says that the result of Factorial is always positive, which Dafny verifies inductively from the function body. To refer to the function's result in the postcondition, use the function itself, as shown in the example.

By default, a function is ghost, and cannot be called from non-ghost code. To make it non-ghost, replace the keyword function with the two keywords function method.

By default, a function has an implicit receiver parameter, this. This parameter can be removed by preceding the function declaration with the keyword static. A static function F in a class C can be invoked by F.M(…). This can give a convenient way to declare a number of helper functions in a separate class.

Classes

A class is defined as follows:

```
class C {
  // member declarations go here
}
```

where the members of the class (fields, methods, and functions) are defined (as described above) inside the curly braces.

Datatypes

An inductive datatype is a type whose values are created using a fixed set of constructors. A datatype Tree with constructors Empty and Node is declared as follows:

```
datatype Tree = Empty | Node(Tree, int, Tree);
```

The constructors are separated by vertical bars. Parameter-less constructors need not use parentheses, as is shown here for Empty.

For each constructor Ct, the datatype implicitly declares a boolean member Ct?, which returns true for those values that have been constructed using Ct. For example, after the code snippet:

```
var t0 := Empty;
var t1 := Node(t0, 5, t0);
```

the expression t1.Node? evaluates to true and t0.Node? evaluates to false. Two datatype values are equal if they have been created using the same constructor and the same parameters to that constructor. Therefore, for parameter-less constructors like Empty, t.Empty? gives the same result as t == Empty.

A constructor can optionally declare a destructor for any of its parameters, which is done by introducing a name for the parameter. For example, if Tree were declared as:

```
datatype Tree = Empty | Node(left: Tree, data: int, right: Tree);
```

then t1.data == 5 and t1.left == t0 hold after the code snippet above.

Generics

Dafny supports generic types. That is, any class, inductive datatype, method, and function can have type parameters. These are declared in angle brackets after the name of what is being declared. For example:

```
class Multiset<T> { /*…*/ }
datatype Tree<T> = Empty | Node(Tree<T>, T, Tree<T>);
method Find<T>(key: T, collection: Tree<T>) { /*…*/ }
function IfThenElse<T>(b: bool, x: T, y: T): T { /*…*/ }
```

Statements

Here are examples of the most common statements in Dafny.

```
var LocalVariables := ExprList;
Lvalues := ExprList;
assert BoolExpr;
print PrintList;

if (BoolExpr0) {
  Stmts0
} else if (BoolExpr1) {
  Stmts1
} else {
  Stmts2
}

while (BoolExpr)
  invariant Inv;
  modifies Frame;
```

```
  decreases Rank;
{
  Stmts
}

match (Expr) {
  case Empty => Stmts0
  case Node(l, d, r) => Stmts1
}

break;
return;
```

The var statement introduces local variables (which are not allowed to shadow other variables declared inside the same set of most tightly enclosing curly braces). Each variable can optionally be followed by :T for any type T, which explicitly gives the preceding variable the type T (rather than being inferred). The ExprList with initial values is optional. To declare the variables as ghost variables, precede the declaration with the keyword ghost.

The assignment statement assigns each right-hand side in ExprList to the corresponding left-hand side in Lvalues. These assignments are performed in parallel, so the left-hand sides must denote distinct L-values. Each right-hand side can be an expression or an object creation of one of the following forms:

```
new T
new T.Init(ExprList)
new T[SizeExpr]
new T[SizeExpr0, SizeExpr1]
```

The first form allocates an object of type T. The second form additionally invokes an initialization method or constructor on the newly allocated object. The other forms show examples of array allocations, in particular a one- and a two-dimensional array of T values, respectively.

The entire right-hand side of an assignment can also be a method call, in which case the left-hand sides are the actual out-parameters (omitting the := if there are no out-parameters).

The assert statement claims that the given expression evaluates to true (which is checked by the verifier).

The print statement outputs to standard output the values of the given print expressions. A print expression is either an expression or a string literal (where \n is used to denote a newline character).

The if statement is the usual one. The example shows stringing together alternatives using else if. The else branch is optional, as usual.

The while statement is the usual loop, where the invariant declaration gives a loop invariant, the modifies clause restricts the modification frame of the loop, and the decreases clause introduces a variant function for the loop. By default, the loop invariant is true, the modification frame is the same as in the enclosing context (usually the modifies clause of the enclosing method), and the variant function is guessed from the loop guard.

The match statement evaluates the source Expr, an expression whose type is an inductive datatype, and then executes the case corresponding to which constructor was used to create the source datatype value, binding the constructor parameters to the given names.

The break statement can be used to exit loops, and the return statement can be used to exit a method.

Expressions

The expressions in Dafny are quite similar to those in Java-like languages. Here are some noteworthy differences.

In addition to the short-circuiting boolean operators && (and) and || (or), Dafny has

a short-circuiting implication operator ==> and an if-and-only-if operator <==>. As suggested by their widths, <==> has lower binding power than ==>, which in turn has lower binding power than && and ||.

Dafny comparison expressions can be chaining, which means that comparisons "in the same direction" can be strung together. For example,

```
0 <= i < j <= a.Length == N
```

has the same meaning as:

```
0 <= i && i < j && j <= a.Length && a.Length == N
```

Note that boolean equality can be expressed using both == and <==>. There are two differences between these. First, == has a stronger binding power than <==>. Second, == is chaining while <==> is associative. That is, a == b == c is the same as a == b && b == c, whereas a <==> b <==> c is the same as a <==> (b <==> c), which is also the same as (a <==> b) <==> c.

Operations on integers are the usual ones, except that / (integer division) and % (integer modulo) follow the Euclidean definition, which means that % always results in a non-negative number. (Hence, when the first argument to / or % is negative, the result is different than what you get in C, Java, or C#, see http://en.wikipedia.org/wiki/Modulo_operation.)

Dafny expressions include universal and existential quantifiers, which have the form:

```
forall x :: Expr
```

and likewise for exists, where x is a bound variable (which can be declared with an explicit type, as in x: T) and Expr is a boolean expression.

Operations on sets include + (union), * (intersection), and - (set difference), as well as the set comparison operators < (proper subset), <= (subset), their duals > and >=, and !! (disjointness). The expression x in S says that x is a member of set S, and x !in S is a convenient way of writing !(x in S). To make a set from some elements, enclose them in curly braces. For example, {x,y} is the set consisting of x and y (which is a singleton set if x == y), {x} is the singleton set containing x, and {} is the empty set.

Operations on sequences include + (concatenation) and the comparison operators < (proper prefix) and <= (prefix). Membership can be checked like for sets: x in S and x !in S. The length of a sequence S is denoted |S|, and the elements of such a sequence have indices from 0 to less than |S|. The expression S[j] denotes the element at index j of sequence S. The expression S[m..n], where 0 <= m <= n <= |S|, returns a sequence whose elements are the n-m elements of S starting at index m (that is, from S[m], S[m+1], … up to but not including S[n]). The expression S[m..] (often called "drop m") is the same as S[m..|S|], that is, it returns the sequence whose elements are all but the first m elements of S. The expression S[..n] (often called "take n") is the same as S[0..n], that is, it returns the sequence that consists of the first n elements of S. If j is a valid index into sequence S, then the expression S[j := x] is the sequence that is like S except that it has x at index j. Finally, to make a sequence from some elements, enclose them in square brackets. For example, [x,y] is the sequence consisting of the two elements x and y such that [x,y][0] == x and [x,y][1] == y, [x] is the singleton sequence whose only element is x, and [] is the empty sequence.

The if-then-else expression has the form:

```
if BoolExpr then Expr0 else Else1
```

where Expr0 and Expr1 are any expressions of the same type. Unlike the if statement, the if-then-else expression does not require parentheses around the guard expression, uses the then keyword, and must include an explicit else branch.

The match statement also has an analogous match expression, which has a form like:

```
match Expr
case Empty => Expr0
```

case Node(l, d, r) => Expr1

As with the if statement versus the if-then-else expression, note that the match expression does not require parentheses around the source expression and does not surround the cases with curly braces. A match expression can only be used in the body of function definitions, where it must either be the entire body or be the entire expression for a case in an enclosing match expression; furthermore, the source expression must be a parameter of the enclosing function.

Contact | Terms | Trademarks | Privacy and Cookies | Code of Conduct | Feedback ©2013 Microsoft Corporation. All rights reserved. Microsoft