Draft Dafny Reference Manual

Manuscript Dafny Reference

2016-01-28 05:22

Richard L. Ford
richford@microsoft.com

K. Rustan M. Leino
leino@microsoft.com

Abstract

This is the Dafny reference manual which describes the Dafny programming language and how to use the Dafny verification system. Parts of this manual are more tutorial in nature in order to help the user understand how to do proofs with Dafny.

Contents

0. Introduction

Dafny [18] is a programming language with built-in specification constructs. The Dafny static program verifier can be used to verify the functional correctness of programs.

The Dafny programming language is designed to support the static verification of programs. It is imperative, sequential, supports generic classes, methods and functions, dynamic allocation, inductive and co-inductive datatypes, and specification constructs. The specifications include pre- and postconditions, frame specifications (read and write sets), and termination metrics. To further support specifications, the language also offers updatable ghost variables, recursive functions, and types like sets and sequences. Specifications and ghost constructs are used only during verification; the compiler omits them from the executable code.

The Dafny verifier is run as part of the compiler. As such, a programmer interacts with it much in the same way as with the static type checkerâ€"when the tool produces errors, the programmer responds by changing the programâ€™s type declarations, specifications, and statements.

The easiest way to try out Dafny is in your web browser at rise4fun[15]. Once you get a bit more serious, you may prefer to download it to run it on your machine. Although Dafny can be run from the command line (on Windows or other platforms), the preferred way to run it is in Microsoft Visual Studio 2012 (or newer) or using emacs, where the Dafny verifier runs in the background while the programmer is editing the program.

The Dafny verifier is powered by Boogie [0, 16, 23] and Z3[4].

From verified programs, the Dafny compiler produces code (`.dll` or `.exe`) for the .NET platform via intermediate C# files. However, the facilities for interfacing with other .NET code are minimal.

This is the reference manual for the Dafny verification system. It is based on the following references: [5, 13–15, 18, 20, 22]

The main part of the reference manual is in top down order except for an initial section that deals with the lowest level constructs.

0.0. Dafny Example

To give a flavor of Dafny, here is the solution to a competition problem.

```
// VSComp 2010, problem 3, find a 0 in a linked list and return how many
// nodes were skipped until the first 0 (or end-of-list) was found.
// Rustan Leino, 18 August 2010.
//
// The difficulty in this problem lies in specifying what the return
// value 'r' denotes and in proving that the program terminates.  Both of
// these are addressed by declaring a ghost field 'List' in each
// linked-list node, abstractly representing the linked-list elements
// from the node to the end of the linked list.  The specification can
// now talk about that sequence of elements and can use 'r' as an index
// into the sequence, and termination can be proved from the fact that
// all sequences in Dafny are finite.
//
// We only want to deal with linked lists whose 'List' field is properly
// filled in (which can only happen in an acyclic list, for example).  To
// that avail, the standard idiom in Dafny is to declare a predicate
// 'Valid()' that is true of an object when the data structure
// representing object's abstract value is properly formed.  The
// definition of 'Valid()' is what one intuitively would think of as the
// ''object invariant'', and it is mentioned explicitly in method pre-
// and postconditions.  As part of this standard idiom, one also declared
// a ghost variable 'Repr' that is maintained as the set of objects that
// make up the representation of the aggregate object--in this case, the
// Node itself and all its successors.

class Node {
```

```
  ghost var List: seq<int>
  ghost var Repr: set<Node>
  var head: int
  var next: Node

  predicate Valid()
    reads this, Repr
  {
    this in Repr &&
    1 <= |List| && List[0] == head &&
    (next == null ==> |List| == 1) &&
    (next != null ==>
      next in Repr && next.Repr <= Repr && this !in next.Repr &&
      next.Valid() && next.List == List[1..])
  }

  static method Cons(x: int, tail: Node) returns (n: Node)
    requires tail == null || tail.Valid()
    ensures n != null && n.Valid()
    ensures if tail == null then n.List == [x]
                            else n.List == [x] + tail.List
  {
    n := new Node;
    n.head, n.next := x, tail;
    if (tail == null) {
      n.List := [x];
      n.Repr := {n};
    } else {
      n.List := [x] + tail.List;
      n.Repr := {n} + tail.Repr;
    }
  }
}

method Search(ll: Node) returns (r: int)
  requires ll == null || ll.Valid()
  ensures ll == null ==> r == 0
  ensures ll != null ==>
```

```
            0 <= r && r <= |ll.List| &&
            (r < |ll.List| ==> ll.List[r] == 0 && 0 !in ll.List[..r]) &&
            (r == |ll.List| ==> 0 !in ll.List)
{
  if (ll == null) {
    r := 0;
  } else {
    var jj,i := ll,0;
    while (jj != null && jj.head != 0)
      invariant jj != null ==> jj.Valid() && i + |jj.List| == |ll.List| &&
                              ll.List[i..] == jj.List
      invariant jj == null ==> i == |ll.List|
      invariant 0 !in ll.List[..i]
      decreases |ll.List| - i
    {
      jj := jj.next;
      i := i + 1;
    }
    r := i;
  }
}

method Main()
{
  var list: Node := null;
  list := list.Cons(0, list);
  list := list.Cons(5, list);
  list := list.Cons(0, list);
  list := list.Cons(8, list);
  var r := Search(list);
  print "Search returns ", r, "\n";
  assert r == 1;
}
```

1. Lexical and Low Level Grammar

Dafny uses the Coco/R lexer and parser generator for its lexer and parser (http://www.ssw.uni-linz.ac.at/Research/Projects/Coco)[27]. The Dafny input file to Coco/R is the `Dafny.atg` file in

the source tree. A Coco/R input file consists of code written in the target language (e.g. C#) intermixed with these special sections:

0. The Characters section which defines classes of characters that are used in defining the lexer (Section 1.0).
1. The Tokens section which defines the lexical tokens (Section 1.1).
2. The Productions section which defines the grammar. The grammar productions are distributed in the later parts of this document in the parts where those constructs are explained.

The grammar presented in this document was derived from the `Dafny.atg` file but has been simplified by removing details that, though needed by the parser, are not needed to understand the grammar. In particular, the following transformation have been performed.

- The semantics actions, enclosed by "(." and ".)", where removed.
- There are some elements in the grammar used for error recovery ("SYNC"). These were removed.
- There are some elements in the grammar for resolving conflicts ("IF(b)"). These have been removed.
- Some comments related to Coco/R parsing details have been removed.
- A Coco/R grammar is an attributed grammar where the attributes enable the productions to have input and output parameters. These attributes were removed except that boolean input parameters that affect the parsing are kept.

  - In our representation we represent these in a definition by giving the names of the parameters following the non-terminal name. For example `entity1(allowsX)`.
  - In the case of uses of the parameter, the common case is that the parameter is just passed to a lower-level non-terminal. In that case we just give the name, e.g. `entity2(allowsX)`.
  - If we want to given an explicit value to a parameter, we specify it in a keyword notation like this: `entity2(allowsX: true)`.
  - In some cases the value to be passed depends on the grammatical context. In such cases we give a description of the conditions under which the parameter is true, enclosed in parenthesis. For example:
    `FunctionSignatureOrEllipsis_(allowGhostKeyword: ("method" present))`means that the `allowGhostKeyword` parameter is true if the "method" keyword was given in the associated `FunctionDecl`.
  - Where a parameter affects the parsing of a non-terminal we will explain the effect of the parameter.

The names of character sets and tokens start with a lower case letter but the names of grammar non-terminals start with an upper-case letter.

The grammar uses Extended BNF notation. See the Coco/R Referenced manual for details. But in summary:

- identifiers starting with a lower case letter denote terminal symbols,
- identifiers starting with an upper case letter denote nonterminal symbols.
- Strings denote themselves.
- `=` separates the sides of a production, e.g. `A = a b c`
- In the Coco grammars "." terminates a production, but for readability in this document a production starts with the defined identifier in the left margin and may be continued on subsequent lines if they are indented.
- `|` separates alternatives, e.g. `a b | c | d e` means `a b` or `c` or `d e`
- `( )` groups alternatives, e.g. (a | b) c means a c or b c
- `[ ]` option, e.g. `[a] b` means `a b` or `b`
- `{ }` iteration (0 or more times), e.g. `{a} b` means `b` or `a b` or `a a b` or …
- We allow `|` inside `[ ]` and `{ }`. So `[a | b]` is short for `[(a | b)]` and `{a | b}` is short for `{(a | b)}`.
- The first production defines the name of the grammar, in this case `Dafny`.

In addition to the Coco rules, for the sake of readability we have adopted these additional conventions.

- We allow `-` to be used. `a - b` means it matches if it matches `a` but not `b`.
- To aid in explaining the grammar we have added some additional productions that are not present in the original grammar. We name these with a trailing underscore. If you inline these where they are referenced, the result should let you reconstruct the original grammar.

For the convenience of the reader, any references to character sets, tokens, or grammar non-terminals in this document are hyper-links that will link to the definition of the entity.

1.0. Character Classes

This section defines character classes used later in the token definitions. In this section backslash is used to start an escape sequence, so for example '\n' denotes the single linefeed character.

```
letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"
```

At present, a letter is an ASCII upper or lowercase letter. Other Unicode letters are not supported.

```
digit = "0123456789"
```

A digit is just one of the base-10 digits.

```
posDigit = "123456789"
```

A `posDigit` is a digit, excluding 0.

```
hexdigit = "0123456789ABCDEFabcdef"
```

A `hexdigit` character is a digit or one of the letters from 'A' to 'F' in either case.

```
special = "'_?"
```

The *special* characters are the characters in addition to alphanumeric characters that are allowed to appear in a Dafny identifier. These are

- `"'"` because mathematicians like to put primes on identifiers and some ML programmers like to start names of type parameters with a "'".
- "_" because computer scientists expect to be able to have underscores in identifiers.
- "?" because it is useful to have "?" at the end of names of predicates, e.g. "Cons?".

```
cr       = '\r'
```

A carriage return character.

```
lf       = '\n'
```

A line feed character.

```
tab      = '\t'
```

A tab character.

```
space = ' '
```

A space character.

```
nondigitIdChar = letter + special
```

The characters that can be used in an identifier minus the digits.

```
idchar = nondigitIdChar + digit
```

The characters that can be used in an identifier.

```
nonidchar = ANY - idchar
```

Any character except those that can be used in an identifier.

```
charChar = ANY - '\'' - '\\' - cr - lf
```

Characters that can appear in a character constant.

```
stringChar = ANY - '"' - '\\' - cr - lf
```

Characters that can appear in a string constant.

```
verbatimStringChar = ANY - '"'
```

Characters that can appear in a verbatim string.

1.0.0. Comments

Comments are in two forms.

- They may go from "/*" to "*/" and be nested.
- They may go from "//" to the end of the line.

1.1. Tokens

As with most languages, Dafny syntax is defined in two levels. First the stream of input characters is broken up into *tokens*. Then these tokens are parsed using the Dafny grammar. The Dafny tokens are defined in this section.

1.1.0. Reserved Words

The following reserved words appear in the Dafny grammar and may not be used as identifiers of user-defined entities:

```
reservedword =
    "abstract" | "array" | "as" | "assert" | "assume" | "bool" | "break" |
    "calc" | "case" | "char" | "class" | "codatatype" | "colemma" |
    "constructor" | "copredicate" | "datatype" | "decreases" |
    "default" | "else" | "ensures" | "exists" | "extends" | "false" |
    "forall" | "free" | "fresh" | "function" | "ghost" | "if" | "imap" | "import" |
    "in" | "include" | "inductive" | "int" | "invariant" | "iset" | "iterator" | "label" |
    "lemma" | "map" | "match" | "method" | "modifies" | "modify" |
    "module" | "multiset" | "nat" | "new" | "newtype" | "null" | "object" |
    "old" | "opened" | "predicate" | "print" | "protected" |
    "reads" | "real" | "refines" | "requires" | "return" | "returns" | "seq" |
```

```
    "set" | "static" | "string" | "then" | "this" | "trait" | "true" | "type" |
    "var" | "where" | "while" | "yield" | "yields" | arrayToken

arrayToken = "array" [ posDigit { digit }]
```

An arrayToken is a reserved word that denotes an array type of given rank. array is an array type of rank 1 (aka a vector). array2 is the type of two-dimensional arrays, etc.

TODO: Is "_" is reserved word?

1.1.1. Identifiers

```
ident = nondigitIdChar { idchar } - arraytoken - chartoken - reservedword
```

In general Dafny identifiers are sequences of idChar characters where the first character is a nondigitIdChar. However tokens that fit this pattern are not identifiers if they look like an array type token, a character literal, or a reserved work.

1.1.2. Digits

```
digits = digit {['_'] digit}
```

A sequence of decimal digits, possibly interspersed with underscores for readability. Example: 1_234_567.

```
hexdigits = "0x" hexdigit {['_'] hexdigit}
```

A hexadecimal constant, possibly interspersed with underscores for readability. Example: 0xffff_ffff.

```
decimaldigits = digit {['_'] digit} '.' digit {['_'] digit}
```

A decimal fraction constant, possibly interspersed with underscores for readability. Example: 123_456.789_123.

1.1.3. Escaped Character

In this section the "\" characters are literal.

```
escapedChar =
    ( "\'" | "\"" | "\\" | "\0" | "\n" | "\r" | "\t"
      | "\u" hexdigit hexdigit hexdigit hexdigit
    )
```

In Dafny character or string literals escaped characters may be used to specify the presence of the delimiting quote, or back slash, or null, or new line, or carriage return or tab, or the Unicode character with given hexadecimal representation.

1.1.4. Character Constant Token

```
charToken = "'" ( charChar | escapedChar ) "'"
```

A character constant is enclosed by "'" and includes either a character from the charChar set, or an escaped character. Note that although Unicode letters are not allowed in Dafny identifiers, Dafny does support Unicode in its character and string constants and in its data. A character constant has type char.

1.1.5. String Constant Token

```
stringToken =
    '"' { stringChar | escapedChar }  '"'
  | '@' '"' { verbatimStringChar | '"' '"' } '"'
```

A string constant is either a normal string constant or a verbatim string constant. A normal string constant is enclosed by '"' and can contain characters from the stringChar set and escapes.

A verbatim string constant is enclosed between '@"' and '"' and can consists of any characters (including newline characters) except that two successive double quotes give a way to escape one quote character inside the string.

1.2. Low Level Grammar Productions

1.2.0. Identifier Variations

```
Ident = ident
```

The Ident non-terminal is just an ident token and represents an ordinary identifier.

```
DotSuffix =
  ( ident | digits | "requires" | "reads" )
```

When using the *dot* notation to denote a component of a compound entity the token following the ".", in addition to being an identifier, can also be a natural number, or one of the keywords requires or reads.

- Digits can be used to name fields of classes and destructors of datatypes. For example, the built-in tuple datatypes have destructors named 0, 1, 2, etc. Note that as a field or destructor name, internal underscores matter, so 10 is different from 1_0.
- `m.requires` is used to denote the precondition for method m.
- `m.reads` is used to denote the things that method m may read.

```
NoUSIdent = ident - "_" { idChar }
```

A `NoUSIdent` is an identifier except that identifiers with a **leading** underscore are not allowed. The names of user-defined entities are required to be `NoUSIdent`s. We introduce more mnemonic names for these below (e.g. `ClassName`).

```
WildIdent = NoUSIdent | "_"
```

Identifier, disallowing leading underscores, except the "wildcard" identifier "_". When "_" appears it is replaced by a unique generated identifier distinct from user identifiers.

1.2.1. NoUSIdent Synonyms

In the productions for the declaration of user-defined entities the name of the user-defined entity is required to be an identifier that does not start with an underscore, i.e., a `NoUSIdent`. To make the productions more mnemonic, we introduce the following synonyms for `NoUSIdent`.

```
ModuleName = NoUSIdent
ClassName = NoUSIdent
TraitName = NoUSIdent
DatatypeName = NoUSIdent
DatatypeMemberName = NoUSIdent
NewtypeName = NoUSIdent
NumericTypeName = NoUSIdent
SynonymTypeName = NoUSIdent
IteratorName = NoUSIdent
TypeVariableName = NoUSIdent
MethodName = NoUSIdent
FunctionName = NoUSIdent
PredicateName = NoUSIdent
CopredicateName = NoUSIdent
LabelName = NoUSIdent
AttributeName = NoUSIdent
FieldIdent = NoUSIdent
```

A `FieldIdent` is one of the ways to identify a field. The other is using digits.

1.2.2. Qualified Names

A qualified name starts with the name of the top-level entity and then is followed by zero or more DotSuffixs which denote a component. Examples:

- Module.MyType1
- MyTuple.1
- MyMethod.requires

The grammar does not actually have a production for qualified names except in the special case of a qualified name that is known to be a module name, i.e. a QualifiedModuleName.

1.2.3. Identifier-Type Combinations

In this section, we describe some nonterminals that combine an identifier and a type.

```
IdentType = WildIdent ":" Type
```

In Dafny, a variable or field is typically declared by giving its name followed by a colon and its type. An IdentType is such a construct.

```
GIdentType(allowGhostKeyword) = [ "ghost" ] IdentType
```

A GIdentType is a typed entity declaration optionally preceded by "ghost". The *ghost* qualifier means the entity is only used during verification but not in the generated code. Ghost variables are useful for abstractly representing internal state in specifications. If allowGhostKeyword is false then "ghost" is not allowed.

```
LocalIdentTypeOptional = WildIdent [ ":" Type ]
```

A LocalIdentTypeOptional is used when declaring local variables. In such a case a value may be specified for the variable in which case the type may be omitted because it can be inferred from the initial value. The initial value value may also be omitted.

```
IdentTypeOptional = WildIdent [ ":" Type ]
```

A IdentTypeOptional is typically used in a context where the type of the identifier may be inferred from the context. Examples are in pattern matching or quantifiers.

```
TypeIdentOptional = [ "ghost" ] ( NoUSIdent | digits ) ":" ] Type
```

TypeIdentOptionals are used in FormalsOptionalIds. This represents situations where a type is given but there may not be an identifier.

```
FormalsOptionalIds = "(" [TypeIdentOptional  { "," TypeIdentOptional } ] ")"
```

A FormalsOptionalIds is a formal parameter list in which the types are required but the names of the parameters is optional. This is used in algebraic datatype definitions.

1.2.4. Numeric Literals

```
Nat = ( digits | hexdigits )
```

A Nat represents a natural number expressed in either decimal or hexadecimal.

```
Dec = (decimaldigits )
```

A Dec represents a decimal fraction literal.

2. Programs

```
Dafny = { IncludeDirective_ } { TopDecl } EOF
```

At the top level, a Dafny program (stored as files with extension `.dfy`) is a set of declarations. The declarations introduce (module-level) methods and functions, as well as types (classes, traits, inductive and co-inductive datatypes, new_types, type synonyms, opaque types, and iterators) and modules, where the order of introduction is irrelevant. A class also contains a set of declarations, introducing fields, methods, and functions.

When asked to compile a program, Dafny looks for the existence of a Main() method. If a legal Main() method is found, the compiler will emit a `.EXE`; otherwise, it will emit a `.DLL`.

(If there is more than one Main(), Dafny will try to emit an .EXE, but this may cause the C# compiler to complain. One could imagine improving this functionality so that Dafny will produce a polite error message in this case.)

In order to be a legal Main() method, the following must be true:

- The method takes no parameters
- The method is not a ghost method
- The method has no requires clause
- The method has no modifies clause
- If the method is an instance (that is, non-static) method in a class, then the enclosing class must not declare any constructor

Note, however, that the following are allowed:

- The method is allowed to be an instance method as long as the enclosing class does not declare any constructor. In this case, the runtime system will allocate an object of the enclosing class and will invoke Main() on it.

- The method is allowed to have `ensures` clauses
- The method is allowed to have `decreases` clauses, including a `decreases *`. (If Main() has a `decreases *`, then its execution may go on forever, but in the absence of a `decreases *` on Main(), Dafny will have verified that the entire execution will eventually terminate.)

An invocation of Dafny may specify a number of source files. Each Dafny file follows the grammar of the `Dafny` non-terminal.

It consists of a sequence of optional *include* directives followed by top level declarations followed by the end of the file.

2.0. Include Directives

```
IncludeDirective_ = "include" stringToken
```

Include directives have the form `"include" stringToken` where the string token is either a normal string token or a verbatim string token. The `stringToken` is interpreted as the name of a file that will be included in the Dafny source. These included files also obey the `Dafny` grammar. Dafny parses and processes the transitive closure of the original source files and all the included files, but will not invoke the verifier on these unless they have been listed explicitly on the command line.

2.1. Top Level Declarations

```
TopDecl = { { DeclModifier }
  ( SubModuleDecl
  | ClassDecl
  | DatatypeDecl
  | NewtypeDecl
  | SynonymTypeDecl
  | IteratorDecl
  | TraitDecl
  | ClassMemberDecl(moduleLevelDecl: true)
  }
```

Top-level declarations may appear either at the top level of a Dafny file, or within a `SubModuleDecl`. A top-level declaration is one of the following types of declarations which are described later.

The `ClassDecl`, `DatatypeDecl`, `NewtypeDecl`, `SynonymTypeDecl`, `IteratorDecl`, and `TraitDecl` declarations are type declarations and are describe in Section 5. Ordinarily `ClassMemberDecl`s appear in class declarations but they can also appear at the top level. In that case they are

included as part of an implicit top-level class and are implicitly `static` (but cannot be declared as static). In addition a `ClassMemberDecl` that appears at the top level cannot be a `FieldDecl`.

2.2. Declaration Modifiers

```
DeclModifier =
  ( "abstract" | "ghost" | "static" | "protected"
  | "extern" [ stringToken]
  )
```

Top level declarations may be preceded by zero or more declaration modifiers. Not all of these are allowed in all contexts.

The "abstract" modifiers may only be used for module declarations. An abstract module can leave some entities underspecified. Abstract modules are not compiled to C#.

The ghost modifier is used to mark entities as being used for specification only, not for compilation to code.

The static modifier is used for class members that that are associated with the class as a whole rather than with an instance of the class.

The protected modifier is used to control the visibility of the body of functions.

The extern modifier is used to alter the CompileName of entities. The CompileName is the name for the entity when translating to Boogie or C#.

The following table shows modifiers that are available for each of the kinds of declaration. In the table we use already-ghost to denote that the item is not allowed to have the ghost modifier because it is already implicitly ghost.

Declaration	allowed modifiers
module	abstract
class	extern
trait	-
datatype or codatatype	-
field	ghost
newtype	-
synonym types	-
iterators	-
method	ghost static extern
lemma, colemma, comethod	already-ghost static protected
inductive lemma	already-ghost static
constructor	-
function (non-method)	already-ghost static protected
function method	already-ghost static protected extern
predicate (non-method)	already-ghost static protected
predicate method	already-ghost static protected extern
inductive predicate	already-ghost static protected
copredicate	already-ghost static protected

3. Modules

```
SubModuleDecl = ( ModuleDefinition_ | ModuleImport_ )
```

Structuring a program by breaking it into parts is an important part of creating large programs. In Dafny, this is accomplished via *modules*. Modules provide a way to group together related types, classes, methods, functions, and other modules together, as well as control the scope of declarations. Modules may import each other for code reuse, and it is possible to abstract over modules to separate an implementation from an interface.

3.0. Declaring New Modules

```
ModuleDefinition_ = "module" { Attribute } ModuleName
       [ [  "exclusively" ] "refines" QualifiedModuleName ]
       "{" { TopDecl } "}"
QualifiedModuleName = Ident { "." Ident }
```

A qualified name that is known to refer to a module.

A new module is declared with the `module` keyword, followed by the name of the new module, and a pair of curly braces ({}) enclosing the body of the module:

```
module Mod {
  ...
}
```

A module body can consist of anything that you could put at the top level. This includes classes, datatypes, types, methods, functions, etc.

```
module Mod {
  class C {
    var f: int
    method m()
  }
  datatype Option = A(int) | B(int)
  type T
  method m()
  function f(): int
}
```

You can also put a module inside another, in a nested fashion:

```
module Mod {
  module Helpers {
    class C {
      method doIt()
      var f: int
    }
  }
}
```

Then you can refer to the members of the `Helpers` module within the `Mod` module by prefixing them with "Helpers.". For example:

```
module Mod {
  module Helpers { ... }
  method m() {
    var x := new Helpers.C;
    x.doIt();
    x.f := 4;
  }
```

```
}
```

Methods and functions defined at the module level are available like classes, with just the module name prefixing them. They are also available in the methods and functions of the classes in the same module.

```
module Mod {
  module Helpers {
    function method addOne(n: nat): nat {
      n + 1
    }
  }
  method m() {
    var x := 5;
    x := Helpers.addOne(x); // x is now 6
  }
}
```

3.1. Importing Modules

```
ModuleImport_ = "import" ["opened" ] ModuleName
    [ "=" QualifiedModuleName
    | "as" QualifiedModuleName ["default" QualifiedModuleName ]
    ]
    [ ";" ]
```

Declaring new submodules is useful, but sometimes you want to refer to things from an existing module, such as a library. In this case, you can *import* one module into another. This is done via the `import` keyword, and there are a few different forms, each of which has a different meaning. The simplest kind is the concrete import, and has the form `import A = B`. This declaration creates a reference to the module `B` (which must already exist), and binds it to the new name `A`. Note this new name, i.e. `A`, is only bound in the module containing the import declaration; it does not create a global alias. For example, if `Helpers` was defined outside of `Mod`, then we could import it:

```
module Helpers {
  ...
}
module Mod {
  import A = Helpers
```

```
  method m() {
    assert A.addOne(5) == 6;
  }
}
```

Note that inside `m()`, we have to use `A` instead of `Helpers`, as we bound it to a different name. The name `Helpers` is not available inside `m()`, as only names that have been bound inside `Mod` are available. In order to use the members from another module, it either has to be declared there with `module` or imported with `import`.

We don't have to give `Helpers` a new name, though, if we don't want to. We can write `import Helpers = Helpers` if we want to, and Dafny even provides the shorthand `import Helpers` for this behavior. You can't bind two modules with the same name at the same time, so sometimes you have to use the = version to ensure the names do not clash.

The `QualifiedModuleName` in the `ModuleImport_` starts with a sibling module of the importing module, or with a submodule of the importing module. There is no wya to refer to the parent module, only sibling modules (and their submodules).

3.2. Opening Modules

Sometimes, prefixing the members of the module you imported with the name is tedious and ugly, even if you select a short name when importing it. In this case, you can import the module as `opened`, which causes all of its members to be available without adding the module name. The `opened` keyword must immediately follow `import`, if it is present. For example, we could write the previous example as:

```
module Mod {
  import opened Helpers
  method m() {
    assert addOne(5) == 6;
  }
}
```

When opening modules, the newly bound members will have low priority, so they will be hidden by local definitions. This means if you define a local function called `addOne`, the function from `Helpers` will no longer be available under that name. When modules are opened, the original name binding is still present however, so you can always use the name that was bound to get to anything that is hidden.

```
module Mod {
  import opened Helpers
  function addOne(n: nat): nat {
```

```
    n - 1
  }
  method m() {
    assert addOne(5) == 6; // this is now false,
                           // as this is the function just defined
    assert Helpers.addOne(5) == 6; // this is still true
  }
}
```

If you open two modules that both declare members with the same name, then neither member can be referred to without a module prefix, as it would be ambiguous which one was meant. Just opening the two modules is not an error, however, as long as you don't attempt to use members with common names. The `opened` keyword can be used with any kind of `import` declaration, including the module abstraction form.

3.3. Module Abstraction

Sometimes, using a specific implementation is unnecessary; instead, all that is needed is a module that implements some interface. In that case, you can use an *abstract* module import. In Dafny, this is written `import A as B`. This means bind the name `A` as before, but instead of getting the exact module `B`, you get any module which is a *adheres* of `B`. Typically, the module `B` may have abstract type definitions, classes with bodyless methods, or otherwise be unsuitable to use directly. Because of the way refinement is defined, any refinement of `B` can be used safely. For example, if we start with:

```
module Interface {
  function method addSome(n: nat): nat
    ensures addSome(n) > n
}
module Mod {
  import A as Interface
  method m() {
    assert 6 <= A.addSome(5);
  }
}
```

then we can be more precise if we know that `addSome` actually adds exactly one. The following module has this behavior. Further, the postcondition is stronger, so this is actually a refinement of the Interface module.

```
module Implementation {
  function method addSome(n: nat): nat
    ensures addSome(n) == n + 1
  {
    n + 1
  }
}
```

We can then substitute `Implementation` for `A` in a new module, by declaring a refinement of `Mod` which defines `A` to be `Implementation`.

```
module Mod2 refines Mod {
  import A = Implementation
  ...
}
```

You can also give an implementation directly, without introducing a refinement, by giving a default to the abstract import:

```
module Interface {
  function method addSome(n: nat): nat
    ensures addSome(n) > n
}
module Mod {
  import A as Interface default Implementation
  method m() {
    assert 6 <= A.addSome(5);
  }
}
module Implementation {
  function method addSome(n: nat): nat
    ensures addSome(n) == n + 1
  {
    n + 1
  }
}
module Mod2 refines Mod {
  import A as Interface default Implementation
  ...
}
```

Regardless of whether there is a default, the only things known about `A` in this example is that it has a function `addSome` that returns a strictly bigger result, so even with the default we still can't prove that `A.addSome(5) == 6`, only that `6 <= A.addSome(5)`.

When you refine an abstract import into a concrete one, or giving a default, Dafny checkes that the concrete module is a refinement of the abstract one. This means that the methods must have compatible signatures, all the classes and datatypes with their constructors and fields in the abstract one must be present in the concrete one, the specifications must be compatible, etc.

3.4. Module Ordering and Dependencies

Dafny isn't particular about which order the modules appear in, but they must follow some rules to be well formed. As a rule of thumb, there should be a way to order the modules in a program such that each only refers to things defined **before** it in the source text. That doesn't mean the modules have to be given in that order. Dafny will figure out that order for you, assuming you haven't made any circular references. For example, this is pretty clearly meaningless:

```
import A = B
import B = A
```

You can have import statements at the toplevel, and you can import modules defined at the same level:

```
import A = B
method m() {
  A.whatever();
}
module B { ... }
```

In this case, everything is well defined because we can put `B` first, followed by the `A` import, and then finally `m()`. If there is no ordering, then Dafny will give an error, complaining about a cyclic dependency.

Note that when rearranging modules and imports, they have to be kept in the same containing module, which disallows some pathological module structures. Also, the imports and submodules are always considered to be first, even at the toplevel. This means that the following is not well formed:

```
method doIt() { }
module M {
  method m() {
    doIt();
  }
```

```
}
```

because the module `M` must come before any other kind of members, such as methods. To define global functions like this, you can put them in a module (called `Globals`, say) and open it into any module that needs its functionality. Finally, if you import via a path, such as `import A = B.C`, then this creates a dependency of `A` on `B`, as we need to know what `B` is (is it abstract or concrete, or a refinement?).

3.5. Name Resolution

When Dafny sees something like `A<T>.B<U>.C<V>`, how does it know what each part refers to? The process Dafny uses to determine what identifier sequences like this refer to is name resolution. Though the rules may seem complex, usually they do what you would expect. Dafny first looks up the initial identifier. Depending on what the first identifier refers to, the rest of the identifier is looked up in the appropriate context.

In terms of the grammar, sequences like the above are represented as a `NameSegment` followed by 0 or more `Suffix`es. A `Suffix` is more general and the form shown above would be for when the `Suffix` is an `AugmentedDotSuffix_`.

The resolution is different depending on whether it is in an expression context or a type context.

3.5.0. Expression Context Name Resolution

The leading `NameSegment` is resolved using the first following rule that succeeds.

0. Local variables, parameters and bound variables. These are things like `x`, `y`, and `i` in `var x;, ... returns (y: int)`, and `forall i :: ...`. The declaration chosen is the match from the innermost matching scope.

1. If in a class, try to match a member of the class. If the member that is found is not static an implicit `this` is inserted. This works for fields, functions, and methods of the current class (if in a static context, then only static methods and functions are allowed). You can refer to fields of the current class either as `this.f` or `f`, assuming of course that `f` hasn't be hidden by one of the above. You can always prefix this if needed, which cannot be hidden. (Note, a field whose name is a string of digits must always have some prefix.)

2. If there is no `Suffix`, then look for a datatype constructor, if unambiguous. Any datatypes that don't need qualification (so the datatype name itself doesn't need a prefix), and also have a uniquely named constructor, can be referred to just by its name. So if

`datatype List = Cons(List) | Nil` is the only datatype that declares `Cons` and `Nil` constructors, then you can write `Cons(Cons(Nil))`. If the constructor name is not unique, then you need to prefix it with the name of the datatype (for example `List.Cons(List.Nil)))`. This is done per constructor, not per datatype.

3. Look for a member of the enclosing module.

4. Module-level (static) functions and methods

TODO: Not sure about the following paragraph. Opened modules are treated at each level, after the declarations in the current module. Opened modules only affect steps 2, 3 and 5. If a ambiguous name is found, an error is generated, rather than continuing down the list. After the first identifier, the rules are basically the same, except in the new context. For example, if the first identifier is a module, then the next identifier looks into that module. Opened modules only apply within the module it is opened into. When looking up into another module, only things explicitly declared in that module are considered.

To resolve expression `E.id`:
First resolve expression E and any type arguments.

- If `E` resolved to a module `M`:

  0. If `E.id<T>` is not followed by any further suffixes, look for unambiguous datatype constructor.
  1. Member of module M: a sub-module (including submodules of imports), class, datatype, etc.
  2. Static function or method.

- If `E` denotes a type:

  3. Look up id as a member of that type

- If `E` denotes an expression:

  4. Let T be the type of E. Look up id in T.

3.5.1. Type Context Name Resolution

In a type context the priority of `NameSegment` resolution is:

1. Type parameters.

2. Member of enclosing module (type name or the name of a module).

To resolve expression `E.id`:

- If `E` resolved to a module `M`:

  0. Member of module M: a sub-module (including submodules of imports), class, datatype, etc.

- If `E` denotes a type:

  1. If `allowDanglingDotName`: Return the type of `E` and the given `E.id`, letting the caller try to make sense of the final dot-name. TODO: I don't under this sentence. What is `allowDanglingDotName`?

4. Specifications

Specifications describe logical properties of Dafny methods, functions, lambdas, iterators and loops. They specify preconditions, postconditions, invariants, what memory locations may be read or modified, and termination information by means of *specification clauses*. For each kind of specification zero or more specification clauses (of the type accepted for that type of specification) may be given, in any order.

We document specifications at these levels:

- At the lowest level are the various kinds of specification clauses, e.g. a `RequiresClause_`.
- Next are the specifications for entities that need them, e.g. a `MethodSpec`.
- At the top level are the entity declarations that include the specifications, e.g. `MethodDecl`.

This section documents the first two of these in a bottom-up manner. We first document the clauses and then the specifications that use them.

4.0. Specification Clauses

4.0.0. Requires Clause

```
RequiresClause_ =
    "requires" Expression(allowLemma: false, allowLambda: false)
```

The **requires** clauses specify preconditions for methods, functions, lambda expressions and iterators. Dafny checks that the preconditions are met at all call sites. The callee may then assume the preconditions hold on entry.

If no **requires** clause is specified it is taken to be `true`.

If more than one **requires** clause is given, then the precondition is the conjunction of all of the expressions from all of the **requires** clauses.

4.0.1. Ensures Clause

```
EnsuresClause_ =
    "ensures" { Attribute } Expression(allowLemma: false, allowLambda: false)
ForAllEnsuresClause_ =
    "ensures" Expression(allowLemma: false, allowLambda: true)
FunctionEnsuresClause_ =
    "ensures" Expression(allowLemma: false, allowLambda: false)
```

An **ensures** clause specifies the post condition for a method, function or iterator.

If no **ensures** clause is specified it is taken to be `true`.

If more than one **ensures** clause is given, then the postcondition is the conjunction of all of the expressions from all of the **ensures** clauses.

TODO: In the present sources `FunctionEnsuresClause_` differs from `EnsuresClause_` only in that it is not allowed to specify `Attribute`s. This seems like a bug and will likely be fixed in a future version.

4.0.2. Decreases Clause

```
DecreasesClause_(allowWildcard, allowLambda) =
    "decreases" { Attribute } DecreasesList(allowWildcard, allowLambda)
FunctionDecreasesClause_(allowWildcard, allowLambda) =
    "decreases" DecreasesList(allowWildcard, allowLambda)

DecreasesList(allowWildcard, allowLambda) =
  PossiblyWildExpression(allowLambda)
  { "," PossiblyWildExpression(allowLambda) }
```

If `allowWildcard` is false but one of the `PossiblyWildExpression`s is a wild-card, an error is reported.

TODO: A `FunctionDecreasesClause_` is not allowed to specify `Attribute`s. this will be fixed in a future version.

Decreases clauses are used to prove termination in the presence of recursion. if more than one **decreases** clause is given it is as if a single **decreases** clause had been given with the collected list of arguments. That is,

```
decreases A, B
decreases C, D
```

is equivalent to

```
decreases A, B, C, D
```

If any of the expressions in the **decreases** clause are wild (i.e. "*") then proof of termination will be skipped.

Termination metrics in Dafny, which are declared by **decreases** clauses, are lexicographic tuples of expressions. At each recursive (or mutually recursive) call to a function or method, Dafny checks that the effective **decreases** clause of the callee is strictly smaller than the effective **decreases** clause of the caller.

What does "strictly smaller" mean? Dafny provides a built-in well-founded order for every type and, in some cases, between types. For example, the Boolean "false" is strictly smaller than "true", the integer 78 is strictly smaller than 102, the set `{2,5}` is strictly smaller than the set `{2,3,5}`, and for "s" of type `seq<Color>` where `Color` is some inductive datatype, the color `s[0]` is strictly less than `s` (provided `s` is nonempty).

What does "effective decreases clause" mean? Dafny always appends a "top" element to the lexicographic tuple given by the user. This top element cannot be syntactically denoted in a Dafny program and it never occurs as a run-time value either. Rather, it is a fictitious value, which here we will denote \top, such that each value that can ever occur in a Dafny program is strictly less than \top. Dafny sometimes also prepends expressions to the lexicographic tuple given by the user. The effective decreases clause is any such prefix, followed by the user-provided decreases clause, followed by \top. We said "user-provided decreases clause", but if the user completely omits a "decreases" clause, then Dafny will usually make a guess at one, in which case the effective decreases clause is any prefix followed by the guess followed by \top. (If you're using the Dafny IDE in Visual Studio, you can hover the mouse over the name of a recursive function or method, or the "while" keyword for a loop, to see the "decreases" clause that Dafny guessed, if any.)

Here is a simple but interesting example: the Fibonacci function.

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

In this example, if you hover your mouse over the function name you will see that Dafny has supplied a `**decreases** n` clause.

Let's take a look at the kind of example where a mysterious-looking decreases clause like

"Rank, 0" is useful.

Consider two mutually recursive methods, `A` and `B`:

```
method A(x: nat)
{
  B(x);
}

method B(x: nat)
{
  if x != 0 { A(x-1); }
}
```

To prove termination of `A` and `B`, Dafny needs to have effective decreases clauses for A and B such that:

- the measure for the callee `B(x)` is strictly smaller than the measure for the caller `A(x)`, and
- the measure for the callee `A(x-1)` is strictly smaller than the measure for the caller `B(x)`.

Satisfying the second of these conditions is easy, but what about the first? Note, for example, that declaring both `A` and `B` with "decreases x" does not work, because that won't prove a strict decrease for the call from `A(x)` to `B(x)`.

Here's one possibility (for brevity, we will omit the method bodies):

```
method A(x: nat)
  decreases x, 1

method B(x: nat)
  decreases x, 0
```

For the call from `A(x)` to `B(x)`, the lexicographic tuple `"x, 0"` is strictly smaller than `"x, 1"`, and for the call from `B(x)` to `A(x-1)`, the lexicographic tuple `"x-1, 1"` is strictly smaller than `"x, 0"`.

Two things to note: First, the choice of "0" and "1" as the second components of these lexicographic tuples is rather arbitrary. It could just as well have been "false" and "true", respectively, or the sets `{2,5}` and `{2,3,5}`. Second, the keyword **decreases** often gives rise to an intuitive English reading of the declaration. For example, you might say that the recursive calls in the definition of the familiar Fibonacci function `Fib(n)` "decreases n". But when the lexicographic tuple contains constants, the English reading of the declaration becomes mysterious and may give rise to questions like "how can you decrease the constant 0?". The keyword is just that—a

keyword. It says "here comes a list of expressions that make up the lexicographic tuple we want to use for the termination measure". What is important is that one effective decreases clause is compared against another one, and it certainly makes sense to compare something to a constant (and to compare one constant to another).

We can simplify things a little bit by remembering that Dafny appends \top to the user-supplied decreases clause. For the A-and-B example, this lets us drop the constant from the **decreases** clause of A:

```
method A(x: nat)
  decreases x

method B(x: nat)
  decreases x, 0
```

The effective decreases clause of `A` is `"x, \top"` and the effective decreases clause of `B` is `"x, 0, \top"`. These tuples still satisfy the two conditions `(x, 0, \top) < (x, \top)` and `(x-1, \top) < (x, 0, \top)`. And as before, the constant "0" is arbitrary; anything less than \top (which is any Dafny expression) would work.

Let's take a look at one more example that better illustrates the utility of `\top`. Consider again two mutually recursive methods, call them `Outer` and `Inner`, representing the recursive counterparts of what iteratively might be two nested loops:

```
method Outer(x: nat)
{
  // set y to an arbitrary non-negative integer
  var y :| 0 <= y;
  Inner(x, y);
}

method Inner(x: nat, y: nat)
{
  if y != 0 {
    Inner(x, y-1);
  } else if x != 0 {
    Outer(x-1);
  }
}
```

The body of `Outer` uses an assign-such-that statement to represent some computation that takes place before `Inner` is called. It sets "y" to some arbitrary non-negative value. In a more concrete

example, `Inner` would do some work for each "y" and then continue as `Outer` on the next smaller "x".

Using a **decreases** clause `"x, y"` for `Inner` seems natural, but if we don't have any bound on the size of the `"y"` computed by `Outer`, there is no expression we can write in **decreases** clause of `Outer` that is sure to lead to a strictly smaller value for `"y"` when `Inner` is called. `\top` to the rescue. If we arrange for the effective decreases clause of `Outer` to be `"x, \top"` and the effective decreases clause for `Inner` to be `"x, y, \top"`, then we can show the strict decreases as required. Since `\top` is implicitly appended, the two decreases clauses declared in the program text can be:

```
method Outer(x: nat)
  decreases x

method Inner(x: nat, y: nat)
  decreases x, y
```

Moreover, remember that if a function or method has no user-declared **decreases** clause, Dafny will make a guess. The guess is (usually) the list of arguments of the function/method, in the order given. This is exactly the decreases clauses needed here. Thus, Dafny successfully verifies the program without any explicit decreases clauses:

```
method Outer(x: nat)
{
  var y :| 0 <= y;
  Inner(x, y);
}

method Inner(x: nat, y: nat)
{
  if y != 0 {
    Inner(x, y-1);
  } else if x != 0 {
    Outer(x-1);
  }
}
```

The ingredients are simple, but the end result may seem like magic. For many users, however, there may be no magic at all – the end result may be so natural that the user never even has to bothered to think about that there was a need to prove termination in the first place.

4.0.3. Framing

```
FrameExpression(allowLemma, allowLambda) =
  ( Expression(allowLemma, allowLambda) [ FrameField ]
  | FrameField )

FrameField = "`" Ident

PossiblyWildFrameExpression(allowLemma) =
    ( "*" | FrameExpression(allowLemma, allowLambda: false) )
```

Frame expressions are used to denote the set of memory locations that a Dafny program element may read or write. A frame expression is a set expression. The form `{}` (that is, the empty set) says that no memory locations may be modified, which is also the default if no **modifies** clause is given explicitly.

Note that framing only applies to the heap, or memory accessed through references. Local variables are not stored on the heap, so they cannot be mentioned (well, they are not in scope in the declaration) in reads annotations. Note also that types like sets, sequences, and multisets are value types, and are treated like integers or local variables. Arrays and objects are reference types, and they are stored on the heap (though as always there is a subtle distinction between the reference itself and the value it points to.)

The `FrameField` construct is used to specify a field of a class object. The identifier following the back-quote is the name of the field being referenced. If the `FrameField` is preceded by an expression the expression must be a reference to an object having that field. If the `FrameField` is not preceded by an expression then the frame expression is referring to that field of the current object. This form is only used from a method of a class.

The use of `FrameField` is discouraged as in practice it has not been shown to either be more concise or to perform better. Also, there's (unfortunately) no form of it for array elements—one could imagine

```
modifies a`[j]
```

Also, `FrameField` is not taken into consideration for lambda expressions.

4.0.4. Reads Clause

```
FunctionReadsClause_ =
  "reads"
  PossiblyWildFrameExpression (allowLemma: false)
  { "," PossiblyWildFrameExpression(allowLemma: false) }
LambdaReadsClause_ =
```

```
  "reads" PossiblyWildFrameExpression(allowLemma: true)
IteratorReadsClause_ =
  "reads" { Attribute }
  FrameExpression(allowLemma: false, allowLambda: false)
  { "," FrameExpression(allowLemma: false, allowLambda: false) }
PossiblyWildExpression(allowLambda) =
    ( "*" | Expression(allowLemma: false, allowLambda) )
```

Functions are not allowed to have side effects but may be restricted in what they can read. The *reading frame* of a function (or predicate) is all the memory locations that the function is allowed to read. The reason we might limit what a function can read is so that when we write to memory, we can be sure that functions that did not read that part of memory have the same value they did before. For example, we might have two arrays, one of which we know is sorted. If we did not put a reads annotation on the sorted predicate, then when we modify the unsorted array, we cannot determine whether the other array stopped being sorted. While we might be able to give invariants to preserve it in this case, it gets even more complex when manipulating data structures. In this case, framing is essential to making the verification process feasible.

It is not just the body of a function that is subject to **reads** checks, but also its precondition and the **reads** clause itself.

A reads clause can list a wildcard ("*"), which allows the enclosing function to read anything. In many cases, and in particular in all cases where the function is defined recursively, this makes it next to impossible to make any use of the function. Nevertheless, as an experimental feature, the language allows it (and it is sound). Note that a "*" makes the rest of the frame expression irrelevant.

A **reads** clause specifies the set of memory locations that a function, lambda, or iterator may read. If more than one **reads** clause is given in a specification the effective read set is the union of the sets specified. If there are no **reads** clauses the effective read set is empty. If `"*"` is given in a **reads** clause it means any memory may be read.

TODO: It would be nice if the different forms of read clauses could be combined. In a future version the single form of read clause will allow a list and attributes.

4.0.5. Modifies Clause

```
ModifiesClause_ =
  "modifies" { Attribute }
  FrameExpression(allowLemma: false, allowLambda: false)
  { "," FrameExpression(allowLemma: false, allowLambda: false) }
```

Frames also affect methods. As you might have guessed, methods are not required to list the

things they read. Methods are allowed to read whatever memory they like, but they are required to list which parts of memory they modify, with a modifies annotation. They are almost identical to their reads cousins, except they say what can be changed, rather than what the value of the function depends on. In combination with reads, modification restrictions allow Dafny to prove properties of code that would otherwise be very difficult or impossible. Reads and modifies are one of the tools that allow Dafny to work on one method at a time, because they restrict what would otherwise be arbitrary modifications of memory to something that Dafny can reason about.

Note that fields of newly allocated objects can always be modified.

It is also possible to frame what can be modified by a block statement by means of the block form of the modify statement (Section 21.16).

A **modifies** clause specifies the set of memory locations that a method, iterator or loop body may modify. If more than one **modifies** clause is given in a specification, the effective modifies set is the union of the sets specified. If no **modifies** clause is given the effective modifies set is empty. A loop can also have a **modifies** clause. If none is given, the loop gets to modify anything the enclosing context is allowed to modify.

4.0.6. Invariant Clause

```
InvariantClause_ =
    "invariant" { Attribute }
    Expression(allowLemma: false, allowLambda: true)
```

An **invariant** clause is used to specify an invariant for a loop. If more than one **invariant** clause is given for a loop the effective invariant is the conjunction of the conditions specified.

The invariant must hold on entry to the loop. And assuming it is valid on entry, Dafny must be able to prove that it then holds at the end of the loop.

4.1. Method Specification

```
MethodSpec =
  { ModifiesClause_
  | RequiresClause_
  | EnsuresClause_
  | DecreasesClause_(allowWildcard: true, allowLambda: false)
  }
```

A method specification is zero or more **modifies**, **requires**, **ensures** or **decreases** clauses, in any order. A method does not have **reads** clauses because methods are allowed to read any

memory.

4.2. Function Specification

```
FunctionSpec =
  { RequiresClause_
  | FunctionReadsClause_
  | FunctionEnsuresClause_
  | FunctionDecreasesClause_(allowWildcard: false, allowLambda: false)
  }
```

A function specification is zero or more **reads**, **requires**, **ensures** or **decreases** clauses, in any order. A function specification does not have **modifies** clauses because functions are not allowed to modify any memory.

4.3. Lambda Specification

```
LambdaSpec_ =
  { LambdaReadsClause_
  | RequiresClause_
  }
```

A lambda specification is zero or more **reads** or **requires** clauses. Lambda specifications do not have **ensures** clauses because the body is never opaque. Lambda specifications do not have **decreases** clauses because they do not have names and thus cannot be recursive. A lambda specification does not have **modifies** clauses because lambdas are not allowed to modify any memory.

4.4. Iterator Specification

```
IteratorSpec =
  { IteratorReadsClause_
  | ModifiesClause_
  | [ "yield" ] RequiresClause_
  | [ "yield" ] EnsuresClause_
  | DecreasesClause_(allowWildcard: false, allowLambda: false)
  }
```

An iterator specification applies both to the iterator's constructor method and to its `MoveNext` method. The **reads** and **modifies** clauses apply to both of them. For the **requires** and **ensures** clauses, if `yield` is not present they apply to the constructor, but if `yield` is present they apply to the `MoveNext` method.

TODO: What is the meaning of a **decreases** clause on an iterator? Does it apply to `MoveNext`? Make sure our description of iterators explains these.

TODO: What is the relationship between the post condition and the `Valid()` predicate?

4.5. Loop Specification

```
LoopSpec =
  { InvariantClause_
  | DecreasesClause_(allowWildcard: true, allowLambda: true)
  | ModifiesClause_
  }
```

A loop specification provides the information Dafny needs to prove properties of a loop. The `InvariantClause_` clause is effectively a precondition and it along with the negation of the loop test condition provides the postcondition. The `DecreasesClause_` clause is used to prove termination.

5. Types

```
Type = DomainType [ "->" Type ]
```

A Dafny type is a domain type (i.e. a type that can be the domain of a function type) optionally followed by an arrow and a range type.

```
DomainType =
  ( BoolType_ | CharType_ | NatType_ | IntType_ | RealType_ | ObjectType_
  | FiniteSetType_ | InfiniteSetType_ | MultisetType_
  | SequenceType_ | StringType_
  | FiniteMapType_ | InfiniteMapType_ | ArrayType_
  | TupleType_ | NamedType_ )
```

The domain types comprise the builtin scalar types, the builtin collection types, tuple types (including as a special case a parenthesized type) and reference types.

Dafny types may be categorized as either value types or reference types.

5.0. Value Types

The value types are those whose values do not lie in the program heap. These are:

- The basic scalar types: `bool`, `char`, `nat`, `int`, `real`
- The built-in collection types: `set`, `multiset`, `seq`, `string`, `map`, `imap`
- Tuple Types
- Inductive and co-inductive types

Data items having value types are passed by value. Since they are not considered to occupy *memory*, framing expressions do not reference them.

5.1. 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 reference types are class types, traits and array types.

The special value `null` is part of every reference type.[0]

5.2. Named Types

```
NamedType_ = NameSegmentForTypeName { "." NameSegmentForTypeName }
```

A `NamedType_` is used to specify a user-defined type by name (possibly module-qualified). Named types are introduced by class, trait, inductive, co-inductive, synonym and opaque type declarations. They are also used to refer to type variables.

```
NameSegmentForTypeName = Ident  [ GenericInstantiation ]
```

A `NameSegmentForTypeName` is a type name optionally followed by a `GenericInstantiation` which supplies type parameters to a generic type, if needed. It is a special case of a `NameSegment` (See Section 22.35) that does not allow a `HashCall`.

The following sections describe each of these kinds of types in more detail.

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

6. Basic types

Dafny offers these basic types: `bool` for booleans, `char` for characters, `int` and `nat` for integers, and `real` for reals.

6.0. Booleans

```
BoolType_ = "bool"
```

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.

6.0.0. Equivalence Operator

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
```

6.0.1. Conjunction and Disjunction

Conjunction is associative and so is disjunction. These operators 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 $\wedge$. The same holds for `||` and $\vee$.

6.0.2. Implication and Reverse Implication

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 5.1), 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 section 22.30.

6.1. Numeric types

```
IntType_ = "int"
RealType_ = "real"
```

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 19. Also, the *subset type* `nat`, representing the non-negative subrange of `int`, is described in Section 20.

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 hexadecimal 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 19.0.

6.2. Characters

```
CharType_ = "char"
```

Dafny supports a type `char` of *characters*. Character literals are enclosed in single quotes, as in `'D'`. Their form is described by the `charToken` nonterminal in the grammar. 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 9.2.4). 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 9.2.4.

The only other operations on characters are obtaining a character by indexing into a string, and the implicit conversion to string when used as a parameter of a `print` statement.

TODO: Are there any conversions between `char` values and numeric values?

7. Type parameters

```
GenericParameters = "<" TypeVariableName [ "(" "==" ")" ]
     { "," TypeVariableName [ "(" "==" ")" ] } ">"
```

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 "`(==)`".[1] 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

[1] 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.

type parameter must be restricted to equality-supporting types, in which case Dafny adds the "`(==)`" automatically.

TODO: Need to describe type inference somewhere.

8. Generic Instantiation

```
GenericInstantiation = "<" Type { "," Type } ">"
```

When a generic entity is used, actual types must be specified for each generic parameter. This is done using a `GenericInstantiation`. If the `GenericInstantiation` is omitted, type inference will try to fill these in.

9. Collection types

Dafny offers several built-in collection types.

9.0. Sets

```
FiniteSetType_ = "set" [ GenericInstantiation ]
InfiniteSetType_ = "iset" [ GenericInstantiation ]
```

For any type `T`, each value of type `set<T>` is a finite set of `T` values.

TODO: 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.

For any type `T`, each value of type `iset<T>` is a potentially infinite set of `T` values.

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 section 22.31.

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 associativity 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>` or `iset<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)`.

9.1. Multisets

```
MultisetType_ = "multiset" [ GenericInstantiation ]
```

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 associativity rules of `+`, `-`, and `*` are like those of the arithmetic operators with the same names. The `+` operator adds the multiplicity of corresponding elements, the `-` operator subtracts them (but 0 is the minimum multiplicity), and the `*` has multiplicity that is the minimum of the multiplicity of the operands.

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
```

9.2. Sequences

```
SequenceType_ = "seq" [ GenericInstantiation ]
```

For any type `T`, a value of type `seq<T>` denotes a *sequence* of `T` elements, that is, a mapping from a finite downward-closed set of natural numbers (called *indices*) to `T` values. (Thinking of it as a map, a sequence is therefore something of a dual of a multiset.)

9.2.0. Sequence Displays

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.

9.2.1. Sequence Relational Operators

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 `>=`.

9.2.2. Sequence Concatenation

Sequences support the following binary operator:

operator	description
`+`	concatenation

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

9.2.3. Other Sequence Expressions

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 $k + 1$ consecutive subsequences from `s`, each of the length indicated by the corresponding length designator, and returns these as a sequence of sequences.[2] 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] == [];
```

[2]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.

```
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.

9.2.4. Strings

```
StringType_ = "string"
```

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 6.2. 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 desirable, for example, when sorting strings.

9.3. Finite and Infinite Maps

```
FiniteMapType_ = "map" [ GenericInstantiation ]
InfiniteMapType_ = "imap" [ GenericInstantiation ]
```

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 (see `MapDisplayExpr`), 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.[3] There is also a *map comprehension expression*, explained in section 22.34.

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

[3] This is likely to change in the future to disallow multiple occurrences of the same key.

shorthand for `!(t in m)`.[4]

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
}
```

10. Types that stand for other types

```
SynonymTypeDecl =
  ( SynonymTypeDefinition_ | OpaqueTypeDefinition_ ) [ ";" ]
```

It is sometimes useful to know a type by several names or to treat a type abstractly. Synonym and opaque types serve this purpose.

10.0. Type synonyms

```
SynonymTypeDefinition_ =
  "type" { Attribute } SynonymTypeName [ GenericParameters ] "=" Type
```

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:

[4] 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.

```
type Replacements<T> = map<T,T>
type Vertex = int
```

As already described in Section 9.2.4, `string` is a built-in type synonym for `seq<char>`, as if it would have been declared as follows:

```
type string = seq<char>
```

10.1. Opaque types

```
OpaqueTypeDefinition_ = "type" { Attribute } SynonymTypeName
  [ "(" "==" ")" ] [ GenericParameters ]
```

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

```
type Y<T>
```

It is 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.

11. Well-founded Functions and Extreme Predicates

This section is a tutorial on well-founded functions and extreme predicates. We place it here in preparation for Section 12 where function and predicate definitions are described.

Recursive functions are a core part of computer science and mathematics. Roughly speaking, when the definition of such a function spells out a terminating computation from given arguments, we may refer to it as a *well-founded function*. For example, the common factorial and Fibonacci functions are well-founded functions.

There are also other ways to define functions. An important case regards the definition of a boolean function as an extreme solution (that is, a least or greatest solution) to some equation.

For computer scientists with interests in logic or programming languages, these *extreme predicates* are important because they describe the judgments that can be justified by a given set of inference rules (see, e.g., [3, 24, 28, 31, 36]).

To benefit from machine-assisted reasoning, it is necessary not just to understand extreme predicates but also to have techniques for proving theorems about them. A foundation for this reasoning was developed by Paulin-Mohring [29] and is the basis of the constructive logic supported by Coq [1] as well as other proof assistants [2, 34]. Essentially, the idea is to represent the knowledge that an extreme predicate holds by the proof term by which this knowledge was derived. For a predicate defined as the least solution, such proof terms are values of an inductive datatype (that is, finite proof trees), and for the greatest solution, a coinductive datatype (that is, possibly infinite proof trees). This means that one can use induction and coinduction when reasoning about these proof trees. Therefore, these extreme predicates are known as, respectively, *inductive predicates* and *coinductive predicates* (or, *co-predicates* for short). Support for extreme predicates is also available in the proof assistants Isabelle [30] and HOL [6].

Dafny supports both well-founded functions and extreme predicates. This section is a tutorial that describes the difference in general terms, and then describes novel syntactic support in Dafny for defining and proving lemmas with extreme predicates. Although Dafny's verifier has at its core a first-order SMT solver, Dafny's logical encoding makes it possible to reason about fixpoints in an automated way.

The encoding for coinductive predicates in Dafny was described previously [21] and is here described in Section 18.2.

11.0. Function Definitions

To define a function $f\colon X \to Y$ in terms of itself, one can write an equation like

$$f \quad = \quad \mathcal{F}(f) \tag{0}$$

where $\mathcal{F}$ is a non-recursive function of type $(X \to Y) \to X \to Y$. Because it takes a function as an argument, $\mathcal{F}$ is referred to as a *functor* (or *functional*, but not to be confused by the category-theory notion of a functor). Throughout, I will assume that $\mathcal{F}(f)$ by itself is well defined, for example that it does not divide by zero. I will also assume that f occurs only in fully applied calls in $\mathcal{F}(f)$; eta expansion can be applied to ensure this. If f is a boolean function, that is, if Y is the type of booleans, then I call f a *predicate*.

For example, the common Fibonacci function over the natural numbers can be defined by the equation

$$\mathit{fib} \quad = \quad \lambda n \bullet \text{ if } n < 2 \text{ then } n \text{ else } \mathit{fib}(n-2) + \mathit{fib}(n-1) \tag{1}$$

With the understanding that the argument n is universally quantified, we can write this equation equivalently as

$$\mathit{fib}(n) \quad = \quad \text{if } n < 2 \text{ then } n \text{ else } \mathit{fib}(n-2) + \mathit{fib}(n-1) \tag{2}$$

The fact that the function being defined occurs on both sides of the equation causes concern that we might not be defining the function properly, leading to a logical inconsistency. In general, there could be many solutions to an equation like (0) or there could be none. Let's consider two ways to make sure we're defining the function uniquely.

11.0.0. Well-founded Functions

A standard way to ensure that equation (0) has a unique solution in f is to make sure the recursion is well-founded, which roughly means that the recursion terminates. This is done by introducing any well-founded relation $\ll$ on the domain of f and making sure that the argument to each recursive call goes down in this ordering. More precisely, if we formulate (0) as

$$f(x) \quad = \quad \mathcal{F}'(f) \tag{3}$$

then we want to check $E \ll x$ for each call $f(E)$ in $\mathcal{F}'(f)$. When a function definition satisfies this *decrement condition*, then the function is said to be *well-founded*.

For example, to check the decrement condition for *fib* in (2), we can pick $\ll$ to be the arithmetic less-than relation on natural numbers and check the following, for any n:

$$2 \leq n \quad \Longrightarrow \quad n-2 \ll n \ \ \wedge \ \ n-1 \ll n \tag{4}$$

Note that we are entitled to using the antecedent $2 \leq n$, because that is the condition under which the else branch in (2) is evaluated.

A well-founded function is often thought of as "terminating" in the sense that the recursive *depth* in evaluating f on any given argument is finite. That is, there are no infinite descending chains of recursive calls. However, the evaluation of f on a given argument may fail to terminate, because its *width* may be infinite. For example, let P be some predicate defined on the ordinals and let *PDownward* be a predicate on the ordinals defined by the following equation:

$$\mathit{PDownward}(o) \quad = \quad P(o) \ \wedge \ \forall p \bullet \ p \ll o \implies \mathit{PDownward}(p) \tag{5}$$

With $\ll$ as the usual ordering on ordinals, this equation satisfies the decrement condition, but evaluating $\mathit{PDownward}(\omega)$ would require evaluating $\mathit{PDownward}(n)$ for every natural number n. However, what we are concerned about here is to avoid mathematical inconsistencies, and that is indeed a consequence of the decrement condition.

11.0.0.0. Example with Well-founded Functions So that we can later see how inductive proofs are done in Dafny, let's prove that for any n, $fib(n)$ is even iff n is a multiple of 3. We split our task into two cases. If $n < 2$, then the property follows directly from the definition of *fib*. Otherwise, note that exactly one of the three numbers $n-2$, $n-1$, and n is a multiple of 3. If n is the multiple of 3, then by invoking the induction hypothesis on $n-2$ and $n-1$, we obtain that $fib(n-2)+fib(n-1)$ is the sum of two odd numbers, which is even. If $n-2$ or $n-1$ is a multiple of 3, then by invoking the induction hypothesis on $n-2$ and $n-1$, we obtain that $fib(n-2)+fib(n-1)$ is the sum of an even number and an odd number, which is odd. In this proof, we invoked the induction hypothesis on $n-2$ and on $n-1$. This is allowed, because both are smaller than n, and hence the invocations go down in the well-founded ordering on natural numbers.

11.0.1. Extreme Solutions

We don't need to exclude the possibility of equation (0) having multiple solutions—instead, we can just be clear about which one of them we want. Let's explore this, after a smidgen of lattice theory.

For any complete lattice $(Y, \leq)$ and any set X, we can by *pointwise extension* define a complete lattice $(X \to Y, \ \dot{\Rightarrow}\ )$, where for any $f, g\colon X \to Y$,

$$f \ \dot{\Rightarrow}\ g \quad \equiv \quad \forall x \bullet f(x) \leq g(x) \tag{6}$$

In particular, if Y is the set of booleans ordered by implication (*false* $\leq$ *true*), then the set of predicates over any domain X forms a complete lattice. Tarski's Theorem [35] tells us that any monotonic function over a complete lattice has a least and a greatest fixpoint. In particular, this means that $\mathcal{F}$ has a least fixpoint and a greatest fixpoint, provided $\mathcal{F}$ is monotonic.

Speaking about the *set of solutions* in f to (0) is the same as speaking about the *set of fixpoints* of functor $\mathcal{F}$. In particular, the least and greatest solutions to (0) are the same as the least and greatest fixpoints of $\mathcal{F}$. In casual speak, it happens that we say "fixpoint of (0)", or more grotesquely, "fixpoint of f" when we really mean "fixpoint of $\mathcal{F}$".

In conclusion of our little excursion into lattice theory, we have that, under the proviso of $\mathcal{F}$ being monotonic, the set of solutions in f to (0) is nonempty, and among these solutions, there is in the $\dot{\Rightarrow}$ ordering a least solution (that is, a function that returns *false* more often than any other) and a greatest solution (that is, a function that returns *true* more often than any other).

When discussing extreme solutions, I will now restrict my attention to boolean functions (that is, with Y being the type of booleans). Functor $\mathcal{F}$ is monotonic if the calls to f in $\mathcal{F}'(f)$ are in *positive positions* (that is, under an even number of negations). Indeed, from now on, I will restrict my attention to such monotonic functors $\mathcal{F}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{g(0)}}{g(2)}}{g(4)}}{g(6)} \qquad\qquad \dfrac{\dfrac{\dfrac{\dfrac{\vdots}{g(-5)}}{g(-3)}}{g(-1)}}{g(1)}$$

Figure 0. Left: a finite proof tree that uses the rules of (9) to establish $g(6)$. Right: an infinite proof tree that uses the rules of (10) to establish $g(1)$.

Let me introduce a running example. Consider the following equation, where x ranges over the integers:

$$g(x) \quad = \quad (x = 0 \ \vee \ g(x-2)) \tag{7}$$

This equation has four solutions in g. With w ranging over the integers, they are:

$$\begin{array}{lcl} g(x) & \equiv & x \in \{w \mid 0 \leq w \ \wedge \ w \text{ even}\} \\ g(x) & \equiv & x \in \{w \mid w \text{ even}\} \\ g(x) & \equiv & x \in \{w \mid (0 \leq w \ \wedge \ w \text{ even}) \ \vee \ w \text{ odd}\} \\ g(x) & \equiv & x \in \{w \mid \mathit{true}\} \end{array} \tag{8}$$

The first of these is the least solution and the last is the greatest solution.

In the literature, the definition of an extreme predicate is often given as a set of *inference rules*. To designate the least solution, a single line separating the antecedent (on top) from conclusion (on bottom) is used:

$$\frac{}{g(0)} \qquad\qquad \frac{g(x-2)}{g(x)} \tag{9}$$

Through repeated applications of such rules, one can show that the predicate holds for a particular value. For example, the *derivation*, or *proof tree*, to the left in Figure 0 shows that $g(6)$ holds. (In this simple example, the derivation is a rather degenerate proof "tree".) The use of these inference rules gives rise to a least solution, because proof trees are accepted only if they are *finite*.

When inference rules are to designate the greatest solution, a double line is used:

$$\frac{}{\;g(0)\;} \qquad\qquad \frac{g(x-2)}{\;g(x)\;} \tag{10}$$

In this case, proof trees are allowed to be infinite. For example, the (partial depiction of the) infinite proof tree on the right in Figure 0 shows that $g(1)$ holds.

Note that derivations may not be unique. For example, in the case of the greatest solution for g, there are two proof trees that establish $g(0)$: one is the finite proof tree that uses the left-hand rule of (10) once, the other is the infinite proof tree that keeps on using the right-hand rule of (10).

11.0.2. Working with Extreme Predicates

In general, one cannot evaluate whether or not an extreme predicate holds for some input, because doing so may take an infinite number of steps. For example, following the recursive calls in the definition (7) to try to evaluate $g(7)$ would never terminate. However, there are useful ways to establish that an extreme predicate holds and there are ways to make use of one once it has been established.

For any $\mathcal{F}$ as in (0), I define two infinite series of well-founded functions, ${}^\flat f_k$ and ${}^\sharp f_k$ where k ranges over the natural numbers:

$$ {}^\flat f_k(x) \quad = \quad \begin{cases} \mathit{false} & \text{if } k = 0 \\ \mathcal{F}({}^\flat f_{k-1})(x) & \text{if } k > 0 \end{cases} \tag{11} $$

$$ {}^\sharp f_k(x) \quad = \quad \begin{cases} \mathit{true} & \text{if } k = 0 \\ \mathcal{F}({}^\sharp f_{k-1})(x) & \text{if } k > 0 \end{cases} \tag{12} $$

These functions are called the *iterates* of f, and I will also refer to them as the *prefix predicates* of f (or the *prefix predicate* of f, if we think of k as being a parameter). Alternatively, we can define ${}^\flat f_k$ and ${}^\sharp f_k$ without mentioning x: Let $\bot$ denote the function that always returns *false*, let $\top$ denote the function that always returns *true*, and let a superscript on $\mathcal{F}$ denote exponentiation (for example, $\mathcal{F}^0(f) = f$ and $\mathcal{F}^2(f) = \mathcal{F}(\mathcal{F}(f))$). Then, (11) and (12) can be stated equivalently as ${}^\flat f_k = \mathcal{F}^k(\bot)$ and ${}^\sharp f_k = \mathcal{F}^k(\top)$.

For any solution f to equation (0), we have, for any k and ℓ such that $k \leq \ell$:

$$ {}^\flat f_k \quad \dot{\Rightarrow} \quad {}^\flat f_\ell \quad \dot{\Rightarrow} \quad f \quad \dot{\Rightarrow} \quad {}^\sharp f_\ell \quad \dot{\Rightarrow} \quad {}^\sharp f_k \tag{13} $$

In other words, every ${}^\flat f_k$ is a *pre-fixpoint* of f and every ${}^\sharp f_k$ is a *post-fixpoint* of f. Next, I define two functions, $f^\downarrow$ and $f^\uparrow$, in terms of the prefix predicates:

$$ f^\downarrow(x) \quad = \quad \exists k \bullet {}^\flat f_k(x) \tag{14} $$

$$ f^\uparrow(x) \quad = \quad \forall k \bullet {}^\sharp f_k(x) \tag{15} $$

By (13), we also have that $f^\downarrow$ is a pre-fixpoint of $\mathcal{F}$ and $f^\uparrow$ is a post-fixpoint of $\mathcal{F}$. The marvelous thing is that, if $\mathcal{F}$ is *continuous*, then $f^\downarrow$ and $f^\uparrow$ are the least and greatest fixpoints of $\mathcal{F}$. These

equations let us do proofs by induction when dealing with extreme predicates. I will explain in Section 11.1.2 how to check for continuity.

Let's consider two examples, both involving function g in (7). As it turns out, g's defining functor is continuous, and therefore I will write $g^{\downarrow}$ and $g^{\uparrow}$ to denote the least and greatest solutions for g in (7).

11.0.2.0. Example with Least Solution The main technique for establishing that $g^{\downarrow}(x)$ holds for some x, that is, proving something of the form $Q \implies g^{\downarrow}(x)$, is to construct a proof tree like the one for $g(6)$ in Figure 0. For a proof in this direction, since we're just applying the defining equation, the fact that we're using a least solution for g never plays a role (as long as we limit ourselves to finite derivations).

The technique for going in the other direction, proving something *from* an established $g^{\downarrow}$ property, that is, showing something of the form $g^{\downarrow}(x) \implies R$, typically uses induction on the structure of the proof tree. When the antecedent of our proof obligation includes a predicate term $g^{\downarrow}(x)$, it is sound to imagine that we have been given a proof tree for $g^{\downarrow}(x)$. Such a proof tree would be a data structure—to be more precise, a term in an *inductive datatype*. For this reason, least solutions like $g^{\downarrow}$ have been given the name *inductive predicate*.

Let's prove $g^{\downarrow}(x) \implies 0 \leq x \ \land \ x$ even. We split our task into two cases, corresponding to which of the two proof rules in (9) was the last one applied to establish $g^{\downarrow}(x)$. If it was the left-hand rule, then $x = 0$, which makes it easy to establish the conclusion of our proof goal. If it was the right-hand rule, then we unfold the proof tree one level and obtain $g^{\downarrow}(x-2)$. Since the proof tree for $g^{\downarrow}(x-2)$ is smaller than where we started, we invoke the *induction hypothesis* and obtain $0 \leq (x-2) \ \land \ (x-2)$ even, from which it is easy to establish the conclusion of our proof goal.

Here's how we do the proof formally using (14). We massage the general form of our proof goal:

$$
\begin{array}{ll}
 & f^{\uparrow}(x) \implies R \\
= & \quad \{ \ (14) \ \} \\
 & (\exists k \bullet {}^{\flat}f_k(x)) \implies R \\
= & \quad \{ \text{ distribute } \implies \text{ over } \exists \text{ to the left } \} \\
 & \forall k \bullet ({}^{\flat}f_k(x) \implies R)
\end{array}
$$

The last line can be proved by induction over k. So, in our case, we prove ${}^{\flat}g_k(x) \implies 0 \leq x \ \land \ x$ even for every k. If $k = 0$, then ${}^{\flat}g_k(x)$ is *false*, so our goal holds trivially. If $k > 0$, then ${}^{\flat}g_k(x) = (x = 0 \ \lor \ {}^{\flat}g_{k-1}(x-2))$. Our goal holds easily for the first disjunct $(x = 0)$. For the other disjunct, we apply the induction hypothesis (on the smaller $k-1$ and with $x-2$) and obtain $0 \leq (x-2) \ \land \ (x-2)$ even, from which our proof goal follows.

11.0.2.1. Example with Greatest Solution We can think of a given predicate $g^{\uparrow}(x)$ as being represented by a proof tree—in this case a term in a *coinductive datatype*, since the proof may be infinite. For this reason, greatest solutions like $g^{\uparrow}$ have been given the name *coinductive predicate*, or *co-predicate* for short. The main technique for proving something from a given proof tree, that is, to prove something of the form $g^{\uparrow}(x) \implies R$, is to destruct the proof. Since this is just unfolding the defining equation, the fact that we're using a greatest solution for g never plays a role (as long as we limit ourselves to a finite number of unfoldings).

To go in the other direction, to establish a predicate defined as a greatest solution, like $Q \implies g^{\uparrow}(x)$, we may need an infinite number of steps. For this purpose, we can use induction's dual, *coinduction*. Were it not for one little detail, coinduction is as simple as continuations in programming: the next part of the proof obligation is delegated to the *coinduction hypothesis*. The little detail is making sure that it is the "next" part we're passing on for the continuation, not the same part. This detail is called *productivity* and corresponds to the requirement in induction of making sure we're going down a well-founded relation when applying the induction hypothesis. There are many sources with more information, see for example the classic account by Jacobs and Rutten [8] or a new attempt by Kozen and Silva that aims to emphasize the simplicity, not the mystery, of coinduction [11].

Let's prove $true \implies g^{\uparrow}(x)$. The intuitive coinductive proof goes like this: According to the right-hand rule of (10), $g^{\uparrow}(x)$ follows if we establish $g^{\uparrow}(x-2)$, and that's easy to do by invoking the coinduction hypothesis. The "little detail", productivity, is satisfied in this proof because we applied a rule in (10) before invoking the coinduction hypothesis.

For anyone who may have felt that the intuitive proof felt too easy, here is a formal proof using (15), which relies only on induction. We massage the general form of our proof goal:

$$
\begin{array}{ll}
 & Q \implies f^{\uparrow}(x) \\
= & \quad \{\ (15)\ \} \\
 & Q \implies \forall k \bullet {}^{\natural}f_k(x) \\
= & \quad \{\ \text{distribute } \implies \text{ over } \forall \text{ to the right}\ \} \\
 & \forall k \bullet Q \implies {}^{\natural}f_k(x)
\end{array}
$$

The last line can be proved by induction over k. So, in our case, we prove $true \implies {}^{\natural}g_k(x)$ for every k. If $k = 0$, then ${}^{\natural}g_k(x)$ is $true$, so our goal holds trivially. If $k > 0$, then ${}^{\natural}g_k(x) = (x = 0 \ \vee \ {}^{\natural}g_{k-1}(x-2))$. We establish the second disjunct by applying the induction hypothesis (on the smaller $k-1$ and with $x-2$).

11.0.3. Other Techniques

Although in this paper I consider only well-founded functions and extreme predicates, it is worth mentioning that there are additional ways of making sure that the set of solutions to (0) is nonempty. For example, if all calls to f in $\mathcal{F}'(f)$ are *tail-recursive calls*, then (under the

assumption that Y is nonempty) the set of solutions is nonempty. To see this, consider an attempted evaluation of $f(x)$ that fails to determine a definite result value because of an infinite chain of calls that applies f to each value of some subset X' of X. Then, apparently, the value of f for any one of the values in X' is not determined by the equation, but picking any particular result values for these makes for a consistent definition. This was pointed out by Manolios and Moore [25]. Functions can be underspecified in this way in the proof assistants ACL2 [10] and HOL [12].

11.1. Functions in Dafny

In this section, I explain with examples the support in Dafny[5] for well-founded functions, extreme predicates, and proofs regarding these.

11.1.0. Well-founded Functions in Dafny

Declarations of well-founded functions are unsurprising. For example, the Fibonacci function is declared as follows:

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

Dafny verifies that the body (given as an expression in curly braces) is well defined. This includes decrement checks for recursive (and mutually recursive) calls. Dafny predefines a well-founded relation on each type and extends it to lexicographic tuples of any (fixed) length. For example, the well-founded relation $x \ll y$ for integers is $x < y \ \wedge \ 0 \leq y$, the one for reals is $x \leq y - 1.0 \ \wedge \ 0.0 \leq y$ (this is the same ordering as for integers, if you read the integer relation as $x \leq y - 1 \ \wedge \ 0 \leq y$), the one for inductive datatypes is structural inclusion, and the one for coinductive datatypes is *false*.

Using a `decreases` clause, the programmer can specify the term in this predefined order. When a function definition omits a `decreases` clause, Dafny makes a simple guess. This guess (which can be inspected by hovering over the function name in the Dafny IDE) is very often correct, so users are rarely bothered to provide explicit `decreases` clauses.

If a function returns `bool`, one can drop the result type `: bool` and change the keyword `function` to `predicate`.

[5] Dafny is open source at dafny.codeplex.com and can also be used online at rise4fun.com/dafny.

11.1.1. Proofs in Dafny

Dafny has `lemma` declarations. These are really just special cases of methods: they can have pre- and postcondition specifications and their body is a code block. Here is the lemma we stated and proved in Section 11.0.0.0:

```
lemma FibProperty(n: nat)
  ensures fib(n) % 2 == 0 <==> n % 3 == 0
{
  if n < 2 {
  } else {
    FibProperty(n-2); FibProperty(n-1);
  }
}
```

The postcondition of this lemma (keyword `ensures`) gives the proof goal. As in any program-correctness logic (e.g., [7]), the postcondition must be established on every control path through the lemma's body. For `FibProperty`, I give the proof by an `if` statement, hence introducing a case split. The then branch is empty, because Dafny can prove the postcondition automatically in this case. The else branch performs two recursive calls to the lemma. These are the invocations of the induction hypothesis and they follow the usual program-correctness rules, namely: the precondition must hold at the call site, the call must terminate, and then the caller gets to assume the postcondition upon return. The "proof glue" needed to complete the proof is done automatically by Dafny.

Dafny features an aggregate statement using which it is possible to make (possibly infinitely) many calls at once. For example, the induction hypothesis can be called at once on all values `n'` smaller than `n`:

```
forall n' | 0 <= n' < n {
  FibProperty(n');
}
```

For our purposes, this corresponds to *strong induction*. More generally, the `forall` statement has the form

```
forall k | P(k)
  ensures Q(k)
{ Statements; }
```

Logically, this statement corresponds to *universal introduction*: the body proves that `Q(k)` holds for an arbitrary `k` such that `P(k)`, and the conclusion of the `forall` statement is then $\forall k \bullet P(k) \implies Q(k)$. When the body of the `forall` statement is a single call (or `calc`

statement), the `ensures` clause is inferred and can be omitted, like in our `FibProperty` example.

Lemma `FibProperty` is simple enough that its whole body can be replaced by the one `forall` statement above. In fact, Dafny goes one step further: it automatically inserts such a `forall` statement at the beginning of every lemma [19]. Thus, `FibProperty` can be declared and proved simply by:

```
lemma FibProperty(n: nat)
  ensures fib(n) % 2 == 0 <==> n % 3 == 0
{ }
```

Going in the other direction from universal introduction is existential elimination, also known as Skolemization. Dafny has a statement for this, too: for any variable `x` and boolean expression `Q`, the *assign such that* statement `x :| Q;` says to assign to `x` a value such that `Q` will hold. A proof obligation when using this statement is to show that there exists an `x` such that `Q` holds. For example, if the fact $\exists k \bullet 100 \leq \mathit{fib}(k) < 200$ is known, then the statement `k :| 100 <= fib(k) < 200;` will assign to `k` some value (chosen arbitrarily) for which `fib(k)` falls in the given range.

11.1.2. Extreme Predicates in Dafny

In this previous subsection, I explained that a `predicate` declaration introduces a well-founded predicate. The declarations for introducing extreme predicates are `inductive predicate` and `copredicate`. Here is the definition of the least and greatest solutions of g from above, let's call them `g` and `G`:

```
inductive predicate g(x: int) { x == 0 || g(x-2) }
copredicate G(x: int) { x == 0 || G(x-2) }
```

When Dafny receives either of these definitions, it automatically declares the corresponding prefix predicates. Instead of the names ${}^{\flat}g_k$ and ${}^{\sharp}g_k$ that I used above, Dafny names the prefix predicates `g#[k]` and `G#[k]`, respectively, that is, the name of the extreme predicate appended with `#`, and the subscript is given as an argument in square brackets. The definition of the prefix predicate derives from the body of the extreme predicate and follows the form in (11) and (12). Using a faux-syntax for illustrative purposes, here are the prefix predicates that Dafny defines automatically from the extreme predicates `g` and `G`:

```
predicate g#[_k: nat](x: int) { _k != 0 && (x == 0 || g#[_k-1](x-2)) }
predicate G#[_k: nat](x: int) { _k != 0 ==> (x == 0 || G#[_k-1](x-2)) }
```

The Dafny verifier is aware of the connection between extreme predicates and their prefix predicates, (14) and (15).

Remember that to be well defined, the defining functor of an extreme predicate must be monotonic, and for (14) and (15) to hold, the functor must be continuous. Dafny enforces the former of these by checking that recursive calls of extreme predicates are in positive positions. The continuity requirement comes down to checking that they are also in *continuous positions*: that recursive calls to inductive predicates are not inside unbounded universal quantifiers and that recursive calls to co-predicates are not inside unbounded existential quantifiers [21, 26].

11.1.3. Proofs about Extreme Predicates

From what I have presented so far, we can do the formal proofs from Sections 11.0.2.0 and 11.0.2.1. Here is the former:

```
lemma EvenNat(x: int)
  requires g(x)
  ensures 0 <= x && x % 2 == 0
{
  var k: nat :| g#[k](x);
  EvenNatAux(k, x);
}
lemma EvenNatAux(k: nat, x: int)
  requires g#[k](x)
  ensures 0 <= x && x % 2 == 0
{
  if x == 0 { } else { EvenNatAux(k-1, x-2); }
}
```

Lemma `EvenNat` states the property we wish to prove. From its precondition (keyword `requires`) and (14), we know there is some `k` that will make the condition in the assign-such-that statement true. Such a value is then assigned to `k` and passed to the auxiliary lemma, which promises to establish the proof goal. Given the condition `g#[k](x)`, the definition of `g#` lets us conclude `k != 0` as well as the disjunction `x == 0 || g#[k-1](x-2)`. The then branch considers the case of the first disjunct, from which the proof goal follows automatically. The else branch can then assume `g#[k-1](x-2)` and calls the induction hypothesis with those parameters. The proof glue that shows the proof goal for `x` to follow from the proof goal with `x-2` is done automatically.

Because Dafny automatically inserts the statement

```
forall k', x' | 0 <= k' < k && g#[k'](x') {
  EvenNatAux(k', x');
}
```

at the beginning of the body of `EvenNatAux`, the body can be left empty and Dafny completes

the proof automatically.

Here is the Dafny program that gives the proof from Section 11.0.2.1:

```
lemma Always(x: int)
  ensures G(x)
{ forall k: nat { AlwaysAux(k, x); } }
lemma AlwaysAux(k: nat, x: int)
  ensures G#[k](x)
{ }
```

While each of these proofs involves only basic proof rules, the setup feels a bit clumsy, even with the empty body of the auxiliary lemmas. Moreover, the proofs do not reflect the intuitive proofs I described in Section 11.0.2.0 and 11.0.2.1. These shortcoming are addressed in the next subsection.

11.1.4. Nicer Proofs of Extreme Predicates

The proofs we just saw follow standard forms: use Skolemization to convert the inductive predicate into a prefix predicate for some `k` and then do the proof inductively over `k`; respectively, by induction over `k`, prove the prefix predicate for every `k`, then use universal introduction to convert to the coinductive predicate. With the declarations `inductive lemma` and `colemma`, Dafny offers to set up the proofs in these standard forms. What is gained is not just fewer characters in the program text, but also a possible intuitive reading of the proofs. (Okay, to be fair, the reading is intuitive for simpler proofs; complicated proofs may or may not be intuitive.)

Somewhat analogous to the creation of prefix predicates from extreme predicates, Dafny automatically creates a *prefix lemma* `L#` from each "extreme lemma" `L`. The pre- and postconditions of a prefix lemma are copied from those of the extreme lemma, except for the following replacements: For an inductive lemma, Dafny looks in the precondition to find calls (in positive, continuous positions) to inductive predicates `P(x)` and replaces these with `P#[_k](x)`. For a co-lemma, Dafny looks in the postcondition to find calls (in positive, continuous positions) to co-predicates `P` (including equality among coinductive datatypes, which is a built-in co-predicate) and replaces these with `P#[_k](x)`. In each case, these predicates `P` are the lemma's *focal predicates*.

The body of the extreme lemma is moved to the prefix lemma, but with replacing each recursive call `L(x)` with `L#[_k-1](x)` and replacing each occurrence of a call to a focal predicate `P(x)` with `P#[_k-1](x)`. The bodies of the extreme lemmas are then replaced as shown in the previous subsection. By construction, this new body correctly leads to the extreme lemma's postcondition.

Let us see what effect these rewrites have on how one can write proofs. Here are the proofs of our running example:

```
inductive lemma EvenNat(x: int)
  requires g(x)
  ensures 0 <= x && x % 2 == 0
{ if x == 0 { } else { EvenNat(x-2); } }
colemma Always(x: int)
  ensures G(x)
{ Always(x-2); }
```

Both of these proofs follow the intuitive proofs given in Sections 11.0.2.0 and 11.0.2.1. Note that in these simple examples, the user is never bothered with either prefix predicates nor prefix lemmas—the proofs just look like "what you'd expect".

Since Dafny automatically inserts calls to the induction hypothesis at the beginning of each lemma, the bodies of the given extreme lemmas `EvenNat` and `Always` can be empty and Dafny still completes the proofs. Folks, it doesn't get any simpler than that!

12. Class Types

```
ClassDecl = "class" { Attribute } ClassName [ GenericParameters ]
  ["extends" Type {"," Type} ]
  "{" { { DeclModifier } ClassMemberDecl(moduleLevelDecl: false) } "}"

ClassMemberDecl(moduleLevelDecl) =
  ( FieldDecl | FunctionDecl |
    MethodDecl(isGhost: ("ghost" was present),
               allowConstructor: !moduleLevelDecl)
  )
```

The `ClassMemberDecl` parameter `moduleLevelDecl` will be true if the member declaration is at the top level or directly within a module declaration. It will be false for `ClassMemberDecl`s that are part of a class or trait declaration. If `moduleLevelDecl` is false `FieldDecl`s are not allowed.

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

```
class C<T> extends J1, ..., Jn
{
  members
}
```

where the list of type parameters `T` is optional and so is "`extends J1, ..., Jn`", which says that the class extends traits `J1` … `Jn`. 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`. A function or method that is not `static` is called an *instance* function or method.

An instance function or 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.

A clas can declare special initializing methods called *constructor methods*. See Section 12.1.

12.0. Field Declarations

```
FieldDecl = "var" { Attribute } FIdentType { "," FIdentType }
```

An FIdentType is used to declare a field. The field name is either an identifier (that is not allowed to start with a leading underscore) or some digits. Digits are used if you want to number your fields, e.g. "0", "1", etc.

```
FIdentType = ( FieldIdent | digits ) ":" Type
```

A field x of some type T is declared as:

```
var x: T
```

A field declaration declares one or more fields of the enclosing class. Each field is a named part of the state of an object of that class. A field declaration is similar to but distinct from a variable declaration statement. Unlike for local variables and bound variables, the type is required and will not be inferred.

Unlike method and function declarations, a field declaration cannot be given at the top level. Fields can be declared in either a class or a trait. A class that inherits from multiple traits will have all the fields declared in any of its parent traits.

Fields that are declared as ghost can only be used in specifications, not in code that will be compiled into executable code.

Fields may not be declared static.

protected is not allowed for fields.

12.1. Method Declarations

```
MethodDecl(isGhost, allowConstructor) =
 MethodKeyword { Attribute } [ MethodName ]
 (  MethodSignature(isGhost)  | SignatureEllipsis_ )
 MethodSpec [ BlockStmt ]
```

The isGhost parameter is true iff the ghost keyword preceded the method declaration.

If the allowConstructor parameter is false then the MethodDecl must not be a constructor declaration.

```
MethodKeyword = ("method" | "lemma" | "colemma"
                | "inductive" "lemma" | "constructor" )
```

The method keyword is used to specify special kinds of methods as explained below.

```
MethodSignature(isGhost) =
    [ GenericParameters ]
    Formals(allowGhost: !isGhost)
    [ "returns" Formals(allowGhost: !isGhost) ]
```

A method signature specifies the method generic parameters, input parameters and return parameters. The formal parameters are not allowed to have `ghost` specified if `ghost` was already specified for the method.

```
SignatureEllipsis_ = "…"
```

A `SignatureEllipsis_` is used when a method or function is being redeclared in module that refines another module. In that case the signature is copied from the module that is being refined. This works because Dafny does not support method or function overloading, so the name of the class method uniquely identifies it without the signature.

```
Formals(allowGhostKeyword) =
  "(" [ GIdentType(allowGhostKeyword)
        { "," GIdentType(allowGhostKeyword) } ] ")"
```

The `Formals` specifies the names and types of the method input or output parameters.

See section 4.1 for a description of `MethodSpec`.

A method declaration adheres to the `MethodDecl` grammar above. Here is an example of a method declaration.

```
method {:att1}{:att2} M<T1, T2>(a: A, b: B, c: C) returns (x: X, y: Y, z: Z)
  requires Pre
  modifies Frame
  ensures Post
  decreases Rank
{
  Body
}
```

where `:att1` and `:att2` are attributes of the method, `T1` and `T2` are type parameters of the method (if generic), `a, b, c` are the methodâ€™s in-parameters, `x, y, z` are the methodâ€™s out-parameters, `Pre` is a boolean expression denoting the methodâ€™s precondition, `Frame` denotes a set of objects whose fields may be updated by the method, `Post` is a boolean expression denoting the methodâ€™s postcondition, `Rank` is the methodâ€™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 methodâ€™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.

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(â€¦).

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.

An ordinary method is declared with the `method` keyword. Section 12.1.0 explains methods that instead use the `constructor` keyword. Section 12.1.1 discusses methods that are declared with the `lemma` keyword. Methods declared with the `inductive lemma` keywords are discussed later in the context of inductive predicates (see 18.0). Methods declared with the `colemma` keyword are discussed later in the context of co-inductive types, in section 18.2.4.1.

A method without is body is *abstract*. A method is allowed to be abstract under the following circumstances:

- It contains an `{:axiom}` attribute
- It contains an `{:imported}` attribute
- It contains a `{:decl}` attribute
- It is a declaration in an abstract module. Note that when there is no body, Dafny assumes that the *ensures* clauses are true without proof.

12.1.0. Constructors

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 construc-

tor. 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.

12.1.1. Lemmas

Sometimes there are steps of logic required to prove a program correct, but they are too complex for Dafny to discover and use on its own. When this happens, we can often give Dafny assistance by providing a lemma. This is done by declaring a method with the `lemma` keyword. Lemmas are implicitly ghost methods and the `ghost` keyword cannot be applied to them.

For an example, see the `FibProperty` lemma in Section 11.1.1.

See the Dafny Lemmas tutorial for more examples and hints for using lemmas.

12.2. Function Declarations

```
FunctionDecl =
  ( "function" [ "method" ] { Attribute }
    FunctionName
    FunctionSignatureOrEllipsis_(allowGhostKeyword: ("method" present))
  | "predicate" [ "method" ] { Attribute }
    PredicateName
    PredicateSignatureOrEllipsis_(allowGhostKeyword: ("method" present))
  | "inductive" "predicate" { Attribute }
    PredicateName
    PredicateSignatureOrEllipsis_(allowGhostKeyword: false)
  | "copredicate" { Attribute }
    CopredicateName
    PredicateSignatureOrEllipsis_(allowGhostKeyword: false)
  )
```

```
  FunctionSpec [ FunctionBody ]

FunctionSignatureOrEllipsis_(allowGhostKeyword) =
    FunctionSignature_ | SignatureEllipsis_
FunctionSignature_(allowGhostKeyword) =
    [ GenericParameters ] Formals(allowGhostKeyword) ":" Type

PredicateSignatureOrEllipsis_(allowGhostKeyword) =
    PredicateSignature_(allowGhostKeyword) | SignatureEllipsis_
PredicateSignature_(allowGhostKeyword) =
    [ GenericParameters ] Formals(allowGhostKeyword)

FunctionBody = "{" Expression(allowLemma: true, allowLambda: true) "}"
```

In the above productions, allowGhostKeyword is true if the optional "method" keyword was specified. This allows some of the formal parameters of a function method to be specified as ghost.

See section 4.2 for a description of FunctionSpec.

A Dafny function is a pure mathematical function. It is allowed to read memory that was specified in its reads expression but is not allowed to have any side effects.

Here is an example function declaration:

```
function {:att1}{:att2} F<T1, T2>(a: A, b: B, c: C): T
  requires Pre
  reads Frame
  ensures Post
  decreases Rank
{
  Body
}
```

where :att1 and :att2 are attributes of the function, if any, T1 and T2 are type parameters of the function (if generic), a, b, c are the functionsâ€™s parameters, T is the type of the functionâ€™s result, Pre is a boolean expression denoting the functionâ€™s precondition, Frame denotes a set of objects whose fields the function body may depend on, Post is a boolean expression denoting the functionâ€™s postcondition, Rank is the functionâ€™s variant function, and Body is an expression that defines the function return value. 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 functionâ€™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 `C.F(â€¦)`. This can give a convenient way to declare a number of helper functions in a separate class.

As for methods, a `SignatureEllipsis_` is used when declaring a function in a module refinement. For example, if module `M0` declares function `F`, a module `M1` can be declared to refine `M0` and `M1` can then refine `F`. The refinement function, `M1.F` can have a `SignatureEllipsis_` which means to copy the signature form `M0.F`. A refinement function can furnish a body for a function (if `M0.F` does not provide one). It can also add **ensures** clauses. And if `F` is a predicate, it can add conjuncts to a previously given body.

12.2.0. Function Transparency

A function is said to be *transparent* in a location if the contents of the body of the function is visible at that point. A function is said to be *opaque* at a location if it is not transparent. However the `FunctionSpec` of a function is always available.

A function is usually transparent up to some unrolling level (up to 1, or maybe 2 or 3). If its arguments are all literals it is transparent all the way.

But the transparency of a function is affected by the following:

- whether the function was declared to be protected, and
- whether the function was given the `{:opaque}` attribute (as explained in Section 24.1.12).

The following table summarizes where the function is transparent. The module referenced in the table is the module in which the function is defined.

Protected?	{:opaque}?	Transparent Inside Module	Transparent Outside Module
N	N	Y	Y
Y	N	Y	N
N	Y	N	N

When {:opaque} is specified for function g, g is opaque, however the lemma reveal_g is available to give the semantics of g whether in the defining module or outside.

It currently is not allowed to have both protected and {:opaque} specified for a function.

12.2.1. Predicates

A function that returns a bool results is called a *predicate*. As an alternative syntax, a predicate can be declared by replacing the function keyword with the predicate keyword and omitting a declaration of the return type.

12.2.2. Inductive Predicates and Lemmas

See section 11.1.2 for descriptions of inductive predicates and lemmas.

13. Trait Types

```
TraitDecl = "trait" { Attribute } TraitName [ GenericParameters ]
  "{" { { DeclModifier } ClassMemberDecl(moduleLevelDecl: false) } "}"
```

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 contains two Shape values, of different classes
myShapes := myShapes + [tri];
// move shape 1 to the right by the width of shape 0
myShapes[1].MoveH(myShapes[0].Width());
```

14. Array Types

```
ArrayType_ = arrayToken [ GenericInstantiation ]
```

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

14.0. 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[..]))
```

14.1. 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.

15. Type object

```
ObjectType_ = "object"
```

There is a built-in trait `object` that is like a supertype of all reference types.[6] Every class automatically extends object and so does every user-defined trait. 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>`.

16. Iterator types

```
IteratorDecl = "iterator" { Attribute } IteratorName
  ( [ GenericParameters ]
    Formals(allowGhostKeyword: true)
    [ "yields" Formals(allowGhostKeyword: true) ]
  | "…"
  )
  IteratorSpec [ BlockStmt ]
```

See section 4.4 for a description of `IteratorSpec`.

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

[6] The current compiler restriction that `object` cannot be used as a type parameter needs to be removed.

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 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.[7]

[7] It would make sense to rename the special fields `_reads` and `_modifies` to have the same names as the

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`.

It's regrettably tricky to use iterators. The language really ought to have a `foreach` statement to make this easier. Here is an example showing definition and use of an iterator.

```
iterator Iter<T>(s: set<T>) yields (x: T)
  yield ensures x in s && x !in xs[..|xs|-1];
  ensures s == set z | z in xs;
{
  var r := s;
  while (r != {})
    invariant forall z :: z in xs ==> x !in r;  // r and xs are disjoint
    invariant s == r + set z | z in xs;
  {
    var y :| y in r;
    r, x := r - {y}, y;
    yield;
    assert y == xs[|xs|-1];  // needed as a lemma to prove loop invariant
  }
}

method UseIterToCopy<T>(s: set<T>) returns (t: set<T>)
  ensures s == t;
{
  t := {};
  var m := new Iter(s);
  while (true)
    invariant m.Valid() && fresh(m._new);
    invariant t == set z | z in m.xs;
    decreases s - t;
```

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. Thse changes may be incorporated into a future version of Dafny.

```
  {
    var more := m.MoveNext();
    if (!more) { break; }
    t := t + {m.x};
  }
}
```

17. Function types

```
Type = DomainType "->" Type
```

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 the fact 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
```

Dafny also support anonymous functions by means of *lambda expressions*. See section 22.9.

18. Algebraic 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.

```
DatatypeDecl = ( InductiveDatatypeDecl | CoinductiveDatatypeDecl )
```

18.0. Inductive datatypes

```
InductiveDatatypeDecl_ = "datatype" { Attribute } DatatypeName [ GenericParameters ]
  "=" DatatypeMemberDecl { "|" DatatypeMemberDecl } [ ";" ]
DatatypeMemberDecl = { Attribute } DatatypeMemberName [ FormalsOptionalIds ]
```

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))
```

The datatype update expression also accepts multiple field names, provided these are distinct. For example, a node of some inductive datatype for trees may be updated as follows:

```
node[left := L, right := R]
```

18.1. Tuple types

```
TupleType_ = "(" [ Type { "," Type } ] ")"
```

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*.

18.2. Co-inductive datatypes

```
CoinductiveDatatypeDecl_ = "codatatype" { Attribute } DatatypeName [ GenericParameters ]
  "=" DatatypeMemberDecl { "|" DatatypeMemberDecl } [ ";" ]
```

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.

The paper Co-induction Simply, by Leino and Moskal[20], explains Dafny's implementation and verification of co-inductive types. We capture the key features from that paper in this section but the reader is referred to that paper for more complete details and to supply bibliographic references that we have omitted.

Mathematical induction is a cornerstone of programming and program verification. It arises in data definitions (e.g., some algebraic data structures can be described using induction), it underlies program semantics (e.g., it explains how to reason about finite iteration and recursion), and it gets used in proofs (e.g., supporting lemmas about data structures use inductive proofs). Whereas induction deals with finite things (data, behavior, etc.), its dual, co-induction, deals with possibly infinite things. Co-induction, too, is important in programming and program verification, where it arises in data definitions (e.g., lazy data structures), semantics (e.g., concurrency), and proofs (e.g., showing refinement in a co-inductive big-step semantics). It is thus desirable to have good support for both induction and co-induction in a system for constructing and reasoning about programs.

Co-datatypes and co-recursive functions make it possible to use lazily evaluated data structures (like in Haskell or Agda). Co-predicates, defined by greatest fix-points, let programs state properties of such data structures (as can also be done in, for example, Coq). For the purpose of

writing co-inductive proofs in the language, we introduce co-lemmas. Ostensibly, a co-lemma invokes the co-induction hypothesis much like an inductive proof invokes the induction hypothesis. Underneath the hood, our co-inductive proofs are actually approached via induction: co-lemmas provide a syntactic veneer around this approach.

The following example gives a taste of how the co-inductive features in Dafny come together to give straightforward definitions of infinite matters.

```
// infinite streams
codatatype IStream<T> = ICons(head: T, tail: IStream)

// pointwise product of streams
function Mult(a: IStream<int>, b: IStream<int>): IStream<int>
{ ICons(a.head * b.head, Mult(a.tail, b.tail)) }

// lexicographic order on streams
copredicate Below(a: IStream<int>, b: IStream<int>)
{ a.head <= b.head && ((a.head == b.head) ==> Below(a.tail, b.tail)) }

// a stream is Below its Square
colemma Theorem_BelowSquare(a: IStream<int>)
ensures Below(a, Mult(a, a))
{ assert a.head <= Mult(a, a).head;
  if a.head == Mult(a, a).head {
    Theorem_BelowSquare(a.tail);
  }
}

// an incorrect property and a bogus proof attempt
colemma NotATheorem_SquareBelow(a: IStream<int>)
  ensures Below(Mult(a, a), a); // ERROR
{
  NotATheorem_SquareBelow(a);
}
```

It defines a type `IStream` of infinite streams, with constructor `ICons` and destructors `head` and `tail`. Function `Mult` performs pointwise multiplication on infinite streams of integers, defined using a co-recursive call (which is evaluated lazily). Co-predicate `Below` is defined as a greatest fix-point, which intuitively means that the co-predicate will take on the value true if the recursion goes on forever without determining a different value. The co-lemma states the theorem

`Below(a, Mult(a, a))`. Its body gives the proof, where the recursive invocation of the co-lemma corresponds to an invocation of the co-induction hypothesis.

The proof of the theorem stated by the first co-lemma lends itself to the following intuitive reading: To prove that `a` is below `Mult(a, a)`, check that their heads are ordered and, if the heads are equal, also prove that the tails are ordered. The second co-lemma states a property that does not always hold; the verifier is not fooled by the bogus proof attempt and instead reports the property as unproved.

We argue that these definitions in Dafny are simple enough to level the playing field between induction (which is familiar) and co-induction (which, despite being the dual of induction, is often perceived as eerily mysterious). Moreover, the automation provided by our SMT-based verifier reduces the tedium in writing co-inductive proofs. For example, it verifies `Theorem_BelowSquare` from the program text given aboveâ€" no additional lemmas or tactics are needed. In fact, as a consequence of the automatic-induction heuristic in Dafny, the verifier will automatically verify Theorem_BelowSquare even given an empty body.

Just like there are restrictions on when an *inductive hypothesis* can be invoked, there are restriction on how a *co-inductive* hypothesis can be *used*. These are, of course, taken into consideration by our verifier. For example, as illustrated by the second co-lemma above, invoking the co-inductive hypothesis in an attempt to obtain the entire proof goal is futile. (We explain how this works in section 18.2.4.1) Our initial experience with co-induction in Dafny shows it to provide an intuitive, low-overhead user experience that compares favorably to even the best of todayâ€™s interactive proof assistants for co-induction. In addition, the co-inductive features and verification support in Dafny have other potential benefits. The features are a stepping stone for verifying functional lazy programs with Dafny. Co-inductive features have also shown to be useful in defining language semantics, as needed to verify the correctness of a compiler, so this opens the possibility that such verifications can benefit from SMT automation.

18.2.0. Well-Founded Function/Method Definitions

The Dafny programming language supports functions and methods. A *function* in Dafny is a mathematical function (i.e., it is well-defined, deterministic, and pure), whereas a *method* is a body of statements that can mutate the state of the program. A function is defined by its given body, which is an expression. To ensure that function definitions are mathematically consistent, Dafny insists that recursive calls be well-founded, enforced as follows: Dafny computes the call graph of functions. The strongly connected components within it are *clusters* of mutually recursive definitions arranged in a DAG. This stratifies the functions so that a call from one cluster in the DAG to a lower cluster is allowed arbitrarily. For an intra-cluster call, Dafny prescribes a proof obligation that gets taken through the program verifierâ€™s reasoning engine. Semantically, each function activation is labeled by a *rank*â€"a lexicographic tuple

determined by evaluating the functionâ€™s **decreases** clause upon invocation of the function. The proof obligation for an intra-cluster call is thus that the rank of the callee is strictly less (in a language-defined well-founded relation) than the rank of the caller. Because these well-founded checks correspond to proving termination of executable code, we will often refer to them as â€œtermination checksâ€ . The same process applies to methods.

Lemmas in Dafny are commonly introduced by declaring a method, stating the property of the lemma in the *postcondition* (keyword **ensures**) of the method, perhaps restricting the domain of the lemma by also giving a *precondition* (keyword **requires**), and using the lemma by invoking the method. Lemmas are stated, used, and proved as methods, but since they have no use at run time, such lemma methods are typically declared as *ghost*, meaning that they are not compiled into code. The keyword **lemma** introduces such a method. Control flow statements correspond to proof techniquesâ€”case splits are introduced with if statements, recursion and loops are used for induction, and method calls for structuring the proof. Additionally, the statement:

```
forall x | P(x) { Lemma(x); }
```

is used to invoke `Lemma(x)` on all `x` for which `P(x)` holds. If `Lemma(x)` ensures `Q(x)`, then the forall statement establishes

```
forall x :: P(x) ==> Q(x).
```

18.2.1. Defining Co-inductive Datatypes

Each value of an inductive datatype is finite, in the sense that it can be constructed by a finite number of calls to datatype constructors. In contrast, values of a co-inductive datatype, or co-datatype for short, can be infinite. For example, a co-datatype can be used to represent infinite trees.

Syntactically, the declaration of a co-datatype in Dafny looks like that of a datatype, giving prominence to the constructors (following Coq). The following example defines a co-datatype Stream of possibly infinite lists.

```
codatatype Stream<T> = SNil | SCons(head: T, tail: Stream)
function Up(n: int): Stream<int> { SCons(n, Up(n+1)) }
function FivesUp(n: int): Stream<int>
  decreases 4 - (n - 1) % 5
{
  if (n % 5 == 0) then
    SCons(n, FivesUp(n+1))
  else
```

```
    FivesUp(n+1)
}
```

`Stream` is a co-inductive datatype whose values are possibly infinite lists. Function `Up` returns a stream consisting of all integers upwards of `n` and `FivesUp` returns a stream consisting of all multiples of 5 upwards of `n` . The self-call in `Up` and the first self-call in `FivesUp` sit in productive positions and are therefore classified as co-recursive calls, exempt from termination checks. The second self-call in `FivesUp` is not in a productive position and is therefore subject to termination checking; in particular, each recursive call must decrease the rank defined by the **decreases** clause.

Analogous to the common finite list datatype, Stream declares two constructors, `SNil` and `SCons`. Values can be destructed using match expressions and statements. In addition, like for inductive datatypes, each constructor `C` automatically gives rise to a discriminator `C?` and each parameter of a constructor can be named in order to introduce a corresponding destructor. For example, if `xs` is the stream `SCons(x, ys)`, then `xs.SCons?` and `xs.head == x` hold. In contrast to datatype declarations, there is no grounding check for co-datatypesâ€"since a codatatype admits infinite values, the type is nevertheless inhabited.

18.2.2. Creating Values of Co-datatypes

To define values of co-datatypes, one could imagine a â€œco-functionâ€ language feature: the body of a â€œco-functionâ€ could include possibly never-ending self-calls that are interpreted by a greatest fix-point semantics (akin to a **CoFixpoint** in Coq). Dafny uses a different design: it offers only functions (not â€œco-functionsâ€), but it classifies each intra-cluster call as either *recursive* or *co-recursive*. Recursive calls are subject to termination checks. Co-recursive calls may be never-ending, which is what is needed to define infinite values of a co-datatype. For example, function `Up(n )` in the preceding example is defined as the stream of numbers from `n` upward: it returns a stream that starts with `n` and continues as the co-recursive call `Up(n + 1)`.

To ensure that co-recursive calls give rise to mathematically consistent definitions, they must occur only in productive positions. This says that it must be possible to determine each successive piece of a co-datatype value after a finite amount of work. This condition is satisfied if every co-recursive call is syntactically guarded by a constructor of a co-datatype, which is the criterion Dafny uses to classify intra-cluster calls as being either co-recursive or recursive. Calls that are classified as co-recursive are exempt from termination checks.

A consequence of the productivity checks and termination checks is that, even in the absence of talking about least or greatest fix-points of self-calling functions, all functions in Dafny are deterministic. Since there is no issue of several possible fix-points, the language allows one function to be involved in both recursive and co-recursive calls, as we illustrate by the function `FivesUp`.

18.2.3. Copredicates

Determining properties of co-datatype values may require an infinite number of observations. To that avail, Dafny provides *co-predicates* which are function declarations that use the `copredicate` keyword. Self-calls to a co-predicate need not terminate. Instead, the value defined is the greatest fix-point of the given recurrence equations. Continuing the preceding example, the following code defines a co-predicate that holds for exactly those streams whose payload consists solely of positive integers. The co-predicate definition implicitly also gives rise to a corresponding prefix predicate, `Pos#`. The syntax for calling a prefix predicate sets apart the argument that specifies the prefix length, as shown in the last line; for this figure, we took the liberty of making up a coordinating syntax for the signature of the automatically generated prefix predicate (which is not part of Dafny syntax).

```
copredicate Pos(s: Stream<int>)
{
  match s
  case SNil => true
  case SCons(x, rest) => x > 0 && Pos(rest)
}
// Automatically generated by the Dafny compiler:
predicate Pos#[_k: nat](s: Stream<int>)
  decreases _k
{ if _k = 0 then true else
  match s
  case SNil => true
  case SCons(x, rest) => x > 0 && Pos#[_k-1](rest)
}
```

Some restrictions apply. To guarantee that the greatest fix-point always exists, the (implicit functor defining the) co-predicate must be monotonic. This is enforced by a syntactic restriction on the form of the body of co-predicates: after conversion to negation normal form (i.e., pushing negations down to the atoms), intra-cluster calls of co-predicates must appear only in *positive* positionsâ€”that is, they must appear as atoms and must not be negated. Additionally, to guarantee soundness later on, we require that they appear in *co-friendly* positionsâ€”that is, in negation normal form, when they appear under existential quantification, the quantification needs to be limited to a finite range[8]. Since the evaluation of a co-predicate might not terminate, co-predicates are always ghost. There is also a restriction on the call graph that a cluster

[8]Higher-order function support in Dafny is rather modest and typical reasoning patterns do not involve them, so this restriction is not as limiting as it would have been in, e.g., Coq.

containing a co-predicate must contain only co-predicates, no other kinds of functions.

A **copredicate** declaration of `P` defines not just a co-predicate, but also a corresponding *prefix predicate* `P#`. A prefix predicate is a finite unrolling of a co-predicate. The prefix predicate is constructed from the co-predicate by

- adding a parameter _k of type nat to denote the prefix length,
- adding the clause "**decreases** `_k;`" to the prefix predicate (the co-predicate itself is not allowed to have a decreases clause),
- replacing in the body of the co-predicate every intra-cluster call `Q(args)` to a copredicate by a call `Q#[_k - 1](args)` to the corresponding prefix predicate, and then
- prepending the body with `if _k = 0 then true else`.

For example, for co-predicate `Pos`, the definition of the prefix predicate `Pos#` is as suggested above. Syntactically, the prefix-length argument passed to a prefix predicate to indicate how many times to unroll the definition is written in square brackets, as in `Pos#[k](s)`. In the Dafny grammar this is called a `HashCall`. The definition of `Pos#` is available only at clusters strictly higher than that of `Pos`; that is, `Pos` and `Pos#` must not be in the same cluster. In other words, the definition of `Pos` cannot depend on `Pos#`.

18.2.3.0. Co-Equality Equality between two values of a co-datatype is a built-in co-predicate. It has the usual equality syntax `s == t`, and the corresponding prefix equality is written `s ==#[k] t`. And similarly for `s != t` and `s !=#[k] t`.

18.2.4. Co-inductive Proofs

From what we have said so far, a program can make use of properties of co-datatypes. For example, a method that declares `Pos(s)` as a precondition can rely on the stream `s` containing only positive integers. In this section, we consider how such properties are established in the first place.

18.2.4.0. Properties About Prefix Predicates Among other possible strategies for establishing co-inductive properties we take the time-honored approach of reducing co-induction to induction. More precisely, Dafny passes to the SMT solver an assumption `D(P)` for every co-predicate `P`, where:

```
D(P) = ? x â€¢ P(x) <==> ? k â€¢ P#[k](x)
```

In other words, a co-predicate is true iff its corresponding prefix predicate is true for all finite unrollings.

In Sec. 4 of the paper Co-induction Simply a soundness theorem of such assumptions is given, provided the co-predicates meet the co-friendly restrictions. An example proof of `Pos(Up(n))` for every `n > 0` is here shown:

```
lemma UpPosLemma(n: int)
  requires n > 0
  ensures Pos(Up(n))
{
  forall k | 0 <= k { UpPosLemmaK(k, n); }
}

lemma UpPosLemmaK(k: nat, n: int)
  requires n > 0
  ensures Pos#[k](Up(n))
  decreases k
{
  if k != 0 {
    // this establishes Pos#[k-1](Up(n).tail)
    UpPosLemmaK(k-1, n+1);
  }
}
```

The lemma `UpPosLemma` proves `Pos(Up(n))` for every `n > 0`. We first show `Pos#[k](Up(n ))`, for `n > 0` and an arbitrary `k`, and then use the forall statement to show `? k â€¢ Pos#[k](Up(n))`. Finally, the axiom `D(Pos)` is used (automatically) to establish the co-predicate.

18.2.4.1. Colemmas As we just showed, with help of the `D` axiom we can now prove a co-predicate by inductively proving that the corresponding prefix predicate holds for all prefix lengths `k` . In this section, we introduce *co-lemma* declarations, which bring about two benefits. The first benefit is that co-lemmas are syntactic sugar and reduce the tedium of having to write explicit quantifications over `k` . The second benefit is that, in simple cases, the bodies of co-lemmas can be understood as co-inductive proofs directly. As an example consider the following co-lemma.

```
colemma UpPosLemma(n: int)
  requires n > 0
  ensures Pos(Up(n))
{
```

```
    UpPosLemma(n+1);
  }
```

This co-lemma can be understood as follows: `UpPosLemma` invokes itself co-recursively to obtain the proof for `Pos(Up(n).tail)` (since `Up(n).tail` equals `Up(n+1)`). The proof glue needed to then conclude `Pos(Up(n))` is provided automatically, thanks to the power of the SMT-based verifier.

18.2.4.2. Prefix Lemmas To understand why the above `UpPosLemma` co-lemma code is a sound proof, let us now describe the details of the desugaring of co-lemmas. In analogy to how a **copredicate** declaration defines both a co-predicate and a prefix predicate, a **colemma** declaration defines both a co-lemma and *prefix lemma*. In the call graph, the cluster containing a co-lemma must contain only co-lemmas and prefix lemmas, no other methods or function. By decree, a co-lemma and its corresponding prefix lemma are always placed in the same cluster. Both co-lemmas and prefix lemmas are always ghosts.

The prefix lemma is constructed from the co-lemma by

- adding a parameter `_k` of type `nat` to denote the prefix length,
- replacing in the co-lemmaâ€™s postcondition the positive co-friendly occurrences of co-predicates by corresponding prefix predicates, passing in `_k` as the prefix-length argument,
- prepending `_k` to the (typically implicit) **decreases** clause of the co-lemma,
- replacing in the body of the co-lemma every intra-cluster call `M(args)` to a colemma by a call `M#[_k - 1](args)` to the corresponding prefix lemma, and then
- making the bodyâ€™s execution conditional on `_k != 0`.

Note that this rewriting removes all co-recursive calls of co-lemmas, replacing them with recursive calls to prefix lemmas. These recursive call are, as usual, checked to be terminating. We allow the pre-declared identifier `_k` to appear in the original body of the co-lemma.[9]

We can now think of the body of the co-lemma as being replaced by a **forall** call, for every k , to the prefix lemma. By construction, this new body will establish the colemmaâ€™s declared postcondition (on account of the `D` axiom, and remembering that only the positive co-friendly occurrences of co-predicates in the co-lemmaâ€™s postcondition are rewritten), so there is no reason for the program verifier to check it.

The actual desugaring of our co-lemma `UpPosLemma` is in fact the previous code for the `UpPosLemma` lemma except that `UpPosLemmaK` is named `UpPosLemma#` and modulo a minor syntactic difference in how the `k` argument is passed.

[9]Note, two places where co-predicates and co-lemmas are not analogous are: co-predicates must not make recursive calls to their prefix predicates, and co-predicates cannot mention _k.

In the recursive call of the prefix lemma, there is a proof obligation that the prefixlength argument `_k - 1` is a natural number. Conveniently, this follows from the fact that the body has been wrapped in an `if _k != 0` statement. This also means that the postcondition must hold trivially when `_k = 0`, or else a postcondition violation will be reported. This is an appropriate design for our desugaring, because co-lemmas are expected to be used to establish co-predicates, whose corresponding prefix predicates hold trivially when `_k = 0`. (To prove other predicates, use an ordinary lemma, not a co-lemma.)

It is interesting to compare the intuitive understanding of the co-inductive proof in using a co-lemma with the inductive proof in using the lemma. Whereas the inductive proof is performing proofs for deeper and deeper equalities, the co-lemma can be understood as producing the infinite proof on demand.

19. Newtypes

```
NewtypeDecl = "newtype" { Attribute } NewtypeName "="
  ( NumericTypeName [ ":" Type ] "|" Expression(allowLemma: false, allowLambda: true)
  | Type
  )
```

A new numeric type can be declared with the *newtype* declaration[10], for example:

```
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.[11]

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

[10] Should `newtype` perhaps be renamed to `numtype`?

[11] Would it be useful to also automatically define `predicate N?(m: M) { Q }`?

```
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`. Consequently, 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
```

If possible Dafny will represent values of the newtype using a native data type for the sake of efficiency. This action can be inhibited or a specific native data type selected by using the `(:nativeType)` attribute, as explained in section 24.1.11.

There is a restriction that the value `0` must be part of every newtype.[12]

[12]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.

19.0. 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 6.1.)

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.

20. Subset types

```
NatType_ = "nat"
```

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 19.0.)

Dafny supports one subset type, namely the built-in type `nat`, whose base type is `int`.[13] 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
{
```

[13] A future version of Dafny will support user-defined subset types.

```
  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`.

21. Statements

```
Stmt = ( BlockStmt | AssertStmt | AssumeStmt | PrintStmt | UpdateStmt
  | VarDeclStatement | IfStmt | WhileStmt | MatchStmt | ForallStmt
  | CalcStmt | ModifyStmt | LabeledStmt_ | BreakStmt_ | ReturnStmt
  | YieldStmt | SkeletonStmt
  )
```

Many of Dafny's statements are similar to those in traditional programming languages, but a number of them are significantly different. This grammar production shows the different kinds of Dafny statements. They are described in subsequent sections.

21.0. Labeled Statement

```
LabeledStmt_ = "label" LabelName ":" Stmt
```

A labeled statement is just the keyword `label` followed by and identifier which is the label followed by a colon and a statement. The label may be referenced in a break statement to transfer control to the location after that statement.

21.1. Break Statement

```
BreakStmt_ = "break" ( LabelName | { "break" } ) ";"
```

A break statement breaks out of one or more loops (if the statement consists solely of one or more `break` keywords), or else transfers control to just past the statement bearing the referenced label, if a label was used.

21.2. Block Statement

```
BlockStmt = "{" { Stmt } "}"
```

A block statement is just a sequence of statements enclosed by curly braces.

21.3. Return Statement

```
ReturnStmt = "return" [ Rhs { "," Rhs } ] ";"
```

A return statement can only be used in a method. It is used to terminate the execution of the method. To return a value from a method, the value is assigned to one of the named return values sometime before a return statement. In fact, the return values act very much like local variables, and can be assigned to more than once. Return statements are used when one wants to return before reaching the end of the body block of the method. Return statements can be just the return keyword (where the current value of the out parameters are used), or they can take a list of values to return. If a list is given the number of values given must be the same as the number of named return values.

21.4. Yield Statement

```
YieldStmt = "yield" [ Rhs { "," Rhs } ] ";"
```

A yield statement can only be used in an iterator. See section Iterator types for more details about iterators.

The body of an iterator is a *co-routine*. It is used to yield control to its caller, signaling that a new set of values for the iterator's yield parameters (if any) are available. Values are assigned to the yield parameters at or before a yield statement. In fact, the yield parameters act very much like local variables, and can be assigned to more than once. Yield statements are used when one wants to return new yield parameter values to the caller. Yield statements can be just the **yield** keyword (where the current value of the yield parameters are used), or they can

take a list of values to yield. If a list is given the number of values given must be the same as the number of named return yield parameters.

21.5. Update Statement

```
UpdateStmt = Lhs { "," Lhs }
    ( ":=" Rhs { "," Rhs }
    | ":|" [ "assume" ] Expression(allowLemma: false, allowLambda: true)
    )
    ";""
```

The update statement has two forms. The first more normal form allows for parallel assignment of right-hand-side values to the left-hand side. For example `x,y := y,x` to swap the values of `x` and `y`. Of course the common case will have only one rhs and one lhs.

The form that uses "`:|`" assigns some values to the left-hand-side variables such that the boolean expression on the right hand side is satisfied. This can be used to make a choice as in the following example where we choose an element in a set.

```
function PickOne<T>(s: set<T>): T
  requires s != {}
{
  var x :| x in s; x
}
```

Dafny will report an error if it cannot prove that values exist which satisfy the condition.

In addition, though the choice is arbitrary, given identical circumstances the choice will be made consistently.

In the actual grammar two additional forms are recognized for purposes of error detection. The form:

```
Lhs { Attribute} ;
```

is assumed to be a mal-formed call.

The form

```
Lhs ":"
```

is diagnosed as a label in which the user forgot the **label** keyword.

21.6. Variable Declaration Statement

```
VarDeclStatement = [ "ghost" ] "var" { Attribute }
  (
    LocalIdentTypeOptional { "," { Attribute } LocalIdentTypeOptional }
    [ ":=" Rhs { "," Rhs }
    | { Attribute } ":|" [ "assume" ] Expression(allowLemma: false, allowLambda: true)
    ]
  |
    "(" CasePattern { "," CasePattern } ")"
    ":=" Expression(allowLemma: false, allowLambda: true)
  )
  ";"
```

A `VarDeclStatement` is used to declare one or more local variables in a method or function. The type of each local variable must be given unless the variable is given an initial value in which case the type will be inferred. If initial values are given, the number of values must match the number of variables declared.

Note that the type of each variable must be given individually. The following code

```
var x, y : int;
```

does not declare both `x` and `y` to be of type `int`. Rather it will give an error explaining that the type of `x` is underspecified.

The lefthand side can also contain a tuple of patterns which will be matched against the right-hand-side. For example:

```
function returnsTuple() : (int, int)
{
    (5, 10)
}

function usesTuple() : int
{
    var (x, y) := returnsTuple();
    x + y
}
```

21.7. Guards

```
Guard = ( "*" | "(" "*" ")" | Expression(allowLemma: true, allowLambda: true) )
```

Guards are used in `if` and `while` statements as boolean expressions. Guards take two forms.

The first and most common form is just a boolean expression.

The second form is either `*` or `(*)`. These have the same meaning. An unspecified boolean value is returned. The value returned may be different each time it is executed.

21.8. Binding Guards

```
BindingGuard(allowLambda) =
  IdentTypeOptional { "," IdentTypeOptional } { Attribute }
  ":|" Expression(allowLemma: true, allowLambda)
```

A `BindingGuard` is used as a condition in an `IfStmt`. It binds the identifiers declared in the `IdentTypeOptional`s. If there exists one or more assignments of values to the bound identifiers for which `Expression` is true, then the `BindingGuard` returns true and the identifiers are bound to values that make the `Expression` true.

The identifiers bound by `BindingGuard` are ghost variables and cannot be assigned to non-ghost variables. They are only used in specification contexts.

Here is an example:

```
predicate P(n: int)
{
  n % 2 == 0
}

method M1() returns (ghost y: int)
    requires ∃x :: P(x)
    ensures P(y)
{
  if x : int :| P(x) {
      y := x;
  }
}
```

21.9. If Statement

```
IfStmt = "if"
  ( IfAlternativeBlock
  |
    ( BindingGuard(allowLambda: true)
    | Guard
```

```
    | "…"
    )
    BlockStmt [ "else" ( IfStmt | BlockStmt ) ]
  )
```

In the simplest form an `if` statement uses a guard that is a boolean expression. It then has the same form as in C# and other common programming languages. For example

```
if x < 0 {
  x := -x;
}
```

If the guard is an asterisk then a non-deterministic choice is made:

```
if * {
  print "True";
} else {
  print "False";
}
```

```
IfAlternativeBlock =
   "{" { "case"
      (
        BindingGuard(allowLambda:false)
      | Expression(allowLemma: true, allowLambda: false)
      ) "=>" { Stmt } } "}" .
```

The `if` statement using the `IfAlternativeBlock` form is similar to the `if ... fi` construct used in the book "A Discipline of Programming" by Edsger W. Dijkstra. It is used for a multi-branch `if`.

For example:

```
if {
  case x <= y => max := y;
  case y <= x => max := y;
}
```

In this form the expressions following the `case` keyword are called *guards*. The statement is evaluated by evaluating the guards in an undetermined order until one is found that is `true` or else all have evaluated to `false`. If none of them evaluate to `true` then the `if` statement does nothing. Otherwise the statements to the right of `=>` for the guard that evaluated to `true` are executed.

21.10. While Statement

```
WhileStmt = "while"
  ( LoopSpecWhile WhileAlternativeBlock
  | ( Guard | "…" ) LoopSpec
      ( BlockStmt
      | "…"
      | /* go body-less */
      )
  )

WhileAlternativeBlock =
   "{" { "case" Expression(allowLemma: true, allowLambda: false) "=>" { Stmt } } "}" .
```

See section 4.5 for a description of LoopSpec.

The `while` statement is Dafny's only loop statement. It has two general forms.

The first form is similar to a while loop in a C-like language. For example:

```
var i := 0;
while i < 5 {
  i := i + 1;
}
```

In this form the condition following the `while` is one of these:

- A boolean expression. If true it means execute one more iteration of the loop. If false then terminate the loop.
- An asterisk (`*`), meaning non-deterministically yield either `true` or `false` as the value of the condition
- An ellipsis (`...`), which makes the while statement a *skeleton* `while` statement. TODO: What does that mean?

The *body* of the loop is usually a block statement, but it can also be a *skeleton*, denoted by ellipsis, or missing altogether. TODO: Wouldn't a missing body cause problems? Isn't it clearer to have a block statement with no statements inside?

The second form uses the `WhileAlternativeBlock`. It is similar to the `do ... od` construct used in the book "A Discipline of Programming" by Edsger W. Dijkstra. For example:

```
while
  decreases if 0 <= r then r else -r;
{
  case r < 0 =>
```

```
      r := r + 1;
    case 0 < r =>
      r := r - 1;
  }
```

For this form the guards are evaluated in some undetermined order until one is found that is true, in which case the corresponding statements are executed. If none of the guards evaluates to true then the loop execution is terminated.

21.10.0. Loop Specifications

For some simple loops such as those mentioned previously Dafny can figure out what the loop is doing without more help. However in general the user must provide more information in order to help Dafny prove the effect of the loop. This information is provided by a `LoopSpec`. A `LoopSpec` provides information about invariants, termination, and what the loop modifies. `LoopSpecs` are explained in section 4.5. However the following sections present additional rationale and tutorial on loop specifications.

21.10.0.0. Loop Invariants `While` loops present a problem for Dafny. There is no way for Dafny to know in advance how many times the code will go around the loop. But Dafny needs to consider all paths through a program, which could include going around the loop any number of times. To make it possible for Dafny to work with loops, you need to provide loop invariants, another kind of annotation.

A loop invariant is an expression that holds upon entering a loop, and after every execution of the loop body. It captures something that is invariant, i.e. does not change, about every step of the loop. Now, obviously we are going to want to change variables, etc. each time around the loop, or we wouldn't need the loop. Like pre- and postconditions, an invariant is a property that is preserved for each execution of the loop, expressed using the same boolean expressions we have seen. For example,

```
var i := 0;
while i < n
  invariant 0 <= i
{
  i := i + 1;
}
```

When you specify an invariant, Dafny proves two things: the invariant holds upon entering the loop, and it is preserved by the loop. By preserved, we mean that assuming that the invariant holds at the beginning of the loop, we must show that executing the loop body once makes the

invariant hold again. Dafny can only know upon analyzing the loop body what the invariants say, in addition to the loop guard (the loop condition). Just as Dafny will not discover properties of a method on its own, it will not know any but the most basic properties of a loop are preserved unless it is told via an invariant.

21.10.0.1. Loop Termination Dafny proves that code terminates, i.e. does not loop forever, by using `decreases` annotations. For many things, Dafny is able to guess the right annotations, but sometimes it needs to be made explicit. In fact, for all of the code we have seen so far, Dafny has been able to do this proof on its own, which is why we haven't seen the decreases annotation explicitly yet. There are two places Dafny proves termination: loops and recursion. Both of these situations require either an explicit annotation or a correct guess by Dafny.

A `decreases` annotation, as its name suggests, gives Dafny an expression that decreases with every loop iteration or recursive call. There are two conditions that Dafny needs to verify when using a decreases expression:

- that the expression actually gets smaller, and
- that it is bounded.

Many times, an integral value (natural or plain integer) is the quantity that decreases, but other things that can be used as well. In the case of integers, the bound is assumed to be zero. For example, the following is a proper use of decreases on a loop (with its own keyword, of course):

```
while 0 < i
  invariant 0 <= i
  decreases i
{
  i := i - 1;
}
```

Here Dafny has all the ingredients it needs to prove termination. The variable i gets smaller each loop iteration, and is bounded below by zero. This is fine, except the loop is backwards from most loops, which tend to count up instead of down. In this case, what decreases is not the counter itself, but rather the distance between the counter and the upper bound. A simple trick for dealing with this situation is given below:

```
while i < n
  invariant 0 <= i <= n
  decreases n - i
{
  i := i + 1;
}
```

This is actually Dafny's guess for this situation, as it sees `i < n` and assumes that `n - i` is the quantity that decreases. The upper bound of the loop invariant implies that `0 <= n â€" i`, and gives Dafny a lower bound on the quantity. This also works when the bound `n` is not constant, such as in the binary search algorithm, where two quantities approach each other, and neither is fixed.

If the **decreases** clause of a loop specified "*", then no termination check will be performed. Use of this feature is sound only with respect to partial correctness.

21.10.0.2. Loop Framing In some cases we also must specify what memory locations the loop body is allowed to modify. This is done using a `modifies` clause. See the discussion of framing in methods for a fuller discussion.

21.11. Match Statement

```
MatchStmt = "match" Expression(allowLemma: true, allowLambda: true)
  ( "{" { CaseStatement  } "}"
  | { CaseStatement }
  )

CaseStatement = CaseBinding_ "=>" { Stmt }
```

The `match` statement is used to do case analysis on a value of inductive or co-inductive type. The form with no leading `Ident` is for matching tuples. The expression after the `match` keyword is the (co)inductive value being matched. The expression is evaluated and then matched against each of the case clauses.

There must be a case clause for each constructor of the data type. The identifier after the `case` keyword in a case clause, if present, must be the name of one of the data type's constructors. If the constructor takes parameters then a parenthesis-enclosed list of identifiers (with optional type) must follow the constructor. There must be as many identifiers as the constructor has parameters. If the optional type is given it must be the same as the type of the corresponding parameter of the constructor. If no type is given then the type of the corresponding parameter is the type assigned to the identifier.

When an inductive value that was created using constructor expression `C1(v1, v2)` is matched against a case clause `C2(x1, x2)`, there is a match provided that `C1` and `C2` are the same constructor. In that case `x1` is bound to value `v1` and `x2` is bound to `v2`. The identifiers in the case pattern are not mutable. Here is an example of the use of a `match` statement.

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

```
// Return the sum of the data in a tree.
method Sum(x: Tree) returns (r: int)
{
  match x {
    case Empty => r := -1;
    case Node(t1 : Tree, d, t2) => {
      var v1 := Sum(t1);
      var v2 := Sum(t2);
      r := v1 + d + v2;
    }
 }
}
```

Note that the `Sum` method is recursive yet has no `decreases` annotation. In this case it is not needed because Dafny is able to deduce that `t1` and `t2` are *smaller* (structurally) than `x`. If `Tree` had been coinductive this would not have been possible since `x` might have been infinite.

21.12. Assert Statement

```
AssertStmt =
    "assert" { Attribute }
    ( Expression(allowLemma: false, allowLambda: true)
    | "…"
    ) ";"
```

`Assert` statements are used to express logical proposition that are expected to be true. Dafny will attempt to prove that the assertion is true and give an error if not. Once it has proved the assertion it can then use its truth to aid in following deductions. Thus if Dafny is having a difficult time verifying a method the user may help by inserting assertions that Dafny can prove, and whose true may aid in the larger verification effort.

If the proposition is ... then (TODO: what does this mean?).

21.13. Assume Statement

```
AssumeStmt =
    "assume" { Attribute }
    ( Expression(allowLemma: false, allowLambda: true)
    | "…"
    ) ";"
```

The `Assume` statement lets the user specify a logical proposition that Dafny may assume to be true without proof. If in fact the proposition is not true this may lead to invalid conclusions.

An `Assume` statement would ordinarily be used as part of a larger verification effort where verification of some other part of the program required the proposition. By using the `Assume` statement the other verification can proceed. Then when that is completed the user would come back and replace the `assume` with `assert`.

If the proposition is ... then (TODO: what does this mean?).

21.14. Print Statement

```
PrintStmt =
    "print" Expression(allowLemma: false, allowLambda: true)
    { "," Expression(allowLemma: false, allowLambda: true) } ";"
```

The `print` statement is used to print the values of a comma-separated list of expressions to the console. The generated C# code uses the `System.Object.ToString()` method to convert the values to printable strings. The expressions may of course include strings that are used for captions. There is no implicit new line added, so to get a new line you should include "\n" as part of one of the expressions. Dafny automatically creates overrides for the ToString() method for Dafny data types. For example,

```
datatype Tree = Empty | Node(left: Tree, data: int, right: Tree)
method Main()
{
  var x : Tree := Node(Node(Empty, 1, Empty), 2, Empty);
  print "x=", x, "\n";
}
```

produces this output:

```
x=Tree.Node(Tree.Node(Tree.Empty, 1, Tree.Empty), 2, Tree.Empty)
```

21.15. Forall Statement

```
ForallStmt = "forall"
  ( "(" [ QuantifierDomain ] ")"
  | [ QuantifierDomain ]
  )
  { [ "free" ] ForAllEnsuresClause_ }
  [ BlockStmt ]
```

The `forall` statement executes ensures expressions or a body in parallel for all quantified values in the specified range. The use of the `parallel` keyword is deprecated. Use `forall` instead. There are several variant uses of the `forall` statement. And there are a number of restrictions.

In particular a `forall` statement can be classified as one of the following:

- *Assign* - the `forall` statement is used for simultaneous assignment. The target must be an array element or an object field.
- *Call* - The body consists of a single call to a method without side effects
- *Proof* - The `forall` has `ensure` expressions which are effectively quantified or proved by the body (if present).

An *assign* `forall` statement is to perform simultaneous assignment. The following is an excerpt of an example given by Leino in Developing Verified Programs with Dafny. When the buffer holding the queue needs to be resized, the `forall` statement is used to simultaneously copy the old contents into the new buffer.

```
class {:autocontracts} SimpleQueue<Data>
{
  ghost var Contents: seq<Data>;
  var a: array<Data>; // Buffer holding contents of queue.
  var m: int          // Index head of queue.
  var n: int;         // Index just past end of queue
  ...
  method Enqueue(d: Data)
    ensures Contents == old(Contents) + [d]
  {
    if n == a.Length {
      var b := a;
      if m == 0 { b := new Data[2 * a.Length]; }
      forall (i | 0 <= i < n - m) {
          b[i] := a[m + i];
      }
      a, m, n := b, 0, n - m;
    }
    a[n], n, Contents := d, n + 1, Contents + [d];
  }
}
```

Here is an example of a *call* `forall` statement and the callee. This is contained in the CloudMake-ConsistentBuilds.dfy test in the Dafny repository.

```
forall (cmd', deps', e' | Hash(Loc(cmd', deps', e')) == Hash(Loc(cmd, deps, e))) {
  HashProperty(cmd', deps', e', cmd, deps, e);
}

ghost method HashProperty(cmd: Expression, deps: Expression, ext: string,
    cmd': Expression, deps': Expression, ext': string)
  requires Hash(Loc(cmd, deps, ext)) == Hash(Loc(cmd', deps', ext'))
  ensures cmd == cmd' && deps == deps' && ext == ext'
```

From the same file here is an example of a *proof* `forall` statement.

```
forall (p | p in DomSt(stCombinedC.st) && p in DomSt(stExecC.st))
  ensures GetSt(p, stCombinedC.st) == GetSt(p, stExecC.st)
{
  assert DomSt(stCombinedC.st) <= DomSt(stExecC.st);
  assert stCombinedC.st == Restrict(DomSt(stCombinedC.st), stExecC.st);
}
```

More generally the statement

```
forall x | P(x) { Lemma(x); }
```

is used to invoke `Lemma(x)` on all `x` for which `P(x)` holds. If `Lemma(x)` ensures `Q(x)`, then the forall statement establishes

```
forall x :: P(x) ==> Q(x).
```

The `forall` statement is also used extensively in the desugared forms of co-predicates and co-lemmas. See section 18.2.

TODO: List all of the restrictions on the `forall` statement.

21.16. Modify Statement

```
ModifyStmt =
  "modify" { Attribute }
  ( FrameExpression(allowLemma: false, allowLambda: true)
    { "," FrameExpression(allowLemma: false, allowLambda: true) }
  | "…"
  )
  ( BlockStmt | ";" )
```

The `modify` statement has two forms which have two different purposes.

When the `modify` statement ends with a semi-colon rather than a block statement its effect is to say that some undetermined modifications have been made to any or all of the memory locations specified by the frame expressions. In the following example, a value is assigned to field `x` followed by a `modify` statement that may modify any field in the object. After that we can no longer prove that the field `x` still has the value we assigned to it.

```
class MyClass {
  var x: int;
  method N()
    modifies this
  {
    x := 18;
    modify this;
    assert x == 18;  // error: cannot conclude this here
  }
}
```

When the `modify` statement is followed by a block statement we are instead specifying what can be modified in that block statement. Namely, only memory locations specified by the frame expressions of the block `modify` statement may be modified. Consider the following example.

```
class ModifyBody {
  var x: int;
  var y: int;
  method M0()
    modifies this
  {
    modify {} {
      x := 3;  // error: violates modifies clause of the modify statement
    }
  }
  method M1()
    modifies this
  {
    modify {} {
      var o := new ModifyBody;
      o.x := 3;  // fine
    }
  }
  method M2()
```

```
    modifies this
  {
    modify this {
      x := 3;
    }
  }
}
```

The first `modify` statement in the example has an empty frame expression so it cannot modify any memory locations. So an error is reported when it tries to modify field `x`.

The second `modify` statement also has an empty frame expression. But it allocates a new object and modifies it. Thus we see that the frame expressions on a block `modify` statement only limits what may be modified of existing memory. It does not limit what may be modified in new memory that is allocated.

The third `modify` statement has a frame expression that allows it to modify any of the fields of the current object, so the modification of field `x` is allowed.

21.17. Calc Statement

```
CalcStmt = "calc" { Attribute } [ CalcOp ] "{" CalcBody "}"
CalcBody = { CalcLine [ CalcOp ] Hints }
CalcLine = Expression(allowLemma: false, allowLambda: true) ";"
Hints = { ( BlockStmt | CalcStmt ) }
CalcOp =
  ( "==" [ "#" "[" Expression(allowLemma: true, allowLambda: true) "]" ]
  | "<" | ">"
  | "!=" | "<=" | ">="
  | "<==>" | "==>" | "<=="
  )
```

The `calc` statement supports *calculational proofs* using a language feature called *program-oriented calculations* (poC). This feature was introduced and explained in the Verified Calculations paper by Leino and Polikarpova[22]. Please see that paper for a more complete explanation of the `calc` statement. We here mention only the highlights.

Calculational proofs are proofs by stepwise formula manipulation as is taught in elementary algebra. The typical example is to prove an equality by starting with a left-hand-side, and through a series of transformations morph it into the desired right-hand-side.

Non-syntactic rules further restrict hints to only ghost and side-effect free statements, as well as impose a constraint that only chain-compatible operators can be used together in a

calculation. The notion of chain-compatibility is quite intuitive for the operators supported by poC; for example, it is clear that "<" and ">" cannot be used within the same calculation, as there would be no relation to conclude between the first and the last line. See the paper for a more formal treatment of chain-compatibility.

Note that we allow a single occurrence of the intransitive operator "!=" to appear in a chain of equalities (that is, "!=" is chain-compatible with equality but not with any other operator, including itself). Calculations with fewer than two lines are allowed, but have no effect. If a step operator is omitted, it defaults to the calculation-wide operator, defined after the `calc` keyword. If that operator if omitted, it defaults to equality.

Here is an example using `calc` statements to prove an elementary algebraic identity. As it turns out Dafny is able to prove this without the `calc` statements, but it helps to illustrate the syntax.

```
lemma docalc(x : int, y: int)
  ensures (x + y) * (x + y) == x * x + 2 * x * y + y * y
{
  calc {
    (x + y) * (x + y); ==
      // distributive law: (a + b) * c == a * c + b * c
      x * (x + y) + y * (x + y); ==
      // distributive law: a * (b + c) == a * b + a * c
      x * x + x * y + y * x + y * y; ==
      calc {
        y * x; ==
        x * y;
    }
      x * x + x * y + x * y + y * y; ==
      calc {
        x * y + x * y; ==
        // a = 1 * a
        1 * x * y + 1 * x * y; ==
        // Distributive law
        (1 + 1) * x * y; ==
        2 * x * y;
      }
      x * x + 2 * x * y + y * y;
  }
}
```

Here we started with `(x + y) * (x + y)` as the left-hand-side expressions and gradually transformed it using distributive, commutative and other laws into the desired right-hand-side.

The justification for the steps are given as comments, or as nested `calc` statements that prove equality of some sub-parts of the expression.

The == to the right of the semicolons show the relation between that expression and the next. Because of the transitivity of equality we can then conclude that the original left-hand-side is equal to the final expression.

We can avoid having to supply the relational operator between every pair of expressions by giving a default operator between the `calc` keyword and the opening brace as shown in this abbreviated version of the above calc statement:

```
calc == {
  (x + y) * (x + y);
  x * (x + y) + y * (x + y);
  x * x + x * y + y * x + y * y;
  x * x + x * y + x * y + y * y;
  x * x + 2 * x * y + y * y;
}
```

And since equality is the default operator we could have omitted it after the `calc` keyword. The purpose of the block statements or the `calc` statements between the expressions is to provide hints to aid Dafny in proving that step. As shown in the example, comments can also be used to aid the human reader in cases where Dafny can prove the step automatically.

21.18. Skeleton Statement

```
SkeletonStmt =
  "…"
  ["where" Ident {"," Ident } ":="
    Expression(allowLemma: false, allowLambda: true)
    {"," Expression(allowLemma: false, allowLambda: true) }
  ] ";"
```

22. Expressions

The grammar of Dafny expressions follows a hierarchy that reflects the precedence of Dafny operators. The following table shows the Dafny operators and their precedence in order of increasing binding power.

operator	description
`;`	In LemmaCall;Expression
`<==>`, ⇔	equivalence (if and only if)
`==>`, ⇒ `<==`, ⇐	implication (implies) reverse implication (follows from)
`&&`, ∧ `||`, ∨	conjunction (and) disjunction (or)
`!`, ¬	negation (not)

`==`
`==#[k]`
`!=`
`!=#[k]`
`<`
`<=`
`>=`
`>`
`in`
`!in`
`!!`
	equality
	prefix equality (co-inductive)
	disequality
	prefix disequality (co-inductive)
	less than
	at most
	at least
	greater than
	collection membership
	collection non-membership
	disjointness

`+` `-`	addition (plus) subtraction (minus)

`*`
`/`
`%`
	multiplication (times)
	division (divided by)
	modulus (mod)

`-` `!`, ¬ Primary Expressions	arithmetic negation (unary minus) logical negation

We are calling the `UnaryExpression`s that are neither arithmetic nor logical negation the *primary expressions*. They are the most tightly bound.

In the grammar entries below we explain the meaning when the operator for that precedence level is present. If the operator is not present then we just descend to the next precedence level.

22.0. Top-level expressions

```
Expression(allowLemma, allowLambda) =
    EquivExpression(allowLemma, allowLambda)
    [ ";" Expression(allowLemma, allowLambda) ]
```

The "allowLemma" argument says whether or not the expression to be parsed is allowed to have the form S;E where S is a call to a lemma. "allowLemma" should be passed in as "false" whenever the expression to be parsed sits in a context that itself is terminated by a semi-colon.

The "allowLambda" says whether or not the expression to be parsed is allowed to be a lambda expression. More precisely, an identifier or parenthesized-enclosed comma-delimited list of identifiers is allowed to continue as a lambda expression (that is, continue with a "reads", "requires", or "=>") only if "allowLambda" is true. This affects function/method/iterator specifications, if/while statements with guarded alternatives, and expressions in the specification of a lambda expression itself.

Sometimes an expression will fail unless some relevant fact is known. In the following example the `F_Fails` function fails to verify because the `Fact(n)` divisor may be zero. But preceding the expression by a lemma that ensures that the denominator is not zero allows function `F_Succeeds` to succeed.

```
function Fact(n: nat): nat
{
  if n == 0 then 1 else n * Fact(n-1)
}

lemma L(n: nat)
  ensures 1 <= Fact(n)
{
}

function F_Fails(n: nat): int
{
  50 / Fact(n)  // error: possible division by zero
}

function F_Succeeds(n: nat): int
{
  L(n);
  50 / Fact(n)
}
```

22.1. Equivalence Expressions

```
EquivExpression(allowLemma, allowLambda) =
```

```
  ImpliesExpliesExpression(allowLemma, allowLambda)
  { "<==>" ImpliesExpliesExpression(allowLemma, allowLambda) }
```

An EquivExpression that contains one or more "<==>"s is a boolean expression and all the contained ImpliesExpliesExpression must also be boolean expressions. In that case each "<==>" operator tests for logical equality which is the same as ordinary equality.

See section 6.0.0 for an explanation of the <==> operator as compared with the == operator.

22.2. Implies or Explies Expressions

```
ImpliesExpliesExpression(allowLemma, allowLambda) =
  LogicalExpression(allowLemma, allowLambda)
  [ (  "==>" ImpliesExpression(allowLemma, allowLambda)
    | "<==" LogicalExpression(allowLemma, allowLambda)
           { "<==" LogicalExpression(allowLemma, allowLambda) }
    )
  ]

ImpliesExpression(allowLemma, allowLambda) =
  LogicalExpression(allowLemma, allowLambda)
  [  "==>" ImpliesExpression(allowLemma, allowLambda) ]
```

See section 6.0.2 for an explanation of the ==> and <== operators.

22.3. Logical Expressions

```
LogicalExpression(allowLemma, allowLambda) =
  RelationalExpression(allowLemma, allowLambda)
  [ ( "&&" RelationalExpression(allowLemma, allowLambda)
         { "&&" RelationalExpression(allowLemma, allowLambda) }
    | "||" RelationalExpression(allowLemma, allowLambda)
         { "||" RelationalExpression(allowLemma, allowLambda) }
    )
  ]
```

See section 6.0.1 for an explanation of the && (or $\wedge$) and || (or $\vee$) operators.

22.4. Relational Expressions

```
RelationalExpression(allowLemma, allowLambda) =
  Term(allowLemma, allowLambda)
  [ RelOp Term(allowLemma, allowLambda)
         { RelOp Term(allowLemma, allowLambda) } ]

RelOp =
  ( "==" [ "#" "[" Expression(allowLemma: true, allowLambda: true) "]" ]
  | "<" | ">" | "<=" | ">="
  | "!=" [ "#" "[" Expression(allowLemma: true, allowLambda: true) "]" ]
  | "in"
  | "!in"
  | "!!"
  )
```

The relation expressions that have a `RelOp` compare two or more terms. As explained in section 6, `==`, `!=`, `<`, `>`, `<=`, and `>=` and their corresponding Unicode equivalents are *chaining*.

The `in` and `!in` operators apply to collection types as explained in section 9 and represent membership or non-membership respectively.

The `!!` represents disjointness for sets and multisets as explained in sections 9.0 and 9.1.

Note that `x ==#[k] y` is the prefix equality operator that compares co-inductive values for equality to a nesting level of k, as explained in section 18.2.3.0.

22.5. Terms

```
Term(allowLemma, allowLambda) =
  Factor(allowLemma, allowLambda)
  { AddOp Factor(allowLemma, allowLambda) }
AddOp = ( "+" | "-" )
```

`Terms` combine `Factors` by adding or subtracting. Addition has these meanings for different types:

- Arithmetic addition for numeric types (section 6.1).
- Union for sets and multisets (sections 9.0 and 9.1)
- Concatenation for sequences (section 9.2)

Subtraction is arithmetic subtraction for numeric types, and set or multiset difference for sets and multisets.

22.6. Factors

```
Factor(allowLemma, allowLambda) =
  UnaryExpression(allowLemma, allowLambda)
  { MulOp UnaryExpression(allowLemma, allowLambda) }
MulOp = ( "*" | "/" | "%" )
```

A `Factor` combines `UnaryExpression`s using multiplication, division, or modulus. For numeric types these are explained in section 6.1.

Only `*` has a non-numeric application. It represents set or multiset intersection as explained in sections 9.0 and 9.1.

22.7. Unary Expressions

```
UnaryExpression(allowLemma, allowLambda) =
  ( "-" UnaryExpression(allowLemma, allowLambda)
  | "!" UnaryExpression(allowLemma, allowLambda)
  | PrimaryExpression_(allowLemma, allowLambda)
  )
```

A `UnaryExpression` applies either numeric (section 6.1) or logical (section 6.0) negation to its operand.

22.8. Primary Expressions

```
PrimaryExpression_(allowLemma, allowLambda) =
  ( MapDisplayExpr { Suffix }
  | LambdaExpression(allowLemma)
  | EndlessExpression(allowLemma, allowLambda)
  | NameSegment { Suffix }
  | SeqDisplayExpr { Suffix }
  | SetDisplayExpr { Suffix }
  | MultiSetExpr { Suffix }
  | ConstAtomExpression { Suffix }
  )
```

After descending through all the binary and unary operators we arrive at the primary expressions which are explained in subsequent sections. As can be seen, a number of these can be followed by 0 or more Suffixes to select a component of the value.

If the `allowLambda` is false then LambdaExpressions are not recognized in this context.

22.9. Lambda expressions

```
LambdaExpression(allowLemma) =
  ( WildIdent
  | "(" [ IdentTypeOptional { "," IdentTypeOptional } ] ")"
  )
  LambdaSpec_
  LambdaArrow Expression(allowLemma, allowLambda: true)

LambdaArrow = ( "=>" | "->" )
```

See section 4.3 for a description of LambdaSpec.

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)
```

22.10. Left-Hand-Side Expressions

```
Lhs =
  ( NameSegment { Suffix }
  | ConstAtomExpression Suffix { Suffix }
  )
```

A left-hand-side expression is only used on the left hand side of an `UpdateStmt`.

TODO: Try to give examples showing how these kinds of left-hand-sides are possible.

22.11. Right-Hand-Side Expressions

```
Rhs =
  ( ArrayAllocation_
  | ObjectAllocation_
  | Expression(allowLemma: false, allowLambda: true)
  | HavocRhs_
  )
  { Attribute }
```

An `Rhs` is either array allocation, an object allocation, an expression, or a havoc right-hand-side, optionally followed by one or more `Attribute`s.

Right-hand-side expressions appear in the following constructs: `ReturnStmt`, `YieldStmt`, `UpdateStmt`, or `VarDeclStatement`. These are the only contexts in which arrays or objects may be allocated, or in which havoc may be produced.

22.12. Array Allocation

```
ArrayAllocation_ = "new" Type "[" Expressions "]"
```

This allocates a new single or multi-dimensional array as explained in section 14.

22.13. Object Allocation

```
ObjectAllocation_ = "new" Type [ "(" [ Expressions ] ")" ]
```

This allocated a new object of a class type as explained in section 12.

22.14. Havoc Right-Hand-Side

```
HavocRhs_ = "*"
```

A havoc right-hand-side produces an arbitrary value of its associated type. To get a more constrained arbitrary value the "assign-such-that" operator (`:|`) can be used. See section 21.5.

22.15. Constant Or Atomic Expressions

```
ConstAtomExpression =
  ( LiteralExpression_
  | FreshExpression_
  | OldExpression_
  | CardinalityExpression_
  | NumericConversionExpression_
  | ParensExpression
  )
```

A `ConstAtomExpression` represent either a constant of some type, or an atomic expression. A `ConstAtomExpression` is never an l-value. Also, a `ConstAtomExpression` is never followed by an open parenthesis (but could very well have a suffix that starts with a period or a square bracket). (The "Also…" part may change if expressions in Dafny could yield functions.)

22.16. Literal Expressions

```
LiteralExpression_ =
 ( "false" | "true" | "null" | Nat | Dec |
   charToken | stringToken | "this")
```

A literal expression is a boolean literal, a null object reference, an unsigned integer or real literal, a character or string literal, or "this" which denote the current object in the context of an instance method or function.

22.17. Fresh Expressions

```
FreshExpression_ = "fresh" "(" Expression(allowLemma: true, allowLambda: true) ")"
```

`fresh(e)` returns a boolean value that is true if the objects referenced in expression `e` were all freshly allocated in the current method invocation. The argument of `fresh` must be either an object reference or a collection of object references.

22.18. Old Expressions

```
OldExpression_ = "old" "(" Expression(allowLemma: true, allowLambda: true) ")"
```

An *old expression* is used in postconditions. `old(e)` evaluates to the value expression `e` had on entry to the current method.

22.19. Cardinality Expressions

```
CardinalityExpression_ = "|" Expression(allowLemma: true, allowLambda: true) "|"
```

For a collection expression `c`, `|c|` is the cardinality of `c`. For a set or sequence the cardinality is the number of elements. For a multiset the cardinality is the sum of the multiplicities of the elements. For a map the cardinality is the cardinality of the domain of the map. Cardinality is not defined for infinite maps. For more see section 9.

22.20. Numeric Conversion Expressions

```
NumericConversionExpression_ =
    ( "int" | "real" ) "(" Expression(allowLemma: true, allowLambda: true) ")"
```

Numeric conversion expressions give the name of the target type followed by the expression being converted in parentheses. This production is for `int` and `real` as the target types but this also applies more generally to other numeric types, e.g. `newtypes`. See section 19.0.

22.21. Parenthesized Expression

```
ParensExpression =
  "(" [ Expressions ] ")"
```

A `ParensExpression` is a list of zero or more expressions enclosed in parentheses.

If there is exactly one expression enclosed then the value is just the value of that expression.

If there are zero or more than one the result is a `tuple` value. See section 18.1.

22.22. Sequence Display Expression

```
SeqDisplayExpr = "[" [ Expressions ] "]"
```

A sequence display expression provide a way to constructing a sequence with given values. For example

```
[1, 2, 3]
```

is a sequence with three elements in it. See section 9.2 for more information on sequences.

22.23. Set Display Expression

```
SetDisplayExpr = [ "iset" ] "{" [ Expressions ] "}"
```

A set display expression provide a way to constructing a set with given elements. If the keyword `iset` is present then a potentially infinite set is constructed.

For example

```
{1, 2, 3}
```

is a set with three elements in it. See section 9.0 for more information on sets.

22.24. Multiset Display or Cast Expression

```
MultiSetExpr =
    "multiset"
    ( "{" [ Expressions ] "}"
    | "(" Expression(allowLemma: true, allowLambda: true) ")"
    )
```

A multiset display expression provide a way to constructing a multiset with given elements and multiplicity. For example

```
multiset{1, 1, 2, 3}
```

is a multiset with three elements in it. The number 1 has a multiplicity of 2, the others a multiplicity of 1.

On the other hand, a multiset cast expression converts a set or a sequence into a multiset as shown here:

```
var s : set<int> := {1, 2, 3};
var ms : multiset<int> := multiset(s);
ms := ms + multiset{1};
var sq : seq<int> := [1, 1, 2, 3];
var ms2 : multiset<int> := multiset(sq);
assert ms == ms2;
```

See section 9.1 for more information on multisets.

22.25. Map Display Expression

```
MapDisplayExpr = ("map" | "imap" ) "[" [ MapLiteralExpressions ] "]"
MapLiteralExpressions =
    Expression(allowLemma: true, allowLambda: true)
    ":=" Expression(allowLemma: true, allowLambda: true)
    { "," Expression(allowLemma: true, allowLambda: true)
          ":=" Expression(allowLemma: true, allowLambda: true)
    }
```

A map display expression builds a finite or potentially infinite map from explicit `MapLiteralExpressions`. For example:

```
var m := map[1 := "a", 2 := "b"];
ghost var im := imap[1 := "a", 2 := "b"];
```

Note that `imap`s may only appear in ghost contexts. See section 9.3 for more details on maps and imaps.

22.26. Endless Expression

```
EndlessExpression(allowLemma, allowLambda) =
  ( IfExpression_(allowLemma, allowLambda)
  | MatchExpression(allowLemma, allowLambda)
  | QuantifierExpression(allowLemma, allowLambda)
  | SetComprehensionExpr(allowLemma, allowLambda)
  | StmtInExpr Expression(allowLemma, allowLambda)
  | LetExpr(allowLemma, allowLambda)
  | MapComprehensionExpr(allowLemma, allowLambda)
  )
```

EndlessExpression gets it name from the fact that all its alternate productions have no terminating symbol to end them, but rather they all end with an Expression at the end. The various EndlessExpression alternatives are described below.

22.27. If Expression

```
IfExpression_(allowLemma, allowLambda) =
    "if" Expression(allowLemma: true, allowLambda: true)
    "then" Expression(allowLemma: true, allowLambda: true)
    "else" Expression(allowLemma, allowLambda)
```

The IfExpression is a conditional expression. It first evaluates the expression following the if. If it evaluates to true then it evaluates the expression following the then and that is the result of the expression. If it evaluates to false then the expression following the else is evaluated and that is the result of the expression. It is important that only the selected expression is evaluated as the following example shows.

```
var k := 10 / x; // error, may divide by 0.
var m := if x != 0 then 10 / x else 1; // ok, guarded
```

22.28. Case Bindings and Patterns

```
CaseBinding_ =
  "case"
  ( Ident [ "(" CasePattern { "," CasePattern } ")" ]
  | "(" CasePattern { "," CasePattern } ")"
  )

CasePattern =
  ( Ident "(" [ CasePattern { "," CasePattern } ] ")"
  | "(" [ CasePattern { "," Casepattern } ] ")"
  | IdentTypeOptional
  )
```

Case bindings and patterns are used for (possibly nested) pattern matching on inductive or coinductive values. The CaseBinding_ construct is used in CaseStatement and CaseExpressions. Besides its use in CaseBinding_, CasePatterns are used in LetExprs and VarDeclStatements.

When matching an inductive or coinductive value in a MatchStmt or MatchExpression, there must be a CaseBinding_ for each constructor. A tuple is considered to have a single constructor.

The Ident of the CaseBinding_ must match the name of a constructor (or in the case of a tuple the Ident is absent and the second alternative is chosen). The CasePatterns inside the parenthesis are then matched against the argument that were given to the constructor when the value was constructed. The number of CasePatterns must match the number of parameters to the constructor (or the arity of the tuple).

The CasePatterns may be nested. The set of non-constructor-name identifiers contained in a CaseBinding_ must be distinct. They are bound to the corresponding values in the value being matched.

22.29. Match Expression

```
MatchExpression(allowLemma, allowLambda) =
  "match" Expression(allowLemma, allowLambda)
  ( "{" { CaseExpression(allowLemma: true, allowLambda: true) } "}"
  | { CaseExpression(allowLemma, allowLambda) }
  )

CaseExpression(allowLemma, allowLambda) =
    CaseBinding_ "=>" Expression(allowLemma, allowLambda)
```

A MatchExpression is used to conditionally evaluate and select an expression depending on the value of an algebraic type, i.e. an inductive type, or a co-inductive type.

The Expression following the match keyword is called the *selector*. There must be a CaseExpression for each constructor of the type of the selector. The Ident following the case keyword in a CaseExpression is the name of a constructor of the selector's type. It may be absent if the expression being matched is a tuple since these have no constructor name.

If the constructor has parameters then in the CaseExpression the constructor name must be followed by a parenthesized list of CasePatterns. If the constructor has no parameters then the CaseExpression must not have a following CasePattern list. All of the identifiers in the CasePatterns must be distinct. If types for the identifiers are not given then types are inferred from the types of the constructor's parameters. If types are given then they must agree with the types of the corresponding parameters.

A MatchExpression is evaluated by first evaluating the selector. Then the CaseClause is selected for the constructor that was used to construct the evaluated selector. If the constructor had parameters then the actual values used to construct the selector value are bound to the identifiers in the identifier list. The expression to the right of the => in the CaseClause is then evaluated in the environment enriched by this binding. The result of that evaluation is the result of the MatchExpression.

Note that the braces enclosing the CaseClauses may be omitted.

22.30. Quantifier Expression

```
QuantifierExpression(allowLemma, allowLambda) =
    ( "forall" | "exists" ) QuantifierDomain "::"
    Expression(allowLemma, allowLambda)

QuantifierDomain =
  IdentTypeOptional { "," IdentTypeOptional } { Attribute }
  [ "|" Expression(allowLemma: true, allowLambda: true) ]
```

A QuantifierExpression is a boolean expression that specifies that a given expression (the one following the "::") is true for all (for **forall**) or some (for **exists**) combination of values of the quantified variables, namely those in the QuantifierDomain.

Here are some examples:

```
assert forall x : nat | x <= 5 :: x * x <= 25;
(forall n :: 2 <= n ==> (∃d :: n < d && d < 2*n))
```

or using the Unicode symbols:

```
assert ∀ x : nat | x <= 5 • x * x <= 25;
(∀ n • 2 <= n ==> (∃ d • n < d && d < 2*n))
```

The quantifier identifiers are *bound* within the scope of the expressions in the QuantifierExpression.

It types are not given for the quantified identifiers then Dafny attempts to infer their types from the context of the expressions. It this is not possible the program is in error.

22.31. Set Comprehension Expressions

```
SetComprehensionExpr(allowLemma, allowLambda) =
  [ "set" | "iset" ]
  IdentTypeOptional { "," IdentTypeOptional } { Attribute }
  "|" Expression(allowLemma, allowLambda)
  [ "::" Expression(allowLemma, allowLambda) ]
```

A set comprehension expression is an expressions that yields a set (possibly infinite if iset is used) that satisfies specified conditions. There are two basic forms.

If there is only one quantified variable the optional "::" Expression need not be supplied, in which case it is as if it had been supplied and the expression consists solely of the quantified

variable. That is,

```
set x : T | P(x)
```

is equivalent to

```
set x : T | P(x) :: x
```

For the full form

```
var S := set x1:T1, x2:T2 ... | P(x1, x2, ...) :: Q(x1, x2, ...)
```

the elements of `S` will be all values resulting from evaluation of `Q(x1, x2, ...)` for all combinations of quantified variables `x1, x2, ...` such that predicate `P(x1, x2, ...)` holds. For example,

```
var S := set x:nat, y:nat | x < 2 && y < 2 :: (x, y)
```

would yield `S == {(0, 0), (0, 1), (1, 0), (1,1) }`

The types on the quantified variables are optional and if not given Dafny will attempt to infer them from the contexts in which they are used in the `P` or `Q` expressions.

If a finite set was specified ("set" keyword used), Dafny must be able to prove that the result is finite otherwise the set comprehension expression will not be accepted.

Set comprehensions involving reference types such as

```
set o: object | true
```

are allowed in ghost contexts. In particular, in ghost contexts, the check that the result is finite should allow any set comprehension where the bound variable is of a reference type. In non-ghost contexts, it is not allowed, because–even though the resulting set would be finite–it is not pleasant or practical to compute at run time.

22.32. Statements in an Expression

```
StmtInExpr = ( AssertStmt | AssumeStmt | CalcStmt )
```

A `StmtInExpr` is a kind of statement that is allowed to precede an expression in order to ensure that the expression can be evaluated without error. For example:

```
assume x != 0; 10/x
```

`Assert`, `assume` and `calc` statements can be used in this way.

22.33. Let Expression

```
LetExpr(allowLemma, allowLambda) =
    [ "ghost" ] "var" CasePattern { "," CasePattern }
    ( ":=" | { Attribute } ":|" )
    Expression(allowLemma: false, allowLambda: true)
    { "," Expression(allowLemma: false, allowLambda: true) } ";"
    Expression(allowLemma, allowLambda)
```

A `let` expression allows binding of intermediate values to identifiers for use in an expression. The start of the `let` expression is signaled by the `var` keyword. They look much like a local variable declaration except the scope of the variable only extends to the enclosed expression.

For example:

```
var sum := x + y; sum * sum
```

In the simple case the `CasePattern` is just an identifier with optional type (which if missing is inferred from the rhs).

The more complex case allows destructuring of constructor expressions. For example:

```
datatype Stuff = SCons(x: int, y: int) | Other
function GhostF(z: Stuff): int
  requires z.SCons?
{
  var SCons(u, v) := z; var sum := u + v; sum * sum
}
```

22.34. Map Comprehension Expression

```
MapComprehensionExpr(allowLemma, allowLambda) =
  ( "map" | "imap" ) IdentTypeOptional { Attribute }
  [ "|" Expression(allowLemma: true, allowLambda: true) ]
  "::" Expression(allowLemma, allowLambda)
```

A `MapComprehensionExpr` defines a finite or infinite map value by defining a domain (using the `IdentTypeOptional` and the optional condition following the "|") and for each value in the domain, giving the mapped value using the expression following the "::".

For example:

```
function square(x : int) : int { x * x }
method test()
{
  var m := map x : int | 0 <= x <= 10 :: x * x;
```

```
  ghost var im := imap x : int :: x * x;
  ghost var im2 := imap x : int :: square(x);
}
```

Dafny maps must be finite, so the domain must be constrained to be finite. But imaps may be infinite as the example shows. The last example shows creation of an infinite map that gives the same results as a function.

22.35. Name Segment

```
NameSegment = Ident [ GenericInstantiation | HashCall ]
```

A NameSegment names a Dafny entity by giving its declared name optionally followed by information to make the name more complete. For the simple case it is just an identifier.

If the identifier is for a generic entity it is followed by a GenericInstantiation which provides actual types for the type parameters.

To reference a prefix predicate (see section 18.2.3) or prefix lemma (see section 18.2.4.2), the identifier must be the name of the copredicate or colemma and it must be followed by a HashCall.

22.36. Hash Call

```
HashCall = "#" [ GenericInstantiation ]
  "[" Expression(allowLemma: true, allowLambda: true) "]"
  "(" [ Expressions ] ")"
```

A HashCall is used to call the prefix for a copredicate or colemma. In the non-generic case it just insert "#[k]" before the call argument list where k is the number of recursion levels.

In the case where the colemma is generic, the generic type argument is given before. Here is an example:

```
codatatype Stream<T> = Nil | Cons(head: int, stuff: T, tail: Stream)

function append(M: Stream, N: Stream): Stream
{
  match M
  case Nil => N
  case Cons(t, s, M') => Cons(t, s, append(M', N))
}
```

```
function zeros<T>(s : T): Stream<T>
{
  Cons(0, s, zeros(s))
}

function ones<T>(s: T): Stream<T>
{
  Cons(1, s, ones(s))
}

copredicate atmost(a: Stream, b: Stream)
{
  match a
  case Nil => true
  case Cons(h,s,t) => b.Cons? && h <= b.head && atmost(t, b.tail)
}

colemma {:induction false} Theorem0<T>(s: T)
  ensures atmost(zeros(s), ones(s))
{
  // the following shows two equivalent ways to getting essentially the
  // co-inductive hypothesis
  if (*) {
    Theorem0#<T>[_k-1](s);
  } else {
    Theorem0(s);
  }
}
```

where the `HashCall` is `"Theorem0#<T>[_k-1](s);"`. See sections 18.2.3 and 18.2.4.2.

22.37. Suffix

```
Suffix =
  ( AugmentedDotSuffix_
  | DatatypeUpdateSuffix_
  | SubsequenceSuffix_
```

```
  | SlicesByLengthSuffix_
  | SequenceUpdateSuffix_
  | SelectionSuffix_
  | ArgumentListSuffix_
  )
```

The Suffix non-terminal describes ways of deriving a new value from the entity to which the suffix is appended. There are six kinds of suffixes which are described below.

22.37.0. Augmented Dot Suffix

```
AugmentedDotSuffix_ = ". " DotSuffix [ GenericInstantiation | HashCall ]
```

An augmented dot suffix consists of a simple DotSuffix optionally followed by either

- a GenericInstantiation (for the case where the item selected by the DotSuffix is generic), or
- a HashCall for the case where we want to call a prefix copredicate or colemma. The result is the result of calling the prefix copredicate or colemma.

22.37.1. Datatype Update Suffix

```
DatatypeUpdateSuffix_ =
  "." "(" MemberBindingUpdate { "," MemberBindingUpdate } ")"

MemberBindingUpdate =
  ( ident | digits ) ":=" Expression(allowLemma: true, allowLambda: true)
```

A datatype update suffix is used to produce a new datatype value that is the same as an old datatype value except that the value corresponding to a given destructor has the specified value. In a MemberBindingUpdate, the ident or digits is the name of a destructor (i.e. formal parameter name) for one of the constructors of the datatype. The expression to the right of the ":=" is the new value for that formal.

All of the destructors in a DatatypeUpdateSuffix_ must be for the same constructor, and if they do not cover all of the destructors for that constructor then the datatype value being updated must have a value derived from that same constructor.

Here is an example:

```
module NewSyntax {
datatype MyDataType = MyConstructor(myint:int, mybool:bool)
                    | MyOtherConstructor(otherbool:bool)
```

```
                    | MyNumericConstructor(42:int)

  method test(datum:MyDataType, x:int)
      returns (abc:MyDataType, def:MyDataType, ghi:MyDataType, jkl:MyDataType)
      requires datum.MyConstructor?;
      ensures abc == datum.(myint := x + 2);
      ensures def == datum.(otherbool := !datum.mybool);
      ensures ghi == datum.(myint := 2).(mybool := false);
      // Resolution error: no non_destructor in MyDataType
      //ensures jkl == datum.(non_destructor := 5);
      ensures jkl == datum.(42 := 7);
  {
      abc := MyConstructor(x + 2, datum.mybool);
      abc := datum.(myint := x + 2);
      def := MyOtherConstructor(!datum.mybool);
      ghi := MyConstructor(2, false);
      jkl := datum.(42 := 7);

      assert abc.(myint := abc.myint - 2) == datum.(myint := x);
  }
  }
```

22.37.2. Subsequence Suffix

```
SubsequenceSuffix_ =
  "[" [ Expression(allowLemma: true, allowLambda: true) ]
      ".." [ Expression(allowLemma: true, allowLambda: true) ]
  "]"
```

A subsequence suffix applied to a sequence produces a new sequence whose elements are taken from a contiguous part of the original sequence. For example, expression `s[lo..hi]` for sequence `s`, and integer-based numerics `lo` and `hi` satisfying `0 <= lo <= hi <= |s|`. See section 9.2.3 for details.

22.37.3. Slices By Length Suffix

```
SlicesByLengthSuffix_ =
  "[" Expression(allowLemma: true, allowLambda: true)
      ":" Expression(allowLemma: true, allowLambda: true)
```

```
      { ":" Expression(allowLemma: true, allowLambda: true) }
      [ ":" ]
  "]"
```

Applying a SlicesByLengthSuffix_ to a sequence produces a sequence of subsequences of the original sequence. See section 9.2.3 for details.

22.37.4. Sequence Update Suffix

```
SequenceUpdateSuffix_ =
  "[" Expression(allowLemma: true, allowLambda: true)
      ":=" Expression(allowLemma: true, allowLambda: true)
  "]"
```

For a sequence `s` and expressions `i` and `v`, the expression `s[i := v]` is the same as the sequence `s` except that at index `i` it has value `v`.

22.37.5. Selection Suffix

```
SelectionSuffix_ =
  "[" Expression(allowLemma: true, allowLambda: true)
      { "," Expression(allowLemma: true, allowLambda: true) }
  "]"
```

If a SelectionSuffix_ has only one expression in it, it is a zero-based index that may be used to select a single element of a sequence or from a single-dimensional array.

If a SelectionSuffix_ has more than one expression in it, then it is a list of indices to index into a multi-dimensional array. The rank of the array must be the same as the number of indices.

22.37.6. Argument List Suffix

```
ArgumentListSuffix_ = "(" [ Expressions ] ")"
```

An argument list suffix is a parenthesized list of expressions that are the arguments to pass to a method or function that is being called. Applying such a suffix caused the method or function to be called and the result is the result of the call.

22.38. Expression Lists

```
Expressions =
    Expression(allowLemma: true, allowLambda: true)
    { "," Expression(allowLemma: true, allowLambda: true) }
```

The `Expressions` non-terminal represents a list of one or more expressions separated by a comma.

23. Module Refinement

TODO: Write this section.

24. Attributes

```
Attribute = "{" ":" AttributeName [ Expressions ] "}"
```

Dafny allows many of its entities to be annotated with *Attributes*. The grammar shows where the attribute annotations may appear.

Here is an example of an attribute from the Dafny test suite:

```
{:MyAttribute "hello", "hi" + "there", 57}
```

In general an attribute may have any name the user chooses. It may be followed by a comma separated list of expressions. These expressions will be resolved and type-checked in the context where the attribute appears.

24.0. Dafny Attribute Implementation Details

In the Dafny implementation the `Attributes` type holds the name of the attribute, a list of `Expression` arguments and a link to the previous Attributes object for that Dafny entity. So for each Dafny entity that has attributes we have a list of them.

Dafny stores attributes on the following kinds of entities: Declaration (base class), ModuleDefinition, Statement, AssignmentRhs, LocalVariable, LetExpr, ComprehensionExpr, MaybeFreeExpression, Specification.

TODO: Dafny internals information should go into a separate document on Dafny internals.

24.1. Dafny Attributes

All entities that Dafny translates to Boogie have their attributes passed on to Boogie except for the `{:axiom}` attribute (which conflicts with Boogie usage) and the `{:trigger}` attribute which

is instead converted into a Boogie quantifier *trigger*. See Section 11 of [16].

Dafny has special processing for some attributes. For some attributes the setting is only looked for on the entity of interest. For others we start at the entity and if the attribute is not there, look up in the hierarchy (enclosing class and enclosing modules). The latter case is checked by the ContainsBoolAtAnyLevel method in the Dafny source. The attribute declaration closest to the entity overrides those further away.

For attributes with a single boolean expression argument, the attribute with no argument is interpreted as if it were true.

The attributes that are processed specially by Dafny are described in the following sections.

24.1.0. assumption

This attribute can only be placed on a local ghost bool variable of a method. Its declaration cannot have a rhs, but it is allowed to participate as the lhs of exactly one assignment of the form: `b := b && expr;`. Such a variable declaration translates in the Boogie output to a declaration followed by an `assume b` command. TODO: What is the motivation for this?

24.1.1. autoReq boolExpr

For a function declaration, if this attribute is set true at the nearest level, then its `requires` clause is strengthed sufficiently so that it may call the functions that it calls.

For following example

```
function f(x:int) : bool
  requires x > 3
{
  x > 7
}

// Should succeed thanks to auto_reqs
function {:autoReq} g(y:int, b:bool) : bool
{
  if b then f(y + 2) else f(2*y)
}
```

the `{:autoReq}` attribute causes Dafny to deduce a `requires` clause for g as if it had been declared

```
function g(y:int, b:bool) : bool
  requires if b then y + 2 > 3 else 2 * y > 3
```

```
{
  if b then f(y + 2) else f(2*y)
}
```

24.1.2. autocontracts

Dynamic frames [9, 17, 32, 33] are frame expressions that can vary dynamically during program execution. AutoContracts is an experimental feature that will fill much of the dynamic-frames boilerplate into a class.

From the user's perspective, what needs to be done is simply:

- mark the class with {:autocontracts}
- declare a function (or predicate) called Valid()

AutoContracts will then:

- Declare:

```
ghost var Repr: set(object);
```

- For function/predicate Valid(), insert:

```
reads this, Repr
```

- Into body of Valid(), insert (at the beginning of the body):

```
this in Repr && null !in Repr
```

- and also insert, for every array-valued field A declared in the class:

```
(A != null ==> A in Repr) &&
```

- and for every field F of a class type T where T has a field called Repr, also insert:

```
(F != null ==> F in Repr && F.Repr SUBSET Repr && this !in Repr)
```

- Except, if A or F is declared with {:autocontracts false}, then the implication will not be added.

- For every constructor, add:

```
modifies this
ensures Valid() && fresh(Repr - {this})
```

- At the end of the body of the constructor, add:

```
Repr := {this};
if (A != null) { Repr := Repr + {A}; }
if (F != null) { Repr := Repr + {F} + F.Repr; }
```

- For every method, add:

```
requires Valid()
modifies Repr
ensures Valid() && fresh(Repr - old(Repr))
```

- At the end of the body of the method, add:

```
if (A != null) { Repr := Repr + {A}; }
if (F != null) { Repr := Repr + {F} + F.Repr; }
```

24.1.3. axiom

The `{:axiom}` attribute may be placed on a function or method. It means that the post-condition may be assumed to be true without proof. In that case also the body of the function or method may be omitted.

The `{:axiom}` attribute is also used for generated `reveal_*` lemmas as shown in Section 24.1.12.

24.1.4. compile

The `{:compile}` attribute takes a boolean argument. It may be applied to any top-level declaration. If that argument is false then that declaration will not be compiled into .Net code.

24.1.5. decl

The `{:decl}` attribute may be placed on a method declaration. It inhibits the error message that has would be given when the method has a `ensures` clauses but no body.

TODO: There are no examples of this in the Dafny tests. What is the motivation for this?

24.1.6. fuel

The fuel attributes is used to specify how much "fuel" a function should have, i.e., how many times Z3 is permitted to unfold it's definition. The new {:fuel} annotation can be added to the function itself, it which case it will apply to all uses of that function, or it can overridden within the scope of a module, function, method, iterator, calc, forall, while, assert, or assume. The general format is:

```
{:fuel functionName,lowFuel,highFuel}
```

When applied as an annotation to the function itself, omit functionName. If highFuel is omitted, it defaults to lowFuel + 1.

The default fuel setting for recursive functions is 1,2. Setting the fuel higher, say, to 3,4, will give more unfoldings, which may make some proofs go through with less programmer assistance (e.g., with fewer assert statements), but it may also increase verification time, so use it with care. Setting the fuel to 0,0 is similar to making the definition opaque, except when used with all literal arguments.

24.1.7. heapQuantifier

The `{:heapQuantifier}` attribute may be used on a `QuantifierExpression`. When it appears in a quantifier expression it is as if a new heap-valued quantifier variable was added to the quantification. Consider this code that is one of the invariants of a while loop.

```
invariant forall u {:heapQuantifier} :: f(u) == u + r
```

The quantifier is translated into the following Boogie:

```
(forall q$heap#8: Heap, u#5: int ::
    {:heapQuantifier}
    $IsGoodHeap(q$heap#8) && ($Heap == q$heap#8 || $HeapSucc($Heap, q$heap#8))
      ==> $Unbox(Apply1(TInt, TInt, f#0, q$heap#8, $Box(u#5))): int == u#5 + r#0);
```

What this is saying is that the quantified expression, `f(u) == u + r`, which may depend on the heap, is also valid for any good heap that is either the same as the current heap, or that is derived from it by heap update operations.

TODO: I think this means that the quantified expression is actually independent of the heap. Is that true?

24.1.8. imported

If a `MethodDecl` or `FunctionDecl` has an `{:imported}` attribute, then it is allowed to have a empty body even though it has an **ensures** clause. Ordinarily a body would be required in

order to provide the proof of the **ensures** clause (but the `(:axiom)` attribute also provides this facility, so the need for `(:imported)` is not clear.) A method or function declaration may be given the `(:imported)` attribute. This suppresses the error message that would be given if a method or function with an `ensures` clause does not have a body.

TODO: When would this be used? An example would be helpful.

TODO: When is this useful or valid?

24.1.9. induction

The `{:induction}` attribute controls the application of proof by induction to two contexts. Given a list of variables on which induction might be applied, the `{:induction}` attribute selects a sublist of those variables (in the same order) to which to apply induction.

TODO: Would there be any advantage to taking the order from the attribute, rather than preserving the original order? That would seem to give the user more control.

The two contexts are:

- A method, in which case the bound variables are all the in-parameters of the method.
- A quantifier expression, in which case the bound variables are the bound variables of the quantifier expression.

The form of the `{:induction}` attribute is one of the following:

- `{:induction}` – apply induction to all bound variables
- `{:induction false}` – suppress induction, that is, don't apply it to any bound variable
- `{:induction L}` where `L` is a list consisting entirely of bound variables – apply induction to the specified bound variables
- `{:induction X}` where `X` is anything else – treat the same as {:induction}, that is, apply induction to all bound variables. For this usage conventionally `X` is `true`.

Here is an example of using it on a quantifier expression:

```
ghost method Fill_J(s: seq<int>)
  requires forall i :: 1 <= i < |s| ==> s[i-1] <= s[i]
  ensures forall i,j {:induction j} :: 0 <= i < j < |s| ==> s[i] <= s[j]
{
}
```

24.1.10. layerQuantifier

When Dafny is translating a quantified expression, if it has a `{:layerQuantifier}` attribute an additional quantifier variable is added to the quantifier bound variables. This variable as the

predefined *LayerType*. A `{:layerQuantifier}` attribute may be placed on a quantifier expression. Translation of Dafny into Boogie defines a *LayerType* which has defined zero and successor constructors.

The Dafny source has the comment that "if a function is recursive, then make the reveal lemma quantifier a layerQuantifier." And in that case it adds the attribute to the quantifier.

There is no explicit user of the `{:layerQuantifier}` attribute in the Dafny tests. So I believe this attribute is only used internally by Dafny and not externally.

TODO: Need more complete explanation of this attribute.

24.1.11. nativeType

The `{:nativeType}` attribute may only be used on a `NewtypeDecl` where the base type is an integral type. It can take one of the following forms:

- `{:nativeType}` - With no parameters it has no effect and the `NewtypeDecl` have its default behavior which is to choose a native type that can hold any value satisfying the constraints, if possible, otherwise BigInteger is used.
- `{:nativeType true}` - Also gives default `NewtypeDecl` behavior, but gives an error if base type is not integral.
- `{:nativeType false}` - Inhibits using a native type. BigInteger is used for integral types and BitRational for real types.
- `{:nativeType "typename"}` - This form has an native integral type name as a string literal. Acceptable values are: "byte", "sbyte", "ushort", "short", "uint", "int", "ulong" and "long". An error is reported if the given data type cannot hold all the values that satisfy the constraint.

24.1.12. opaque

Ordinarily the body of a function is transparent to its users but sometimes it is useful to hide it. If a function `f` is given the `{:opaque}` attribute then Dafny hides the body of the function, so that it can only be seen within its recursive clique (if any), or if the programmer specifically asks to see it via the `reveal_f()` lemma.

We create a lemma to allow the user to selectively reveal the function's body
That is, given:

```
function {:opaque} foo(x:int, y:int) : int
  requires 0 <= x < 5
  requires 0 <= y < 5
  ensures foo(x, y) < 10
{ x + y }
```

We produce:

```
lemma {:axiom} reveal_foo()
  ensures forall x:int, y:int {:trigger foo(x,y)} ::
       0 <= x < 5 && 0 <= y < 5 ==> foo(x,y) == foo_FULL(x,y)
```

where `foo_FULL` is a copy of `foo` which does not have its body hidden. In addition `foo_FULL` is given the `{:opaque_full}` and `{:auto_generated}` attributes in addition to the `{:opaque}` attribute (which it got because it is a copy of `foo`).

24.1.13. opaque full

The `{:opaque_full}` attribute is used to mark the *full* version of an opaque function. See Section 24.1.12.

24.1.14. prependAssertToken

This is used internally in Dafny as part of module refinement. It is an attribute on an assert statement. The Dafny code has the following comment:

```
// Clone the expression, but among the new assert's attributes, indicate
// that this assertion is supposed to be translated into a check.  That is,
// it is not allowed to be just assumed in the translation, despite the fact
// that the condition is inherited.
```

TODO: Decide if we want to describe this in more detail, or whether the functionality is already adequately described where refinement is described.

24.1.15. tailrecursion

This attribute is used on a method declarations. It has a boolean argument.

If specified with a false value it means the user specifically requested no tail recursion, so none is done.

If specified with a true value, or if not specified then tail recursive optimization will be attempted subject to the following conditions:

- It is an error if the method is a ghost method and tail recursion was explicitly requested.
- Only direct recursion is supported, not mutually recursive methods.
- If `{:tailrecursion true}` was specified but the code does not allow it an error message is given.

24.1.16. timeLimitMultiplier

This attribute may be placed on a method or function declaration and has an integer argument. If `{:timeLimitMultiplier X}` was specified a `{:timelimit Y}` attributed is passed on to Boogie where `Y` is `X` times either the default verification time limit for a function or method, or times the value specified by the Boogie `timelimit` command-line option.

24.1.17. trigger

Trigger attributes are used on quantifiers and comprehensions. They are translated into Boogie triggers.

24.1.18. typeQuantifier

The `{:typeQuantifier}` must be used on a quantifier if it quantifies over types.

24.2. Boogie Attributes

Use the Boogie "/attrHelp" option to get the list of attributes that Boogie recognizes and their meaning. Here is the output at the time of this writing. Dafny passes attributes that have been specified to the Boogie.

```
Boogie: The following attributes are supported by this implementation.

  ---- On top-level declarations ------------------------------------------

    {:ignore}
      Ignore the declaration (after checking for duplicate names).

    {:extern}
      If two top-level declarations introduce the same name (for example, two
      constants with the same name or two procedures with the same name), then
      Boogie usually produces an error message.  However, if at least one of
      the declarations is declared with :extern, one of the declarations is
      ignored.  If both declarations are :extern, Boogie arbitrarily chooses
      one of them to keep; otherwise, Boogie ignore the :extern declaration
      and keeps the other.

    {:checksum <string>}
      Attach a checksum to be used for verification result caching.
```

```
---- On implementations and procedures -------------------------------------

   {:inline N}
     Inline given procedure (can be also used on implementation).
     N should be a non-negative number and represents the inlining depth.
     With /inline:assume call is replaced with "assume false" once inlining depth is reached
     With /inline:assert call is replaced with "assert false" once inlining depth is reached
     With /inline:spec call is left as is once inlining depth is reached.
     With the above three options, methods with the attribute {:inline N} are not verified.
     With /inline:none the entire attribute is ignored.

   {:verify false}
     Skip verification of an implementation.

   {:vcs_max_cost N}
   {:vcs_max_splits N}
   {:vcs_max_keep_going_splits N}
     Per-implementation versions of
     /vcsMaxCost, /vcsMaxSplits and /vcsMaxKeepGoingSplits.

   {:selective_checking true}
     Turn all asserts into assumes except for the ones reachable from
     assumptions marked with the attribute {:start_checking_here}.
     Thus, "assume {:start_checking_here} something;" becomes an inverse
     of "assume false;": the first one disables all verification before
     it, and the second one disables all verification after.

   {:priority N}
     Assign a positive priority 'N' to an implementation to control the order
     in which implementations are verified (default: N = 1).

   {:id <string>}
     Assign a unique ID to an implementation to be used for verification
     result caching (default: "<impl. name>:0").

   {:timeLimit N}
     Set the time limit for a given implementation.
```

```
---- On functions ------------------------------------------------------

   {:builtin "spec"}
   {:bvbuiltin "spec"}
     Rewrite the function to built-in prover function symbol 'fn'.

   {:inline}
   {:inline true}
     Expand function according to its definition before going to the prover.

   {:never_pattern true}
     Terms starting with this function symbol will never be
     automatically selected as patterns. It does not prevent them
     from being used inside the triggers, and does not affect explicit
     trigger annotations. Internally it works by adding {:nopats ...}
     annotations to quantifiers.

   {:identity}
   {:identity true}
     If the function has 1 argument and the use of it has type X->X for
     some X, then the abstract interpreter will treat the function as an
     identity function.  Note, the abstract interpreter trusts the
     attribute--it does not try to verify that the function really is an
     identity function.

---- On variables ------------------------------------------------------

   {:existential true}
     Marks a global Boolean variable as existentially quantified. If
     used in combination with option /contractInfer Boogie will check
     whether there ∃a Boolean assignment to the existentials
     that makes all verification conditions valid.  Without option
     /contractInfer the attribute is ignored.

---- On assert statements ----------------------------------------------

   {:subsumption n}
```

```
      Overrides the /subsumption command-line setting for this assertion.

    {:split_here}
      Verifies code leading to this point and code leading from this point
      to the next split_here as separate pieces.  May help with timeouts.
      May also occasionally double-report errors.

---- The end ------------------------------------------------------------
```

However a scan of Boogie's sources shows it checks for the following attributes.

- {:$}
- {:$renamed$}
- {:InlineAssume}
- {:PossiblyUnreachable}
- {:__dominator_enabled}
- {:__enabled}
- {:a##post##}
- {:absdomain}
- {:ah}
- {:assumption}
- {:assumption_variable_initialization}
- {:atomic}
- {:aux}
- {:both}
- {:bvbuiltin}
- {:candidate}
- {:captureState}
- {:checksum}
- {:constructor}
- {:datatype}
- {:do_not_predicate}
- {:entrypoint}
- {:existential}
- {:exitAssert}
- {:expand}
- {:extern}
- {:hidden}

- {:ignore}
- {:inline}
- {:left}
- {:linear}
- {:linear_in}
- {:linear_out}
- {:msg}
- {:name}
- {:originated_from_invariant}
- {:partition}
- {:positive}
- {:post}
- {:pre}
- {:precondition_previous_snapshot}
- {:qid}
- {:right}
- {:selective_checking}
- {:si_fcall}
- {:si_unique_call}
- {:sourcefile}
- {:sourceline}
- {:split_here}
- {:stage_active}
- {:stage_complete}
- {:staged_houdini_tag}
- {:start_checking_here}
- {:subsumption}
- {:template}
- {:terminates}
- {:upper}
- {:verified_under}
- {:weight}
- {:yields}

25. Dafny User's Guide

25.0. Installing Dafny From Binaries

25.1. Building Dafny from Source

The current version of Dafny only works with Visual Studio 2012, so if you intend to run Dafny from withing Visual Studio you must install Visual Studio 2012.

Dafny performs its verification by translating the Dafny source into the Boogie intermediate verification language. So Dafny references data structures defined in the Boogie project. So the first step is to clone and build Boogie from sources. See https://github.com/boogie-org/boogie.

Follow these steps.

Let *work* be a working directory.

Clone Boogie using

```
cd work
git clone https://github.com/boogie-org/boogie.git
```

Build Boogie using the directions from the Boogie web site, which for Windows currently are:

1. Open Source\Boogie.sln in Visual Studio
2. Right click the Boogie solution in the Solution Explorer and click Enable NuGet Package Restore. You will probably get a prompt asking to confirm this. Choose Yes.
3. Click BUILD > Build Solution.

Clone Dafny using Mercurial. The Dafny directory must be a sibling of the Boogie directory in order for it to find the Boogie files it needs.

```
cd work
hg clone https://hg.codeplex.com/dafny
```

Download and install the Visual Studio 2012 SDK from

- https://www.microsoft.com/en-us/download/details.aspx?id=30668.

This is needed to build the Visual Studio Extension that runs Dafny from within Visual Studio 2012.

Build the command-line Dafny executables. 1. Open dafny\Source\Dafny.sln in Visual Studio 2. Click BUILD > Build Solution.

Build and install the Dafny Visual Studio extensions

1. Open dafny/Source/DafnyExtension.sln in Visual Studio
2. Click BUILD > Build Solution.

3. This builds DafnyLanguageService.vsix and DafnyMenu.vsix in the dafny/Binaries directory.
4. Install these by clicking on them from Windows Explorer. When prompted, only check installing into Visual Studio 2012.

25.2. Using Dafny From Visual Studio

To test your installation, you can open Dafny test files from the dafny/Test subdirectory in Visual Studio 2012. You will want to use "VIEW/Error List" to ensure that you see any errors that Dafny detects, and "VIEW/Output" to see the result of any compilation.

An example of a valid Dafny test is

```
dafny\Test\vstte2012\Tree.dfy
```

You can choose "Dafny/Compile" to compile the Dafny program to C#. Doing that for the above test produces `Tree.cs` and `Tree.dll` (since this test does not have a main program).

The following file:

```
D:\gh\dafny\Test\dafny0\Array.dfy
```

is an example of a Dafny file with verification errors. The source will show red squiggles or dots where there are errors, and the Error List window will describe the errors.

25.3. Using Dafny From the Command Line

25.3.0. Dafny Command Line Options

The command `Dafny.exe /?` gives the following description of options that can be passed to Dafny.

```
---- Dafny options ------------------------------------------------------

Multiple .dfy files supplied on the command line are concatenated into one
Dafny program.

/dprelude:<file>
            choose Dafny prelude file
/dprint:<file>
            print Dafny program after parsing it
            (use - as <file> to print to console)
/printMode:<Everything|NoIncludes|NoGhost>
            NoIncludes disables printing of {:verify false} methods incorporated via the
```

```
            include mechanism, as well as datatypes and fields included from other files.
            NoGhost disables printing of functions, ghost methods, and proof statements
            in implementation methods.  It also disables anything NoIncludes disables.
/rprint:<file>
            print Dafny program after resolving it
            (use - as <file> to print to console)
/dafnyVerify:<n>
            0 - stop after typechecking
            1 - continue on to translation, verification, and compilation
/compile:<n>  0 - do not compile Dafny program
            1 (default) - upon successful verification of the Dafny
                program, compile Dafny program to .NET assembly
                Program.exe (if the program has a Main method) or
                Program.dll (othewise), where Program.dfy is the name
                of the last .dfy file on the command line
            2 - always attempt to compile Dafny program to C# program
                out.cs, regardless of verification outcome
            3 - if there is a Main method and there are no verification
                errors, compiles program in memory (i.e., does not write
                an output file) and runs it
/spillTargetCode:<n>
            0 (default) - don't write the compiled Dafny program (but
                still compile it, if /compile indicates to do so)
            1 - write the compiled Dafny program as a .cs file
/dafnycc     Disable features not supported by DafnyCC
/noCheating:<n>
            0 (default) - allow assume statements and free invariants
            1 - treat all assumptions as asserts, and drop free.
/induction:<n>
            0 - never do induction, not even when attributes request it
            1 - only apply induction when attributes request it
            2 - apply induction as requested (by attributes) and also
                for heuristically chosen quantifiers
            3 (default) - apply induction as requested, and for
                heuristically chosen quantifiers and ghost methods
/inductionHeuristic:<n>
            0 - least discriminating induction heuristic (that is, lean
                toward applying induction more often)
```

```
              1,2,3,4,5 - levels in between, ordered as follows as far as
                  how discriminating they are:  0 < 1 < 2 < (3,4) < 5 < 6
              6 (default) - most discriminating
/noIncludes   Ignore include directives
/noNLarith    Reduce Z3's knowledge of non-linear arithmetic (*,/,%).
              Results in more manual work, but also produces more predictable behavior.
/autoReqPrint:<file>
              Print out requirements that were automatically generated by autoReq.
/noAutoReq    Ignore autoReq attributes
/allowGlobals Allow the implicit class '_default' to contain fields, instance functions,
              and instance methods.  These class members are declared at the module scope,
              outside of explicit classes.  This command-line option is provided to simply
              a transition from the behavior in the language prior to version 1.9.3, from
              which point onward all functions and methods declared at the module scope are
              implicitly static and fields declarations are not allowed at the module scope.
              The reference manual is written assuming this option is not given.


/nologo       suppress printing of version number, copyright message
/env:<n>      print command line arguments
                0 - never, 1 (default) - during BPL print and prover log,
                2 - like 1 and also to standard output
/wait         await Enter from keyboard before terminating program
/xml:<file>   also produce output in XML format to <file>

---- Boogie options --------------------------------------------------------

Multiple .bpl files supplied on the command line are concatenated into one
Boogie program.

/proc:<p>      : limits which procedures to check
/noResolve     : parse only
/noTypecheck   : parse and resolve only

/print:<file>  : print Boogie program after parsing it
                 (use - as <file> to print to console)
/pretty:<n>
              0 - print each Boogie statement on one line (faster).
```

```
            1 (default) - pretty-print with some line breaks.
/printWithUniqueIds : print augmented information that uniquely
               identifies variables
/printUnstructured : with /print option, desugars all structured statements
/printDesugared : with /print option, desugars calls

/overlookTypeErrors : skip any implementation with resolution or type
                    checking errors

/loopUnroll:<n>
            unroll loops, following up to n back edges (and then some)
/soundLoopUnrolling
            sound loop unrolling
/printModel:<n>
            0 (default) - do not print Z3's error model
            1 - print Z3's error model
            2 - print Z3's error model plus reverse mappings
            4 - print Z3's error model in a more human readable way
/printModelToFile:<file>
            print model to <file> instead of console
/mv:<file>    Specify file where to save the model in BVD format
/enhancedErrorMessages:<n>
            0 (default) - no enhanced error messages
            1 - Z3 error model enhanced error messages

/printCFG:<prefix> : print control flow graph of each implementation in
                   Graphviz format to files named:
                     <prefix>.<procedure name>.dot

/useBaseNameForFileName : When parsing use basename of file for tokens instead
                        of the path supplied on the command line

---- Inference options -------------------------------------------------------

/infer:<flags>
            use abstract interpretation to infer invariants
            The default is /infer:i
               <flags> are as follows (missing <flags> means all)
```

```
                i = intervals
                c = constant propagation
                d = dynamic type
                n = nullness
                p = polyhedra for linear inequalities
                t = trivial bottom/top lattice (cannot be combined with
                    other domains)
                j = stronger intervals (cannot be combined with other
                    domains)
            or the following (which denote options, not domains):
                s = debug statistics
            0..9 = number of iterations before applying a widen (default=0)
/noinfer     turn off the default inference, and overrides the /infer
            switch on its left
/checkInfer   instrument inferred invariants as asserts to be checked by
            theorem prover
/interprocInfer
            perform interprocedural inference (deprecated, not supported)
/contractInfer
            perform procedure contract inference
/instrumentInfer
            h - instrument inferred invariants only at beginning of
                loop headers (default)
            e - instrument inferred invariants at beginning and end
                of every block (this mode is intended for use in
                debugging of abstract domains)
/printInstrumented
            print Boogie program after it has been instrumented with
            invariants

---- Debugging and general tracing options ---------------------------------

/trace       blurt out various debug trace information
/traceTimes   output timing information at certain points in the pipeline
/tracePOs     output information about the number of proof obligations
            (also included in the /trace output)
/log[:method] Print debug output during translation
```

```
/break        launch and break into debugger

---- Verification-condition generation options ------------------------------

/liveVariableAnalysis:<c>
              0 = do not perform live variable analysis
              1 = perform live variable analysis (default)
              2 = perform interprocedural live variable analysis
/noVerify     skip VC generation and invocation of the theorem prover
/verifySnapshots:<n>
              verify several program snapshots (named <filename>.v0.bpl
              to <filename>.vN.bpl) using verification result caching:
              0 - do not use any verification result caching (default)
              1 - use the basic verification result caching
              2 - use the more advanced verification result caching
/verifySeparately
              verify each input program separately
/removeEmptyBlocks:<c>
              0 - do not remove empty blocks during VC generation
              1 - remove empty blocks (default)
/coalesceBlocks:<c>
              0 = do not coalesce blocks
              1 = coalesce blocks (default)
/vc:<variety> n = nested block (default for /prover:Simplify),
              m = nested block reach,
              b = flat block, r = flat block reach,
              s = structured, l = local,
              d = dag (default, except with /prover:Simplify)
              doomed = doomed
/traceverify  print debug output during verification condition generation
/subsumption:<c>
              apply subsumption to asserted conditions:
              0 - never, 1 - not for quantifiers, 2 (default) - always
/alwaysAssumeFreeLoopInvariants
              usually, a free loop invariant (or assume
              statement in that position) is ignored in checking contexts
              (like other free things); this option includes these free
              loop invariants as assumes in both contexts
```

```
/inline:<i>   use inlining strategy <i> for procedures with the :inline
              attribute, see /attrHelp for details:
                none
                assume (default)
                assert
                spec
/printInlined
              print the implementation after inlining calls to
              procedures with the :inline attribute (works with /inline)
/lazyInline:1
              Use the lazy inlining algorithm
/stratifiedInline:1
              Use the stratified inlining algorithm
/fixedPointEngine:<engine>
              Use the specified fixed point engine for inference
/recursionBound:<n>
              Set the recursion bound for stratified inlining to
              be n (default 500)
/inferLeastForUnsat:<str>
              Infer the least number of constants (whose names
              are prefixed by <str>) that need to be set to
              true for the program to be correct. This turns
              on stratified inlining.
/smoke        Soundness Smoke Test: try to stick assert false; in some
              places in the BPL and see if we can still prove it
/smokeTimeout:<n>
              Timeout, in seconds, for a single theorem prover
              invocation during smoke test, defaults to 10.
/causalImplies
              Translate Boogie's A ==> B into prover's A ==> A && B.
/typeEncoding:<m>
              how to encode types when sending VC to theorem prover
                 n = none (unsound)
                 p = predicates (default)
                 a = arguments
                 m = monomorphic
/monomorphize
              Do not abstract map types in the encoding (this is an
```

```
            experimental feature that will not do the right thing if
            the program uses polymorphism)
/reflectAdd   In the VC, generate an auxiliary symbol, elsewhere defined
            to be +, instead of +.

---- Verification-condition splitting ------------------------------------

/vcsMaxCost:<f>
            VC will not be split unless the cost of a VC exceeds this
            number, defaults to 2000.0. This does NOT apply in the
            keep-going mode after first round of splitting.
/vcsMaxSplits:<n>
            Maximal number of VC generated per method. In keep
            going mode only applies to the first round.
            Defaults to 1.
/vcsMaxKeepGoingSplits:<n>
            If set to more than 1, activates the keep
            going mode, where after the first round of splitting,
            VCs that timed out are split into <n> pieces and retried
            until we succeed proving them, or there is only one
            assertion on a single path and it timeouts (in which
            case error is reported for that assertion).
            Defaults to 1.
/vcsKeepGoingTimeout:<n>
            Timeout in seconds for a single theorem prover
            invocation in keep going mode, except for the final
            single-assertion case. Defaults to 1s.
/vcsFinalAssertTimeout:<n>
            Timeout in seconds for the single last
            assertion in the keep going mode. Defaults to 30s.
/vcsPathJoinMult:<f>
            If more than one path join at a block, by how much
            multiply the number of paths in that block, to accomodate
            for the fact that the prover will learn something on one
            paths, before proceeding to another. Defaults to 0.8.
/vcsPathCostMult:<f1>
/vcsAssumeMult:<f2>
            The cost of a block is
```

```
                (<assert-cost> + <f2>*<assume-cost>) *
                (1.0 + <f1>*<entering-paths>)
            <f1> defaults to 1.0, <f2> defaults to 0.01.
            The cost of a single assertion or assumption is
            currently always 1.0.
/vcsPathSplitMult:<f>
            If the best path split of a VC of cost A is into
            VCs of cost B and C, then the split is applied if
            A >= <f>*(B+C), otherwise assertion splitting will be
            applied. Defaults to 0.5 (always do path splitting if
            possible), set to more to do less path splitting
            and more assertion splitting.
/vcsDumpSplits
            For split #n dump split.n.dot and split.n.bpl.
            Warning: Affects error reporting.
/vcsCores:<n>
            Try to verify <n> VCs at once. Defaults to 1.
/vcsLoad:<f>  Sets vcsCores to the machine's ProcessorCount * f,
            rounded to the nearest integer (where 0.0 <= f <= 3.0),
            but never to less than 1.

---- Prover options -------------------------------------------------------

/errorLimit:<num>
            Limit the number of errors produced for each procedure
            (default is 5, some provers may support only 1)
/timeLimit:<num>
            Limit the number of seconds spent trying to verify
            each procedure
/errorTrace:<n>
            0 - no Trace labels in the error output,
            1 (default) - include useful Trace labels in error output,
            2 - include all Trace labels in the error output
/vcBrackets:<b>
            bracket odd-characteredidentifier names with |'s.  <b> is:
              0 - no (default with non-/prover:Simplify),
              1 - yes (default with /prover:Simplify)
/prover:<tp>  use theorem prover <tp>, where <tp> is either the name of
```

```
            a DLL containing the prover interface located in the
            Boogie directory, or a full path to a DLL containing such
            an interface. The standard interfaces shipped include:
                SMTLib (default, uses the SMTLib2 format and calls Z3)
                Z3 (uses Z3 with the Simplify format)
                Simplify
                ContractInference (uses Z3)
                Z3api (Z3 using Managed .NET API)
/proverOpt:KEY[=VALUE]
            Provide a prover-specific option (short form /p).
/proverLog:<file>
            Log input for the theorem prover.  Like filenames
            supplied as arguments to other options, <file> can use the
            following macros:
                @TIME@    expands to the current time
                @PREFIX@  expands to the concatenation of strings given
                          by /logPrefix options
                @FILE@    expands to the last filename specified on the
                          command line
            In addition, /proverLog can also use the macro '@PROC@',
            which causes there to be one prover log file per
            verification condition, and the macro then expands to the
            name of the procedure that the verification condition is for.
/logPrefix:<str>
            Defines the expansion of the macro '@PREFIX@', which can
            be used in various filenames specified by other options.
/proverLogAppend
            Append (not overwrite) the specified prover log file
/proverWarnings
            0 (default) - don't print, 1 - print to stdout,
            2 - print to stderr
/proverMemoryLimit:<num>
            Limit on the virtual memory for prover before
            restart in MB (default:100MB)
/restartProver
            Restart the prover after each query
/proverShutdownLimit<num>
            Time between closing the stream to the prover and
```

```
            killing the prover process (default: 0s)
/platform:<ptype>,<location>
            ptype = v11,v2,cli1
            location = platform libraries directory

Simplify specific options:
/simplifyMatchDepth:<num>
            Set Simplify prover's matching depth limit

Z3 specific options:
/z3opt:<arg>  specify additional Z3 options
/z3multipleErrors
            report multiple counterexamples for each error
/useArrayTheory
            use Z3's native theory (as opposed to axioms).  Currently
            implies /monomorphize.
/useSmtOutputFormat
            Z3 outputs a model in the SMTLIB2 format.
/z3types     generate multi-sorted VC that make use of Z3 types
/z3lets:<n>   0 - no LETs, 1 - only LET TERM, 2 - only LET FORMULA,
            3 - (default) any
/z3exe:<path>
            path to Z3 executable

CVC4 specific options:
/cvc4exe:<path>
            path to CVC4 executable
```

26. References

References

[0] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods*

for Components and Objects: 4th International Symposium, FMCO 2005, volume 4111, pages 364–387. Springer, September 2006.

[1] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development — Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.

[2] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda — a functional language with dependent types. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009*, volume 5674 of *Lecture Notes in Computer Science*, pages 73–78. Springer, August 2009.

[3] Juanito Camilleri and Tom Melham. Reasoning with inductively defined relations in the HOL theorem prover. Technical Report 265, University of Cambridge Computer Laboratory, 1992.

[4] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*, volume 4963, pages 337–340. Springer, March–April 2008.

[5] K. Rustan M. Leino et al. Dafny source code. Available at http://dafny.codeplex.com.

[6] John Harrison. Inductive definitions: Automation and application. In E. Thomas Schubert, Phillip J. Windley, and Jim Alves-Foss, editors, *TPHOLs 1995*, volume 971 of *LNCS*, pages 200–213. Springer, 1995.

[7] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12 (10): 576–580,583, October 1969.

[8] Bart Jacobs and Jan Rutten. An introduction to (co)algebra and (co)induction. In Davide Sangiorgi and Jan Rutten, editors, *Advanced Topics in Bisimulation and Coinduction*, number 52 in Cambridge Tracts in Theoretical Computer Science, pages 38–99. Cambridge University Press, October 2011.

[9] Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on Formal Methods*, volume 4085, pages 268–283. Springer, August 2006.

[10] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[11] Dexter Kozen and Alexandra Silva. Practical coinduction. Technical Report http://hdl.handle.net/1813/30510, Comp. and Inf. Science, Cornell Univ., 2012.

[12] Alexander Krauss. *Automating Recursive Definitions and Termination Proofs in Higher-Order Logic*. PhD thesis, Technische Universität München, 2009.

[13] K. Rustan M. Leino. Main microsoft research dafny web page, a. Available at http://research.microsoft.com/en-us/projects/dafny.

[14] K. Rustan M. Leino. Dafny quick reference, b. Available at http://research.microsoft.com/en-us/projects/dafny/reference.aspx.

[15] K. Rustan M. Leino. Try dafny in your browser, c. Available at http://rise4fun.com/Dafny.

[16] K. Rustan M. Leino. This is Boogie 2. Manuscript KRML 178, 2008. Available at http://research.microsoft.com/~leino/papers.html.

[17] K. Rustan M. Leino. Dynamic-frame specifications in dafny. JML seminar, Dagstuhl, Germany, 2009. Available at http://research.microsoft.com/en-us/um/people/leino/papers/dafny-jml-dagstuhl-2009.pptx.

[18] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *LPAR-16*, volume 6355, pages 348–370. Springer, April 2010.

[19] K. Rustan M. Leino. Automating induction with an SMT solver. In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation — 13th International Conference, VMCAI 2012*, volume 7148, pages 315–331. Springer, January 2012.

[20] K. Rustan M. Leino and Michal Moskal. Co-induction simply: Automatic co-inductive proofs in a program verifier. Manuscript KRML 230, 2014a. Available at http://research.microsoft.com/en-us/um/people/leino/papers/krml230.pdf.

[21] K. Rustan M. Leino and Michał Moskal. Co-induction simply — automatic co-inductive proofs in a program verifier. In *FM 2014*, volume 8442 of *LNCS*, pages 382–398. Springer, May 2014b.

[22] K. Rustan M. Leino and Nadia Polikarpova. Verified calculations. Manuscript KRML 231, 2013. Available at http://research.microsoft.com/en-us/um/people/leino/papers/krml231.pdf.

[23] K. Rustan M. Leino and Philipp Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010*, volume 6015, pages 312–327. Springer, March 2010.

[24] Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Information and Computation*, 207 (2): 284–304, February 2009.

[25] Panagiotis Manolios and J Strother Moore. Partial functions in ACL2. *Journal of Automated Reasoning*, 31 (2): 107–127, 2003.

[26] Robin Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., 1982. ISBN 0387102353.

[27] Hanspeter Mössenböck, Markus Löberbauer, and Albrecht Wöß. The compiler generator coco/r. Open source from University of Linz, 2013. Available at http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/.

[28] Tobias Nipkow and Gerwin Klein. *Concrete Semantics with Isabelle/HOL*. Springer, 2014.

[29] Christine Paulin-Mohring. Inductive definitions in the system Coq — rules and properties. In *TLCA '93*, volume 664 of *LNCS*, pages 328–345. Springer, 1993.

[30] Lawrence C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In Alan Bundy, editor, *CADE-12*, volume 814 of *LNCS*, pages 148–161. Springer, 1994.

[31] Benjamin C. Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Catalin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Software Foundations*. http://www.cis.upenn.edu/~bcpierce/sf, version 3.2 edition, January 2015.

[32] Jan Smans, Bart Jacobs, Frank Piessens, and Wolfram Schulte. Automatic verifier for Java-like programs based on dynamic frames. In José Luiz Fiadeiro and Paola Inverardi, editors, *Fundamental Approaches to Software Engineering, 11th International Conference, FASE 2008*, volume 4961, pages 261–275. Springer, March–April 2008.

[33] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In Sophia Drossopoulou, editor, *ECOOP 2009 — Object-Oriented Programming, 23rd European Conference*, volume 5653, pages 148–172. Springer, July 2009.

[34] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. In *ICFP 2011*, pages 266–278. ACM, September 2011.

[35] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5: 285–309, 1955.

[36] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.