Dafny Reference Manual

K. Rustan M. Leino, Richard L. Ford, David R. Cok

February 15, 2022

Abstract: This is the Dafny reference manual; it 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.

Acknowledgements

Rustan Leino is the designer of the Dafny language and the chief implementor of the dafny tools. Leino, Richard Ford, and David Cok are the principal authors of this document. Many others contributed to the implementation and to documenting various aspects of the language, as described on the project website at https://github.com/dafny-lang/dafny.

Contents

Abstract: This is the Dafny reference manual; it 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.

(Link to current document as pdf)
(Link to current document as html)

1. Introduction

Dafny (Leino 2010) is a programming language with built-in specification constructs, so that verifying a program's correctness with respect to those specifications is a natural part of writing software. The Dafny static program verifier can be used to verify the functional correctness of programs. This document is a reference manual for the programming language and a user guide for the dafny tool that performs verification and compilation to an executable form.

The Dafny programming language is designed to support the static verification of programs. It is imperative, sequential, supports generic classes, inheritance and abstraction, methods and functions, dynamic allocation, inductive and coinductive 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 in much the same way as with the static type checker—when the tool produces errors, the programmer responds by changing the program's type declarations, specifications, and statements.

(This document typically uses "Dafny" to refer to the programming language and "dafny" to refer to the software tool that verifies and compiles programs in the Dafny language.)

The easiest way to try out Dafny is to download it to run it on your machine as you follow along with the Dafny tutorial. Dafny can be run from the command line (on Linux, MacOS, Windows or other platforms) or from an IDE such as emacs or an editor such as VSCode, which can provide syntax highlighting without the built-in verification.

The verifier is powered by Boogie (Barnett et al. 2006; Leino 2008b; Leino and Rümmer 2010) and Z3 (Moura and Bjørner 2008).

From verified programs, the `dafny` compiler can produce code for a number of different backends: the .NET platform via intermediate C# files, Java, Javascript, Go, and C++. Each language provides a basic Foreign Function

Interface (through uses of `:extern`) and a supporting runtime library. However, there is no automatic FFI generator, so `:extern` stubs must be written by hand.

This reference manual for the Dafny verification system is based on the following references: (Leino 2010), (Leino 2008a), (Leino and Polikarpova 2013), (Leino and Moskal 2014a), Co-induction Simply.

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

The details of using (and contributing to) the dafny tool are described in the User Guide (Section 24).

1.1. 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 end, the standard idiom in Dafny is to
// declare a predicate 'Valid()' that is true of an object when
// the data structure representing that 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 declares 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? // Node? means a Node value or null

  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.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;
}
```

2. 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)(Mössenböck, Löberbauer, and Wöß 2013). 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 (C# for the `dafny` tool) intermixed with these special sections:

0. The Characters section which defines classes of characters that are used in defining the lexer.
1. The Tokens section which defines the lexical tokens.
2. The Productions section which defines the grammar. The grammar productions are distributed in the later parts of this document in the places 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 transformations have been performed.

- The semantics actions, enclosed by "(." and ".)", were 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 give 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; 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. In summary:

- identifiers starting with a lower case letter denote terminal symbols
- identifiers starting with an upper case letter denote nonterminal symbols
- strings (a sequence of characters enclosed by double quote characters) denote the sequence of enclosed characters
- `=` 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.

2.1. Dafny Input

Dafny source code files are readable text encoded as UTF-8 Unicode (because this is what the Coco/R-generated scanner and parser read). All program text other than the contents of comments, character, string and verbatim string literals are printable and white-space ASCII characters, that is, ASCII characters in the range `!` to `~`, plus space, tab, cr and nl (ASCII, 9, 10, 13, 32) characters.

However, a current limitation of the Coco/R tool used by `dafny` is that only printable and white-space ASCII characters can be used. Use `\u` escapes in string and character literals to insert unicode characters. Unicode in comments will work fine unless the unicode is interpreted as an end-of-comment indication. Unicode in verbatim strings will likely not be interpreted as intended. [Outstanding issue #818].

2.2. Tokens and whitespace

The characters used in a Dafny program fall into four groups:

- White space characters
- alphanumerics: letters, digits, underscore (_), apostrophe ('), and question mark (?)
- punctuation: `(){}[],.`;`
- operator characters (the other printable characters)

Each Dafny token consists of a sequence of consecutive characters from just one of these groups, excluding white-space. White-space is ignored except that it separates tokens.

A sequence of alphanumeric characters (with no preceding or following additional alphanumeric characters) is a *single* token. This is true even if the token is syntactially or semantically invalid and the sequence could be separated into more than one valid token. For example, `assert56` is one identifier token, not a keyword `assert` followed by a number; `ifb!=0` begins with the token `ifb` and not with the keyword `if` and token `b`; `0xFFFFZZ` is an illegal token, not a valid hex number `0xFFFF` followed by an identifier `ZZ`. White-space must be used to separate two such tokens in a program.

Somewhat differently, operator tokens need not be separated. Only specific sequences of operator characters are recognized and these are somewhat context-sensitive. For example, in `seq<set<int>>`, the grammar knows that `>>` is two individual `>` tokens terminating the nested type parameter lists; the right shift operator `>>` would never be valid here. Similarly, the sequence `==>` is always one token; even if it were invalid in its context, separating it into `==` and `>` would always still be invalid.

In summary, except for required white space between alphanumeric tokens, adding white space between tokens or removing white space can never result in changing the meaning of a Dafny program. For the rest of this document, we consider Dafny programs as sequences of tokens.

2.3. Character Classes

This section defines character classes used later in the token definitions. In this section a backslash is used to start an escape sequence; so for example `'\n'` denotes the single linefeed character. Also in this section, double quotes enclose the set of characters constituting a character class; enclosing single quotes are used when there is just one character in the class. `+` indicates the union of two character classes; `-` is the set-difference between the two classes. `ANY` designates all unicode characters.

```
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"
posDigitFrom2 = "23456789"
```

A `posDigit` is a digit, excluding 0. `posDigitFrom2` excludes both 0 and 1.

```
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, and
- `?` 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. Here the scanner generator will interpret `ANY` as any unicode character. However, `nonidchar` is used only to mark the end of the `!in` token; in this context any character other than whitespace or printable ASCII will trigger a subsequent scanning or parsing error.

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

Characters that can appear in a character constant. See the discussion on unicode support.

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

Characters that can appear in a string constant. See the discussion on unicode support.

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

Characters that can appear in a verbatim string. See the discussion on unicode support.

2.4. Comments

Comments are in two forms.

- They may go from `/*` to `*/` .
- They may go from `//` to the end of the line.

Comments may be nested, but note that the nesting of multi-line comments is behavior that is different from most programming languages. In Dafny,

```
method m() {
  /* comment
     /* nested comment
     */
     rest of outer comment
  */
}
```

is permitted; this feature is convenient for commenting out blocks of program statements that already have multi-line comments within them. Other than looking for end-of-comment delimiters, the contents of a comment are not interpreted. Comments may contain any unicode character, but see the discussion on unicode support for more information.

Note that the nesting is not fool-proof. In

```
method m() {
  /* var i: int;
     // */ line comment
     var j: int;
```

```
  */
}
```

and

```
method m() {
  /* var i: int;
     var s: string := "a*/b";
     var j: int;
   */
}
```

the `*/` inside the line comment and the string are seen as the end of the outer comment, leaving trailing text that will provoke parsing errors.

2.5. 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.

2.5.1. Reserved Words

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

```
reservedword =
    "abstract" | "allocated" | "as" | "assert" | "assume" |
    "bool" | "break" | "by" |
    "calc" | "case" | "char" | "class" | "codatatype" |
    "colemma" | "const" | "constructor" | "copredicate" |
    "datatype" | "decreases" |
    "else" | "ensures" | "exists" | "export" | "extends" |
    "false" | "forall" | "fresh" | "function" | "ghost" |
    "if" | "imap" | "import" | "in" | "include" | "inductive" |
    "int" | "invariant" | "is" | "iset" | "iterator" |
    "label" | "lemma" | "map" | "match" | "method" |
    "modifies" | "modify" | "module" | "multiset" |
    "nameonly" | "nat" | "new" | "newtype" | "null" |
    "object" | "object?" | "old" | "opened" | "ORDINAL"
    "predicate" | "print" | "provides" |
    "reads" | "real" | "refines" | "requires" | "return" |
    "returns" | "reveal" | "reveals" |
    "seq" | "set" | "static" | "string" |
    "then" | "this" | "trait" | "true" | "twostate" | "type" |
    "unchanged" | "var" | "while" | "witness" |
    "yield" | "yields" |
```

```
    arrayToken | bvToken

arrayToken = "array" [ posDigitFrom2 | posDigit digit { digit }]["?"]

bvToken = "bv" ( 0 | 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. `array1` and `array1?` are not reserved words; they are just ordinary identifiers. Similarly, `bv0`, `bv1`, and `bv8` are reserved words, but `bv02` is an ordinary identifier.

2.5.2. Identifiers

```
ident = nondigitIdChar { idchar } - 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 a character literal or a reserved word (including array or bit-vector type tokens). Also, `ident` tokens that begin with an `_` are not permitted as user identifiers.

2.5.3. Digits

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

A sequence of decimal digits, possibly interspersed with underscores for readability (but not beginning or ending with an underscore). Example: `1_234_567`.

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

A hexadecimal constant, possibly interspersed with underscores for readability (but not beginning or ending with an underscore). Example: `0xffff_ffff`.

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

A decimal fraction constant, possibly interspersed with underscores for readability (but not beginning or ending with an underscore). Example: `123_456.789_123`.

2.5.4. 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 a single- or double-quote character, backslash, null, new line, carriage return, tab, or a Unicode character with given hexadecimal representation.

2.5.5. 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, string, and verbatim strings constants and in its comments. A character constant has type `char`.

2.5.6. 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 consist of any characters (including newline characters) except that two successive double quotes represent one quote character inside the string. This is the mechanism for escaping a double quote character, which is the only character needing escaping in a verbatim string.

2.5.7. Ellipsis

```
ellipsis = "..."
```

The ellipsis symbol is typically used to designate something missing that will later be inserted through refinement or is already present in a parent declaration..

2.6. Low Level Grammar Productions

2.6.1. 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 "." may be an identifier, 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 a digit sequence is treated as a string, not a number: internal underscores matter, so `10` is different from `1_0` and from `010`.
- `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 or, in some contexts, a `digits`. 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. This wildcard has several uses in the language, but it is not used as part of expressions.

2.6.2. 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` and other identifier-related symbols.

```
IdentOrDigits = Ident | digits
NoUSIdentOrDigits = NoUSIdent | digits
ModuleName = NoUSIdent
ClassName = NoUSIdent    // also traits
DatatypeName = NoUSIdent
DatatypeMemberName = NoUSIdentOrDigits
NewtypeName = NoUSIdent
SynonymTypeName = NoUSIdent
IteratorName = NoUSIdent
TypeVariableName = NoUSIdent
MethodFunctionName = NoUSIdentOrDigits
```

```
LabelName = NoUSIdentOrDigits
AttributeName = NoUSIdent
ExportId = NoUSIdentOrDigits
TypeNameOrCtorSuffix = NoUSIdentOrDigits
```

Some parsing constexts

2.6.3. Qualified Names

```
QualifiedModuleName = ModuleName { "." ModuleName }
```

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

- `Module.MyType1`
- `MyTuple.1`
- `MyMethod.requires`
- `A.B.C.D`

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`.

2.6.4. 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.

```
FIdentType = NoUSIdentOrDigits ":" Type
```

A `FIdentType` is used to declare a field. The Type is required because there is no initializer.

```
CIdentType = NoUSIdentOrDigits [ ":" Type ]
```

A `CIdentType` is used for a `const` declaration. The Type is optional because it may be inferred from the initializer.

```
GIdentType(allowGhostKeyword, allowNewKeyword, allowNameOnlyKeyword, allowDefault) =
    { "ghost" | "new" | "nameonly" } IdentType
    [ ":=" Expression(allowLemma: true, allowLambda: true) ]
```

A `GIdentType` is a typed entity declaration optionally preceded by `ghost` or `new`. The *ghost* qualifier means the entity is only used during verification

and 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. If `allowNewKeyword` is false, then `new` is not allowed. If `allowNameOnlyKeyword` is false, then `nameonly` is not allowed. If `allowDefault` is false, then `:= Expression` is not allowed.

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

A `LocalIdentTypeOptional` is used when declaring local variables. If a value is specified for the variable, the type may be omitted because it can be inferred from the initial value. An initial value is not required.

```
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" | "nameonly" } [ NoUSIdentOrDigits ":" ] Type
    [ ":=" Expression(allowLemma: true, allowLambda: true) ]
```

`TypeIdentOptional`s are used in `FormalsOptionalIds`. This represents situations where a type is given but there may not be an identifier. The default-value expression `:= Expression` is allowed only if `NoUSIdentOrDigits :` is also provided. If modifier `nameonly` is given, then `NoUSIdentOrDigits` must also be used.

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

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

2.6.5. 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.

3. 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) constants, methods, functions, lemmas, types (classes, traits, inductive and co-inductive datatypes, newtypes, 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 an executable appropriate to the target langauge; otherwise it will emit a library or individual files. The conditions for a legal Main() method are described in the User Guide (Section 24.9.1). If there is more than one Main(), Dafny will emit an error message.

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

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

3.1. 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 the included files unless they have been listed explicitly on the command line.

The file name may be a path using the customary `/`, `.`, and `..` specifiers. The interpretation of the name (e.g., case-sensitivity) will depend on the underlying operating system. A path not beginning with `/` is looked up in the underlying file system relative to the current working directory (the one in which the dafny tool is invoked). Paths beginning with a device designator (e.g., `C:`) are only permitted on Windows systems.

3.2. Top Level Declarations

```
TopDecl = {
  { DeclModifier }
```

```
  ( SubModuleDecl
  | ClassDecl
  | DatatypeDecl
  | NewtypeDecl
  | SynonymTypeDecl  // includes opaque types
  | 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 various kinds of declarations described later. Top-level declarations are implicitly members of a default (unnamed) top-level module.

The `ClassDecl`, `DatatypeDecl`, `NewtypeDecl`, `SynonymTypeDecl`, `IteratorDecl`, and `TraitDecl` declarations are type declarations and are described in Section 6 and the following sections. 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`.

3.3. Declaration Modifiers

```
DeclModifier = ( "abstract" | "ghost" | "static" )
```

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.

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

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

4. Modules

```
SubModuleDecl = ( ModuleDefinition | ModuleImport | ModuleExport )
```

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, as well as to 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.

4.1. Declaring New Modules

```
ModuleDefinition = "module" { Attribute } ModuleQualifiedName
        [ "refines" ModuleQualifiedName ]
        "{" { TopDecl } "}"

ModuleQualifiedName = ModuleName { "." ModuleName }
```

A `ModuleQualifiedName` is a qualified name that is expected to refer to a module; a *qualified name* is a sequence of `.`-separated identifiers, which designates a program entity by representing increasingly-nested scopes.

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
  }
}
```

Note that everything declared at the top-level (in all the files constituting the program) is implicitly part of a single implicit unnamed global module.

4.2. Declaring nested modules standalone

As described in the previous section, module declarations can be nested. It is also permitted to declare a nested module *outside* of its "enclosing" module. So instead of

```
module A {
  module B {
  }
}
```

one can write

```
module A {
}
module A.B {
}
```

The second module is completely separate; for example, it can be in a different file. This feature provides flexibility in writing and maintenance; for example, it can reduce the size of module `A` by extracting module `A.B` into a separate body of text.

However, it can also lead to confusion and program authors need to take care. It may not be apparent to a reader of module `A` that module `A.B` exists; the existence of `A.B` might cause names to be resolved differently and the semantics of the program might be (silently) different if `A.B` is present or absent.

4.3. Importing Modules

```
ModuleImport =
    "import"
    [ "opened" ]
    ( QualifiedModuleExport
    | ModuleName "=" QualifiedModuleExport
    | ModuleName ":" QualifiedModuleExport
    )

QualifiedModuleExport =
    ModuleQualifiedName [ "`" ModuleExportSuffix ]

ModuleExportSuffix =
    ( ExportId
    | "{" ExportId { "," ExportId } "}"
    )
```

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, which has two forms with different meanings. The simplest form is the concrete import, which 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`. This form can also be used to create a reference to a nested module, as in `import A = B.C`. The other form, using a `:`, is described in Section 4.6.

As modules in the same scope must have different names, this ability to bind a module to a new name allows disambiguating separately developed external modules that have the same name. Note that the new name is only bound in

the scope 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, that other module either has to be declared there with `module` or imported with `import`. (As described below, the resolution of the `ModuleQualifiedName` that follows the `=` in the `import` statement or the `refines` in a module declaration uses slightly different rules.)

We don't have to give `Helpers` a new name, though, if we don't want to. We can write `import Helpers = Helpers` to import the module under its own name; 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. When importing nested modules, `import B.C` means `import C = B.C`; the implicit name is always the last name segment of the module designation.

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

Import statements may occur at the top-level of a program (that is, in the implicit top-level module of the program) as well. There they serve simply as a way to give a new name, perhaps a shorthand name, to a module. For example,

```
module MyModule { ... } // declares module MyModule
import MyModule  // error: cannot add a moduled named MyModule
                 // because there already is one
import M = MyModule // OK. M and MyModule are equivalent
```

4.4. 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, if present, must immediately follow `import`. 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 have lower priority than 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. However, if the ambiguous references actually refer to the same declaration, then they are permitted. The `opened` keyword may be used with any kind of `import` declaration, including the module abstraction form.

An `import opened` may occur at the top-level as well. For example,

```
module MyModule { ... } // declares MyModule
import opened MyModule // does not declare a new module, but does
                       // make all names in MyModule available in
                       // the current scope, without needing
                       // qualification
import opened M = MyModule // names in MyModule are available in
                       // the current scope without qualification
                       // or qualified with either M or MyModule
```

The Dafny style guidelines suggest using opened imports sparingly. They are best used when the names being imported have obvious and unambiguous meanings and when using qualified names would be verbose enough to impede understanding.

4.5. Export Sets and Access Control

```
ModuleExport =
  "export"
  [ ExportId ]
  [ "..." ]
  {
    "provides" ( ExportSignature { "," ExportSignature } | "*" )
  | "reveals"  ( ExportSignature { "," ExportSignature } | "*" )
  | "extends"  ExportId { "," ExportId }
  }

ExportSignature = TypeNameOrCtorSuffix [ "." TypeNameOrCtorSuffix ]
```

In some programming languages, keywords such as `public`, `private`, and `protected` are used to control access to (that is, visibility of) declared program entities. In Dafny, modules and export sets provide that capability. Modules combine declarations into logically related groups. Export sets then permit selectively exposing subsets of declarations; another module can import the export set appropriate to its needs. A user can define as many export sets as are needed to provide different kinds of access to the module's declarations. Each export set designates a list of names, which must be names that are declared in the module (or in a refinement parent).

By default all the names declared in a module are available outside the module using the `import` mechanism. An *export set* enables a module to disallow the use of some declarations outside the module.

Export sets have names; those names are used in `import` statements to designate which export set of a module is being imported. If a module `M` has export sets `E1` and `E2`, we can write `import A = M`E1` to create a module alias `A` that contains only the names in `E1`. Or we can write `import A = M`{E1,E2}` to import the union of names in `E1` and `E2` as module alias `A`. As before, `import M`E1` is an abbreviation of `import M = M`E1`.

If no export set is given in an import statement, the default export set of the module is used.

There are various defaults that apply differently in different cases. The following description is with respect to an example module `M`:

`M` has no export sets declared. Then another module may simply `import Z = M` to obtain access to all of M's declarations.

`M` has one or more named export sets (e.g., `E`, `F`). Then another module can write `import Z = M`E` or `import Z = M`{E,F}` to obtain access to the names that are listed in export set `E` or to the union of those in export sets `E` and `F`, respectively. If no export set has the same name as the module, then an export set designator must be used: in that case you cannot write simply `import Z = M`.

`M` has an unnamed export set, along with other export sets (e.g., `E`). The unnamed export set is the default export set and implicitly has the same name as the module. Because there is a default export set, another module may write either `import Z = M` or `import Z = M`M` to import the names in that default export set. You can also still use the other export sets with the explicit designator: `import Z = M`E`

`M` declares an export set with the same name as the module. This is equivalent to declaring an export set without a name. `import M` and `import M`M` perform the same function in either case; the export set with or without the name of the module is the default export set for the module.

Note that names of module aliases (declared by import statements) are just like other names in a module; they can be included or omitted from export sets. Names brought into a module by *refinement* are treated the same as locally declared names and can be listed in export set declarations. However, names brought into a module by `import opened` (either into a module or a refinement parent of a module) may not be further exported. For example,

```
module A {
  const a := 10;
  const z := 10;
}
module B {
  import opened Z = A // includes a, declares Z
  const b := Z.a; // OK
}
module C {
  import opened B // includes b, Z, but not a
  //assert b == a; // error: a is not known
  //assert b == B.a; // error: B.a is not valid
  //assert b == A.a; // error: A is not known
  assert b == Z.a; // OK: module Z is known and includes a
}
```

However, in the above example,

- if `A` has one export set `export Y reveals a` then the import in module `B` is invalid because `A` has no default export set;
- if `A` has one export set `export Y reveals a` and `B` has `import Z = A`Y` then B's import is OK. So is the use of `Z.a` in the assert because `B` declares

`Z` and `C` brings in `Z` through the `import opened` and `Z` contains `a` by virtue of its declaration. (The alias `Z` is not able to have export sets; all of its names are visible.)

- if `A` has one export set `export provides z` then `A` does have a default export set, so the import in `B` is OK, but neither the use of `a` in `B` nor as `Z.a` in C would be valid, because `a` is not in `Z`.

The default export set is important in the resolution of qualified names, as described in Section 4.8.

4.5.1. Provided and revealed names

Names can be exported from modules in two ways, designated by `provides` and `reveals` in the export set declaration.

When a name is exported as *provided*, then inside a module that has imported the name only the name is known, not the details of the name's declaration.

For example, in the following code the constant `a` is exported as provided.

```
module A {
  export provides a
  const a := 10;
  const b := 20;
}

module B {
  import A
  method m() {
    assert A.a == 10; // a is known, but not its value
    // assert A.b == 20; // b is not known through A`A
  }
}
```

Since `a` is imported into module `B` through the default export set `A`A`, it can be referenced in the assert statement. The constant `b` is not exported, so it is not available. But the assert about `a` is not provable because the value of `a` is not known in module `B`.

In contrast, if `a` is exported as *revealed*, as shown in the next example, its value is known and the assertion can be proved.

```
module A {
  export reveals a
  const a := 10;
  const b := 20;
}

module B {
```

```
  import A
  method m() {
    assert A.a == 10; // a and its value are known
    // assert A.b == 20; // b is not known through A`A
  }
}
```

The following list presents the difference between *provides* and *reveals* for each kind of declaration.

- const: type always known, but value not known when only provided
- function, predicate: signature always known, but body not known when not revealed
- method: TODO
- lemma: TODO
- iterator: TODO
- class, trait: TODO
- opaque type: TODO
- subset type, newtype: TODO
- datatype: TODO
- module: module names may only be provided
- export set: names of export sets are always visible and are not subject to export set rules, that is, export set names may not be put in the provides or reveals lists in export set declarations.

A few other notes:

- Using a `*` instead of a list of names means that all local names (except export set names) in the module are exported.
- If no export sets are declared, then the implicit export set is `export reveals *`
- A module acquires all the export sets from its refinement parent.
- Names acquired by a module from its refinement parent are also subject to export lists. (These are local names just like those declared directly.)
- Names acquired by a module via an `import opened` declaration are not re-exportable, though the new module alias name (such as the `C` in `import C = A.B`) is a local name.

4.5.2. Extends list

An export set declaration may include an *extends* list, which is a list of one or more export set names from the same module containing the declaration (including export set names obtained from a refinement parent). The effect is to include in the declaration the union of all the names in the export sets in the extends list, along with any other names explicitly included in the declaration. So for example in

```
module M {
  const a := 10;
  const b := 10;
  const c := 10;
  export A reveals a
  export B reveals b
  export C reveals c extends A, B
}
```

export set C will contain the names `a`, `b`, and `c`.

4.6. 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 : B`. This means bind the name `A` as before, but instead of getting the exact module `B`, you get any module which *adheres* to `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
}
abstract module Mod {
  import A : Interface
  method m() {
    assert 6 <= A.addSome(5);
  }
}
```

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
  ...
}
```

When you refine an abstract import into a concrete one Dafny checks 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.

A module that includes an abstract import must be declared `abstract`.

4.7. Module Ordering and Dependencies

Dafny isn't particular about the textual order in which modules are declared, but they must follow some rules to be well formed. In particular, there must be a way to order the modules in a program such that each only refers to things defined **before** it in the ordering. That doesn't mean the modules have to be given textually in that order in the source text. 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 // error: circular
```

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 permitted 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 before their containing module, even at the toplevel. This means that the following is not well formed:

```
method doIt() { }
module M {
  method m() {
    doIt(); // error: M precedes 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`, and `B` itself depends on its own nested module `B.C`.

4.8. 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. The form shown above contains three instances of `AugmentedDotSuffix_`.

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

4.8.1. Modules and name spaces

A module is a collection of declarations, each of which has a name. These names are held in two namespaces.

- The names of export sets
- The names of all other declarations, including submodules and aliased modules

In addition names can be classified as *local* or *imported*.

- Local declarations of a module are the declarations that are explicit in the module and the local declarations of the refinement parent. This includes, for example, the `N` of `import N =` in the refinement parent, recursively.
- Imported names of a module are those brought in by `import opened` plus the imported names in the refinement parent.

Within each namespace, the local names are unique. Thus a module may not reuse a name that a refinement parent has declared (unless it is a refining declaration, which replaces both declarations, as described in Section 0).

Local names take precedence over imported names. If a name is used more than once among imported names (coming from different imports), then it is ambiguous to *use* the name without qualification.

4.8.2. Module Id Context Name Resolution

A qualified name may be used to refer to a module in an import statement or a refines clause of a module declaration. Such a qualified name is resolved as follows, with respect to its syntactic location within a module `Z`:

0. The leading `NameSegment` is resolved as a local or imported module name of `Z`, if there is one with a matching name. The target of a `refines` clause does not consider local names, that is, in `module Z refines A.B.C`, any contents of `Z` are not considered in finding `A`.

1. Otherwise, it is resolved as a local or imported module name of the most enclosing module of `Z`, iterating outward to each successive enclosing module until a match is found or the default toplevel module is reached without a match. No consideration of export sets, default or otherwise, is used in this step. Howecver, if at any stage a matching name is found that is not a module declaration, the resolution fails. See the examples below.

2a. Once the leading `NameSegment` is resolved as say module `M`, the next `NameSegment` is resolved as a local or imported module name within `M` The resolution is restricted to the default export set of `M`.

2b. If the resolved module name is a module alias (from an `import` statement) then the target of the alias is resolved as a new qualified name with respect to its syntactic context (independent of any resolutions or modules so far). Since `Z` depends on `M`, any such alias target will already have been resolved, beccause modules are resolved in order of dependency.

3. Step 2 is iterated for each `NameSegment` in the qualified module id, resulting in a module that is the final resolution of the complete qualified id.

Ordinarily a module must be *imported* in order for its constitutent declarations to be visible inside a given module `M`. However, for the resolution of qualified names this is not the case.

This example shows that the resolution of the refinement parent does not use any local names:

```
module A {
  const a := 10
}

module B refines A { // the top-level A, not the submodule A
  module A { const a := 30 }
  method m() { assert a == 10; } // true
}
```

In the example, the `A` in `refines A` refers to the global `A`, not the submodule `A`.

A module is a collection of declarations, each of which has a name. These names are held in two namespaces.

- The names of export sets
- The names of all other declarations, including submodules and aliased modules

In addition names can be classified as *local* or *imported*.

- Local declarations of a module are the declarations that are explicit in the module and the local declarations of the refinement parent. This includes, for example, the `N` of `import N =` in the refinement parent, recursively.
- Imported names of a module are those brought in by `import opened` plus the imported names in the refinement parent.

Within each namespace, the local names are unique. Thus a module may not reuse a name that a refinement parent has declared (unless it is a refining declaration, which replaces both declarations, as described in Section 21).

Local names take precedence over imported names. If a name is used more than once among imported names (coming from different imports), then it is ambiguous to *use* the name without qualification, unless they refer to the same entity or to equal types.

4.8.3. Module Id Context Name Resolution

A qualified name may be used to refer to a module in an import statement or a refines clause of a module declaration. Such a qualified name is resolved as follows, with respect to its syntactic location within a module `Z`:

0. The leading `NameSegment` is resolved as a local or imported module name of `Z`, if there is one with a matching name. The target of a `refines` clause does not consider local names, that is, in `module Z refines A.B.C`, any contents of `Z` are not considered in finding `A`.

1. Otherwise, it is resolved as a local or imported module name of the most enclosing module of `Z`, iterating outward to each successive enclosing module until a match is found or the default toplevel module is reached without a match. No consideration of export sets, default or otherwise, is used in this step. However, if at any stage a matching name is found that is not a module declaration, the resolution fails. See the examples below.

2a. Once the leading `NameSegment` is resolved as say module `M`, the next `NameSegment` is resolved as a local or imported module name within `M` The resolution is restricted to the default export set of `M`.

2b. If the resolved module name is a module alias (from an `import` statement) then the target of the alias is resolved as a new qualified name with respect to its syntactic context (independent of any resolutions or modules so far). Since

`Z` depends on `M`, any such alias target will already have been resolved, beccause modules are resolved in order of dependency.

3. Step 2 is iterated for each `NameSegment` in the qualified module id, resulting in a module that is the final resolution of the complete qualified id.

Ordinarily a module must be *imported* in order for its constitutent declarations to be visible inside a given module `M`. However, for the resolution of qualified names this is not the case.

Ths example shows that the resolution of the refinement parent does not use any local names:

```
module A {
  const a := 10
}

module B refines A { // the top-level A, not the submodule A
  module A { const a := 30 }
  method m() { assert a == 10; } // true
}
```

The `A` in `refines A` refers to the submodule `A`, not the global `A`.

4.8.4. 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` is not 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 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. In each module, names from opened modules are also potential martches, but only after names declared in the module. If a ambiguous name is found or name of the wrong kind (e.g. a module instead of an expression identifier), 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.

4.8.5. 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`?

5. 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`, which typically consist of a sequence of specification clauses.
- 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.

5.1. Specification Clauses

5.1.1. Requires Clause

```
RequiresClause(allowLabel) =
  "requires" { Attribute }
  [ LabelName ":" ]  // Label allowed only if allowLabel is true
  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, with a collected list of all the given Attributes. The order of conjunctions (and hence the order of `requires` clauses with respect to each other) can be important: earlier conjuncts can set conditions that establish that later conjuncts are well-defined.

5.1.2. Ensures Clause

```
EnsuresClause(allowLambda) =
  "ensures" { Attribute } Expression(allowLemma: false,
                                     allowLambda)
```

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, with a collected list of all the given Attributes. The order of conjunctions (and hence the order of `ensures` clauses with respect to each other) can be important: earlier conjuncts can set conditions that establish that later conjuncts are well-defined.

5.1.3. Decreases Clause

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

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

PossiblyWildExpression(allowLambda, allowWild) =
  ( "*"  // if allowWild is false, using '*' provokes an error
  | Expression(allowLemma: false, allowLambda)
  )
```

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

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 and a collected list of Attributes. That is,

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

is equivalent to

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

Note that changing the order of multiple `decreases` clauses will change the order of the expressions within the equivalent single `decreases` clause, and will therefore have different semantics.

Loops and compiled methods (but not functions and not ghost methods, including lemmas) can be specified to be possibly non-terminating. This is done by declaring the method or loop with `decreases *`, which causes the proof of termination to be skipped. If a `*` is present in a `decreases` clause, no other expressions are allowed in the `decreases` clause. A method that contains a possibly non-terminating loop or a call to a possibly non-terminating method must itself be declared as possibly non-terminating.

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 $\top$.

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, Dafny supplies 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 the `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 be bothered to think about that there was a need to prove termination in the first place.

TODO: Should there be user-level syntax to invoke this termination ordering

5.1.4. Framing

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

FrameField = "`" IdentOrDigits

PossiblyWildFrameExpression(allowLemma, allowLambda, allowWild) =
  ( "*"  // error if !allowWild and '*'
  | FrameExpression(allowLemma, allowLambda)
  )
```

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 within a method of a class or trait.

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.

5.1.5. Reads Clause

```
ReadsClause(allowLemma, allowLambda, allowWild) =
  "reads"
  { Attribute }
  PossiblyWildFrameExpression(allowLemma, allowLambda, allowWild)
  { "," PossiblyWildFrameExpression(allowLemma, allowLambda, allowWild) }
```

Functions are not allowed to have side effects; they may also 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). If a `reads` clause uses `*`, then the `reads` clause is not allowed to mention anything else (since anything else would be irrelevant, anyhow).

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.

TO BE WRITTEN: multiset of objects allowed in reads clauses

5.1.6. Modifies Clause

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

Frames also affect methods. 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. These 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.

If an object is newly allocated within the body of a method or within the scope of a `modifies` statement or a loop's `modifies` clause, then the fields of that object may 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 (cf. Section 19.22).

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 may modify anything the enclosing context is allowed to modify.

Note that *modifies* here is used in the sense of *writes*. That is, a field that may not be modified may not be written to, even with the same value it already has or even if the value is restored later. The terminology and semantics varies among specification languages. Some define frame conditions in this sense (a) of *writes* and others in the sense (b) that allows writing a field with the same value or changing the value so long as the original value is restored by the end of the scope. For example, JML defines `assignable` and `modifies` as synonyms in the sense (a), though KeY interprets JML's `assigns/modifies` in sense (b).

ACSL and ACSL++ use the `assigns` keyword, but with *modify* (b) semantics. Ada/SPARK's dataflow contracts encode *write* (a) semantics.

5.1.7. 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.

5.2. Method Specification

```
MethodSpec =
  { ModifiesClause(allowLambda: false)
  | RequiresClause(allowLabel: true)
  | EnsuresClause(allowLambda: false)
  | 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.

5.3. Function Specification

```
FunctionSpec =
  { RequiresClause(allowLabel: true)
  | ReadsClause(allowLemma: false, allowLambda: false,
                                   allowWild: true)
  | EnsuresClause(allowLambda: false)
  | DecreasesClause(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.

5.4. Lambda Specification

```
LambdaSpec =
  { ReadsClause(allowLemma: true, allowLambda: false,
                                  allowWild: true)
  | RequiresClause(allowLabel: false)
  }
```

// TODO - the above grammar is not quite right for Requires

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.

5.5. Iterator Specification

```
IteratorSpec =
  { ReadsClause(allowLemma: false, allowLambda: false,
                                  allowWild: false)
  | ModifiesClause(allowLambda: false)
  | [ "yield" ] RequiresClause(allowLabel: !isYield)
  | [ "yield" ] EnsuresClause(allowLambda: false)
  | 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?

5.6. Loop Specification

```
LoopSpec =
  { InvariantClause_
  | DecreasesClause(allowWildcard: true, allowLambda: true)
  | ModifiesClause(allowLambda: true)
  }
```

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.

5.7. Auto-generated boilerplate specifications

TO BE WRITTEN - {:autocontracts}

6. Types

```
Type = DomainType_ | ArrowType_
```

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

```
DomainType_ =
  ( BoolType_ | CharType_ | IntType_ | RealType_
  | OrdinalType_ | BitVectorType_ | ObjectType_
  | FiniteSetType_ | InfiniteSetType_
  | MultisetType_
  | FiniteMapType_ | InfiniteMapType_
  | SequenceType_
  | NatType_
  | StringType_
  | 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.

6.1. Value Types

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

- The basic scalar types: `bool`, `char`, `int`, `real`, `ORDINAL`, bitvector types
- The built-in collection types: `set`, `iset`, `multiset`, `seq`, `string`, `map`, `imap`
- Tuple Types
- Inductive and co-inductive types
- Function (arrow) types
- Subset and newtypes that are based on value types

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

The `nat` type is a pre-defined subset type of `int`.

Dafny does not include types themselves as values, nor is there a type of types.

6.2. 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. Dafny supports both reference types that contain the special `null` value (*nullable types*) and reference types that do not (*non-null types*).

6.3. 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` (Section 20.40) that does not allow a `HashCall`.

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

7. Basic types

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

7.1. Booleans

```
BoolType_ = "bool"
```

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

Type `bool` supports the following operations:

operator	description
`<==>`	equivalence (if and only if)
`==>`	implication (implies)
`<==`	reverse implication (follows from)
`&&`	conjunction (and)
`||`	disjunction (or)
`==`	equality
`!=`	disequality
`!`	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.

7.1.1. Equivalence Operator

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

7.1.2. Conjunction and Disjunction

Conjunction and disjunction are associative. 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 `&`. The same holds for `||` and `|`.

7.1.3. 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, ==> is right associative
C <== B <== A
(C <== B) <== A // parentheses redundant, <== 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 6.2), which means the following two expressions are well-formed:

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

The contrapositives 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 20.34.

7.2. 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`, either *subset types* or *newtypes*, are described in Section 11.3 and Section 12.

There is one built-in *subset type*, `nat`, representing the non-negative subrange of `int`.

The language includes a literal for each 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 literals, use the unary minus operator, as in `-12`, but not -(12).

There are also literals for some of the reals. These are written as a decimal point with a nonempty sequence of decimal digits on both sides, optionally prefixed by a `-` character. For example, `1.0`, `1609.344`, `-12.5`, 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 Angstrom
```

In addition to equality and disequality, numeric types support the following relational operations, which have the same precedence as equality:

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) – int only
`-`	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 less strongly than any of these operators. There is no implicit conversion between `int` and `real`: use `as int` or `as real` conversions to write an explict conversion (cf. Section 20.10).

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 value, 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 `Floor` that returns the *floor* of the real value (as an int 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.Floor == 3
(-2.5).Floor == -3
-2.5.Floor == -2 // This is -(2.5.Floor)
r.Floor as real <= r
r <= r' ==> r.Floor <= r'.Floor
```

Note in the third line that member access (like `.Floor`) binds stronger than unary minus. The fourth line uses the conversion function `as real` from `int` to `real`, as described in Section 20.10.

TODO: Need syntax for real literals with exponents

7.3. Bit-vector Types

```
BitVectorType_ = bvToken
```

Dafny includes a family of bit-vector types, each type having a specific, constant length, the number of bits in its values. Each such type is distinct and is designated by the prefix `bv` followed (without white space) by a positive integer (without leading zeros) stating the number of bits. For example, `bv1`, `bv8`, and `bv32` are legal bit-vector type names. The type `bv0` is also legal; it is a bit-vector type with no bits and just one value, `0x0`.

Constant literals of bit-vector types are given by integer literals converted automatically to the designated type, either by an implicit or explicit conversion operation or by initialization in a declaration. Dafny checks that the constant literal is in the correct range. For example,

```
const i: bv1 := 1
const j: bv8 := 195
const k: bv2 := 5 // error - out of range
const m := (194 as bv8) | (7 as bv8)
```

Bit-vector values can be converted to and from `int` and other bit-vector types, as long as the values are in range for the target type. Bit-vector values are always considered unsigned.

Bit-vector operations include bit-wise operators and arithmetic operators (as well as equality, disequality, and comparisons). The arithmetic operations truncate the high-order bits from the results; that is, they perform unsigned arithmetic modulo 2^{number of bits}, like 2's-complement machine arithmetic.

operator	description
`<<`	bit-limited bit-shift left
`>>`	unsigned bit-shift right
`+`	bit-limited addition
`-`	bit-limited subtraction
`*`	bit-limited multiplication
`&`	bit-wise and
`|`	bit-wise or
`^`	bit-wise exclusive-or
`-`	bit-limited negation (unary minus)
`!`	bit-wise complement

The groups of operators lower in the table above bind more tightly.[1] All operators bind more tightly than equality, disequality, and comparisons. All binary operators are left-associative, but the bit-wise `&`, `|`, and `^` do not associate together (parentheses are required to disambiguate).

The right-hand operand of bit-shift operations is an `int` value, must be non-negative, and no more than the number of bits in the type. There is no signed right shift as all bit-vector values correspond to non-negative integers.

Here are examples of the various operations (all the assertions are true except where indicated):

```
const i: bv4 := 9
const j: bv4 := 3

method m() {
  assert (i & j) == (1 as bv4);
  assert (i | j) == (11 as bv4);
  assert (i ^ j) == (10 as bv4);
  assert !i == (6 as bv4);
```

[1] The binding power of shift and bit-wise operations is different than in C-like languages.

```
  assert -i == (7 as bv4);
  assert (i + i) == (2 as bv4);
  assert (j - i) == (10 as bv4);
  assert (i * j) == (11 as bv4);
  assert (i as int) / (j as int) == 3;
  assert (j << 1) == (6 as bv4);
  assert (i << 1) == (2 as bv4);
  assert (i >> 1) == (4 as bv4);
  assert i == 9; // auto conversion of literal to bv4
  assert i * 4 == j + 8 + 9; // arithmetic is modulo 16
  assert i + j >> 1 == (i + j) >> 1; // + - bind tigher than << >>
  assert i + j ^ 2 == i + (j^2);
  assert i * j & 1 == i * (j&1); // & | ^ bind tighter than + - *
}
```

The following are incorrectly formed:

```
const i: bv4 := 9
const j: bv4 := 3

method m() {
  assert i & 4 | j == 0 ; // parentheses required
}

const k: bv4 := 9

method p() {
  assert k as bv5 == 9 as bv6; // error: mismatched types
}
```

These produce assertion errors:

```
const i: bv4 := 9

method m() {
  assert i as bv3 == 1; // error: i is out of range for bv3
}
const j: bv4 := 9

method n() {
  assert j == 25; // error: 25 is out of range for bv4
}
```

7.4. Ordinal type

```
OrdinalType_ = "ORDINAL"
```

TO BE WRITTEN

7.5. 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 the following:

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*, where each *x* is a hexadecimal digit

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 in string literals (Section 10.3.5). 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 10.3.5.

Character values can be converted to and from `int` values using the `as int` and `as char` conversion operations. The result is what would be expected in other programming languages, namely, the `int` value of a `char` is the ACSII or unicode numeric value.

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.

8. Type parameters

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

// The optional Variance indicator is permitted only if allowVariance is true
Variance = ( "*" | "+" | "!" | "-" )

TypeParameterCharacteristics = "(" TPCharOption { "," TPCharOption } ")"

TPCharOption = ( "==" | "0" | "00" | "!" "new" )
```

Many of the types, 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.

Dafny has some inference support that makes certain signatures less cluttered (described in Section 23.2).

8.1. Declaring restrictions on type parameters

It is sometimes necessary to restrict type parameters so that they can only be instantiated by certain families of types, that is, by types that have certain properties. These properties are known as *type characteristics*. The following subsections describe the type characteristics that Dafny supports.

In some cases, type inference will infer that a type-parameter must be restricted in a particular way, in which case Dafny will add the appropriate suffix, such as `(==)`, automatically.

If more than one restriction is needed, they are either listed comma-separated, inside the parentheses or as multiple parenthesized elements: `T(==,0)` or `T(==)(0)`.

8.1.1. Equality-supporting type parameters: `T(==)`

Designating a type parameter with the `(==)` suffix indicates that the parameter may only be replaced in non-ghost contexts with types that are known to support run-time equality comparisons (`==` and `!=`). All types support equality in ghost contexts, as if, for some types, the equality function is ghost.

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 when used in a non-ghost context. Again, note that *all* types support equality in *ghost* contexts; the difference is only for non-ghost (that is, compiled) code. Co-inductive datatypes, arrow types, and inductive datatypes with ghost parameters are examples of types that are not equality supporting.

8.1.2. Auto-initializable types: `T(0)`

At every access of a variable `x` of a type `T`, Dafny ensures that `x` holds a legal value of type `T`. If no explicit initialization is given, then an arbitrary value is assumed by the verifier and supplied by the compiler, that is, the variable is *auto-initialized*. For example,

```
method m() {
  var n: nat; // Auto-initialized to an arbitrary value of type `nat`
  assert n >= 0; // true, regardless of the value of n
  var i: int;
  assert i >= 0; // possibly false, arbitrary ints may be negative
}
```

For some types (known as *auto-init types*), the compiler can choose an initial value, but for others it does not. Variables and fields whose type the compiler does not auto-initialize are subject to *definite-assignment* rules. These ensure that the program explicitly assigns a value to a variable before it is used. For more details see Section 23.6 and the `-definiteAssignment` command-line option.

Dafny supports auto-init as a type characteristic. To restrict a type parameter to auto-init types, mark it with the `(0)` suffix. For example,

```
method AutoInitExamples<A(0), X>() returns (a: A, x: X)
{
  // 'a' does not require an explicit initialization, since A is auto-init
  // error: out-parameter 'x' has not been given a value
}
```

In this example, an error is reported because out-parameter `x` has not been assigned—since nothing is known about type `X`, variables of type `X` are subject to definite-assignment rules. In contrast, since type parameter `A` is declared to be restricted to auto-init types, the program does not need to explicitly assign any value to the out-parameter `a`.

8.1.3. Nonempty types: `T(00)`

Auto-init types are important in compiled contexts. In ghost contexts, it may still be important to know that a type is nonempty. Dafny supports a type characteristic for nonempty types, written with the suffix `(00)`. For example,

```
method NonemptyExamples<B(00), X>() returns (b: B, ghost g: B, ghost h: X)
{
  // error: non-ghost out-parameter 'b' has not been given a value
  // ghost out-parameter 'g' is fine, since its type is nonempty
  // error: 'h' has not been given a value
}
```

Because of `B`'s nonempty type characteric, ghost parameter `g` does not need to be explicitly assigned. However, Dafny reports an error for the non-ghost `b`, since `B` is not an auto-init type, and reports an error for `h`, since the type `X` could be empty.

Note that every auto-init type is nonempty.

8.1.4. Non-heap based: `T(!new)`

Dafny makes a distinction between types whose values are on the heap, i.e. references, like classes and arrays, and those that are strictly value-based, like basic types and datatypes. The practical implication is that references depend on allocation state (e.g., are affected by the `old` operation) whereas non-reference values are not. Thus it can be relevant to know whether the values of a type parameter are heap-based or not. This is indicated by the mode suffix `(!new)`.

A type parameter characterized by `(!new)` is *recursively* independent of the allocation state. For example, a datatype is not a reference, but for a parameterized data type such as

```
datatype Result<T> = Failure(error: string) | Success(value: T)
```

the instantiation `Result<int>` satisfies `(!new)`, whereas `Result<array<int>>` does not.

Note that this characteristic of a type parameter is operative for both verification and compilation. Also, opaque types at the topmost scope are always implicitly `(!new)`.

Here are some examples:

```
datatype Result<T> = Failure(error: string) | Success(v: T)
datatype ResultN<T(!new)> = Failure(error: string) | Success(v: T)

class C {}

method m() {
```

```
  var x1: Result<int>;
  var x2: ResultN<int>;
  var x3: Result<C>;
  var x4: ResultN<C>; // error
  var x5: Result<array<int>>;
  var x6: ResultN<array<int>>; // error
}
```

8.2. Type parameter variance

TO BE WRITTEN: Type parameter variance

9. 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 (cf. Section 23.2).

10. Collection types

Dafny offers several built-in collection types.

10.1. Sets

```
FiniteSetType_ = "set" [ GenericInstantiation ]

InfiniteSetType_ = "iset" [ GenericInstantiation ]
```

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

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

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 20.35.

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

operator	description
`*`	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	result type	description
`|s|`	`nat`	set cardinality (not for `iset`)
`e in s`	`bool`	set membership
`e !in s`	`bool`	set non-membership

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

10.2. 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	result type	description
`|s|`	`nat`	multiset cardinality
`e in s`	`bool`	multiset membership
`e !in s`	`bool`	multiset non-membership
`s[e]`	`nat`	multiplicity of `e` in `s`
`s[e := n]`	`multiset<T>`	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
```

10.3. 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.

10.3.1. 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 also a sequence comprehension expression (Section 20.27):

```
seq(5, i => i*i)
```

is equivalent to `[0, 1, 4, 9, 16]`.

10.3.2. 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 `>=`.

10.3.3. Sequence Concatenation

Sequences support the following binary operator:

operator	description
`+`	concatenation

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

10.3.4. 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	result type	description
`|s|`	`nat`	sequence length
`s[i]`	`T`	sequence selection
`s[i := e]`	`seq<T>`	sequence update
`e in s`	`bool`	sequence membership
`e !in s`	`bool`	sequence non-membership
`s[lo..hi]`	`seq<T>`	subsequence
`s[lo..]`	`seq<T>`	drop
`s[..hi]`	`seq<T>`	take
`s[`*slices*`]`	`seq<seq<T>>`	slice
`multiset(s)`	`multiset<T>`	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. If *slices* is terminated by a colon, then the length of the last slice extends until the end of `s`, that is, its length is `|s|` minus the sum of the given length designators. For example,

the following equalities hold, for any sequence `s` of length at least `10`:

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

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

10.3.5. 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 (Section 7.5). 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 do 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.

10.4. 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 in a map display expression, only the last occurrence appears in the resulting map.[2] There is also a *map comprehension expression*, explained in Section 20.39.

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	result type	description
`|fm|`	`nat`	map cardinality
`m[d]`	`U`	map selection
`m[t := u]`	`map<T,U>`	map update
`t in m`	`bool`	map domain membership
`t !in m`	`bool`	map domain non-membership
`m.Keys`	(i)`set<T>`	the domain of `m`
`m.Values`	(i)`set<U>`	the range of `m`
`m.Items`	(i)`set<(T,U)>`	set of pairs (t,u) in `m`

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

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

The expressions `m.Keys`, `m.Values`, and `m.Items` return, as sets, the domain, the range, and the 2-tuples holding the key-value associations in the map. Note that `m.Values` will have a different cardinality than `m.Keys` and `m.Items` if different keys are associated with the same value. If `m` is an `imap`, then these expressions return `iset` values.

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 := ComputeJTCoefficient(K); // do expensive computation
  cache := cache[K := coeff];  // update the cache
}
```

Dafny also overloads the `+` and `-` binary operators for maps. The `+` operator merges two maps or imaps of the same type, as if each (key,value) pair of the RHS is added in turn to the LHS (i)map. In this use, `+` is not commutative; if a key exists in both (i)maps, it is the value from the RHS (i)map that is present in the result.

The `-` operator implements a map difference operator. Here the LHS is a `map<K,V>` or `imap<K,V>` and the RHS is a `set<K>` (but not an `iset`); the operation removes from the LHS all the (key,value) pairs whose key is a member of the RHS set.

10.5. Iterating over collections

Collections are very commonly used in programming and one frequently needs to iterate over the elements of a collection. Dafny does not have built-in iterator methods, but the idioms by which to do so are straightforward. The subsections below give some introductory examples; more detail can be found in this power user note.

TODO: Add examples of using a iterator class TODO: Should a foreach statment be added to Dafny

[3] This is likely to change in the future as follows: The `in` and `!in` operations will no longer be supported on maps, with `x in m` replaced by `x in m.Keys`, and similarly for `!in`.

10.5.1. Sequences and arrays

Sequences and arrays are indexable and have a length. So the idiom to iterate over the contents is well-known. For an array:

```
  var i := 0;
  var sum := 0;
  while i < a.Length {
    sum := sum + a[i];
    i := i + 1;
  }
}
```

For a sequence, the only difference is the length operator:

```
  var i := 0;
  var sum := 0;
  while i < |s| {
    sum := sum + s[i];
    i := i + 1;
  }
```

The `forall` statement (Section 19.21) can also be used with arrays where parallel assigment is needed:

```
  var rev := new int[s.Length];
  forall i | 0 <= i < s.Length {
    rev[i] := s[s.Length-i-1];
  }
```

10.5.2. Sets

There is no intrinsic order to the elements of a set. Nevertheless, we can extract an arbitrary element of a nonempty set, performing an iteration as follows:

```
// s is a set<int>
  var ss := s;
  while ss != {}
    decreases |ss|
  {
    var i: int :| i in ss;
    ss := ss - {i};
    print i, "\n";
  }
```

Because `isets` may be infinite, Dafny does not permit iteration over an `iset`.

10.5.3. Maps

Iterating over the contents of a `map` uses the component sets: `Keys`, `Values`, and `Items`. The iteration loop follows the same patterns as for sets:

```
var items := m.Items;
while items != {}
  decreases |items|
{
  var item :| item in items;
  items := items - { item };
  print item.0, " ", item.1, "\n";
}
```

There are no mechanisms currently defined in Dafny for iterating over `imaps`.

11. Types that stand for other types

```
SynonymTypeDecl =
  SynonymTypeDecl_ | OpaqueTypeDecl_ | SubsetTypeDecl_
```

It is sometimes useful to know a type by several names or to treat a type abstractly. There are several mechanisms in Dafny to do this:

- (Section 11.1) A typical *synonym type*, in which a type name is a synonym for another type
- (Section 11.2) An *opaque type*, in which a new type name is declared as an uninterpreted type
- (Section 11.3) A *subset type*, in which a new type name is given to a subset of the values of a given type

11.1. Type synonyms

```
SynonymTypeName = NoUSIdent

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

A *type synonym* declaration:

```
type Y<T> = G
```

declares `Y<T>` to be a synonym for the type `G`. If the `= G` is omitted then the declaration just declares a name as an uninterpreted *opaque* type, as described in Section 11.2. Such types may be given a definition elsewhere in the Dafny program.

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

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

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

The new type name itself may have type characteristics declared, though these are typically inferred from the definition, if there is one.

As already described in Section 10.3.5, `string` is a built-in type synonym for `seq<char>`, as if it would have been declared as follows:

```
type string(==,0,!new) = seq<char>
```

If the implicit declaration did not include the type characteristics, they would be inferred in any case.

11.2. Opaque types

```
OpaqueTypeDecl_ =
  "type" { Attribute } SynonymTypeName
   { TypeParameterCharacteristics }
   [ GenericParameters ]
   [ TypeMembers ]

TypeMembers =
  "{"
  {
    { DeclModifier }
    ClassMemberDecl(allowConstructors: false,
                    isValueType: true,
                    moduleLevelDecl: false,
                    isWithinAbstractModule: module.IsAbstract)
  }
  "}"
```

An opaque type is a special case of a type synonym that is underspecified. Such a type is declared simply by:

```
type Y<T>
```

Its definition can be revealed in a refining module. The name `Y` can be immediately followed by a type characteristics suffix (Section 8.1). Because there is no defining RHS, the type characterics cannot be inferred and so must be stated. If, in some refining module, a definition of the type is given, the type characteristics must match those of the new definition.

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.

Even as an opaque type, the type may be given members such as constants, methods or functions.

11.3. Subset types

```
SubsetTypeDecl_ =
  "type"
  { Attribute }
  SynonymTypeName [ GenericParameters ]
  "="
  LocalIdentTypeOptional
  "|"
  Expression(allowLemma: false, allowLambda: true)
  [ "ghost" "witness" Expression(allowLemma: false, allowLambda: true)
  | "witness" Expression((allowLemma: false, allowLambda: true)
  | "witness" "*"
  ]
```

```
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. This condition is checked by the verifier, not by the type checker. Similarly, assignments from one subset type to another (both with the same base type) are also permitted, as long as it can be established that the value being assigned satisfies the predicate defining the receiving 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 20.10.)

Dafny builds in three families of subset types, as described next.

11.3.1. Type`nat`

The built-in type `nat`, which represents the non-negative integers (that is, the natural numbers), is a subset type:

```
type nat = n: int | 0 <= n
```

A simple example that puts subset type `nat` to good use is the standard Fibonacci function:

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

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

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

11.3.2. Non-null types

Every class, trait, and iterator declaration `C` gives rise to two types.

One type has the name `C?` (that is, the name of the class, trait, or iterator declaration with a `?` character appended to the end). The values of `C?` are the references to `C` objects, and also the value `null`. In other words, `C?` is the type of *possibly null* references (aka, *nullable* references) to `C` objects.

The other type has the name `C` (that is, the same name as the class, trait, or iterator declaration). Its values are the references to `C` objects, and does not contain the value `null`. In other words, `C` is the type of *non-null* references to `C` objects.

The type `C` is a subset type of `C?`:

```
type C = c: C? | c != null
```

(It may be natural to think of the type `C?` as the union of type `C` and the value `null`, but, technically, Dafny defines `C` as a subset type with base type `C?`.)

From being a subset type, we get that `C` is a subtype of `C?`. Moreover, if a class or trait `C` extends a trait `B`, then type `C` is a subtype of `B` and type `C?` is a subtype of `B?`.

Every possibly-null reference type is a subtype of the built-in possibly-null trait type `object?`, and every non-null reference type is a subtype of the built-in non-null trait type `object`. (And, from the fact that `object` is a subset type of `object?`, we also have that `object` is a subtype of `object?`.)

Arrays are references and array types also come in these two flavors. For example, `array?` and `array2?` are possibly-null (1- and 2-dimensional) array types, and `array` and `array2` are their respective non-null types.

Note that `?` is not an operator. Instead, it is simply the last character of the name of these various possibly-null types.

11.3.3. Arrow types: `->`, `-->`, and `~>`

The built-in type `->` stands for total functions, `-->` stands for partial functions (that is, functions with possible `requires` clauses), and `~>` stands for all functions. More precisely, these are type constructors that exist for any arity (`() -> X`, `A -> X`, `(A, B) -> X`, `(A, B, C) -> X`, etc.).

For a list of types `TT` and a type `U`, the values of the arrow type `(TT) ~> U` are functions from `TT` to 'U. This includes functions that may read the heap and functions that are not defined on all inputs. It is not common to need this generality (and working with such general functions is difficult). Therefore, Dafny defines two subset types that are more common (and much easier to work with).

The type `(TT) --> U` denotes the subset of `(TT) ~> U` where the functions do not read the (mutable parts of the) heap. Values of type `(TT) --> U` are called *partial functions*, and the subset type `(TT) --> U` is called the *partial arrow type*. (As a mnemonic to help you remember that this is the partial arrow, you may think of the little gap between the two hyphens in `-->` as showing a broken arrow.)

The built-in partial arrow type is defined as follows (here shown for arrows with arity 1):

```
type A --> B = f: A ~> B | forall a :: f.reads(a) == {}
```

(except that what is shown here left of the `=` is not legal Dafny syntax). That is, the partial arrow type is defined as those functions `f` whose reads frame is empty for all inputs. More precisely, taking variance into account, the partial arrow type is defined as

```
type -A --> +B = f: A ~> B | forall a :: f.reads(a) == {}
```

The type `(TT) -> U` is, in turn, a subset type of `(TT) --> U`, adding the restriction that the functions must not impose any precondition. That is, values of type `(TT) -> U` are *total functions*, and the subset type `(TT) -> U` is called the *total arrow type*.

The built-in total arrow type is defined as follows (here shown for arrows with arity 1):

```
type -A -> +B = f: A --> B | forall a :: f.requires(a)
```

That is, the total arrow type is defined as those partial functions `f` whose precondition evaluates to `true` for all inputs.

Among these types, the most commonly used are the total arrow types. They are also the easiest to work with. Because they are common, they have the

simplest syntax (`->`).

Note, informally, we tend to speak of all three of these types as arrow types, even though, technically, the `~>` types are the arrow types and the `-->` and `->` types are subset types thereof. The one place where you may need to remember that `-->` and `->` are subset types is in some error messages. For example, if you try to assign a partial function to a variable whose type is a total arrow type and the verifier is not able to prove that the partial function really is total, then you'll get an error say the subset-type constraint may not be satisfied.

For more information about arrow types, see Section 17.

12. Newtypes

```
NewtypeDecl = "newtype" { Attribute } NewtypeName "="
  [ ellipsis ]
  ( LocalIdentTypeOptional
    "|"
    Expression(allowLemma: false, allowLambda: true)
    [ "ghost" "witness" Expression(allowLemma: false, allowLambda: true)
    | "witness" Expression((allowLemma: false, allowLambda: true)
    | "witness" "*"
    ]
  | Type
  )
  [ TypeMembers ]
```

A newtype is like a type synonym or subset type except that it declares a wholly new type name that is distinct from its base type.

A new type can be declared with the *newtype* declaration, for example:

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

where `M` is a 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`. At present, Dafny only supports `int` and `real` as base types of newtypes.

A newtype is a type that supports the same operations as its base type. The newtype is distinct from and incompatible with other 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.

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

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

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

```
newtype int32 = x | -0x8000_0000 <= x < 0x8000_0000
```

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.

The incompatibility of a newtype and its basetype is intentional, as newtypes are meant to be used as distinct types from the basetype. If numeric types are desired that mix more readily with the basetype, the subset types described in Section 11.3 may be more appropriate.

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 is to use a conversion operation, described in Section 12.1, on `c` to convert it to the base type:

```
-128 <= c as int < 128
```

If possible, Dafny compilers will represent values of the newtype using a native 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 22.1.12.

Furthermore, for the compiler to be able to make an appropriate choice of representation, the constants in the defining expression as shown above must be known constants at compile-time. They need not be numeric literals; combinations of basic operations and symbolic constants are also allowed as described in Section 20.46.

12.1. Conversion operations

For every type `N`, there is a conversion operation with the name `as N`, described more fully in Section 20.10. It is a partial function defined when the given

value, which can be of any 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 have no fractional part. (To round a real-based numeric value down to the nearest integer, use the `.Floor` member, see Section 7.2.)

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

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.

The `as N` conversion operation is grammatically a suffix operation like `.field` and array indexing, but binds less tightly than unary operations: `- x as int` is `(- x) as int`; `a + b as int` is `a + (b as int)`.

The `as N` conversion can also be used with reference types. For example, if `C` is a class, `c` is an expression of type `C`, and `o` is an expression of type `object`, then `c as object` and `c as object?` are upcasts and `o is C` is a downcast. A downcast requires the LHS expression to have the RHS type, as is enforced by the verifier.

For some types (in particular, reference types), there is also a corresponding `is` operation (Section 20.10) that tests whether a value is valid for a given type.

13. Class Types

```
ClassDecl = "class" { Attribute } ClassName [ GenericParameters ]
  ["extends" Type {"," Type} | ellipsis ]
  "{" { { DeclModifier }
        ClassMemberDecl(allowConstructors: true,
                        isValueType: false,
                        moduleLevelDecl: false,
                        isWithinAbstractModule: false) }
  "}"

ClassMemberDecl(allowConstructors, isValueType,
                moduleLevelDecl, isWithinAbstractModule) =
  ( FieldDecl(isValueType) // allowed iff moduleLevelDecl is false
  | ConstantFieldDecl(moduleLevelDecl)
  | FunctionDecl(isWithinAbstractModule)
  | MethodDecl(isGhost: "ghost" was present,
               allowConstructors, isWithinAbstractModule)
  )
```

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 true `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. The text "`extends J1, ..., Jn`" is also optional and 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
  var x: int
  method Example() returns (b: bool)
  {
    var x: int;
    b := f == this.f;
  }
}
```

so the method body always assigns `true` to the out-parameter `b`. However, in this example, `x` and `this.x` are different because the field `x` is shadowed by the declaration of the local variable `x`. There is no semantic difference between qualified and unqualified accesses to the same receiver and member.

A `C` instance is created using `new`. There are three forms of `new`, depending on whether or not the class declares any *constructors* (see Section 13.3.2):

```
c := new C;
c := new C.Init(args);
c := new C(args);
```

For a class with no constructors, the first two forms can be used. The first form simply allocates a new instance of a `C` object, initializing its fields to values of their respective types (and initializing each `const` field with a RHS to its specified value). The second form additionally invokes an *initialization method* (here, named `Init`) on the newly allocated object and the given arguments. It is therefore a shorthand for

```
c := new C;
c.Init(args);
```

An initialization method is an ordinary method that has no out-parameters and that modifies no more than `this`.

For a class that declares one or more constructors, the second and third forms of `new` can be used. For such a class, the second form invokes the indicated constructor (here, named `Init`), which allocates and initializes the object. The third form is the same as the second, but invokes the *anonymous constructor* of the class (that is, a constructor declared with the empty-string name).

13.1. Field Declarations

```
FieldDecl(isValueType) =
  "var" { Attribute } FIdentType { "," FIdentType }
```

A `FieldDecl` is not permitted in a value type (i.e., if `isValueType` is true).

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.

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.

13.2. Constant Field Declarations

```
ConstantFieldDecl(moduleLeavelDecl) =
  "const" { Attribute } CIdentType [ ellipsis ]
   [ ":=" Expression(allowLemma: false, allowLambda:true) ]
```

A `const` declaration declares a name bound to a value, which value is fixed after initialization.

The declaration must either have a type or an initializing expression (or both). If the type is omitted, it is inferred from the initializing expression.

- A const declaration may include the `ghost` and `static` modifiers, but no others.
- A const declaration may appear within a module or within any declaration that may contain members (class, trait, datatype, newtype).
- If it is in a module, it is implicitly `static`, and may not also be declared `static`.
- If the declaration has an initializing expression that is a ghost expression, then the ghost-ness of the declaration is inferred; the `ghost` modifier may be omitted.

13.3. Method Declarations

```
MethodDecl(isGhost, allowConstructors, isWithinAbstractModule) =
  MethodKeyword_ { Attribute } [ MethodFunctionName ]
```

```
  ( MethodSignature_(isGhost, isExtreme: true iff this is a least
                                    or greatest lemma declaration)
  | ellipsis
  )
  MethodSpec(isConstructor: true iff
                       this is a constructor declaration)
  [ 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"
                 | "constructor"
                 | "lemma"
                 | "twostate" "lemma"
                 | "least" "lemma"
                 | "greatest" "lemma"
                 )
```

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

```
MethodSignature_(isGhost, isExtreme) =
  [ GenericParameters ]
  [ KType ]    // permitted only if isExtreme == true
  Formals(allowGhostKeyword: !isGhost, allowNewKeyword: isTwostateLemma, allowDefault: true)
  [ "returns" Formals(allowGhostKeyword: !isGhost, allowNewKeyword: false, allowDefault: fal
```

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.

A `ellipsis` is used when a method or function is being redeclared in a module that refines another module. (cf. Section 21) 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.

```
KType = "[" ( "nat" | "ORDINAL" ) "]"
```

The *k-type* may be specified only for least and greatest lemmas and is described in Section 18.3. // TODO - check this is the correct reference

```
Formals(allowGhostKeyword, allowNewKeyword, allowDefault) =
  "(" [ GIdentType(allowGhostKeyword, allowNewKeyword, allowNameOnlyKeyword: true, allowDefa
        { "," GIdentType(allowGhostKeyword, allowNewKeyword, allowNameOnlyKeyword: true, all
```

```
      ]
  ")"
```

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

See Section 5.2 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 method's in-parameters, `x, y, z` are the method's out-parameters, `Pre` is a boolean expression denoting the method's precondition, `Frame` denotes a set of objects whose fields may be updated by the method, `Post` is a boolean expression denoting the method's postcondition, `Rank` is the method's variant function, and `Body` is a list of statements that implements the method. `Frame` can be a list of expressions, each of which is a set of objects or a single object, the latter standing for the singleton set consisting of that one object. The method's frame is the union of these sets, plus the set of objects allocated by the method body. For example, if `c` and `d` are parameters of a class type `C`, then

```
modifies {c, d}
modifies {c} + {d}
modifies c, {d}
modifies c, d
```

all mean the same thing.

13.3.1. Ordinary methods

A method can be declared as ghost by preceding the declaration with the keyword `ghost` and as static by preceding the declaration with the keyword `static`. The default is non-static (i.e., instance) and non-ghost. An instance method has an implicit receiver parameter, `this`. A static method M in a class C can be invoked by `C.M(…)`.

An ordinary method is declared with the `method` keyword. Section [#sec-

constructors] explains methods that instead use the `constructor` keyword. Section [#sec-lemmas] 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 [#sec-inductive-datatypes]). Methods declared with the `colemma` keyword are discussed later in the context of co-inductive types, in section [#sec-colemmas].

A method without a 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. (TODO: `:extern` attribute?)

13.3.2. Constructors

To write structured object-oriented programs, one often relies on objects being constructed only in certain ways. For this purpose, Dafny provides *constructor (method)s*. A constructor is declared with the keyword `constructor` instead of `method`; constructors are permitted only in classes. A constructor is allowed to be declared as `ghost`, in which case it can only be used in ghost contexts.

A constructor can only be called at the time an object is allocated (see object-creation examples below). Moreover, when a class contains a constructor, every call to `new` for a class must be accompanied by a call to one of its constructors. A class may declare no constructors or one or more constructors.

13.3.2.1. Classes with no explicit constructors For a class that declares no constructors, an instance of the class is created with

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

This allocates an object and initializes its fields to values of their respective types (and initializes each `const` field with a RHS to its specified value). The RHS of a `const` field may depend on other `const` or `var` fields, but circular dependencies are not allowed.

This simple form of `new` is allowed only if the class declares no constructors, which is not possible to determine in every scope. It is easy to determine whether or not a class declares any constructors if the class is declared in the same module that performs the `new`. If the class is declared in a different module and that module exports a constructor, then it is also clear that the class has a constructor (and thus this simple form of `new` cannot be used). (Note that an export set that `reveals` a class `C` also exports the anonymous constructor of `C`, if any.) But if the module that declares `C` does not export any constructors for `C`, then callers outside the module do not know whether or not `C` has a constructor.

Therefore, this simple form of `new` is allowed only for classes that are declared in the same module as the use of `new`.

The simple `new C` is allowed in ghost contexts. Also, unlike the forms of `new` that call a constructor or initialization method, it can be used in a simultaneous assignment; for example

```
c, d, e := new C, new C, 15;
```

is legal.

As a shorthand for writing

```
c := new C;
c.Init(args);
```

where `Init` is an initialization method (see the top of Section [#sec-class-types]), one can write

```
c := new C.Init(args);
```

but it is more typical in such a case to declare a constructor for the class.

(The syntactic support for initialization methods is provided for historical reasons. It may be deprecated in some future version of Dafny. In most cases, a constructor is to be preferred.)

13.3.2.2. Classes with one or more constructors Like other class members, constructors have names. And like other members, their names must be distinct, even if their signatures are different. Being able to name constructors promotes names like `InitFromList` or `InitFromSet` (or just `FromList` and `FromSet`). Unlike other members, one constructor is allowed to be *anonymous*; in other words, an *anonymous constructor* is a constructor whose name is essentially the empty string. For example:

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

The named constructor is invoked as

```
  i := new Item.I(42);
```

The anonymous constructor is invoked as

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

dropping the ".".

13.3.2.3. Two-phase constructors The body of a constructor contains two sections, an initialization phase and a post-initialization phase, separated by a `new;` statement. If there is no `new;` statement, the entire body is the initialization phase. The initialization phase is intended to initialize field variables. In this phase, uses of the object reference `this` are restricted; a program may use `this`

- as the receiver on the LHS,
- as the entire RHS of an assignment to a field of `this`,
- and as a member of a set on the RHS that is being assigned to a field of `this`.

A `const` field with a RHS is not allowed to be assigned anywhere else. A `const` field without a RHS may be assigned only in constructors, and more precisely only in the initialization phase of constructors. During this phase, a `const` field may be assigned more than once; whatever value the `const` field has at the end of the initialization phase is the value it will have forever thereafter.

For a constructor declared as `ghost`, the initialization phase is allowed to assign both ghost and non-ghost fields. For such an object, values of non-ghost fields at the end of the initialization phase are in effect no longer changeable.

There are no restrictions on expressions or statements in the post-initialization phase.

13.3.3. 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 23.5.2.

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

13.3.4. Two-state lemmas and functions

The heap is an implicit parameter to every function, though a function is only allowed to read those parts of the mutable heap that it admits to in its `reads` clause. Sometimes, it is useful for a function to take two heap parameters, for example, so the function can return the difference between the value of a field in the two heaps. Such a *two-state function* is declared by `twostate function` (or `twostate predicate`, which is the same as a `twostate function` that returns a `bool`). A two-state function is always ghost. It is appropriate to think of these two implicit heap parameters as representing a "current" heap and an "old" heap.

For example, the predicate

```
twostate predicate Increasing(c: Cell)
  reads c
{
  old(c.data) <= c.data
}
```

returns `true` if the value of `c.data` has not been reduced from the old state to the current. Dereferences in the current heap are written as usual (e.g., `c.data`) and must, as usual, be accounted for in the function's `reads` clause. Dereferences in the old heap are enclosed by `old` (e.g., `old(c.data)`), just like when one dereferences a method's initial heap. The function is allowed to read anything in the old heap; the `reads` clause only declares dependencies on locations in the current heap. Consequently, the frame axiom for a two-state function is sensitive to any change in the old-heap parameter; in other words, the frame axiom says nothing about two invocations of the two-state function with different old-heap parameters.

At a call site, the two-state function's current-heap parameter is always passed in as the caller's current heap. The two-state function's old-heap parameter is by default passed in as the caller's old heap (that is, the initial heap if the caller is a method and the old heap if the caller is a two-state function). While there is never a choice in which heap gets passed as the current heap, the caller can use any preceding heap as the argument to the two-state function's old-heap parameter. This is done by labeling a state in the caller and passing in the label, just like this is done with the built-in `old` function.

For example, the following assertions all hold:

```
method Caller(c: Cell)
  modifies c
{
  c.data := c.data + 10;
  label L:
  assert Increasing(c);
  c.data := c.data - 2;
  assert Increasing(c);
  assert !Increasing@L(c);
}
```

The first call to `Increasing` uses `Caller`'s initial state as the old-heap parameter, and so does the second call. The third call instead uses as the old-heap parameter the heap at label `L`, which is why the third call returns `false`. As shown in the example, an explicitly given old-heap parameter is given after an `@`-sign (which follows the name of the function and any explicitly given type parameters) and before the open parenthesis (after which the ordinary parameters are given).

A two-state function is allowed to be called only from a two-state context, which

means a method, a two-state lemma (see below), or another two-state function. Just like a label used with an `old` expressions, any label used in a call to a two-state function must denote a program point that *dominates* the call. This means that any control leading to the call must necessarily have passed through the labeled program point.

Any parameter (including the receiver parameter, if any) passed to a two-state function must have been allocated already in the old state. For example, the second call to `Diff` in method `M` is illegal, since `d` was not allocated on entry to `M`:

```
twostate function Diff(c: Cell, d: Cell): int
  reads d
{
  d.data - old(c.data)
}

method M(c: Cell) {
  var d := new Cell(10);
  label L:
  ghost var x := Diff@L(c, d);
  ghost var y := Diff(c, d); // error: d is not allocated in old state
}
```

A two-state function can declare that it only assumes a parameter to be allocated in the current heap. This is done by preceding the parameter with the `new` modifier, as illustrated in the following example, where the first call to `DiffAgain` is legal:

```
twostate function DiffAgain(c: Cell, new d: Cell): int
  reads d
{
  d.data - old(c.data)
}

method P(c: Cell) {
  var d := new Cell(10);
  ghost var x := DiffAgain(c, d);
  ghost var y := DiffAgain(d, c); // error: d is not allocated in old state
}
```

A *two-state lemma* works in an analogous way. It is a lemma with both a current-heap parameter and an old-heap parameter, it can use `old` expressions in its specification (including in the precondition) and body, its parameters may use the `new` modifier, and the old-heap parameter is by default passed in as the caller's old heap, which can be changed by using an `@`-parameter.

Here is an example of something useful that can be done with a two-state lemma:

```
function SeqSum(s: seq<Cell>): int
  reads s
{
  if s == [] then 0 else s[0].data + SeqSum(s[1..])
}

twostate lemma IncSumDiff(s: seq<Cell>)
  requires forall c :: c in s ==> Increasing(c)
  ensures old(SeqSum(s)) <= SeqSum(s)
{
  if s == [] {
  } else {
    calc {
      old(SeqSum(s));
    ==  // def. SeqSum
      old(s[0].data + SeqSum(s[1..]));
    ==  // distribute old
      old(s[0].data) + old(SeqSum(s[1..]));
    <=  { assert Increasing(s[0]); }
      s[0].data + old(SeqSum(s[1..]));
    <=  { IncSumDiff(s[1..]); }
      s[0].data + SeqSum(s[1..]);
    ==  // def. SeqSum
      SeqSum(s);
    }
  }
}
```

A two-state function can be used as a first-class function value, where the receiver (if any), type parameters (if any), and old-heap parameter are determined at the time the first-class value is mentioned. While the receiver and type parameters can be explicitly instantiated in such a use (for example, `p.F<int>` for a two-state instance function `F` that takes one type parameter), there is currently no syntactic support for giving the old-heap parameter explicitly. A caller can work around this restriction by using (fancy-word alert!) eta-expansion, meaning wrapping a lambda expression around the call, as in `x => p.F<int>@L(x)`. The following example illustrates using such an eta-expansion:

```
class P {
  twostate function F<X>(x: X): X
}

method EtaExample(p: P) returns (ghost f: int -> int) {
  label L:
  f := x => p.F<int>@L(x);
}
```

TO BE WRITTEN - unchanged predicate

13.4. Function Declarations

```
FunctionDecl(isWithinAbstractModule) =
  ( [ "twostate" ] "function" [ "method" ] { Attribute }
    MethodFunctionName
    FunctionSignatureOrEllipsis_(allowGhostKeyword:
                                          ("method" present),
                                 allowNewKeyword:
                                          "twostate" present)
  | "predicate" [ "method" ] { Attribute }
    MethodFunctionName
    PredicateSignatureOrEllipsis_(allowGhostKeyword:
                                          ("method" present),
                                  allowNewKeyword:
                                          "twostate" present)
  | ( "least" | "greatest" ) "predicate" { Attribute }
    MethodFunctionName
    PredicateSignatureOrEllipsis_(allowGhostKeyword: false,
                          allowNewKeyword: "twostate" present))
  )
  FunctionSpec
  [ FunctionBody ]

FunctionSignatureOrEllipsis_(allowGhostKeyword) =
  FunctionSignature_(allowGhostKeyword) | ellipsis

FunctionSignature_(allowGhostKeyword, allowNewKeyword) =
  [ GenericParameters ]
  Formals(allowGhostKeyword, allowNewKeyword)
  ":"
  ( Type
  | "(" GIdentType(allowGhostKeyword: false,
                   allowNewKeyword: false,
                   allowNameOnlyKeyword: false,
                   allowDefault: false)
    ")"
  )

PredicateSignatureOrEllipsis_(allowGhostKeyword) =
  PredicateSignature_(allowGhostKeyword) | ellipsis

PredicateSignature_(allowGhostKeyword) =
  [ GenericParameters ] [ KType ] Formals(allowGhostKeyword,
```

```
                                                allowNewKeyword)

FunctionBody = "{" Expression(allowLemma: true, allowLambda: true)
               "}" [ "by" "method" BlockStmt ]
```

A function with a `by method` clause declares a *function-by-method*. A function-by-method gives a way to implement a (deterministic, side-effect free) function by a method (whose body may be nondeterministic and may allocate objects that it modifies). This can be useful if the best implementation uses nondeterminism (for example, because it uses `:|` in a nondeterministic way) in a way that does not affect the result, or if the implementation temporarily makes use of some mutable data structures, or if the implementation is done with a loop. For example, here is the standard definition of the Fibonacci function but with an efficient implementation that uses a loop:

```
function Fib(n: nat): nat {
  if n < 2 then n else Fib(n - 2) + Fib(n - 1)
} by method {
  var x, y := 0, 1;
  for i := 0 to n
    invariant x == Fib(i) && y == Fib(i + 1)
  {
    x, y := y, x + y;
  }
  return x;
}
```

The `by method` clause is allowed only for the `function` or `predicate` declarations (without `method`, `twostate`, `least`, and `greatest`, but possibly with `static`). The method inherits the in-parameters, attributes, and `requires` and `decreases` clauses of the function. The method also gets one out-parameter, corresponding to the function's result value (and the name of it, if present). Finally, the method gets an empty `modifies` clause and a postcondition `ensures r == F(args)`, where `r` is the name of the out-parameter and `F(args)` is the function with its arguments. In other words, the method body must compute and return exactly what the function says, and must do so without modifying any previously existing heap state.

The function body of a function-by-method is allowed to be ghost, but the method body must be compilable. In non-ghost contexts, the compiler turns a call of the function-by-method into a call that leads to the method body.

Note, the method body of a function-by-method may contain `print` statements. This means that the run-time evaluation of an expression may have print effects. Dafny does not track print effects, but this is the only situation that an expression can have a print effect.

13.4.1. Functions

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 5.3 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 function's parameters, `T` is the type of the function's result, `Pre` is a boolean expression denoting the function's precondition, `Frame` denotes a set of objects whose fields the function body may depend on, `Post` is a boolean expression denoting the function's postcondition, `Rank` is the function's variant function, and `Body` is an expression that defines the function's 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, such as when the function is recursive. 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.

Within a postcondition, the result of the function is designated by a call of the function, such as `Factorial(n)` in the example above. Alternatively, a name

for the function result can be given in the signature, as in the following rewrite of the example above.

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

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`".

Like methods, functions can be either *instance* (which they are be default) or *static* (when the function declaration contains the keyword `static`). An instance function, but not a static function, has an implicit receiver parameter, `this`. A static function `F` in a class `C` can be invoked by `C.F(…)`. This provides a convenient way to declare a number of helper functions in a separate class.

As for methods, a `...` is used when declaring a function in a module refinement (cf. Section 21). 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 `...` which means to copy the signature from `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.

13.4.2. Predicates

A function that returns a `bool` result 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.

13.4.3. Function Transparency

A function is said to be *transparent* in a location if 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 whether the function was given the `{:opaque}` attribute (as explained in Section 22.1.13).

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

`{:opaque}`?	Transparent Inside Module	Transparent Outside Module
N	Y	Y
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.

13.4.4. Least/Greatest (CoInductive) Predicates and Lemmas

See Section 23.5.3 for descriptions of inductive predicates and lemmas.

14. Trait Types

```
TraitDecl =
  "trait" { Attribute } ClassName [ GenericParameters ]
  [ "extends" Type { "," Type } | ellipsis ]
  "{"
   { { DeclModifier } ClassMemberDecl(allowConstructors: true,
                                      isValueType: false,
                                      moduleLevelDecl: false,
                                      isWithinAbstractModule: false) }
  "}"
```

A *trait* is an abstract superclass, similar to an "interface" or "mixin".[4]

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

```
trait J
{
  _members_
}
```

where *members* can include fields, functions, methods and declarations of nested traits, 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 a variable of type `J`; a value of type `J` is assignable to a variable of a reference type `C` that extends `J` only if the verifier can prove that the reference does indeed refer to an object of allocated type `C`. 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. There is further discussion on this point in Section 14.2.

`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 implement a trait `J`, and a reference to such a `C` object can be used as a value of type `J`.

14.1. Type `object`

```
ObjectType_ = "object" | "object?"
```

There is a built-in trait `object` that is implicitly extended by all classes and traits. It produces two types: the type `object?` that is a supertype of all reference types and a subset type `object` that is a supertype of all non-null

[4] Traits are new to Dafny and are likely to evolve for a while.

reference types. This includes reference types like arrays and iterators that do not permit explicit extending of traits. 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>`.

It serves no purpose (but does no harm) to explicitly list the trait `object` as an extendee in a class or trait declaration.

Traits `object?` and `object` contain no members.

The dynamic allocation of objects is done using `new C…`, where `C` is the name of a class. The name `C` is not allowed to be a trait, except that it is allowed to be `object`. The construction `new object` allocates a new object (of an unspecified class type). The construction can be used to create unique references, where no other properties of those references are needed. (`new object?` makes no sense; always use `new object` instead because the result of `new` is always non-null.)

14.2. Inheritance

The purpose of traits is to be able to express abstraction: a trait encapsulates a set of behaviors; classes and traits that extend it *inherit* those behaviors, perhaps specializing them.

A trait or class may extend multiple other traits. The traits syntactically listed in a trait or class's `extends` clause are called its *direct parents*; the *transitive parents* of a trait or class are its direct parents, the transitive parents of its direct parents, and the `object` trait (if it is not itself `object`). These are sets of traits, in that it does not matter if there are repetitions of a given trait in a class or trait's direct or transitive parents. However, if a trait with type parameters is repeated, it must have the same actual type parameters in each instance. Furthermore, a trait may not be in its own set of transitive parents; that is, the graph of traits connected by the directed *extends* relationship may not have any cycles.

A class or trait inherits (as if they are copied) all the instance members of its transitive parents. However, since names may not be overloaded in Dafny, different members (that is, members with different type signatures) within the set of transitive parents and the class or trait itself must have different names.[5] This restriction does mean that traits from different sources that coincidentally use the same name for different purposes cannot be combined by being part of the set of transitive parents for some new trait or class.

A declaration of member `C.M` in a class or trait *overrides* any other declarations of the same name (and signature) in a transitive parent. `C.M` is then called an override; a declaration that does not override anything is called an *original declaration*.

[5] It is possible to conceive of a mechanism for disambiguating conflicting names, but this would add complexity to the language that does not appear to be needed, at least as yet.

Static members of a trait may not be redeclared; thus, if there is a body it must be declared in the trait; the compiler will require a body, though the verifier will not.

Where traits within an extension hierarchy do declare instance members with the same name (and thus the same signature), some rules apply. Recall that, for methods, every declaration includes a specification; if no specification is given explicitly, a default specification applies. Instance method declarations in traits, however, need not have a body, as a body can be declared in an override.

For a given non-static method M,

- A trait or class may not redeclare M if it has a transitive parent that declares M and provides a body.
- A trait may but need not provide a body if all its transitive parents that declare M do not declare a body.
- A trait or class may not have more than one transitive parent that declares M with a body.
- A class that has one or more transitive parents that declare M without a body and no transitive parent that declares M with a body must itself redeclare M with a body if it is compiled. (The verifier alone does not require a body.)
- Currently (and under debate), the following restriction applies: if `M` overrides two (or more) declarations, `P.M` and `Q.M`, then either `P.M` must override `Q.M` or `Q.M` must override `P.M`.

The last restriction above is the current implementation. It effectively limits inheritance of a method M to a single "chain" of declarations and does not permit mixins.

Each of any method declarations explicitly or implicitly includes a specification. In simple cases, those syntactially separate specifications will be copies of each other (up to renaming to take account of differing formal parameter names). However they need not be. The rule is that the specifications of M in a given class or trait must be *as strong as* M's specifications in a transitive parent. Here *as strong as* means that it must be permitted to call the subtype's M in the context of the supertype's M. Stated differently, where P and C are a parent trait and a child class or trait, respectively, then, under the precondition of `P.M`,

- C.M's `requires` clause must be implied by P.M's `requires` clause
- C.M's `ensures` clause must imply P.M's `ensures` clause
- C.M's `reads` set must be a subset of P.M's `reads` set
- C.M's `modifies` set must be a subset of P.M's `modifies` set
- C.M's `decreases` expression must be smaller than or equal to P.M's `decreases` expression

Non-static const and field declarations are also inherited from parent traits. These may not be redeclared in extending traits and classes. However, a trait need not initalize a const field with a value. The class that extends a trait

that declares such a const field without an initializer can initialize the field in a constructor. If the declaring trait does give an initial value in the declaration, the extending class or trait may not either redeclare the field or give it a value in a constructor.

When names are inherited from multiple traits, they must be different. If two traits declare a common name (even with the same signature), they cannot both be extendees of the same class or trait.

14.3. Example of traits

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 is used by all classes that extend `Shape`. Here are two classes that each extend `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());
```

15. Array Types

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

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

15.1. 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`. They can be initialized using an ordered list of expressions enclosed in square brackets, as follows:

```
a := new T[] [t1, t2, t3, t4];
```

The length of an array is retrieved using the immutable `Length` member. For example, the array allocated above satisfies:

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

Once an array is allocated, its length cannot be changed.

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, with 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[..]))
```

15.2. 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>`. (Accordingly, `array0` and `array1` are just normal identifiers, not type names.)

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 for 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.

16. Iterator types

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

See Section 5.5 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 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);
```

TODO: Add example of using the iterator

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 is recorded in the following immutable fields:

```
ghost var _reads: set<object>
ghost var _modifies: set<object>
ghost var _decreases0: T0
ghost var _decreases1: T1
// ...
```

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

```
ghost var _new: set<object>;
```

to which any objects allocated on behalf of the iterator body are 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 is regrettably tricky to use iterators. The language really ought to have a `foreach` statement to make this easier. Here is an example showing a 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]; // a lemma to help prove loop invariant
```

[6] It would make sense to rename the special fields `_reads` and `_modifies` to have the same names as the corresponding keywords, `reads` and `modifies`, as is done for function values. Also, the various `_decreases\(_i_\)` fields can be combined into one field named `decreases` whose type is a n-tuple. Thse changes may be incorporated into a future version of Dafny.

```
  }
}

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

TODO: The section above can use some rewriting, a summary of the defined members of an iterator, and more examples. Probably also a redesign.

17. Arrow types

```
ArrowType_ = ( DomainType_ "~>" Type
             | DomainType_ "-->" Type
             | DomainType_ "->" Type
             )
```

Functions are first-class values in Dafny. The types of function values are called *arrow types* (aka, *function types*). Arrow types have the form `(TT) ~> U` where `TT` is a (possibly empty) comma-delimited list of types and `U` is a type. `TT` 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, arr: array<bool>): real
  requires x < 1000
  reads arr
```

is `(int, array<bool>) ~> real`.

As seen in the example above, the functions that are values of a type `(TT) ~> U` can have a precondition (as indicated by the `requires` clause) and can read values in the heap (as indicated by the `reads` clause). As described in Section 11.3.3, the subset type `(TT) --> U` denotes partial (but heap-independent) functions and the subset type `(TT) -> U` denotes total functions.

A function declared without a `reads` clause is known by the type checker to be a partial function. For example, the type of

```
function F(x: int, b: bool): real
  requires x < 1000
```

is `(int, bool) --> real`. Similarly, a function declared with neither a `reads` clause nor a `requires` clause is known by the type checker to be a total function. For example, the type of

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

is `(int, bool) -> real`. In addition to functions declared by name, Dafny also supports anonymous functions by means of *lambda expressions* (see Section 20.13).

To simplify the appearance of the basic case where a function's domain consists of a list of exactly one non-function, non-tuple type, the parentheses around the domain type can be dropped in this case. For example, you may write just `T -> U` for a total arrow type. 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 arrows are right associative. For example, `A -> B -> C` means `A -> (B -> C)`, whereas the other association requires explicit parentheses: `(A -> B) -> C`. As another example, `A -> B --> C ~> D` means `A -> (B --> (C ~> D))`.

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
```

The arrow types themselves do not divide its parameters into ghost versus non-ghost. Instead, a function used as a first-class value is considered to be ghost if either the function or any of its arguments is ghost. The following example program illustrates:

```
function method F(x: int, ghost y: int): int
{
  x
}

method Example() {
  ghost var f: (int, int) -> int;
  var g: (int, int) -> int;
  var h: (int) -> int;
  var x: int;
  f := F;
  x := F(20, 30);
  g := F; // error: tries to assign ghost to non-ghost
  h := F; // error: wrong arity (and also tries to assign ghost to non-ghost)
}
```

In addition to 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
```

In these examples, if `f` instead had type `T --> U` or `T -> U`, then the type of `f.reads` is `T -> set<object?>` and the type of `f.requires` is `T -> bool`.

Dafny also supports anonymous functions by means of *lambda expressions*. See Section 20.13.

17.1. Tuple types

```
TupleType = "(" [ [ "ghost" ] Type { "," [ "ghost" ] Type } ] ")"
```

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

```
datatype Pair<T,U> = Pair(0: T, 1: U)
```

Dafny provides the type `(T, U)` and the constructor `(t, u)`, as if the datatype's name were "" (i.e., an empty string) and its type arguments are given in round parentheses, and as if the constructor name were the empty string. 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 more. 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*.

The `ghost` modifier can be used to mark tuple components as being used for specification only:

```
var pair: (int, ghost int) := (1, ghost 2);
```

18. Algebraic Datatypes

```
DatatypeDecl =
  ( "datatype" | "codatatype" )
  { Attribute }
  DatatypeName [ GenericParameters ]
  "=" [ ellipsis ]
      [ "|" ] DatatypeMemberDecl
      { "|" DatatypeMemberDecl }
      [ TypeMembers ]

DatatypeMemberDecl =
  { Attribute } DatatypeMemberName [ FormalsOptionalIds ]
```

Dafny offers two kinds of algebraic datatypes, those defined inductively (with `datatype`) and those defined co-inductively (with `codatatype`). 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.

18.1. Inductive datatypes

The values of inductive datatypes can be seen as finite trees where the leaves are values of basic types, numeric types, reference types, co-inductive datatypes, or arrow 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 may 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?`.

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)
```

18.2. Co-inductive datatypes

TODO: This section and particularly the subsections need rewriting using the least and greatest terminology, and to make the text fit better into the overall reference manual.

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 the language 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(Leino and Moskal 2014a), 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 are omitted here.

18.3. Co-induction

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 is 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: 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<T>)

// 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);
}
```

The example 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 above— 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 restrictions on how a *co-inductive* hypothesis can be *used*. These are, of course, taken into consideration by Dafny's 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 18.3.5.2) 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 today'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.

18.3.1. 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; the clusters are 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 is taken through the program verifier's reasoning engine. Semantically, each function activation is labeled by a *rank*—a lexicographic tuple determined by evaluating the function'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 "termination checks". 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* (key-

word `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 techniques—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).
```

18.3.2. 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` auto-

matically 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-datatypes—since a codatatype admits infinite values, the type is nevertheless inhabited.

18.3.3. Creating Values of Co-datatypes

To define values of co-datatypes, one could imagine a "co-function" language feature: the body of a "co-function" 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 "co-functions"), 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 cannot be multiple fix-points, the language allows one function to be involved in both recursive and co-recursive calls, as we illustrate by the function `FivesUp`.

18.3.4. Copredicates

Determining properties of co-datatype values may require an infinite number of observations. To that end, 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* positions—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* positions—that is, in negation normal form, when they appear under existential quantification, the quantification needs to be limited to a finite range[7]. 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 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

[7] 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.

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#`.

18.3.4.1. 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`.

18.3.5. 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.

18.3.5.1. 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 • P(x) <==> ? k • 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 • Pos#[k](Up(n))`. Finally, the axiom `D(Pos)` is used (automatically) to establish the co-predicate.

18.3.5.2. 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.

18.3.5.3. 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-lemma'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 body'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.[8]

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 colemma's declared postcondition (on account of the `D` axiom, and remembering that only the positive co-friendly occurrences of co-predicates in the co-lemma'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.

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.

[8]Note, two places where co-predicates and co-lemmas are not analogous are (a) co-predicates must not make recursive calls to their prefix predicates and (b) co-predicates cannot mention `_k`.

19. Statements

```
Stmt =
  ( AssertStmt | AssumeStmt | BlockStmt | BreakStmt
  | CalcStmt | ExpectStmt | ForallStmt | IfStmt
  | LabeledStmt | MatchStmt | ModifyStmt
  | PrintStmt | ReturnStmt | RevealStmt | SkeletonStmt
  | UpdateStmt | UpdateFailureStmt
  | VarDeclStatement | WhileStmt | ForLoopStmt | YieldStmt
  )
```

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.

19.1. Labeled Statement

```
LabeledStmt = "label" LabelName ":" Stmt
```

A labeled statement is just the keyword `label` followed by an identifier which is the label, followed by a colon and a statement. The label may be referenced in a break statement that is within the labeled statement to transfer control to the location after the labeled statement. The label is not allowed to be the same as any previous dominating label.

The label may also be used in an `old` expression (Section 20.24). In this case the label must have been encountered during the control flow in route to the `old` expression. That is, again, the label must dominate the use of the label.

19.2. Break Statement

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

A break statement provides a means to transfer control in a way different than the usual nested control structures. There are two forms of break statement: with and without a label.

If a label is used, the break statement must be enclosed in a statement with that label and the result is to transfer control to the statement after the labeled statement. For example, such a break statement can be used to exit a sequence of statements in a block statement before reaching the end of the block.

For example,

```
L: {
  var n := ReadNext();
  if n < 0  { break L; }
```

```
  DoSomething(n);
}
```

is equivalent to

```
{
  var n: ReadNext();
  if 0 <= n {
    DoSomething(n);
  }
}
```

If no label is specified and the statement lists `n` occurrences of `break`, then the statement must be enclosed in at least `n` levels of loops. Control continues after exiting `n` enclosing loops. For example,

```
var i := 0;
while i < 10 {
  var j := 0;
  while j < 10 {
    var k := 0;
    while k < 10 {
      if (j + k == 15) break break;
      k := k + 1;
    }
    j := j + 1;
  }
  // control continues here after the break, exiting two loops
  i := i + 1;
}
```

19.3. Block Statement

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

A block statement is just a sequence of statements enclosed by curly braces. Local variables declared in the block end their scope at the end of the block.

19.4. 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 out-parameters sometime before a return statement. In fact, the out-parameters 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 values of the out-parameters are used), or they can take a list of expressions to return. If a list is given, the number of expressions given must be the same as the number of named out-parameters. These expressions are evaluated, then they are assigned to the out-parameters, and then the method terminates.

19.5. Yield Statement

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

A yield statement can only be used in an iterator. See Section 16 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 (out-)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 values of the yield parameters are used), or they can take a list of expressions to yield. If a list is given, the number of expressions given must be the same as the number of named iterator out-parameters. These expressions are then evaluated, then they are assigned to the yield parameters, and then the iterator yields.

19.6. Update and Call Statements

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

If more than one left-hand side is used, these must denote different l-values, unless the corresponding right-hand sides also denote the same value.

The update statement serves several logical purposes.

1) The form

```
Lhs {Attribute} ";"
```

is assumed to be a call to a method with no out-parameters.

2) The form

```
  Lhs { , Lhs } ":=" Rhs ";"
```

can occur in the `UpdateStmt` grammar when there is a single Rhs that takes the special form of a `Lhs` that is a call. This is the only case where the number of left-hand sides can be different than the number of right-hand sides in the `UpdateStmt`. In that case the number of left-hand sides must match the number of out-parameters of the method that is called or there must be just one `Lhs` to the left of the `:=`, which then is assigned a tuple of the out-parameters. Note that the result of a method call is not allowed to be used as an argument of another method call, as if it were an expression.

3) This is the typical parallel-assignment form, in which no call is involved:

```
  Lhs { , Lhs } ":=" Rhs { "," Rhs } ";"
```

Thise `UpdateStmt` is a parallel assignment of right-hand-side values to the left-hand sides. For example, `x,y := y,x` swaps the values of `x` and `y`. If more than one left-hand side is used, these must denote different l-values, unless the corresponding right-hand sides also denote the same value. There must be an equal number of left-hand sides and right-hand sides in this case. Of course, the most common case will have only one `Rhs` and one `Lhs`.

4) The form

```
Lhs { "," Lhs } :| [ "assume" ] Expression<false,false>
```

using "`:|`" 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. The given boolean expression need not constrain the LHS values uniquely.

```
method Sum(X: set<int>) returns (s: int)
{
  s := 0; var Y := X;
  while Y != {}
    decreases Y
  {
    var y: int;
    y :| y in Y;
    s, Y := s + y, Y - {y};
```

```
  }
}
```

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

In addition, as the choice is arbitrary, assignment statements using `:|` may be non-deterministic when executed.

Note that the form

```
    Lhs ":"
```

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

19.7. Update with Failure Statement (:-)

```
UpdateFailureStmt  =
    [ Lhs { "," Lhs } ]
    ":-"
    [ "expect"  | "assert" | "assume" ]
    Expression(allowLemma: false, allowLambda: false)
    { "," Rhs }
    ";"
```

A `:-` statement is an alternate form of the `:=` statement that allows for abrupt return if a failure is detected. This is a language feature somewhat analogous to exceptions in other languages.

An update-with-failure statement uses *failure-compatible* types. A failure-compatible type is a type that has the following members (each with no in-parameters and one out-parameter):

- a function method `IsFailure()` that returns a `bool`
- an optional function method `PropagateFailure()` that returns a value assignable to the first out-parameter of the caller
- an optional method or function `Extract()`

A failure-compatible type with an `Extract` member is called *value-carrying*.

To use this form of update,

- if the RHS of the update-with-failure statement is a method call, the first out-parameter of the callee must be failure-compatible
- if instead the RHS of the update-with-failure statement is one or more expressions, the first of these expressions must be a value with a failure-compatible type
- the caller must have a first out-parameter whose type matches the output of `PropagateFailure` applied to the first output of the callee, unless an `expect`, `assume`, or `assert` keyword is used after `:-` (cf. Section 19.7.7).

- if the failure-compatible type of the RHS does not have an `Extract` member, then the LHS of the `:-` statement has one less expression than the RHS (or than the number of out-parameters from the method call)
- if the failure-compatible type of the RHS does have an `Extract` member, then the LHS of the `:-` statement has the same number of expressions as the RHS (or as the number of out-parameters from the method call) and the type of the first LHS expression must be assignable from the return type of the `Extract` member
- the `IsFailure` and `PropagateFailure` methods may not be ghost
- the LHS expression assigned the output of the `Extract` member is ghost precisely if `Extract` is ghost

The following subsections show various uses and alternatives.

19.7.1. Failure compatible types

A simple failure-compatible type is the following:

```
datatype Status =
| Success
| Failure(error: string)
{
  predicate method IsFailure() { this.Failure?  }
  function method PropagateFailure(): Status
    requires IsFailure()
  {
    Failure(this.error)
  }
}
```

A commonly used alternative that carries some value information is something like this generic type:

```
datatype Outcome<T> =
| Success(value: T)
| Failure(error: string)
{
  predicate method IsFailure() {
    this.Failure?
  }
  function method PropagateFailure<U>(): Outcome<U>
    requires IsFailure()
  {
    Failure(this.error) // this is Outcome<U>.Failure(...)
  }
  function method Extract(): T
    requires !IsFailure()
  {
```

```
    this.value
  }
}
```

19.7.2. Simple status return with no other outputs

The simplest use of this failure-return style of programming is to have a method call that just returns a non-value-carrying `Status` value:

```
method Callee(i: int) returns (r: Status)
{
  if i < 0 { return Failure("negative"); }
  return Success;
}

method Caller(i: int) returns (rr: Status)
{
  :- Callee(i);
  ...
}
```

Note that there is no LHS to the `:-` statement. If `Callee` returns `Failure`, then the caller immediately returns, not executing any statements following the call of `Callee`. The value returned by `Caller` (the value of `rr` in the code above) is the result of `PropagateFailure` applied to the value returned by `Callee`, which is often just the same value. If `Callee` does not return `Failure` (that is, returns a value for which `IsFailure()` is `false`) then that return value is forgotten and execution proceeds normally with the statements following the call of `Callee` in the body of `Caller`.

The desugaring of the `:- Callee(i);` statement is

```
var tmp;
tmp := Callee(i);
if tmp.IsFailure() {
  rr := tmp.PropagateFailure();
  return;
}
```

In this and subsequent examples of desugaring, the `tmp` variable is a new, unique variable, unused elsewhere in the calling member.

19.7.3. Status return with additional outputs

The example in the previous subsection affects the program only through side effects or the status return itself. It may well be convenient to have additional out-parameters, as is allowed for `:=` updates; these out-parameters behave just as for `:=`. Here is an example:

```
method Callee(i: int) returns (r: Status, v: int, w: int)
{
  if i < 0 { return Failure("negative"), 0, 0; }
  return Success, i+i, i*i;
}

method Caller(i: int) returns (rr: Status, k: int)
{
  var j: int;
  j, k :- Callee(i);
  k := k + k;
  ...
}
```

Here `Callee` has two outputs in addition to the `Status` output. The LHS of the `:-` statement accordingly has two l-values to receive those outputs. The recipients of those outputs may be any sort of l-values; here they are a local variable and an out-parameter of the caller. Those outputs are assigned in the `:-` call regardless of the `Status` value:

- If `Callee` returns a failure value as its first output, then the other outputs are assigned, the *caller's* first out-parameter (here `rr`) is assigned the value of `PropagateFailure`, and the caller returns.
- If `Callee` returns a non-failure value as its first output, then the other outputs are assigned and the caller continues execution as normal.

The desugaring of the `j, k :- Callee(i);` statement is

```
var tmp;
tmp, j, k := Callee(i);
if tmp.IsFailure() {
  rr := tmp.PropagateFailure();
  return;
}
```

19.7.4. Failure-returns with additional data

The failure-compatible return value can carry additional data as shown in the `Outcome<T>` example above. In this case there is a (first) LHS l-value to receive this additional data.

```
method Callee(i: int) returns (r: Outcome<nat>, v: int)
{
  if i < 0 { return Failure("negative"), i+i; }
  return Success(i), i+i;
}

method Caller(i: int) returns (rr: Outcome<int>, k: int)
```

```
{
  var j: int;
  j, k :- Callee(i);
  k := k + k;
  ...
}
```

Suppose `Caller` is called with an argument of `10`. Then `Callee` is called with argument `10` and returns `r` and `v` of `Outcome<nat>.Success(10)` and `20`. Here `r.IsFailure()` is `false`, so control proceeds normally. The `j` is assigned the result of `r.Extract()`, which will be `10`, and `k` is assigned `20`. Control flow proceeds to the next line, where `k` now gets the value `40`.

Suppose instead that `Caller` is called with an argument of `-1`. Then `Callee` is called with the value `-1` and returns `r` and `v` with values `Outcome<nat>.Failure("negative")` and `-2`. `k` is assigned the value of `v` (-2). But `r.IsFailure()` is `true`, so control proceeds directly to return from `Caller`. The first out-parameter of `Caller` (`rr`) gets the value of `r.PropagateFailure()`, which is `Outcome<int>.Failure("negative")`; `k` already has the value `-2`. The rest of the body of `Caller` is skipped. In this example, the first out-parameter of `Caller` has a failure-compatible type so the exceptional return will propagate up the call stack. It will keep propagating up the call stack as long as there are callers with this first special output type and calls that use `:-` and the return value keeps having `IsFailure()` true.

The desugaring of the `j, k :- Callee(i);` statement in this example is

```
var tmp;
tmp, k := Callee(i);
if tmp.IsFailure() {
  rr := tmp.PropagateFailure();
  return;
}
j := tmp.Extract();
```

19.7.5. RHS with expression list

Instead of a failure-returning method call on the RHS of the statement, the RHS can instead be a list of expressions. As for a `:=` statement, in this form, the expressions on the left and right sides of `:-` must correspond, just omitting a LHS l-value for the first RHS expression if its type is not value-carrying. The semantics is very similar to that in the previous subsection.

- The first RHS expression must have a failure-compatible type.
- All the assignments of RHS expressions to LHS values except for the first RHS value are made.
- If the first RHS value (say `r`) responds `true` to `r.IsFailure()`, then `r.PropagateFailure()` is assigned to the first out-parameter of the *caller*

and the execution of the caller's body is ended.
- If the first RHS value (say `r`) responds `false` to `r.IsFailure()`, then
  - if the type of `r` is value-carrying, then `r.Extract()` is assigned to the first LHS value of the `:-` statement (if `r` is not value-carrying, then the corresponding LHS l-value is omitted)
  - execution of the caller's body continues with the statement following the `:-` statement.

A RHS with a method call cannot be mixed with a RHS containing multiple expressions.

For example, the desugaring of

```
method m(Status r) returns (rr: Status) {
  var k;
  k :- r, 7;
  ...
}
```

is

```
var k;
var tmp;
tmp, k := r, 7;
if tmp.IsFailure() {
  rr := tmp.PropagateFailure();
  return;
}
```

19.7.6. Failure with initialized declaration.

The `:-` syntax can also be used in initalization, as in

```
var s :- M();
```

This is equivalent to

```
var s;
s :- M();
```

with the semantics as described above.

19.7.7. Keyword alternative

In any of the above described uses of `:-`, the `:-` token may be followed immediately by the keyword `expect`, `assert` or `assume`.

- `assert` means that the RHS evaluation is expected to be successful, but that the verifier should prove that this is so; that is, the verifier should prove `assert !r.IsFailure()` (where `r` is the status return from the callee) (cf. Section 19.16)

- `assume` means that the RHS evaluation should be assumed to be successful, as if the statement `assume !r.IsFailure()` followed the evaluation of the RHS (cf. Section 19.17)
- `expect` means that the RHS evaluation should be assumed to be successful (like using `assume` above), but that the compiler should include a run-time check for success. This is equivalent to including `expect !r.IsFailure()` after the RHS evaluation; that is, if the status return is a failure, the program halts. (cf. Section 19.18)

In each of these cases, there is no abrupt return from the caller. Thus there is no evaluation of `PropagateFailure`. Consequently the first out-parameter of the caller need not match the return type of `PropagateFailure`; indeed, the failure-compatible type returned by the callee need not have a `PropagateFailure` member.

The equivalent desugaring replaces

```
if tmp.IsFailure() {
  rr := tmp.PropagateFailure();
  return;
}
```

with

```
expect !tmp.IsFailure(), tmp;
```

or

```
assert !tmp.IsFailure();
```

or

```
assume !tmp.IsFailure();
```

There is a grammatical nuance that the user should be aware of. The keywords `assert`, `assume`, and `expect` can start an expression. For example, `assert P; E` can be an expression. However, in `e :- assert P; E;` the `assert` is parsed as the keyword associated with `:-`. To have the `assert` considered part of the expression use parentheses: `e :- (assert P; E);`.

19.7.8. Key points

There are several points to note.

- The first out-parameter of the callee is special. It has a special type and that type indicates that the value is inspected to see if an abrupt return from the caller is warranted. This type is often a datatype, as shown in the examples above, but it may be any type with the appropriate members.
- The restriction on the type of caller's first out-parameter is just that it must be possible (perhaps through generic instantiation and type infer-

ence, as in these examples) for `PropagateFailure` applied to the failure-compatible output from the callee to produce a value of the caller's first out-parameter type. If the caller's first out-parameter type is failure-compatible (which it need not be), then failures can be propagated up the call chain. If the keyword form of the statement is used, then no `PropagateFailure` member is needed and there is no restriction on the caller's first out-parameter.
- In the statement `j, k :- Callee(i);`, when the callee's return value has an `Extract` member, the type of `j` is not the type of the first out-parameter of `Callee`. Rather it is a type assignable from the output type of `Extract` applied to the first out-value of `Callee`.
- A method like `Callee` with a special first out-parameter type can still be used in the normal way: `r, k := Callee(i)`. Now `r` gets the first output value from `Callee`, of type `Status` or `Outcome<nat>` in the examples above. No special semantics or exceptional control paths apply. Subsequent code can do its own testing of the value of `r` and whatever other computations or control flow are desired.
- The caller and callee can have any (positive) number of output arguments, as long as the callee's first out-parameter has a failure-compatible type and the caller's first out-parameter type matches `PropagateFailure`.
- If there is more than one LHS, the LHSs must denote different l-values, unless the RHS is a list of expressions and the corresponding RHS values are equal.
- The LHS l-values are evaluated before the RHS method call, in case the method call has side-effects or return values that modify the l-values prior to assignments being made.

It is important to note the connection between the failure-compatible types used in the caller and callee, if they both use them. They do not have to be the same type, but they must be closely related, as it must be possible for the callee's `PropagateFailure` to return a value of the caller's failure-compatible type. In practice this means that one such failure-compatible type should be used for an entire program. If a Dafny program uses a library shared by multiple programs, the library should supply such a type and it should be used by all the client programs (and, effectively, all Dafny libraries). It is also the case that it is inconvenient to mix types such as `Outcome` and `Status` above within the same program. If there is a mix of failure-compatible types, then the program will need to use `:=` statements and code for explicit handling of failure values.

19.7.9. Failure returns and exceptions

The `:-` mechanism is like the exceptions used in other programming languages, with some similarities and differences.

- There is essentially just one kind of 'exception' in Dafny, the variations of the failure-compatible data type.
- Exceptions are passed up the call stack whether or not intervening meth-

ods are aware of the possibility of an exception, that is, whether or not the intervening methods have declared that they throw exceptions. Not so in Dafny: a failure is passed up the call stack only if each caller has a failure-compatible first out-parameter, is itself called in a `:-` statement, and returns a value that responds true to `IsFailure()`.

- All methods that contain failure-return callees must explicitly handle those failures using either `:-` statements or using `:=` statements with a LHS to receive the failure value.

19.8. Variable Declaration Statement

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

CasePatternLocal = ( [ Ident ] "(" CasePatternLocsl { "," CasePatternLocal } ")"
                   | LocalIdentTypeOptional
                   )
```

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 its type can be inferred, either from a given initial value, or from other uses of the variable. 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 if it cannot be inferred from uses of x.

What follows the `LocalIdentTypeOptional` optionally combines the variable declarations with an update statement (cf. Section 19.6). If the RHS is a call, then any variable receiving the value of a formal ghost out-parameter will automatically be declared as ghost, even if the `ghost` keyword is not part of the variable declaration statement.

The left-hand side can also contain a tuple of patterns that 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
}
```

The assignment with failure operator `:-` returns from the method if the value evaluates to a failure value of a failure-compatible type, see Section 19.7.

19.9. 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.

19.10. Binding Guards

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

`IfStmt`s can also take a `BindingGuard`. It checks if there exist values for the given variables that satisfy the given expression. If so, it binds some satisfying values to the variables and proceeds into the "then" branch; otherwise it proceeds with the "else" branch, where the bound variables are not in scope.

In other words, the statement

```
if x :| P { S } else { T }
```

has the same meaning as

```
if exists x :| P { var x :| P; S } else { T }
```

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 exists x :: P(x)
    ensures P(y)
{
  if x : int :| P(x) {
      y := x;
  }
}
```

19.11. If Statement

```
IfStmt = "if"
  ( AlternativeBlock(allowBindingGuards: true)
  |
    ( BindingGuard(allowLambda: true)
    | Guard
    | ellipsis
```

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

AlternativeBlock(allowBindingGuards) =
  ( { AlternativeBlockCase(allowBindingGuards) }
  | "{" { AlternativeBlockCase(allowBindingGuards) } "}"
  )

AlternativeBlockCase(allowBindingGuards) =
      { "case"
      (
        BindingGuard(allowLambda: false) // permitted iff allowBindingGuards == true
      | Expression(allowLemma: true, allowLambda: false)
      ) "=>" { Stmt } } .
```

The simplest form of an `if` statement uses a guard that is a boolean expression. For example,

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

Unlike `match` statements, `if` statements do not have to be exhaustive: omitting the `else` block is the same as including an empty `else` block. To ensure that an `if` statement is exhaustive, use the `if-case` statement documented below.

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

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

The `if-case` statement using the `AlternativeBlock` 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 := x;
  }
```

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` and the statements to the right of `=>` for that guard

are executed. The statement requires at least one of the guards to evaluate to `true` (that is, `if-case` statements must be exhaustive: the guards must cover all cases).

TODO: Describe the … refinement

19.12. While Statement

```
WhileStmt =
  "while"
  ( LoopSpec
    AlternativeBlock(allowBindingGuards: false)
  | ( Guard | ellipsis )
    LoopSpec
    ( BlockStmt
    | ellipsis
    | /* go body-less */
    )
  )
```

Loops need *loop specifications* (`LoopSpec` in the grammar) in order for Dafny to prove that they obey expected behavior. In some cases Dafny can infer the loop specifications by analyzing the code, so the loop specifications need not always be explicit. These specifications are described in Section 19.14.

The general loop statement in Dafny is the familiar `while` 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

The second form uses the `AlternativeBlock`. 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 and the while statement is repeated. If none of the guards evaluates to true, then the loop execution is terminated.

TODO: Describe … refinement

19.13. For Loops

```
ForLoopStmt =
  "for" IdentTypeOptional ":="
    Expression(allowLemma: false, allowLambda: false)
    ( "to" | "downto" )
    ( Expression(allowLemma: false, allowLambda: false)
    | "*"
    )
    LoopSpec
    ( BlockStmt
    | /* go body-less */
    )
  )
```

The `for` statement provides a convenient way to write some common loops.

The statement introduces a local variable `IdentTypeOptional`, which is called the *loop index*. The loop index is in scope in the `LoopSpec` and `BlockStmt`, but not after the `for` loop. Assignments to the loop index are not allowed. The type of the loop index can typically be inferred, so it need not be given explicitly. If the identifier is not used, it can be written as `_`, as illustrated in this repeat-20-times loop:

```
for _ := 0 to 20 {
  Body
}
```

There are four basic variations of the `for` loop:

```
for i: T := lo to hi
  LoopSpec
{ Body }

for i: T := hi downto lo
```

```
  LoopSpec
{ Body }

for i: T := lo to *
  LoopSpec
{ Body }

for i: T := hi downto *
  LoopSpec
{ Body }
```

Semantically, they are defined as the following respective `while` loops:

```
{
  var _lo, _hi := lo, hi;
  assert _lo <= _hi && forall _i: int :: _lo <= _i <= _hi ==> _i is T;
  var i := _lo;
  while i != _hi
    invariant _lo <= i <= _hi
    LoopSpec
    decreases _hi - i
  {
    Body
    i := i + 1;
  }
}

{
  var _lo, _hi := lo, hi;
  assert _lo <= _hi && forall _i: int :: _lo <= _i <= _hi ==> _i is T;
  var i := _hi;
  while i != lo
    invariant _lo <= i <= _hi
    LoopSpec
    decreases i - _lo
  {
    i := i - 1;
    Body
  }
}

{
  var _lo := lo;
  assert forall _i: int :: _lo <= _i ==> _i is T;
  var i := _lo;
  while true
```

```
    invariant _lo <= i
    LoopSpec
  {
    Body
    i := i + 1;
  }
}

{
  var _hi := hi;
  assert forall _i: int :: _i <= _hi ==> _i is T;
  var i := _hi;
  while true
    invariant i <= _hi
    LoopSpec
  {
    i := i - 1;
    Body
  }
}
```

Note that expressions `lo` and `hi` are evaluated just once, before the loop iterations start.

Also, note in all variations that the values of `i` in the body are the values from `lo` to, *but not including*, `hi`. This makes it convenient to write common loops, including these:

```
for i := 0 to a.Length {
  Process(a[i]);
}
for i := a.Length downto 0 {
  Process(a[i]);
}
```

Nevertheless, `hi` must be a legal value for the type of the index variable, since that is how the index variable is used in the invariant.

If the end-expression is not `*`, then no explicit `decreases` is allowed, since such a loop is already known to terminate. If the end-expression is `*`, then the absence of an explicit `decreases` clause makes it default to `decreases *`. So, if the end-expression is `*` and no explicit `decreases` clause is given, the loop is allowed only in methods that are declared with `decreases *`.

The directions `to` or `downto` are contextual keywords. That is, these two words are part of the syntax of the `for` loop, but they are not reserved keywords elsewhere.

19.14. 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. For additional tutorial information see (Koenig and Leino 2012) or the online Dafny tutorial.

19.14.1. Loop invariants

Loops present a problem for specification-based reasoning. There is no way to know in advance how many times the code will go around the loop and a tool cannot reason about every one of a possibly unbounded sequence of unrollings. In order to consider all paths through a program, specification-based program verification tools require loop invariants, which are another kind of annotation.

A loop invariant is an expression that holds just prior to the loop test, that is, 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 (just prior to the loop test), 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 that any but the most basic properties of a loop are preserved unless it is told via an invariant.

19.14.2. 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. 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.

That is, the expression must strictly decrease in a well-founded ordering (cf. Section 23.7).

Many times, an integral value (natural or plain integer) is the quantity that decreases, but other values can be used as well. In the case of integers, the bound is assumed to be zero. For each loop iteration the `decreases` expression at the end of the loop body must be strictly smaller than the value at the beginning of the loop body (after the loop test). For integers, the well-founded relation between `x` and `X` is `x < X && 0 <= X`. Thus if the `decreases` value (`X`) is negative at the loop test, it must exit the loop, since there is no permitted value for `x` to have at the end of the loop body.

For example, the following is a proper use of `decreases` on a loop:

```
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` becomes smaller each loop iteration, and is bounded below by zero. When `i` becomes 0, the lower bound of the well-founded order, control flow exits the loop.

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 - 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 specifies `*`, then no termination check will be performed. Use of this feature is sound only with respect to partial correctness.

19.14.3. Loop framing

The specification of a loop also includes *framing*, which says what the loop modifies. The loop frame includes both local variables and locations in the heap.

For local variables, the Dafny verifier performs a syntactic scan of the loop body to find every local variable or out-parameter that occurs as a left-hand side of an assignment. These variables are called *syntactic assignment targets of the loop*, or *syntactic loop targets* for short. Any local variable or out-parameter that is not a syntactic assignment target is known by the verifier to remain unchanged by the loop.

The heap may or may not be a syntactic loop target. It is when the loop body syntactically contains a statement that can modify a heap location. This includes calls to compiled methods, even if such a method has an empty `modifies` clause, since a compiled method is always allowed to allocate new objects and change their values in the heap.

If the heap is not a syntactic loop target, then the verifier knows the heap remains unchanged by the loop. If the heap *is* a syntactic loop target, then the loop's effective `modifies` clause determines what is allowed to be modified by iterations of the loop body.

A loop can use `modifies` clauses to declare the effective `modifies` clause of the loop. If a loop does not explicitly declare any `modifies` clause, then the effective `modifies` clause of the loop is the effective `modifies` clause of the most tightly enclosing loop or, if there is no enclosing loop, the `modifies` clause of the enclosing method.

In most cases, there is no need to give an explicit `modifies` clause for a loop. The one case where it is sometimes needed is if a loop modifies less than is allowed by the enclosing method. Here are two simple methods that illustrate this case:

```
class Cell {
  var data: int
}

method M0(c: Cell, d: Cell)
  requires c != d
  modifies c, d
  ensures c.data == d.data == 100
{
  c.data, d.data := 100, 0;
  var i := 0;
  while i < 100
    invariant d.data == i
    // Needs "invariant c.data == 100" or "modifies d" to verify
  {
    d.data := d.data + 1;
    i := i + 1;
  }
}

method M1(c: Cell)
  modifies c
  ensures c.data == 100
{
  c.data := 100;
  var i := 0;
  while i < 100
    // Needs "invariant c.data == 100" or "modifies {}" to verify
  {
    var tmp := new Cell;
    tmp.data := i;
    i := i + 1;
  }
}
```

In `M0`, the effective `modifies` clause of the loop is `modifies c, d`. Therefore, the method's postcondition `c.data == 100` is not provable. To remedy the situation, the loop needs to be declared either with `invariant c.data == 100` or with `modifies d`.

Similarly, the effective `modifies` clause of the loop in `M1` is `modifies c`. Therefore, the method's postcondition `c.data == 100` is not provable. To remedy the situation, the loop needs to be declared either with `invariant c.data == 100` or with `modifies {}`.

When a loop has an explicit `modifies` clause, there is, at the top of every

iteration, a proof obligation that

- the expressions given in the `modifies` clause are well-formed, and
- everything indicated in the loop `modifies` clause is allowed to be modified by the (effective `modifies` clause of the) enclosing loop or method.

19.14.4. Body-less methods, functions, loops, and aggregate statements

Methods (including lemmas), functions, loops, and `forall` statements are ordinarily declared with a body, that is, a curly-braces pair that contains (for methods, loops, and `forall`) a list of statements or (for a function) an expression. In each case, Dafny syntactically allows these constructs to be given without a body. This is to allow programmers to temporarily postpone the development of the implementation of the method, function, loop, or aggregate statement.

If a method has no body, there is no difference for callers of the method. Callers still reason about the call in terms of the method's specification. But without a body, the verifier has no method implementation to check against the specification, so the verifier is silently happy. The compiler, on the other hand, will complain if it encounters a body-less method, because the compiler is supposed to generate code for the method, but it isn't clever enough to do that by itself without a given method body. If the method implementation is provided by code written outside of Dafny, the method can be marked with an `{:extern}` annotation, in which case the compiler will no longer complain about the absence of a method body.

A lemma is a special kind of method. Callers are therefore unaffected by the absence of a body, and the verifier is silently happy with not having a proof to check against the lemma specification. Despite a lemma being ghost, it is still the compiler that checks for, and complains about, body-less lemmas. A body-less lemma is an unproven lemma, which is often known as an *axiom*. If you intend to use a lemma as an axiom, omit its body and add the attribute `{:axiom}`, which causes the compiler to suppress its complaint about the lack of a body.

Similarly, calls to a body-less function use only the specification of the function. The verifier is silently happy, but the compiler complains (whether or not the function is ghost). As for methods and lemmas, the `{:extern}` and `{:axiom}` attributes can be used to suppress the compiler's complaint.

By supplying a body for a method or function, the verifier will in effect show the feasibility of the specification of the method or function. By supplying an `{:extern}` or `{:axiom}` attribute, you are taking that responsibility into your own hands. Common mistakes include forgetting to provide an appropriate `modifies` or `reads` clause in the specification, or forgetting that the results of functions in Dafny (unlike in most other languages) must be deterministic.

Just like methods and functions have two sides, callers and implementations, loops also have two sides. One side (analogous to callers) is the context that uses the loop. That context treats the loop in the same way regardless of whether or not the loop has a body. The other side is the loop body, that is, the implementation of each loop iteration. The verifier checks that the loop body maintains the loop invariant and that the iterations will eventually terminate, but if there is no loop body, the verifier is silently happy. This allows you to temporarily postpone the authoring of the loop body until after you've made sure that the loop specification is what you need in the context of the loop.

There is one thing that works differently for body-less loops than for loops with bodies. It is the computation of syntactic loop targets, which become part of the loop frame (see Section 19.14.3). For a body-less loop, the local variables computed as part of the loop frame are the mutable variables that occur free in the loop specification. The heap is considered a part of the loop frame if it is used for mutable fields in the loop specification or if the loop has an explicit `modifies` clause. The IDE will display the computed loop frame in hover text.

For example, consider

```
class Cell {
  var data: int
  const K: int
}

method BodylessLoop(n: nat, c: Cell)
  requires c.K == 8
  modifies c
{
  c.data := 5;
  var a, b := n, n;
  for i := 0 to n
    invariant c.K < 10
    invariant a <= n
    invariant c.data < 10
  assert a == n;
  assert b == n;
  assert c.data == 5;
}
```

The loop specification mentions local variable `a`, and thus `a` is considered part of the loop frame. Since what the loop invariant says about `a` is not strong enough to prove the assertion `a == n` that follows the loop, the verifier complains about that assertion.

Local variable `b` is not mentioned in the loop specification, and thus `b` is not included in the loop frame. Since in-parameter `n` is immutable, it is not included in the loop frame, either, despite being mentioned in the loop specification. For

these reasons, the assertion `b == n` is provable after the loop.

Because the loop specification mentions the mutable field `data`, the heap becomes part of the loop frame. Since the loop invariant is not strong enough to prove the assertion `c.data == 5` that follows the loop, the verifier complains about that assertion. On the other hand, had `c.data < 10` not been mentioned in the loop specification, the assertion would be verified, since field `K` is then the only field mentioned in the loop specification and `K` is immutable.

Finally, the aggregate statement (`forall`) can also be given without a body. Such a statement claims that the given `ensures` clause holds true for all values of the bound variables that satisfy the given range constraint. If the statement has no body, the program is in effect omitting the proof, much like a body-less lemma is omitting the proof of the claim made by the lemma specification. As with the other body-less constructs above, the verifier is silently happy with a body-less `forall` statement, but the compiler will complain.

19.15. Match Statement

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

CaseStmt = "case" ExtendedPattern "=>" { Stmt }
```

[`ExtendedPattern` is defined in Section 20.32.]

The `match` statement is used to do case analysis on a value of an inductive or co-inductive datatype (which includes the built-in tuple types), a base type, or newtype. The expression after the `match` keyword is called the *selector*. The expression is evaluated and then matched against each clause in order until a matching clause is found.

The process of matching the selector expression against the `CaseBinding_`s is the same as for match expressions and is described in Section 20.32.

The code below shows an example 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 := 0;
    case Node(t1, 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.

19.16. Assert Statement

```
AssertStmt =
    "assert"
    { Attribute }
    ( [ LabelName ":" ]
      Expression(allowLemma: false, allowLambda: true)
      ( ";"
      | "by" BlockStmt
      )
    | ellipsis
      ";"
```

`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 the assertion cannot be proven. Once the assertion is proved, its truth may aid in proving subsequent 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 truth may aid in the larger verification effort, much as lemmas might be used in mathematical proofs.

`Assert` statements are ignored by the compiler.

Using `...` as the argument of the statement is part of module refinement, as described in Section 21.

TO BE WRITTEN - assert by statements

19.17. Assume Statement

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

```
    )
    ";"
```

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`.

An `assume` statement cannot be compiled. In fact, the compiler will complain if it finds an `assume` anywhere where it has not been replaced through a refinement step.

Using `...` as the argument of the statement is part of module refinement, as described in Section 21.

19.18. Expect Statement

```
ExpectStmt =
    "expect"
    { Attribute }
    ( Expression(allowLemma: false, allowLambda: true)
    | ellipsis
    )
    [ "," Expression(allowLemma: false, allowLambda: true) ]
    ";"
```

The `expect` statement states a boolean expression that is (a) assumed to be true by the verifier and (b) checked to be true at run-time. That is, the compiler inserts into the run-time executable a check that the given expression is true; if the expression is false, then the execution of the program halts immediately. If a second argument is given, it may be a value of any type. That value is converted to a string (just like the `print` statement) and the string is included in the message emitted by the program when it halts; otherwise a default message is emitted.

Because the expect expression and optional second argument are compiled, they cannot be ghost expressions.

`assume` statements are ignored at run-time. The `expect` statement behaves like `assume` for the verifier, but also inserts a run-time check that the assumption is indeed correct (for the test cases used at run-time).

Here are a few use-cases for the `expect` statement.

A) To check the specifications of external methods.

Consider an external method `Random` that takes a `nat` as input and returns a `nat` value that is less than the input. Such a method could be specified as

```
method {:extern} Random(n: nat) returns (r: nat)
  ensures r < n
```

But because there is no body for `Random` (only the external non-dafny implementation), it cannot be verified that `Random` actually satisfies this specification.

To mitigate this situation somewhat, we can define a wrapper function, `Random'`, that calls `Random` but in which we can put some run-time checks:

```
method {:extern} Random(n: nat) returns (r: nat)

method Random'(n: nat) returns (r: nat)
  ensures r < n
{
  r := Random(n);
  expect r < n;
}
```

Here we can verify that `Random'` satisfies its own specification, relying on the unverified specification of `Random`. But we are also checking at run-time that any input-output pairs for `Random` encountered during execution do satisfy the specification, as they are checked by the `expect` statement.

Note, in this example, two problems still remain. One problem is that the out-parameter of the extern `Random` has type `nat`, but there is no check that the value returned really is non-negative. It would be better to declare the out-parameter of `Random` to be `int` and to include `0 <= r` in the condition checked by the `expect` statement in `Random'`. The other problem is that `Random` surely will need `n` to be strictly positive. This can be fixed by adding `requires n != 0` to `Random'` and `Random`.

B) Run-time testing

Verification and run-time testing are complementary and both have their role in assuring that software does what is intended. Dafny can produce executables and these can be instrumented with unit tests. Annotating a method with the `{:test}` attribute indicates to the compiler that it should produce target code that is correspondingly annotated to mark the method as a unit test (e.g., an XUnit test) in the target language. Within that method one might use `expect` statements (as well as `print` statements) to insert checks that the target program is behaving as expected.

C) Compiler tests

If one wants to assure that compiled code is behaving at run-time consistently with the statically verified code, one can use paired assert/expect statements

with the same expression:

```
assert _P_;
expect _P_;
```

The verifier will check that P is always true at the given point in a program (at the `assert` statement).

At run-time, the compiler will insert checks that the same predicate, in the `expect` statement, is true. Any difference identifies a compiler bug. Note that the `expect` must be after the `assert`. If the `expect` is first, then the verifier will interpret the `expect` like an `assume`, in which case the `assert` will be proved trivially and potential unsoundness will be hidden.

Using `...` as the argument of the `expect` statement is part of module refinement, as described in Section 21.

19.19. 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 code uses target-language-specific idioms to perform this printing. The expressions may of course include strings that are used for captions. There is no implicit new line added, so to add a new line you should include `"\n"` as part of one of the expressions. Dafny automatically creates implementations of methods that convert values to strings for all 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)
```

Note that Dafny does not have method overriding and there is no mechanism to override the built-in value->string conversion. Nor is there a way to explicitly invoke this conversion.

Dafny does not keep track of print effects. `print` statements are allowed only in

non-ghost contexts and not in expressions, with one exception. The exception is that a function-by-method may contain `print` statements, whose effect may be observed as part of the run-time evaluation of such functions.

19.20. Reveal Statement

```
RevealStmt =
    "reveal"
    Expression(allowLemma: false, allowLambda: true)
    { "," Expression(allowLemma: false, allowLambda: true) }
    ";"
```

TODO

19.21. Forall Statement

```
ForallStmt =
  "forall"
  ( "(" [ QuantifierDomain ] ")"
  | [ QuantifierDomain ]
  )
  { EnsuresClause(allowLambda: true) }
  [ BlockStmt ]
```

The `forall` statement executes the body simultaneously for all quantified values in the specified range. 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 ghost 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 performs simultaneous assignment. The left-hand sides must denote different l-values, unless the corresponding right-hand sides also coincide.

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);
}

lemma 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'
```

The following example of a *proof* `forall` statement comes from the same file:

```
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 de-sugared forms of co-predicates and co-lemmas. See section [#sec-co-inductive-datatypes].

19.22. Modify Statement

```
ModifyStmt =
  "modify"
  { Attribute }
  ( FrameExpression(allowLemma: false, allowLambda: true)
    { "," FrameExpression(allowLemma: false, allowLambda: true) }
  | ellipsis
  )
  ( 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 the modifies clause
               // on the line above
    }
  }

  method M1()
    modifies this
  {
    modify {} {
      var o := new ModifyBody;
      o.x := 3;  // fine
    }
  }

  method M2()
    modifies this
  {
    modify this {
      x := 3;
    }
  }

  method M3()
    modifies this
  {
    var k: int;
    modify {} { k := 4; } // fine. k is local
  }
}
```

The first `modify` statement in the example has an empty frame expression so the statement guarded by the modifies clause cannot modify any heap 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 limit what may be modified in already allocated memory. It does not limit what may be modified in new memory that is allocated within the block.

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.

Finally, the fourth example shows that the restrictions imposed by the modify statement do not apply to local variables, only those that are heap-based.

Using `...` as the argument of the statement is part of module refinement, as described in Section 21.

19.23. 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(Leino and Polikarpova 2013). 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 imposing 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 is 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 the example illustrates 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 `==` operators show the relation between the previous 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.

19.24. Skeleton Statement

```
SkeletonStmt =
  ellipsis
  ";"
```

TODO: Move to discussion of refinement?

20. 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
`;`	That is LemmaCall; Expression
`<==>`	equivalence (if and only if)
`==>`	implication (implies)
`<==`	reverse implication (follows from)
`&&`, `&`	conjunction (and)
`||`, `|`	disjunction (or)
`==`	equality
`==#[k]`	prefix equality (co-inductive)
`!=`	disequality
`!=#[k]`	prefix disequality (co-inductive)
`<`	less than
`<=`	at most
`>=`	at least
`>`	greater than
`in`	collection membership
`!in`	collection non-membership
`!!`	disjointness
`<<`	left-shift
`>>`	right-shift
`+`	addition (plus)
`-`	subtraction (minus)
`*`	multiplication (times)
`/`	division (divided by)
`%`	modulus (mod)
`|`	bit-wise or
`&`	bit-wise and
`^`	bit-wise exclusive-or (not equal)
`as` operation	type conversion
`is` operation	type conversion

operator	description
-	arithmetic negation (unary minus)
!	logical negation, bit-wise complement
Primary Expressions	

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.

20.1. 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); // note, this is a lemma call in an expression
  50 / Fact(n)
}
```

20.2. 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 0](#sec-equivalence-operator] for an explanation of the `<==>` operator as compared with the `==` operator.

20.3. 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 7.1.3] for an explanation of the `==>` and `<==` operators.

20.4. Logical Expressions

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

Note that the Dafny grammar allows a conjunction or disjunction to be *prefixed* with `&&` or `||` respectively. This form simply allows a parallel structure to be written:

```
var b: bool :=
  && x != null
  && y != null
  && z != null
  ;
```

This is purely a syntactic convenience allowing easy edits such as reordering lines or commenting out lines without having to check that the infix operators are always where they should be.

See Section 7.1.2 for an explanation of the `&&` and `||` operators.

20.5. Relational Expressions

```
RelationalExpression(allowLemma, allowLambda) =
  ShiftTerm(allowLemma, allowLambda)
  { RelOp ShiftTerm(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 [#sec-basic-types], `==`, `!=`, `<`, `>`, `<=`, and `>=` are *chaining*.

The `in` and `!in` operators apply to collection types as explained in Section 10 and represent membership or non-membership respectively.

The `!!` represents disjointness for sets and multisets as explained in Section 10.1 and Section 10.2.

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 [#sec-co-equality].

20.6. Bit Shifts

```
ShiftTerm(allowLemma, allowLambda) =
  Term(allowLemma, allowLambda)
  { ShiftOp Term(allowLemma, allowLambda) }

ShiftOp = ( "<<" | ">>" )
```

These operators are the left and right shift operators for bit-vector values. They take a bit-vector value and an `int`, shifting the bits by the given amount; the result has the same bit-vector type as the LHS. For the expression to be well-defined, the RHS value must be in the range 0 to the number of bits in the bit-vector type, inclusive.

The operations are left-associative: `a << i >> j` is `(a << i) >> j`. ## 20.7. 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 7.2]).
- Union for sets and multisets (Section 10.1 and Section 10.2)
- Concatenation for sequences (Section 10.3)
- Map merging for maps (Section 10.4).

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

20.8. Factors

```
Factor(allowLemma, allowLambda) =
  BitvectorFactor(allowLemma, allowLambda)
  { MulOp BitvectorFactor(allowLemma, allowLambda) }

MulOp = ( "*" | "/" | "%" )
```

A `Factor` combines `UnaryExpression`s using multiplication, division, or modulus. For numeric types these are explained in Section 7.2. As explained there, `/` and `%` on `int` values represent *Euclidean* integer division and modulus and not the typical C-like programming language operations.

Only `*` has a non-numeric application. It represents set or multiset intersection as explained in Section 10.1 and Section 10.2.

20.9. Bit-vector Operations

```
BitvectorFactor(allowLemma, allowLambda) =
  AsExpression(allowLemma, allowLambda)
  { BVOp AsExpression(allowLemma, allowLambda) }

BVOp = ( "|" | "&" | "^" )
```

These operations take two bit-vector values of the same type, returning a value of the same type. The operations perform bit-wise *or* (`|`), *and* (`&`), and *exclusive-or* (`^`). To perform bit-wise equality, use `^` and `!` (unary complement) together.

These operations associate to the left but do not associate with each other; use parentheses: `a & b | c` is illegal; use `(a & b) | c` or `a & (b | c)` instead.

Bit-vector operations are not allowed in some contexts. The `|` symbol is used both for bit-wise or and as the delimiter in a cardinality expression: an ambiguity arise if the expression E in `| E |` contains a `|`. This situation is easily remedied; just enclose E in parentheses, as in `|(E)|`. The only type-correct way this can happen is if the expression is a comprehension, as in `| set x: int :: x | 0x101 |`.

20.10. As (Conversion) and Is (type test) Expressions

```
AsExpression(allowLemma, allowLambda) =
  UnaryExpression(allowLemma, allowLambda)
  { ( "as" | "is" ) Type }
```

The `as` expression converts the given `UnaryExpression` to the stated `Type`, with the result being of the given type. The following combinations of conversions are permitted:

- Any type to itself
- Any int or real based numeric type or bit-vector type to another int or real based numeric type or bit-vector type
- Any base type to a subset or newtype with that base
- Any subset or newtype or to its base type or a subset or newtype of the same base
- Any type to a subset of newtype that has the type as its base
- Any trait to a class or trait that extends that trait
- Any class or trait to a trait extended by that class or trait

Some of the conversions above are already implicitly allowed, without the `as` operation, such as from a subset type to its base. In any case, it must be able to be proved that the value of the given expression is a legal value of the given type. For example, `5 as MyType` is permited (by the verifier) only if `5` is a legitimate value of`MyType` (which must be a numeric type).

The `as` operation is like a grammatical suffix or postfix operation. However, note that the unary operations bind more tightly than does `as`. That is `- 5 as nat` is `(- 5) as nat` (which fails), whereas `a * b as nat` is `a * (b as nat)`. On the other hand, `- a[4]` is `- (a[4])`.

The `is` expression is grammatically similar to the `as` expression, with the same binding power. The `is` expression is a run-time type test that returns a `bool` value indicating whether the LHS expression is a legal value of the RHS type. The expression can be used to check whether a trait value is of a particular class type. That is, the expression in effect checks the allocated type of a trait.

The RHS type of an `is` expression can always be a supertype of the type of the LHS expression. Other than that, the RHS must be based on a reference type and the LHS expression must be assignable to the RHS type. Furthermore, in order to be compilable, the RHS type must not be a subset type other than a non-null reference type, and the type parameters of the RHS must be uniquely determined from the type parameters of the LHS type. The last restriction is designed to make it possible to perform type tests without inspecting type parameters at run time. For example, consider the following types:

```
trait A { }
trait B<X> { }
class C<Y> extends B<Y> { }
class D<Y> extends B<set<Y>> { }
class E extends B<int> { }
class F<Z> extends A { }
```

A LHS expression of type `B<set<int>>` can be used in a type test where the RHS is `B<set<int>>`, `C<set<int>>`, or `D<int>`, and a LHS expression of type `B<int>` can be used in a type test where the RHS is `B<int>`, `C<int>`, or `E`. Those are always allowed in compiled (and ghost) contexts. For an expression `a` of type `A`, the expression `a is F<int>` is a ghost expression; it can be used in

ghost contexts, but not in compiled contexts.

For an expression `e` and type `t`, `e is t` is the condition determining whether `e as t` is well-defined (but, as noted above, is not always a legal expression).

The repertoire of types allowed in `is` tests may be expanded in the future.

20.11. Unary Expressions

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

A `UnaryExpression` applies either numeric (Section 7.2) or logical (Section 7.1) negation to its operand.

20.12. Primary Expressions

```
PrimaryExpression(allowLemma, allowLambda) =
  ( NameSegment { Suffix }
  | LambdaExpression(allowLemma)
  | MapDisplayExpr { Suffix }
  | SeqDisplayExpr { Suffix }
  | SetDisplayExpr { Suffix }
  | EndlessExpression(allowLemma, allowLambda)
  | 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 `Suffix`es to select a component of the value.

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

20.13. Lambda expressions

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

See Section 5.4 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)
```

20.14. 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` or an Update with Failure Statement.

An example of the first (`NameSegment`) form is:

```
    LibraryModule.F().x
```

An example of the second (`ConstAtomExpression`) form is:

```
    old(o.f).x
```

20.15. 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`, `UpdateFailureStmt`, or `VarDeclStatement`. These are the only contexts in which arrays or objects may be allocated, or in which havoc may be produced.

20.16. Array Allocation

```
ArrayAllocation_ =
  "new" [ Type ] "[" [ Expressions ] "]"
  [ "(" Expression(allowLemma: true, allowLambda: true) ")"
  | "[" [ Expressions ] "]"
  ]
```

This allocates a new single or multi-dimensional array as explained in section Section 15. The initialization portion is optional. One form is an explicit list of values, in which case the dimension is optional:

```
var a := new int[5];
var b := new int[5][2,3,5,7,11];
var c := new int[][2,3,5,7,11];
var d := new int[3][4,5,6,7]; // error
```

The comprehension form requires a dimension and uses a function of type `nat -> T` where `T` is the array element type:

```
var a := new int[5](i => i*i);
```

To allocate a multi-dimensional array, simply give the sizes of each dimension. For example,

```
var m := new real[640, 480];
```

allocates a 640-by-480 two-dimensional array of `real`s. The initialization portion cannot give a display of elements like in the one-dimensional case, but it can use an initialization function. A function used to initialize a n-dimensional array requires a function from n `nat`s to a `T`, where `T` is element type of the array. Here is an example:

```
var diag := new int[30, 30]((i, j) => if i == j then 1 else 0);
```

Array allocation is permitted in ghost contexts. If any expression used to specify a dimension or initialization value is ghost, then the `new` allocation can only be used in ghost contexts. Because the elements of an array are non-ghost, an array allocated in a ghost context in effect cannot be changed after initialization.

20.17. Object Allocation

```
ObjectAllocation_ = "new" Type [ "." TypeNameOrCtorSuffix ]
                              [ "(" [ Bindings ] ")" ]
```

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

20.18. Havoc Right-Hand-Side

```
HavocRhs_ = "*"
```

A havoc right-hand-side produces an arbitrary value of its associated type. To obtain a more constrained arbitrary value the "assign-such-that" operator (`:|`) can be used. See Section 19.6.

20.19. Constant Or Atomic Expressions

```
ConstAtomExpression =
  ( LiteralExpression
  | "this"
  | FreshExpression_
  | AllocatedExpression_
  | UnchangedExpression_
  | OldExpression_
  | CardinalityExpression_
```

```
  | ParensExpression
  )
```

A `ConstAtomExpression` represents either a constant of some type, or an atomic expression. A `ConstAtomExpression` is never an l-value.

20.20. Literal Expressions

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

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

20.21. Fresh Expressions

```
FreshExpression_ =
  "fresh" [ "@" LabelName ]
  "(" Expression(allowLemma: true, allowLambda: true) ")"
```

`fresh(e)` returns a boolean value that is true if the objects denoted by expression `e` were all freshly allocated since the time of entry to the enclosing method.

If the `LabelName` is present, it must denote a label that in the enclosing method's control flow dominates the expression. In this case, `fresh@L(e)` returns `true` if the objects denoted by `e` were all freshly allocated since control flow reached label `L`.

The argument of `fresh` must be either an object reference or a set or sequence of object references.

20.22. Allocated Expressions

```
AllocatedExpression_ =
  "allocated" "(" Expression(allowLemma: true, allowLambda: true) ")"
```

For any expression `e`, the expression `allocated(e)` evaluates to `true` in a state if the value of `e` is available in that state, meaning that it could in principle have been the value of a variable in that state. This can be useful when, for example, `allocated(e)` is evaluated in an `old` state. For instance, if `d` is a local variable holding a datatype value `Cons(r, Nil)` where `r` is an object that was allocated in the enclosing method, then `old(allocated(d))` is `false`.

If the expression `e` is of a reference type, then `!old(allocated(e))` is the same as `fresh(e)`.

20.23. Unchanged Expressions

```
UnchangedExpression_ =
  "unchanged" [ "@" LabelName ]
  "(" FrameExpression(allowLemma: true, allowLambda: true)
      { "," FrameExpression(allowLemma: true, allowLambda: true) }
  ")"
```

The `unchanged` expression returns `true` if and only if every reference denoted by its arguments has the same value for all its fields in the old and current state. For example, if `c` is an object with two fields, `x` and `y`, then `unchanged(c)` is equivalent to

```
c.x == old(c.x) && c.y == old(c.y)
```

Each argument to `unchanged` can be a reference, a set of references, or a sequence of references. If it is a reference, it can be followed by `` `f ``, where `f` is a field of the reference. This form expresses that `f`, not necessarily all fields, has the same value in the old and current state.

The optional `@`-label says to use it as the old-state instead of using the `old` state. That is, using the example `c` from above, the expression `unchanged@Lbl(c)` is equivalent to

```
c.x == old@Lbl(c.x) && c.y == old@Lbl(c.y)
```

Each reference denoted by the arguments of `unchanged` must be non-null and must be allocated in the old-state of the expression.

20.24. Old and Old@ Expressions

```
OldExpression_ =
  "old" [ "@" LabelName ]
  "(" Expression(allowLemma: true, allowLambda: true) ")"
```

An *old expression* is used in postconditions or in the body of a method or in the body or specification of any two-state function or two-state lemma; an *old* expression with a label is used only in the body of a method at a point where the label dominates its use in this expression.

`old(e)` evaluates the argument using the value of the heap on entry to the method; `old@ident(e)` evaluates the argument using the value of the heap at the given statement label.

Note that **old** and **old@** only affect heap dereferences, like `o.f` and `a[i]`. In particular, neither form has any effect on the value returned for local variables or out-parameters (as they are not on the heap).[9] If the value of an entire

[9]The semantics of `old` in Dafny differs from similar constructs in other specification lan-

expression at a particular point in the method body is needed later on in the method body, the clearest means is to declare a ghost variable, initializing it to the expression in question.

The argument of an `old` expression may not contain nested `old`, `fresh`, or `unchanged` expressions, nor two-state functions or two-state lemmas.

Here are some explanatory examples. All `assert` statements verify to be true.

```
class A {

  var value: int

  method m(i: int)
    requires i == 6
    requires value == 42
    modifies this
  {
    var j: int := 17;
    value := 43;
    label L:
    j := 18;
    value := 44;
    label M:
    assert old(i) == 6; // i is local, but can't be changed anyway
    assert old(j) == 18; // j is local and not affected by old
    assert old@L(j) == 18; // j is local and not affected by old
    assert old(value) == 42;
    assert old@L(value) == 43;
    assert old@M(value) == 44 && this.value == 44;
    // value is this.value; 'this' is the same
    // same reference in current and pre state but the
    // values stored in the heap as its fields are different;
    // '.value' evaluates to 42 in the pre-state, 43 at L,
    // and 44 in the current state
  }
}
```

```
class A {
  var value: int
  constructor ()
     ensures value == 10
  {
     value := 10;
  }
}
```

guages like ACSL or JML.

```
class B {
   var a: A
   constructor () { a := new A(); }

   method m()
     requires a.value == 11
     modifies this, this.a
   {
     label L:
     a.value := 12;
     label M:
     a := new A(); // Line X
     label N:
     a.value := 20;
     label P:

     assert old(a.value) == 11;
     assert old(a).value == 12; // this.a is from pre-state,
                                // but .value in current state
     assert old@L(a.value) == 11;
     assert old@L(a).value == 12; // same as above
     assert old@M(a.value) == 12; // .value in M state is 12
     assert old@M(a).value == 12;
     assert old@N(a.value) == 10; // this.a in N is the heap
                                  // reference at Line X
     assert old@N(a).value == 20; // .value in current state is 20
     assert old@P(a.value) == 20;
     assert old@P(a).value == 20;
  }
}
```

The next example demonstrates the interaction between `old` and array elements.

```
class A {
  var z1: array<nat>
  var z2: array<nat>

  method mm()
    requires z1.Length > 10 && z1[0] == 7
    requires z2.Length > 10 && z2[0] == 17
    modifies z2
  {
    var a: array<nat> := z1;
    assert a[0] == 7;
    a := z2;
```

```
    assert a[0] == 17;
    assert old(a[0]) == 17; // a is local with value z2
    z2[0] := 27;
    assert old(a[0]) == 17; // a is local, with current value of
                            // z2; in pre-state z2[0] == 17
    assert old(a)[0] == 27; // a is local, with current value of
                            // z2; z2[0] is currently 27
  }
}
```

20.25. Cardinality Expressions

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

For a finite-collection expression `c`, `|c|` is the cardinality of `c`. For a finite 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 finite map, the cardinality is the cardinality of the domain of the map. Cardinality is not defined for infinite sets or infinite maps. For more, see Section 10.

20.26. 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 17.1.

20.27. Sequence Display Expression

```
SeqDisplayExpr =
  ( "[" [ Expressions ] "]"
  | "seq" [ GenericInstantiation ]
    "(" Expression(allowLemma: true, allowLambda: true)
    "," Expression(allowLemma: true, allowLambda: true)
    ")"
  )
```

A sequence display expression provides a way to construct a sequence with given values. For example

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

is a sequence with three elements in it.

```
seq(k, n => n+1)
```

is a sequence of k elements whose values are obtained by evaluating the second argument (a function) on the indices 0 up to k.

See section [#sec-sequences] for more information on sequences.

20.28. Set Display Expression

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

A set display expression provides a way of constructing a set with given elements. If the keyword `iset` is present, then a potentially infinite set (with the finite set of given elements) is constructed.

For example

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

is a set with three elements in it. See Section 10.1 for more information on sets.

A multiset display expression provides a way of constructing a multiset with given elements and multiplicities. For example

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

is a multiset with three elements in it. The number 1 has a multiplicity of 2, and the numbers 2 and 3 each have a multiplicity of 1.

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 10.2 for more information on multisets.

20.29. 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"];
```

See Section 10.4 for more details on maps and imaps.

20.30. Endless Expression

```
EndlessExpression(allowLemma, allowLambda) =
  ( IfExpression(allowLemma, allowLambda)
  | MatchExpression(allowLemma, allowLambda)
  | QuantifierExpression(allowLemma, allowLambda)
  | SetComprehensionExpr(allowLemma, allowLambda)
  | StmtInExpr Expression(allowLemma, allowLambda)
  | LetExpression(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.

20.31. If Expression

```
IfExpression(allowLemma, allowLambda) =
    "if" ( BindingGuard(allowLambda: true)
         | 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
```

TO BE WRITTEN - binding form

20.32. Case and Extended Patterns

```
CasePattern =
  ( Ident "(" [ CasePattern { "," CasePattern } ] ")"
  | "(" [ CasePattern { "," CasePattern } ] ")"
  | IdentTypeOptional
  )

ExtendedPattern =
  ( PossiblyNegatedLiteralExpression
  | IdentTypeOptional
  | [ Ident ] "(" [ ExtendedPattern { "," ExtendedPattern } ] ")"
  )

PossiblyNegatedLiteralExpression =
  ( "-" ( Nat | Dec )
  | LiteralExpression
  )
```

Case patterns and extended patterns are used for (possibly nested) pattern matching on inductive, coinductive or base type values. The `ExtendedPattern` construct is used in `CaseStatement` and `CaseExpression`s, that is, in `match` statements and expressions. `CasePattern`s are used in `LetExpr`s and `VarDeclStatement`s. The `ExtendedPattern` differs from `CasePattern` is allowing literals and symbolic constants.

When matching an inductive or coinductive value in a `MatchStmt` or `MatchExpression`, the `ExtendedPattern` must correspond to a

- (1) bound variable (a simple identifier),
- (2) a constructor of the type of the value,
- (3) a literal of the correct type, or
- (4) a symbolic constant.

If the extended pattern is

- a parentheses-enclosed possibly-empty list of patterns, then the pattern matches a tuple.
- an identifier followed by a parentheses-enclosed possibly-empty list of patterns, then the pattern matches a constructor.
- a literal, then the pattern matches exactly that literal.
- a simple identifier, then the pattern matches
  - a parameter-less constructor if there is one defined with the correct type and the given name, else
  - the value of a symbolic constant, if a name lookup finds a declaration for a constant with the given name (if the name is declared but with a non-matching type, a type resolution error will occur),
  - otherwise, the identifier is a new bound variable

Any `ExtendedPattern`s inside the parentheses are then matched against the arguments that were given to the constructor when the value was constructed. The number of `ExtendedPattern` must match the number of parameters to the constructor (or the arity of the tuple). When matching a value of base type, the `ExtendedPattern` should either be a `LiteralExpression_` of the same type as the value, or a single identifier matching all values of this type.

The `ExtendedPattern`s and `CasePattern`s may be nested. The set of bound variable identifiers contained in a `CaseBinding_` or `CasePattern` must be distinct. They are bound to the corresponding values in the value being matched. (Thus, for example, one cannot repeat a bound variable to attempt to match a constructor that has two identical arguments.)

20.33. Match Expression

```
MatchExpression(allowLemma, allowLambda) =
  "match" Expression(allowLemma, allowLambda)
  ( "{" { CaseExpression(allowLemma: true, allowLambda: true) } "}"
  | { CaseExpression(allowLemma, allowLambda) }
  )

CaseExpression(allowLemma, allowLambda) =
  "case" ExtendedPattern "=>" 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, a co-inductive type, or a base type.

The `Expression` following the `match` keyword is called the *selector*. The selector is evaluated and then matched against each `CaseExpression` in order until a matching clause is found, as described in the section on `CaseBinding`s.

All of the variables in the `CasePattern`s must be distinct. If types for the identifiers are not given then types are inferred from the types of the construc-

tor'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. The `ExtendedPatterns` of each `CaseClause` are then compared in order with the resulting value until a matching pattern is found. 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.

20.34. 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 ==> (exists d :: n < d < 2*n))
```

The quantifier identifiers are *bound* within the scope of the expressions in the `QuantifierExpression`.

If 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.

20.35. Set Comprehension Expressions

```
SetComprehensionExpr(allowLemma, allowLambda) =
  [ "set" | "iset" ]
  IdentTypeOptional
  { "," IdentTypeOptional }
  { Attribute }
  "|"
```

```
  Expression(allowLemma, allowLambda)
  [ "::" Expression(allowLemma, allowLambda) ]
```

A set comprehension expression is an expression 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)
```

yields `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.

The universe in which set comprehensions are evaluated is the set of all *allocated* objects, of the appropriate type and satisfying the given predicate. For example, given

```
class I {
  var i: int
}

method test() {
  ghost var m := set x: I :: 0 <= x.i <= 10;
}
```

the set `m` contains only those instances of `I` that have been allocated at the point in program execution that `test` is evaluated. This could be no instances, one per value of `x.i` in the stated range, multiple instances of `I` for each value of `x.i`, or any other combination.

20.36. Statements in an Expression

```
StmtInExpr = ( AssertStmt | AssumeStmt | ExpectStmt
             | RevealStmt | 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`, `expect`, 'reveal' and `calc` statements can be used in this way.

20.37. Let Expression

```
LetExpression(allowLemma, allowLambda) =
  (
    [ "ghost" ] "var" CasePattern { "," CasePattern }
    ( ":=" | ":-" | { Attribute } ":|" )
    Expression(allowLemma: false, allowLambda: true)
    { "," 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
}
```

The syntax using `:-` is discussed in the following subsection.

20.38. Let or Fail Expression

The Let expression described in Section 20.37 has a failure variant that simply uses `:-` instead of `:=`. This Let-or-Fail expression also permits propagating failure results. However, in statements Section 19.7, failure results in immediate return from the method; expressions do not have side effects or immediate return mechanisms.

The expression `:- V; E` is desugared into the *expression*

```
var tmp := V;
if tmp.IsFailure()
then tmp.PropagateFailure()
else E
```

The expression `var v :- V; E` is desugared into the *expression*

```
var tmp := V;
if tmp.IsFailure()
then tmp.PropagateFailure()
else var v := tmp.Extract(); E
```

If the RHS is a list of expressions then the desugaring is similar. `var v, v1 :- V, V1; E` becomes

```
var tmp := V;
if tmp.IsFailure()
then tmp.PropagateFailure()
else var v, v1 := tmp.Extract(), V1; E
```

So, if tmp is a failure value, then a corresponding failure value is propagated along; otherwise, the expression is evaluated as normal.

Note that the value of the let-or-fail expression is either `tmp.PropagateFailure()` or `E`, the two sides of the if-then-else expression. Consequently these two expressions must have types that can be joined into one type for the whole let-or-fail expression. Typically that means that `tmp.PropagateFailure()` is a failure value and `E` is a value-carrying success value, both of the same failure-compatible type, as described in Section 19.7.

20.39. Map Comprehension Expression

```
MapComprehensionExpr(allowLemma, allowLambda) =
  ( "map" | "imap" )
  IdentTypeOptional
  { "," IdentTypeOptional }
  { Attribute }
  [ "|" Expression(allowLemma: true, allowLambda: true) ]
  "::"
  Expression(allowLemma, allowLambda)
  [ ":=" 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 finite 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.

If the expression includes the `:=` token, that token separates domain values from range values. For example, in the following code

```
method test()
{
  var m := map x : int | 1 <= x <= 10 :: 2*x := 3*x;
}
```

`m` maps `2` to `3`, `4` to `6`, and so on.

20.40. 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 18.3.4) or prefix lemma (see Section 18.3.5.3), the identifier must be the name of the copredicate or colemma and it must be followed by a `HashCall`.

20.41. Hash Call

```
HashCall = "#" [ GenericInstantiation ]
  "[" Expression(allowLemma: true, allowLambda: true) "]"
  "(" [ Bindings ] ")"
```

A `HashCall` is used to call the prefix for a copredicate or colemma. In the non-generic case, 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<T>)

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 state the
  // co-inductive hypothesis
  if (*) {
    Theorem0#<T>[_k-1](s);
  } else {
    Theorem0(s);
  }
}
```

where the `HashCall` is `"Theorem0#<T>[_k-1](s);"`. See Section 18.3.4 and Section 18.3.5.3.

20.42. 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.

20.42.1. 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.

20.42.2. 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);
}
}
```

20.42.3. 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 [#sec-other-sequence-expressions] for details.

20.42.4. 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 [#sec-other-sequence-expressions] for details.

20.42.5. 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`.

If the type of `s` is `seq<T>`, then `v` must have type `T`. The index `i` can have any integer- or bit-vector-based type (this is one situation in which Dafny implements implicit conversion, as if an `as int` were appended to the index expression). The expression `s[i := v]` has the same type as `s`.

20.42.6. 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.

If the `SelectionSuffix_` is used with an array or a sequence, then each index expression can have any integer- or bit-vector-based type (this is one situation in which Dafny implements implicit conversion, as if an `as int` were appended to the index expression).

20.42.7. 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 causes the method or function to be called and the result is the result of the call.

20.43. 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 commas.

20.44. Parameter Bindings

Method calls, object-allocation calls (`new`), function calls, and datatype constructors can be called with both positional arguments and named arguments.

```
ActualBindings =
    ActualBinding
    { "," ActualBinding }

ActualBinding =
    [ NoUSIdentOrDigits ":=" ]
    Expression(allowLemma: true, allowLambda: true)
```

Positional arguments must be given before any named arguments. Positional arguments are passed to the formals in the corresponding position. Named arguments are passed to the formal of the given name. Named arguments can be given out of order from how the corresponding formal parameters are declared. A formal declared with the modifier `nameonly` is not allowed to be passed positionally. The list of bindings for a call must provide exactly one value for every required parameter and at most one value for each optional parameter, and must never name non-existent formals. Any optional parameter that is not given a value takes on the default value declared in the callee for that optional parameter.

20.45. Formal Parameters and Default-Value Expressions

The formal parameters of a method, constructor in a class, iterator, function, or datatype constructor can be declared with an expression denoting a *default value*. This makes the parameter *optional*, as opposed to *required*. All required parameters must be declared before any optional parameters. All nameless parameters in a datatype constructor must be declared before any `nameonly` parameters.

The default-value expression for a parameter is allowed to mention the other parameters, including `this` (for instance methods and instance functions), but not the implicit `_k` parameter in least and greatest predicates and lemmas. The default value of a parameter may mention both preceding and subsequent parameters, but there may not be any dependent cycle between the parameters and their default-value expressions.

The well-formedness of default-value expressions is checked independent of the precondition of the enclosing declaration. For a function, the parameter default-value expressions may only read what the function's `reads` clause allows. For a datatype constructor, parameter default-value expressions may not read anything. A default-value expression may not be involved in any recursive or mutually recursive calls with the enclosing declaration.

20.46. Compile-Time Constants

In certain situations in Dafny it is helpful to know what the value of a constant is during program analysis, before verification or execution takes place. For example, a compiler can choose an optimized representation of a `newtype` that

is a subset of `int` if it knows the range of possible values of the subset type: if the range is within 0 to less than 256, then an unsigned 8-bit representation can be used.

To continue this example, suppose a new type is defined as

```
const MAX := 47
newtype mytype = x | 0 <= x < MAX*4
```

In this case, we would prefer that Dafny recognize that `MAX*4` is known to be constant with a value of `188`. The kinds of expressions for which such an optimization is possible are called *compile-time constants*. Note that the representation of `mytype` makes no difference semantically, but can affect how compiled code is represented at run time. In addition, though, using a symbolic constant (which may well be used elsewhere as well) improves the self-documentation of the code.

In Dafny, the following expressions are compile-time constants[10], recursively (that is, the arguments of any operation must themselves be compile-time constants):

- int, bit-vector, real, boolean, char and string literals
- int operations: `+ - * / %` and unary `-` and comparisons `< <= > >= == !=`
- real operations: `+ - *` and unary `-` and comparisons '< <= > >= == !='
- bool operations: `&& || ==> <== <==> == !=` and unary '!'
- bit-vector operations: `+ - * / % << >> & | ^` and unary `! -` and comparisons `< <= > >= == !=`
- char operations: `< <= > >= == !=`
- string operations: length: `|...|`, concatenation: `+`, comparisons `< <= == !=`, indexing `[]`
- conversions between: `int real char` bit-vector
- newtype operations: newtype arguments, but not newtype results
- symbolic values that are declared `const` and have an explicit initialization value that is a compile-time constant
- conditional (if-then-else) expressions
- parenthesized expressions

[10] This set of operations that are constant-folded may be enlarged in future versions of Dafny.

21. Refinement

Refinement is the process of replacing something somewhat abstract with something somewhat more concrete. For example, in one module one might declare a type name, with no definition, such as `type T`, and then in a refining module, provide a definition. One could prove general properties about the contents of an (abstract) module, and use that abstract module, and then later provide a more concrete implementation without having to redo all of the proofs.

Dafny supports *module refinement*, where one module is created from another, and in that process the new module may be made more concrete than the previous. More precisely, refinement takes the following form in Dafny. One module declares some program entities. A second module *refines* the first by declaring how to augment or replace (some of) those program entities. The first module is called the *refinement parent*; the second is the *refining* module; the result of combining the two (the original declarations and the augmentation directives) is the *assembled* module or *refinement result*.

Syntactically, the refinement parent is a normal module declaration. The refining module declares which module is its refinement parent with the `refines` clause:

```
module P { // refinement parent
}
module M refines P { // refining module
}
```

The refinement result is created as follows.

0) The refinement result is a module within the same enclosing module as the refining module, has the same name, and in fact replaces the refining module in their shared scope.

1) All the declarations (including import and export declarations) of the parent are copied into the refinement result. These declarations are *not* re-resolved. That is, the assignment of declarations and types to syntactic names is not changed. The refinement result may exist in a different enclosing module and with a different set of imports than the refinement parent, so that if names were reresolved, the result might be different (and possibly not semantically valid). This is why Dafny does not re-resolve the names in their new context.

2) All the declarations of the refining module that have different names than the declarations in the refinement parent are also copied into the refinement result. However, because the refining module is just a set of augmentation directives and mayt refer to names copied from the refinement parent, resolution of names and types of the declarations copied in this step is performed in the context of the full refinement result.

3) Where declarations in the parent and refinement module have the same name, the second refines the first and the combination, a refined declaration, is the result placed in the refinement result module, to the exclusion of the declarations with the same name from the parent and refinement modules.

The way the refinement result declarations are assembled depends on the kind of declaration; the rules are described in subsections below.

So that it is clear that refinment is taking place, refining declarations have some syntactic indicator that they are refining some parent declaration. Typically this is the presence of a `...` token.

21.1. Export set declarations

A refining export set declaration begins with the syntax

```
"export" Ident ellipsis
```

but otherwise contains the same `provides`, `reveals` and `extends` sections, with the ellipsis indicating that it is a refining declaration.

The result declaration has the same name as the two input declarations and the unions of names from each of the `provides`, `reveals`, and `extends` sections, respectively.

An unnamed export set declaration from the parent is copied into the result module with the name of the parent module. The result module has a default export set according to the general rules for export sets, after all of the result module's export set declarations have been assembled.

21.2. Import declarations

Aliasing import declarations are not refined. The result module contains the union of the import declarations from the two input modules. There must be no names in common among them.

Abstract import declarations (declared with `:` instead of `=`, Section 4.6) are refined. The refinement parent contains the abstract import and the refining module contains a regular aliasing import for the same name. Dafny checks that the refining import *adheres* to the abstract import.

21.3. Sub-module declarations

TODO

21.4. Const declarations

A parent `const` declaration may be refined by a refining `const` declaration if

- the parent has no initialization,
- the child has the same type as the parent, and
- one or both of the following holds:
  - the child has an initializing expression
  - the child is declared `ghost` and the parent is not `ghost`, or vice versa

To indicate it is a refining declaration, a refining `const` declaration contains an ellipsis in this syntax:

```
"const" { Attribute } CIdentType "..." [ ":=" Expression ]
```

21.5. Method declarations

TODO

21.6. Lemma declarations

TODO

21.7. Function and predicate declarations

TODO

21.8. Iterator declarations

TODO

21.9. Class and trait declarations

TODO

21.10. Type declarations

– opaque, type synonym, subset, newtype, datatype

TODO

22. 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.

In general, any Dafny entity may have a list of attributes. Dafny does not check that the attributes listed for an entity are appropriate for that entity (which means that misspellings may go silently unnoticed).

22.1. 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 (Leino 2008b).

Dafny has special processing for some attributes. For some attributes, the setting is only looked for on the entity with the attribute. 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 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.

22.1.1. 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. See (Leino and Wüstholz 2015), Section 3, for example uses of the `{:assumption}` attribute in Boogie.

22.1.2. 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)
}
```

22.1.3. autocontracts

Dynamic frames (Kassios 2006; Smans et al. 2008; Smans, Jacobs, and Piessens 2009; Leino 2009) 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 <= 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; }
```

22.1.4. 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 [#sec-opaque].

22.1.5. 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.

22.1.6. 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 an `ensures` clauses but no body. It has been used to declare Dafny interfaces in the MSR IronClad and IronFleet projects. Instead the `extern` keyword should be used (but that is soon to be replaced by the `{:extern}` attribute).

22.1.7. 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.

22.1.8. 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.

22.1.9. 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.

This seems to duplicate what `extern` and `{:decl}` do and would be a good candidate for deprecation.

22.1.10. 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 sub-list of those variables (in the same order) to which to apply induction.

Dafny issue 34 proposes to remove the restriction that the sub-list be in the same order, and would apply induction in the order given in the `{:induction}` attribute.

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:

```
lemma 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]
{
}
```

22.1.11. 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. Dafny issue 35 tracks further effort for this attribute.

22.1.12. 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 datatype cannot hold all the values that satisfy the constraint.

22.1.13. 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`).

22.1.14. opaque_full

The `{:opaque_full}` attribute is used to mark the *full* version of an opaque function. See Section 22.1.13.

22.1.16. tailrecursion

This attribute is used on 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 no argument is 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.

22.1.17. 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.

22.1.18. trigger

Trigger attributes are used on quantifiers and comprehensions. They are translated into Boogie triggers.

22.1.19. typeQuantifier

The `{:typeQuantifier}` attribute must be used on a quantifier if it quantifies over types.

22.2. 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 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 reach
       With /inline:assert call is replaced with "assert false" once inlining depth is reach
       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 verifie
       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
```

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}

23. Advanced Topics

23.1. Type Parameter Completion

http://leino.science/papers/krml270.html

TO BE WRITTEN

23.2. Type Inference

TO BE WRITTEN

23.3. Ghost Inference

TO BE WRITTEN

23.4. Well-founded Functions and Extreme Predicates

TODO: This section needs rewriting

This section is a tutorial on well-founded functions and extreme predicates. We place it here in preparation for Section [#sec-class-types] 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., (Camilleri and Melham 1992; Winskel 1993; Leroy and Grall 2009; Pierce et al. 2015; Nipkow and Klein 2014)).

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 (Paulin-Mohring 1993) and is the basis of the constructive logic supported by Coq (Bertot and Castéran 2004) as well as other proof assistants (Bove, Dybjer, and Norell 2009; Swamy et al. 2011). 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 (Paulson 1994) and HOL (Harrison 1995).

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 (Leino and Moskal 2014b) and is here described in Section [#sec-co-inductive-datatypes].

23.4.1. Function Definitions

To define a function $f : X \to Y$ in terms of itself, one can write an equation like
~ Equation {#eq-general}

$$f = \mathcal{F}(f)$$

~

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

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

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

~ Equation {#eq-fib}

$$\mathit{fib}(n) = \textbf{if } n < 2 \textbf{ then } n \textbf{ else } \mathit{fib}(n-2)$$

~

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 [#eq-general] or there could be none. Let's consider two ways to make sure we're defining the function uniquely.

TO BE WRITTEN - two-state functions and predicates

TO BE WRITTEN - functions with named results

TO BE WRITTEN - various kinds of arrow types: ~> -> –>

23.4.1.1. Well-founded Functions A standard way to ensure that equation [#eq-general] 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 [#eq-general] as

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

then we want to check $E \ll x$ for each call $f(E)$ in $f(x) = \mathcal{F}'(f)$. 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 [#eq-fib], we can pick $\ll$ to be the arithmetic less-than relation on natural numbers and check the following, for any n:

$$2 \leq n \implies n-2 \ll n \ \wedge \ n-1 \ll n$$

Note that we are entitled to use the antecedent $2 \leq n$ because that is the condition under which the else branch in [#eq-fib] 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 $P_{\downarrow}$ be a predicate on the ordinals defined by the following equation:

$$P\_\downarrow = P(o) \ \wedge \ \forall p \bullet \ p \ll o \implies P\_\downarrow(p)$$

With $\ll$ as the usual ordering on ordinals, this equation satisfies the decrement condition, but evaluating $P\_\downarrow(\omega)$ would require evaluating $P\_\downarrow(n)$ 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.

23.4.1.2. 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, $\mathit{fib}(n)$ 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 $\mathit{fib}(n-2)+\mathit{fib}(n-1)$ 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 $\mathit{fib}(n-2)+\mathit{fib}(n-1)$ 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.

23.4.1.3. Extreme Solutions We don't need to exclude the possibility of equation [#eq-general] 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 $f, g\colon X \to Y$,

Equation

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

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 (Tarski 1955) 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 [#eq-general] is the same as speaking about the *set of fixpoints* of functor $\mathcal{F}$. In particular, the least and greatest solutions to [#eq-general] are the same as the least and greatest fixpoints of $\mathcal{F}$. In casual speak, it happens that we say "fixpoint of [#eq-general]", 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 [#eq-general] 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 $\mathcal{F}$.

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

~ Equation {#eq-EvenNat}

$$g(x) = (x = 0 \ \vee \ g(x-2))$$

~

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

Equation

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

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:

Equation {#g-ind-rule} $\frac{}{g(0)}$ $\qquad$ $\frac{g(x-2)}{g(x)}$

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 [#fig-proof-trees] 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 thick line is used:

~ Equation {#g-coind-rule} $\dfrac{}{g(0)}$ $\qquad$ $\dfrac{g(x-2)}{g(x)}$

In this case, proof trees are allowed to be infinite. For example, the left-hand example below shows a finite proof tree that uses the rules of [#g-ind-rule] to establish $g(6)$. On the right is a partial depiction of an infinite proof tree that uses the rules of [#g-coind-rule] to establish $g(1)$.

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

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 [#g-coind-rule] once, the other is the infinite proof tree that keeps on using the right-hand rule of [#g-coind-rule].

23.4.2. 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 [#eq-EvenNat] 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 [#eq-general], I define two infinite series of well-founded functions, ${}^{\flat}f_k$ and ${}^{\sharp}f_k$ where k ranges over the natural numbers:

~ Equation {#eq-least-approx}

$$ {}^{\flat}f_k(x) = \left\{ \begin{array}{ll} \mathit{false} & \text{if } k = 0 \\ \mathcal{F}({}^{\flat}f_{k-1})(x) & \text{if } k > 0 \end{array} \right\} $$

.

~ Equation {#eq-greatest-approx}

$$ {}^{\sharp}f_k(x) = \left\{ \begin{array}{ll} \mathit{true} & \text{if } k = 0 \\ \mathcal{F}({}^{\sharp}f_{k-1})(x) & \text{if } k > 0 \end{array} \right\} $$

.
~

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, [#eq-least-approx] and [#eq-greatest-approx] 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 [#eq-general], we have, for any k and ℓ such that $k \leq \ell$:

Equation {#eq-prefix-postfix}

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

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:

Equation {#eq-least-is-exists}

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

Equation {#eq-greatest-is-forall}

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

By [#eq-prefix-postfix], 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 [#sec-friendliness] how to check for continuity.

Let's consider two examples, both involving function g in [#eq-EvenNat]. 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 [#eq-EvenNat].

23.4.2.1. 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 [#fig-proof-trees]. 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 $g^{\downarrow}(x) \implies 0 \leq x \;\wedge\; x$ even. We split our task into two cases, corresponding to which of the two proof rules in [#g-ind-rule] 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 $0 \leq (x-2) \ \wedge \ (x-2)$ even, from which it is easy to establish the conclusion of our proof goal.

Here's how we do the proof formally using [#eq-least-is-exists]. We massage the general form of our proof goal:

$$
\begin{array}{ll}
 & \mid f^{\uparrow}(x) \implies R \mid \\
= \mid & \quad \{ \text{ [\#eq-least-is-exists] } \} \mid \\
 & \mid (\\
existsk \bullet {}^{\flat}f_{k}(x)) & \implies R \mid \\
= \mid & \quad \{ \text{ distribute } \implies \text{ over } \exists \text{ to the left } \} \mid \\
 & \mid \forall k \bullet ({}^{\flat}f_{k}(x) \implies R) \mid
\end{array}
$$

The last line can be proved by induction over k. So, in our case, we prove ${}^{\flat}g\_k(x) \implies 0 \leq x \wedge x$ even for every k. If $k = 0$, then ${}^{\flat}g\_k(x)$ is `false`, so our goal holds trivially. If $k > 0$, then ${}^{\flat}g\_k(x) = (x = 0 \ \vee \ {}^{\flat}g\_k - 1(x-2))$. 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 $0 \leq (x-2) \ \wedge \ (x-2)$ even, from which our proof goal follows.

23.4.2.2. 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 (Jacobs and Rutten 2011) or a new attempt by Kozen and Silva that aims to emphasize the simplicity, not the mystery, of coinduction (Kozen and Silva 2012).

Let's prove $true \implies g^{\uparrow}(x)$. The intuitive coinductive proof goes like this: According to the right-hand rule of [#g-coind-rule], $g^{\uparrow}(x)$ 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 [#g-coind-rule] before invoking the coinduction hypothesis.

For anyone who may have felt that the intuitive proof felt too easy, here is a formal proof using [#eq-greatest-is-forall], which relies only on induction. We massage the general form of our proof goal:

The last line can be proved by induction over k. So, in our case, we prove

23.4.3. 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 [#eq-general] 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 (Manolios and Moore 2003). Functions can be underspecified in this way in the proof assistants ACL2 (Kaufmann, Manolios, and Moore 2000) and HOL (Krauss 2009).

23.5. Functions in Dafny

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

23.5.1. 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`.

23.5.2. 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 [#sec-fib-example]:

```
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., (Hoare 1969)), 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 (Leino 2012). 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
$existsk\bullet\ 100 \leq fib(k) < 200$ 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.

23.5.3. 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 [#eq-least-approx] and [#eq-greatest-approx]. 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, [#eq-least-is-exists] and [#eq-greatest-is-forall].

Remember that to be well defined, the defining functor of an extreme predicate must be monotonic, and for [#eq-least-is-exists] and [#eq-greatest-is-forall] 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 (Milner 1982; Leino and Moskal 2014b).

23.5.4. Proofs about Extreme Predicates

From what I have presented so far, we can do the formal proofs from Sections [#sec-example-least-solution] and [#sec-example-greatest-solution]. 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 [#eq-least-is-exists], 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 [#sec-example-greatest-solution]:

```
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 [#sec-example-least-solution] and [#sec-example-greatest-solution]. These shortcoming are addressed in the next subsection.

23.5.5. 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 [#sec-example-least-solution] and [#sec-example-greatest-solution]. 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!

23.6. Variable Initialization and Definite Assignment

TO BE WRITTEN – rules for default initialization; resulting rules for constructors; definite assignment rules

23.7. Well-founded Orders

The well-founded order relations for a variety of built-in types in Dafny are given in the following table:

type of `X` and `x`	`x` strictly below `X`
`bool`	`X && !x`
`int`	`x < X && 0 <= X`
`real`	`x <= X - 1.0 && 0.0 <= X`
`set<T>`	`x` is a proper subset of `X`
`multiset<T>`	`x` is a proper multiset-subset of `X`
`seq<T>`	`x` is a consecutive proper sub-sequence of `X`
`map<K, V>`	`x.Keys` is a proper subset of `X.Keys`
inductive datatypes	`x` is structurally included in `X`
reference types	`x == null && X != null`
co-inductive datatypes	`false`
type parameter	`false`
arrow types	`false`

Also, there are a few relations between the rows in the table above. For example, a datatype value `x` sitting inside a set that sits inside another datatype value

`X` is considered to be strictly below `x`. Here's an illustration of that order, in a program that verifies:

```
datatype D = D(s: set<D>)

method TestD(dd: D) {
  var d := dd;
  while d != D({})
    decreases d
  {
    var x :| x in d.s;
    d := x;
  }
}
```

TODO: Write this section; revise the above

24. Dafny User's Guide

Most of this document decribes the Dafny programming language. This section describes the `dafny` tool, a combined verifier and compiler that implements the Dafny language.

The development of the dafny language and tool is a GitHub project at https://github.com/dafny-lang/dafny. The project is open source, with collaborators from various organizations and additional contributors welcome. The software itself is licensed under the MIT license.

24.1. Introduction

The dafny tool implements the following capabilities:

- checking that the input files represent a valid Dafny program (i.e., syntax, grammar and type checking);
- verifying that the program meets its specifications, by translating the program to verification conditions and checking those with Boogie and an SMT solver, typically Z3;
- compiling the program to a target language, such as C#, Java, Javascript, Go (and others in development);
- running the executable produced by the compiler.

Using various command-line flags, the tool can perform various combinations of the last three actions (the first action is always performed).

24.2. Dafny Programs and Files

A Dafny program is a set of modules. Modules can refer to other modules, such as through `import` declarations or `refines` clauses. A Dafny program consists of all the modules needed so that all module references are resolved.

Dafny files (`.dfy`) in the operating system each hold some number of top-level declarations. Thus a full program may be distributed among multiple files. To apply the `dafny` tool to a Dafny program, the `dafny` tool must be given all the files making up a complete program (or, possibly, more than one program at a time). This can be effected either by listing all of the files by name on the command-line or by using `include` directives within a file to stipulate what other files contain modules that the given file needs. Thus the complete set of modules are all the modules in all the files listed on the command-line or referenced, recursively, by `include` directives within those files. It does not matter if files are repeated either as includes or on the command-line.[11]

[11] File names are considered equal if they have the same absolute path, compared as case-sensitive strings (regardless of whether the underlying file-system is case sensitive). Use of symbolic links may make the same file have a different absolute path; this will generally cause duplicate declaration errors.

Note however that although the complete set of files, command-line plus included files, make up the program, by default, only those files listed on the command-line are verified. To do a complete verification, each file must be verified; it may well happen that a verification failure in one file (which is not on the command-line and thus not checked) may hide a verification failure in a file that is being checked. Thus it is important to eventually check all files, preferably in an order in which the files without dependences are checked first, then those that depend on them, etc., until all files are checked.

24.3. Installing Dafny

The instructions for installing dafny and the required dependencies and environment are described on the Dafny wiki: https://github.com/dafny-lang/dafny/wiki/INSTALL. Theu are not repeated here to avoid replicating information that easily becomes inconsistent and out of date.

As of this writing, users can install pre-built Dafny binaries or build directly from the source files maintained in the github project.

Current and past Dafny binary releases can be found at https://github.com/dafny-lang/dafny/releases for each supported platform. Each release is a .zip file with a name combining the release name and the platform. Current platforms are Windows 10, Ubuntu 16ff, and MacOS 10.14ff.

The principal dependency of the dafny tool is that it uses `dotnet`, which is available and must be installed on Linux and Mac platforms to use dafny.

24.4. Dafny Code Style

There are code style conventions for Dafny code, recorded here. Most significantly, code is written without tabs and with a 2 space indentation.

24.5. IDEs for Dafny

Dafny source files are text files and can of course be edited with any text editor. However, some tools provide syntax-aware features:

- There is a Dafny mode for Emacs.
- VSCode, a cross-platform editor for many programming languages has an extension for dafny, installed from within VSCode. VSCode is available here. The extension provides syntax highlighting, in-line parser, type and verification errors, and code navigation.
- An old Visual Studio plugin is no longer supported

Information about installing IDE extensions for Dafny is found on the Dafny INSTALL page in the wiki.

24.6. The Dafny Server

TO BE WRITTEN

24.7. Using Dafny From the Command Line

`dafny` is a conventional command-line tool, operating just like other command-line tools in Windows and Unix-like systems.

- The format of a command-line is determined by the shell program that is executing the command-line (.e.g. bash, the windows shell, COMMAND, etc.). The command-line typically consists of file names and options, in any order, separated by spaces.
- Files are designated by absolute paths or paths relative to the current working directory. A command-line argument not matching a known option is considered a filepath.
- Files containing dafny code must have a `.dfy` suffix.
- There must be at least one `.dfy` file.
- The command-line may contain other kinds of files appropriate to the language that the dafny files are being compiled to.

The command `Dafny.exe /?` gives the current set of options supported by the tool. The most commonly used options are described in Section 24.10.

- Options may begin with either a `/` (as is typical on Windows) or a `-` (as is typical on Linux)
- If an option is repeated (e.g., with a different argument), then the later instance on the command-line supercedes the earlier instance.
- If an option takes an argument, the option name is followed by a `:` and then by the argument value, with no intervening white space; if the argument itself contains white space, the argument must be enclosed in quotes.
- Escape characters are determined by the shell executing the command-line.

The dafny tool performs several tasks:

- Checking the form of the text in a `.dfy` file. This step is always performed, unless the tool is simply asked for help information or version number.
- Running the verification engine to check all implicit and explicit specifications. This step is performed by default, but can be skipped by using the `-noVerify` or `-dafnyVerify:0` option
- Compiling the dafny program to a target language. This step is performed by default if the verification is successful but can be skipped or always executed by using variations of the `-compile` option.
- Whether the source code of the compiled target is written out is controlled by `-spillTargetCode`
- The particular target language used is chosen by `-compileTarget`
- Whether or not the dafny tool attempts to run the compiled code is controlled by `-compile`

The dafny tool terminates with these exit codes:

- 0 – success
- 1 – invalid command-line arguments
- 2 – parse or types errors
- 3 – compilation errors
- 4 – verification errors

Errors in earlier phases of processing typically hide later errors. For example, if a program has parsing errors, verification or compilation will not be attempted. The option `-countVerificationErrors:0` forces the tool to always end with a 0 exit code.

24.8. Verification

There are a great many options that control various aspects of verifying dafny programs. Here we mention only a few:

- Control of output: `-dprint`, `-rprint`, `-stats`, `-compileVerbose`
- Whether to print warnings: `-proverWarnings`
- Control of time: `-timeLimit`
- Control of the prover used: `-prover`

TO BE WRITTEN - advice on use of verifier, debugging verification problems

24.9. Compilation

The `dafny` tool can compile a Dafny program to one of several target languages. Details and idiosyncracies of each of these are described in the following subsections. In general note that,

- the compiled code originating from `dafny` can be compiled with other source and binary code, but only the `dafny`-originated code is verified
- the output file names can be set using `-out`
- for each target language, there is a runtime library that must be used with the `dafny`-generated code when executing that code; the runtime libraries are part of the Binary and Source releases (typically in the Binaries folder)
- names in Dafny are written out as names in the target language. In some cases this can result in naming conflicts. Thus if a Dafny program is intended to be compiled to a target language X, you should avoid using Dafny identifiers that are not legal identifiers in X or that conflict with reserved words in X.

TODO - location of DafnyRuntime files

24.9.1. Main method

To generate a stand-alone executable from a Dafny program, the Dafny program must use a specific method as the executable entry point. That method is

determined as follows:

- If the /Main option is specified on the command-line with an argument of "-", then no entry point is used at all
- If the /Main option is specified on the command-line and its argument is not an empty string, then its argument is interpreted as the fully-qualified name of a method to be used as the entry point. If there is no matching method, an error message is issued.
- Otherwise, the program is searched for a method with the attribute `{:main}`. If exactly one is found, that method is used as the entry point; if more than one method has the `{:main}` attribute, an error message is issued.
- Otherwise, the program is searched for a method with the name `Main`. If more than one is found an error message is issued.

Any abstract modules are not searched for candidate entry points, but otherwise the entry point may be in any module or type. In addition, an entry-point candidate must satisfy the following conditions:

- The method takes no parameters or type parameters
- The method is not a ghost method
- The method has no requires or modifies clauses, unless it is marked `{:main}`
- If the method is an instance (that is, non-static) method and the enclosing type is a class, then that class must not declare any constructor. In this case, the runtime system will allocate an object of the enclosing class and will invoke the entry-point method on it.
- If the method is an instance (that is, non-static) method and the enclosing type is not a class, then the enclosing type must, when instantiated with auto-initializing type parameters, be an auto-initialing type. In this case, the runtime system will invoke the entry-point method on a value of the enclosing type.

Note, however, that the following are allowed:

- 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.)

If no legal candidate entry point is identified, `dafny` will still produce executable output files, but they will need to be linked with some other code in the target language that provides a `main` entry point.

24.9.2. extern declarations

TO BE WRITTEN

24.9.3. C#

TO BE WRITTEN

24.9.4. Java

The Dafny-to-Java compiler writes out the translated files of a file *A*`.dfy` to a directory *A*-java. The `-out` option can be used to choose a different output directory. The file *A*`.dfy` is translated to *A*`.java`, which is placed in the output directory along with helper files. If more than one `.dfy` file is listyed on the command-line, then the output directory name is taken from the first file, and `.java` files are written for each of the `.dfy` files.

TO BE WRITTEN

24.9.5. Javascript

TO BE WRITTEN

24.9.6. Go

TO BE WRITTEN

24.9.7. C++

The C++ back-end is still very preliminary and is available for experimentation only.

TO BE WRITTEN

24.10. Dafny Command Line Options

There are many command-line options to the `dafny` tool. The most current documentation of the options is within the tool itself, using the `/?` option. Here we give an expanded description of the most important options.

Remember that options can be stated with either a leading `/` or a leading `-`.

24.10.1. Help and version information

- `-?` or `-help` : prints out the current list of command-line options and terminates
- `-version` : prints the version of the executable being invoked and terminates

24.10.2. Controlling errors and exit codes

- `-countVerificationErrors:<n>` - if 0 then always exit with a 0 exit code; if 1 (the default) then use the usual exit code

TO BE WRITTEN

24.10.3. Controlling output

- `-dprint:<file>` - print the Dafny program after parsing (use `-` for ‘ to print to the console)
- `-rprint:<file>` - print the Dafny program after type resolution (use `-` for to print to the console)

TO BE WRITTEN

24.10.4. Controlling aspects of the tool being run

- `-deprecation:<n>` - controls warnings about deprecated features
  - 0 - no warnings
  - 1 (default) - issue warnings
  - 2 - issue warnings and advise about alternate syntax
- `-warnShadowing` - emits a warning if the name of a declared variable caused another variable to be shadowed

TO BE WRITTEN

24.10.5. Controlling verification

- `-verifyAllModules` - verify modules that come from include directives

By default, Dafny only verifies files explicitly listed on the command line: if `a.dfy` includes `b.dfy`, a call to `Dafny a.dfy` will detect and report verification errors from `a.dfy` but not from `b.dfy`'s.

With this flag, Dafny will instead verify everything: all input modules and all their transitive dependencies. This way `Dafny a.dfy` will verify `a.dfy` and all files that it includes (here `b.dfy`), as well all files that these files include, etc.

Running Dafny with `/verifyAllModules` on the file containing your main result is a good way to ensure that all its dependencies verify.

24.10.6. Controlling boogie

TO BE WRITTEN

24.10.7. Controlling the prover

TO BE WRITTEN

24.10.8. Controlling compilation

- `-compile:<n>` - controls whether compilation happens
  - 0 - do not compile the program
  - 1 (default) - upon successful verification, compile the program to the target language
  - 2 - always compile, regardless of verification success
  - 3 - if verification is successful, compile the program (like option 1), and then if there is a `Main` method, attempt to run the program
  - 4 - always compile (like option 2), and then if there is a `Main` method, attempt to run the program
- `-compileTarget:<s>` - sets the target programming language for the compiler
  - `cs` - C#
  - `go` - Go
  - `js` - Javascript
  - `java` - Java
  - `cpp` - C++
  - `php` - PHP
- `-spillTargetCode:<n>` - controls whether to write out compiled code in the target language (instead of just holding it in internal temporary memory)
  - 0 (default) - do not write out code
  - 1 - write it out to the target language, if it is being compiled
  - 2 - write the compiled program if it passes verification, regardless of the `-compile` setting
  - 3 - write the compiled program regardless of verification success and the `-compile` setting
- `-out:<file>` - TODO
- `-compileVerbose:<n>` - whether to write out compilation information
  - 0 - do not print any information (silent mode)
  - 1 (default) - print information such as the files being created by the compiler

TO BE WRITTEN

24.10.9. Options intended for debugging

- `-dprelude:<file>` - choose an alternate prelude file
- `-pmtrace` - print pattern-match compiler debugging information
- `-titrace` - print type inference debugging information

TO BE WRITTEN

Sample math B: $a \to b$ or

$$a \to \pi$$

or (a) or [a $\to$]

Colors

```
integer literal:  10
hex literal:      0xDEAD
real literal:     1.1
boolean literal:  true false
char literal:     'c'
string literal:   "abc"
verbatim string:  @"abc"
ident:            ijk
type:             int
generic type:     map<int,T>
operator:         <=
punctuation:      { }
keyword:          while
spec:             requires
comment:          // comment
attribute         {: name }
error:            $
```

Syntax color tests:

```
integer: 0 00 20 01 0_1
float:   .0 1.0 1. 0_1.1_0
bad:     0_
hex:     0x10_abcdefABCDEF
string:   "string \n \t \r \0" "a\"b" "'" "\'" ""
string:   "!@#$%^&*()_-+={}[]|:;\\<>,.?/~`"
string:   "\u1234 "
string:   "     " : "\0\n\r\t\'\"\\"
notstring: "abcde
notstring: "\u123 " : "x\Zz" : "x\ux"
vstring:  @"" @"a" @""""" @"'\" @"\u"
vstring:  @"xx""y y""zz "
vstring:  @" " @"       "
vstring:  @"x
x"
bad:      @!
char:    'a' '\n' '\'' '"' '\"' ' ' '\\'
char:    '\0' '\r' '\t'  '\u1234'
badchar:  $ `
```

```
ids:  '\u123'   '\Z'  '\u'  '\u2222Z'
ids:  '\u123ZZZ'     '\u2222Z'
ids: 'a : a' : 'ab' :  'a'b' : 'a''b'
ids:  a_b _ab ab? _0
id-label:  a@label
literal:  true false null
op:       - ! ~ x  -!~x
op:       a + b - c * d / e % f a+b-c*d/e%f
op:       <= >= < > == != b&&c || ==> <==> <==
# 25. !=# !! in !in
op:       !in   !iné
not op:  !inx
punc:     . , :: | :| := ( ) [ ] { }
types:    int real string char bool nat ORDINAL
types:    object object?
types:    bv1 bv10 bv0
types:    array array2 array20 array10
types:    array? array2? array20? array10?
ids:      array1 array0 array02 bv02 bv_1
ids:      intx natx int0 int_ int? bv1_ bv1x array2x
types:    seq<int>  set < bool >
types:    map<bool,bool>  imap < bool , bool >
types:    seq<Node> seq< Node >
types:    seq<set< real> >
types:    map<set<int>,seq<bool>>
types:    G<A,int> G<G<A>,G<bool>>
types:    seq map imap set iset multiset
ids:      seqx mapx
no arg:  seq < >  seq < , >  seq <bool , , bool >  seq<bool,>
keywords: if while assert assume
spec:     requires  reads modifies
attribute:  {: MyAttribute "asd", 34 }
attribute:  {: MyAttribute }
comment:  // comment
comment:  /* comment */ after
comment:  // comment /* asd */ dfg
comment:  /* comment /* embedded */ tail */ after
comment:  /* comment // embedded */ after
comment: /* comment
   /* inner comment
   */
   outer comment
   */ after
   more after
```

26. References

Barnett, Mike, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2006. "Boogie: A Modular Reusable Verifier for Object-Oriented Programs." In *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, edited by Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, 4111:364–87. Lncs. Springer.

Bertot, Yves, and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development — Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer.

Bove, Ana, Peter Dybjer, and Ulf Norell. 2009. "A Brief Overview of Agda — a Functional Language with Dependent Types." In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009*, edited by Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, 5674:73–78. Lecture Notes in Computer Science. Springer.

Camilleri, Juanito, and Tom Melham. 1992. "Reasoning with Inductively Defined Relations in the HOL Theorem Prover." 265. University of Cambridge Computer Laboratory.

Harrison, John. 1995. "Inductive Definitions: Automation and Application." In *TPHOLs 1995*, edited by E. Thomas Schubert, Phillip J. Windley, and Jim Alves-Foss, 971:200–213. LNCS. Springer.

Hoare, C. A. R. 1969. "An Axiomatic Basis for Computer Programming." *Cacm* 12 (10): 576–80, 583.

Jacobs, Bart, and Jan Rutten. 2011. "An Introduction to (Co)algebra and (Co)induction." In *Advanced Topics in Bisimulation and Coinduction*, edited by Davide Sangiorgi and Jan Rutten, 38–99. Cambridge Tracts in Theoretical Computer Science 52. Cambridge University Press.

Kassios, Ioannis T. 2006. "Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions." In *FM 2006: Formal Methods, 14th International Symposium on Formal Methods*, edited by Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, 4085:268–83. Lncs. Springer.

Kaufmann, Matt, Panagiotis Manolios, and J Strother Moore. 2000. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers.

Koenig, Jason, and K. Rustan M. Leino. 2012. "Getting Started with Dafny: A Guide." In *Software Safety and Security: Tools for Analysis and Verification*, edited by Tobias Nipkow, Orna Grumberg, and Benedikt Hauptmann, 33:152–81. NATO Science for Peace and Security Series d: Information and Communication Security. IOS Press.

Kozen, Dexter, and Alexandra Silva. 2012. "Practical Coinduction." http://hdl.handle.net/1813/30510. Comp.; Inf. Science, Cornell Univ.

Krauss, Alexander. 2009. "Automating Recursive Definitions and Termination Proofs in Higher-Order Logic." PhD thesis, Technische Universität München.

Leino, K. Rustan M. 2008a. "Main Microsoft Research Dafny Web Page."

———. 2008b. "This Is Boogie 2." Manuscript KRML 178.

———. 2009. "Dynamic-Frame Specifications in Dafny." JML seminar,

Dagstuhl, Germany.

———. 2010. "Dafny: An Automatic Program Verifier for Functional Correctness." In *LPAR-16*, edited by Edmund M. Clarke and Andrei Voronkov, 6355:348–70. Lncs. Springer.

———. 2012. "Automating Induction with an SMT Solver." In *Verification, Model Checking, and Abstract Interpretation — 13th International Conference, VMCAI 2012*, edited by Viktor Kuncak and Andrey Rybalchenko, 7148:315–31. Lncs. Springer.

Leino, K. Rustan M., and Michal Moskal. 2014a. "Co-Induction Simply: Automatic Co-Inductive Proofs in a Program Verifier." Manuscript KRML 230.

Leino, K. Rustan M., and Michał Moskal. 2014b. "Co-Induction Simply — Automatic Co-Inductive Proofs in a Program Verifier." In *FM 2014*, 8442:382–98. LNCS. Springer.

Leino, K. Rustan M., and Nadia Polikarpova. 2013. "Verified Calculations." Manuscript KRML 231.

Leino, K. Rustan M., and Philipp Rümmer. 2010. "A Polymorphic Intermediate Verification Language: Design and Logical Encoding." In *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010*, edited by Javier Esparza and Rupak Majumdar, 6015:312–27. Lncs. Springer.

Leino, K. Rustan M., and Valentin Wüstholz. 2015. "Fine-Grained Caching of Verification Results." In *Computer Aided Verification (CAV)*, edited by Daniel Kroening and Corina S. Pasareanu, 9206:380–97. Lecture Notes in Computer Science. Springer.

Leroy, Xavier, and Hervé Grall. 2009. "Coinductive Big-Step Operational Semantics." *Information and Computation* 207 (2): 284–304.

Manolios, Panagiotis, and J Strother Moore. 2003. "Partial Functions in Acl2." *Journal of Automated Reasoning* 31 (2): 107–27.

Milner, Robin. 1982. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc.

Mössenböck, Hanspeter, Markus Löberbauer, and Albrecht Wöß. 2013. "The Compiler Generator Coco/r." Open source from University of Linz.

Moura, Leonardo de, and Nikolaj Bjørner. 2008. "Z3: An Efficient SMT Solver." In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*, edited by C. R. Ramakrishnan and Jakob Rehof, 4963:337–40. Lncs. Springer.

Nipkow, Tobias, and Gerwin Klein. 2014. *Concrete Semantics with Isabelle/HOL*. Springer.

Paulin-Mohring, Christine. 1993. "Inductive Definitions in the System Coq — Rules and Properties." In *TLCA '93*, 664:328–45. LNCS. Springer.

Paulson, Lawrence C. 1994. "A Fixedpoint Approach to Implementing (Co)inductive Definitions." In *CADE-12*, edited by Alan Bundy, 814:148–61. LNCS. Springer.

Pierce, Benjamin C., Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. 2015. *Software Foundations*. Version 3.2. http://www.cis.upenn.edu/~bcpierce/sf.

Smans, Jan, Bart Jacobs, and Frank Piessens. 2009. "Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic." In *ECOOP 2009 — Object-Oriented Programming, 23rd European Conference*, edited by Sophia Drossopoulou, 5653:148–72. Lncs. Springer.

Smans, Jan, Bart Jacobs, Frank Piessens, and Wolfram Schulte. 2008. "Automatic Verifier for Java-Like Programs Based on Dynamic Frames." In *Fundamental Approaches to Software Engineering, 11th International Conference, FASE 2008*, edited by José Luiz Fiadeiro and Paola Inverardi, 4961:261–75. Lncs. Springer.

Swamy, Nikhil, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2011. "Secure Distributed Programming with Value-Dependent Types." In *ICFP 2011*, 266–78. ACM.

Tarski, Alfred. 1955. "A Lattice-Theoretical Fixpoint Theorem and Its Applications." *Pacific Journal of Mathematics* 5: 285–309.

Winskel, Glynn. 1993. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press.