The Whiley Language Specification

Version 0.3.23

David J. Pearce, 2014

Contents

Chapter 1

Introduction

This document provides a specification of the *Whiley Programming Language*. Whiley is a hybrid imperative and functional programming language designed to produce programs with as few errors as possible. Whiley allows explicit specifications to be given for functions, methods and data structures, and employs a *verifying compiler* to check whether programs meet their specifications. As such, Whiley is ideally suited for use in *safety critical system*s. However, there are many benefits to be gained from using Whiley in a general setting (e.g. improved documentation, maintainability, reliability, etc). Finally, this document is *not* intended as a general introduction to the language, and the reader is referred to alternative documents for learning the language[?].

1.1 Background

Reliability of large software systems is a difficult problem facing software engineering, where subtle errors can have disastrous consequences. Infamous examples include: the Therac-25 disaster where a computer-operated X-ray machine gave lethal doses to patients[?]; the 1988 worm which reeked havoc on the internet by exploiting a buffer overrun[?]; the 1991 Patriot missile failure where a rounding error resulted in the missile catastrophically hitting a barracks[?]; and, the Ariane 5 rocket which exploded shortly after launch because of an integer overflow, costing the ESA an estimated $500 million[?].

Around 2003, Hoare proposed the creation of a *verifying compiler* as a grand challenge for computer science[?]. A verifying compiler "*uses automated mathematical and logical reasoning to check the correctness of the programs that it compiles.*" There have been numerous attempts to construct a verifying compiler system, although none has yet made it into the mainstream. Early examples include that of King[?], Deutsch[?], the Gypsy Verification Environment[?] and the Stanford Pascal Verifier[?]. More recently, the Extended Static Checker for Modula-3[?] which became the Extended Static Checker for Java (ESC/Java) — a widely acclaimed and influential work[?]. Building on this success was JML and its associated tooling which provided a standard notation for specifying functions in Java[?]. Finally, Microsoft developed the Spec# system which is built on top of C#[?].

1.2 Goals

The Whiley Programming Language has been designed from scratch in conjunction with a verifying compiler. The intention is to provide an open framework for research in automated software verification. The initial goal is to automatically eliminate common errors, such as *null dereferences*, *array-out-of-bounds*, *divide-by-zero* and more. In the future, the intention is to consider more complex issues, such as termination, proof-carrying code and user-supplied proofs.

1.3 History

Development of the Whiley programming language begun in 2009 by Dr. David J. Pearce, at the time a lecturer in Computer Science at Victoria University of Wellington. The accompanying website `http://whiley.org` went live in 2010, making the first versions of Whiley available for download. Since then, Whiley has been in constant development with the majority of contributions being made by the original author. Several scientific papers have published on different aspects of the language, including:

- **Implementing a Language with Flow-Sensitive and Structural Typing on the JVM**. David J. Pearce and James Noble. In *Proceedings of the Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE)*, 2011.
- **Sound and Complete Flow Typing with Unions, Intersections and Negations**, David J. Pearce. In *Proceedings of the Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, pages 335–354, 2013
- **A Calculus for Constraint-Based Flow Typing**. David J. Pearce. In *Proceedings of the Workshop on Formal Techniques for Java-like Languages (FTFJP)*, Article 7, 2013.
- **Whiley: a Platform for Research in Software Verification**. David J. Pearce and Lindsay Groves. In *Proceedings of the Conference on Software Language Engineering (SLE)*, pages 238-âĂŞ248, 2013
- **Reflections on Verifying Software with Whiley**. David J. Pearce and Lindsay Groves. In *Proceedings of the Workshop on Formal Techniques for Safety-Critical Software (FTSCS)*, 2013

Chapter 2

Lexical Structure

This chapter specifies the lexical structure of the Whiley programming language. Programs in Whiley are organised into one or more *source file*s written in Unicode. The Whiley language uses *indentation syntax* to delimit blocks and statements, rather than curly-braces (or similar) as found in many other languages.

2.1 Line Terminators

A Whiley compiler splits the sequence of (Unicode) input characters into lines by identifying *line terminators*:

```
LineTerminator  ::=  \n | \r | \r \n
```

Here, `\n` represents the ASCII character `LF` (`0xA`), whilst `\r` represents the ASCII character `CR` (`0xD`). The two characters `\r` `\n` taken together form one line terminator.

2.2 Indentation

After splitting the input characters into lines, a Whiley compiler then identifies the *indentation* of each line. This is necessary because Whiley employs indentation syntax meaning that indentation is significant in the meaning of Whiley programs.

```
Indentation  ::=  ^ ( \t | )*
```

Here, ^ demarcates the start of a line and, hence, indentation may only occur at the beginning of a line. Indentation may be compared using the $\leq$ comparator, such that $i \leq ir$ always holds (where i is some indentation and r is either empty or represents additional indentation). In other words, some indentation i is considered less-than-or-equal to another piece of indentation ir which includes the first as a prefix. This comparator is important for delimiting *statement block*s (§5.1).

2.3 Comments

There are two kinds of comments in Whiley: *line comment*s and *block comment*s:

```
/* This is a block comment */
```

The above illustrates a block comment, which is all of the text between `/*` and `*/` inclusive.

```
// This is a line comment
```

The above illustrates a line comment, which is all of the text from `//` up to the end-of-line.

2.4 Identifiers

An identifier is a sequence of one or more *letters* or *digits* which starts with a letter.

```
 Ident  ::=  Letter ( Letter | Digit )*

Letter  ::=  _ | a | ... | z | A | ... | Z

 Digit  ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Letters include lowercase and uppercase alphabetic characters (i.e. `a-z` and `A-Z`) and the underscore (`_`).

2.5 Keywords

The following strings are reserved for use as *keywords* and may not be used as identifiers:

```
Keyword  ::=  all | any | assert | assume | bool | break | byte
           |  case | catch | char | continue | debug
           |  default | do | else | ensures | export | false
           |  finite | for | function | if | import | in | int | is
           |  method | native | new | no | null | package
           |  private | protected | public | real | requires
           |  return | skip | some | string | switch | throw
           |  throws | total | true | try | void | where | while
```

The following strings are reserved for use as *keywords*, but may additionally be used as identifiers in certain contexts:

```
KeywordIdentifier  ::=  constant | from | type
```

2.6 Literals

A *literal* is a source-level entity which describes a value of primitive type (§4.4).

```
Literal  ::=  NullLiteral
           |  BoolLiteral
           |  ByteLiteral
           |  IntLiteral
           |  RealLiteral
           |  CharLiteral
           |  StringLiteral
```

2.6.1 Null Literal

The **`null`** type (§4.4.1) has a single value expressed as the **`null`** literal.

```
NullLiteral  ::=  null
```

2.6.2 Boolean Literals

The **`bool`** type (§4.4.2) has two values expressed as the `true` and `false` literals.

```
BoolLiteral  ::=  true | false
```

2.6.3 Byte Literals

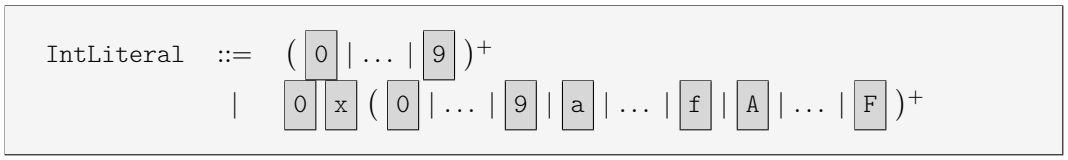
The **`byte`** type (§4.4.3) has 256 values which are expressed as sequences of binary digits, followed by the suffix "`b`" (e.g. `0101b`).

```
ByteLiteral  ::=  ( 0 | 1 )+ b
```

Byte literals do not need to contain exactly eight digits and, when fewer digits are given, are padded out to eight digits by appending zero's from the left (e.g. `00101b` becomes `00000101b`).

2.6.4 Integer Literals

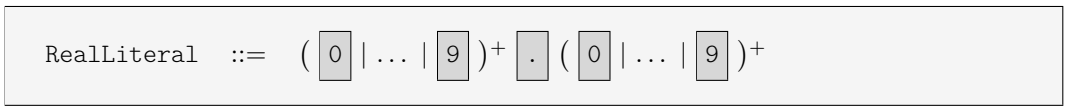
The **int** type (§4.4.4) represents the infinite set of integer values which are expressed as sequences of numeric or hexadecimal digits (e.g. `123456`, `0xffaf`, etc).



Since integer values in Whiley are of arbitrary size (§4.4.4), there is no limit on the size of an integer literal.

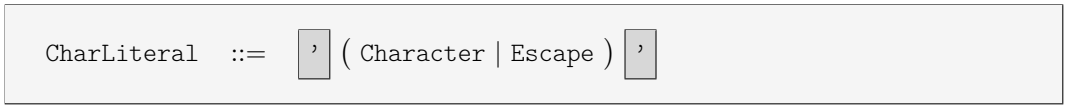
2.6.5 Real Literals

The **real** type (§4.4.5) represents the infinite set of rational values which are expressed as sequences of numeric digits separated by a period (e.g. `1.0`, `0.223`, `12.55`, etc).



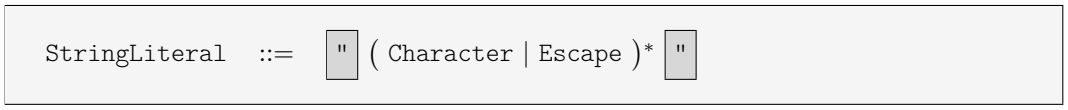
2.6.6 Character Literals

A *character literal* is expressed as a single character or an escape sequence enclosed in single quotes (e.g. 'c').



2.6.7 String Literals

A *string literal* is expressed as a sequence of zero or more characters or escape sequences enclosed in double quotes (e.g. "Hello-World").



Chapter 3

Source Files

Whiley programs are split across one or more source files which are compiled into *WyIL file*s prior to execution. Source files contain declarations which describe the functions, methods, data types and constants which form the program. Source files are grouped together into coherent units called *package*s.

3.1 Compilation Units

Two kinds of *compilation unit* are taken into consideration when compiling a Whiley source file: other source files; and, binary WyIL files. The Whiley Intermediate Language (WyIL) file format is described elsewhere, but defines a binary representation of a Whiley source file. When one or more Whiley source files are compiled together, a *compilation group* is formed. External symbols encountered during compilation are first resolved from the compilation group, and then from previously WyIL files.

3.2 Packages

Programs in Whiley are organised into packages to help reduce name conflicts and provide some grouping of related concepts. A Whiley source file may provide an optional `package` declaration to identify the package it belongs to. This declaration must occur at the beginning of the source file.

```
PackageDecl  ::=  package Ident ( . Ident )*
```

Any source file which does not provide a `package` declaration is considered to be in the *default package*.

3.3 Names

There are four functional entities which can be defined within a Whiley source file: *type declaration*s (§3.5.2), *constant declaration*s (§3.5.3), *function declaration*s (§3.5.4) and *method declaration*s (§3.5.5). These define *named entities* which may be referenced from other compilation units. Every named entity has a unique *fully-qualified* name constructed from the enclosing package name, the source file name and the declared name. For example:

Graphics.whiley

```
package tracer

type Point is { int x, int y }

constant Origin is { x: 0, y: 0 }
```

This declares two named entities: `tracer.Graphics.Point` and `tracer.Graphics.Origin`.

Two named entities may *clash* if they have the same fully qualified name and are in the same category. There are three entity categories: *types*, *constants* and *functions/methods*. The following illustrates a common pattern:

```
type Point is { int x, int y }

function Point(int x, int y) => Point:
    return {x: x, y: y}
```

Here, two named entities share the same fully qualified named. This is permitted because they are in different categories.

3.4 Imports

When performing *name resolution*, a Whiley compiler first attempts to resolve names within the same source file. For any remaining unresolved, the compiler examines imported entities in reverse declaration order. Entities are imported using an `import` declaration:

```
ImportDecl ::= import [ FromSpec ] Ident ( ( . | .. ) ( Ident | * ) )*

FromSpec   ::= ( Ident | * ) from
```

A declaration of the form "`import some.pkg.File`" imports the compilation unit "`File`" residing in package "`some.pkg`". Named entities (e.g. "`Entity`") within that compilation unit can then be referenced using a *partially qualified* name which omits the package component (e.g. "`File.Entity`"). A declaration of the form "`import Entity from some.pkg.File`" imports the named entity "`Entity`" from the compilation unit "`File`" residing in package "`some.pkg`". Note, this does *not* import the compilation unit "`some.pkg.File`" (and, hence, does not subsume the statement "`import some.pkg.File`").

A *wildcard* may be used in place of the compilation unit name (e.g. "`import some.pkg.*`") to import *all* compilation units within the given package. A *package match* may be used in place of some of all of the package component (e.g. "`import some..File`") to import all compilation units with matching name and package *prefix* and/or *suffix*. A *wildcard* may be used in place of the entity name (e.g. "`import * from some.pkg.File`") to import *all* named entities within the given compilation unit.

3.5 Named Declarations

Camel case

3.5.1 Access Control

3.5.2 Type Declarations

A *type declaration* declares a named type within a Whiley source file. The declaration may refer to named types in this or other source files and may also *recursively* refer to itself (either directly or indirectly).

```
TypeDecl  ::=  type Ident is TypePattern [ where Expr ]
```

The optional **where** clause defines a *boolean expression* which holds for any instance of this type. This is often referred to as the type *invariant* or *constraint*. Variables declared within the *type pattern* may be referred to within the optional **where** clause.

Examples. Some simple examples illustrating type declarations are:

```
// Define a simple point type
type Point is { int x, int y }

// Define the type of natural numbers
type nat is (int x) where x >= 0
```

The first declaration defines an unconstrained record type named `Point`, whilst the second defines a constrained integer type `nat`.

Notes. A convention is that type declarations for *records* or *unions of records* begin with an upper case character (e.g. `Point` above). All other type declarations begin with lower case. This reflects the fact that records are most commonly used to describe objects in the domain.

3.5.3 Constant Declarations

A *constant declaration* declares a named constant within a Whiley source file. The declaration may refer to named constants in this or other source files, although it may not refer to itself (either directly or indirectly).

```
ConstantDecl  ::=  constant Ident is Expr
```

The given *constant expression* is evaluated at *compile time* and must produce a constant value. This prohibits the use of function or method calls within the constant expression. However, general operators (e.g. for arithmetic) are permitted.

Examples. Some simple examples to illustrate constant declarations are:

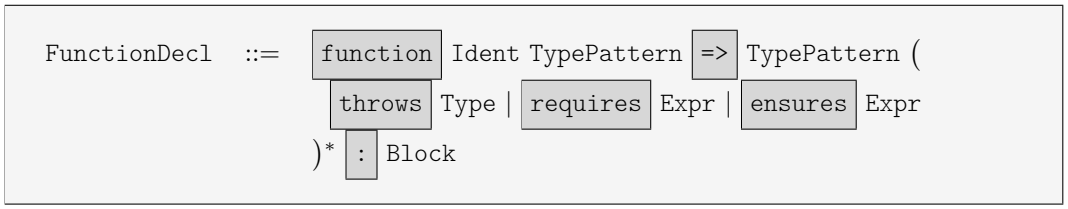
```
// Define the well-known mathematical constant to 10 decimal places.
constant PI is 3.141592654

// Define a constant expression which is twice PI
constant TWO_PI is PI * 2.0
```

The first declaration defines the constant `PI` to have the **real** value `3.141592654`. The second declaration illustrates a more interesting constant expression which is evaluated to `6.283185308` at compile time.

Notes. A convention is that constants are named in upper case with underscores separating words (i.e. as in `TWO_PI` above).

3.5.4 Function Declarations

A *function declaration* defines a function within a Whiley source file. Functions are *pure* and may not have side-effects. This means they are guaranteed to return the same result given the same arguments, and are permitted within specifications (i.e. in type invariants, *loop invariant*s, and function/method *precondition*s or *postcondition*s). Functions may call other functions, but may not call other methods. Functions may not allocate memory on the heap and/or instigate concurrent computation.



The first type pattern (i.e. before "=>") is referred to as the *parameter*, whilst the second is referred to as the *return*. There are three kinds of optional clause which follow:

- **Throws clause**. This defines the exceptions which may be thrown by this function. Multiple clauses may be given, and these are taken together as a union. Furthermore, the convention is to specify **`throws`** clause(s) first.
- **Requires clause(s)**. These define constraints on the permissible values of the parameters on entry to the function, and are often collectively referred to as the precondition. These expressions may refer to any variables declared within the parameter type pattern. Multiple clauses may be given, and these are taken together as a conjunction. Furthermore, the convention is to specify the **`requires`** clause(s) before any **`ensures`** clause(s).
- **Ensures clause(s)**. These define constraints on the permissible values of the function's return value, and are often collectively referred to as the postcondition. These expressions may refer to any variables declared within either the parameter or return type pattern. Multiple clauses may be given, and these are taken together as a conjunction. Furthermore, the convention is to specify **`ensures`** clause(s) last.

Examples. The following function declaration provides a small example to illustrate:

```
function max(int x, int y) => (int z)
// return must be greater than either parameter
ensures x <= z && y <= z
// return must equal one of the parmaeters
ensures x == z || y == z:
    // implementation
    if x > y:
        return x
    else:
        return y
```

This defines the specification and implementation of the well-known `max()` function which returns the largest of its parameters. This does not throw any exceptions, and does not enforce any preconditions on its parameters.

3.5.5 Method Declarations

A *method declaration* defines a method within a Whiley source file. Methods are *impure* and may have side-effects. Thus, they cannot be used within specifications (i.e. in type invariants, loop invariants, and function/method preconditions or postconditions). However, unlike functions, they methods call other functions and/or methods (including `native` methods). They may also allocate memory on the heap, and/or instigate concurrent computation.

```
MethodDecl  ::=  method Ident TypePattern => TypePattern (
                   throws Type | requires Expr | ensures Expr
                 )* : Block
```

The first type pattern (i.e. before "=>") is referred to as the *parameter*, whilst the second is referred to as the *return*. The three optional clauses are defined identically as for function declarations (§3.5.4).

Examples. The following method declaration provides a small example to illustrate:

```
// Define the well-known concept of a linked list
type LinkedList is null | &{ LinkedList next, int data }

// Define a method which inserts a new item onto the end of the list
method insertAfter(LinkedList list, int item) => LinkedList:
    if list is null:
        // reached the end of the list, so allocate new node
        return new { next: null, data: item }
    else:
        // continue traversing the list
        list→next = insertAfter(list→next, item)
        return list
```

Chapter 4

Types & Values

The Whiley programming language is *statically typed*, meaning that every expression has a type determined at compile time. Furthermore, evaluating an expression is guaranteed to yield a value of its type. Whiley's *type system* governs how the type of any variable or expression is determined. Whiley's type system is unusual in that it incorporates *union type*s (§4.11), *intersection type*s (§4.12) and *negation type*s (§4.13), as well as employing *flow typing* and *structural typing*.

4.1 Overview

Types in Whiley are unusual (in part) because there is a large gap between their *syntactic* description and their underlying *semantic* meaning. In most programming languages (e.g. Java), this gap is either small or non-existent and, hence, there is little to worry about. However, in Whiley, we must tread carefully to avoid confusion. The following example attempts to illustrate this gap between the syntax and semantics of types:

```
function id(null|int x) => int|null:
    return x
```

In this function we see two distinct *type descriptor*s expressed in the program text, namely "`int|null`" and "`null|int`". Type descriptors occur at the source-level and describe *type*s which occur at the abstract (or underlying) level. In this particular case, we have two distinct type descriptors which describe the *same* underlying type. We will often refer to types as providing the semantic (i.e. meaning) of type descriptors.

4.2 Type Descriptors

Type descriptors provide syntax for describing types and, in the remaining sections of this chapter, we explore the range of types supported in Whiley. The top-level grammar for type descriptors is:

```
     Type  ::=  UnitType ( , UnitType )*

UnitType  ::=  UnionType
            |  IntersectionType
            |  TermType

TermType  ::=
            |  PrimitiveType
            |  RecordType
            |  ReferenceType
            |  NominalType
            |  CollectionType
            |  NegationType
            |  FunctionType
            |  MethodType
            |  ( Type )
```

4.3 Type Patterns

Type patterns associate variables with types and their subcomponents and can be used to declare variables and/or *destructuring* types into variables. Type patterns are a source-level entity which are similar to type descriptors. The top-level grammar for type patterns is:

```
TypePattern  ::=  Type [ Ident ]
               |  TuplePattern
               |  RecordPattern
               |  RationalPattern
```

Type patterns do not exist for all compound structures — only those where a value is guaranteed to exist which could be associated with a variable.

4.4 Primitive Types

Primitive types are the atomic building blocks of all types in Whiley.

```
PrimitiveType  ::=
                 |  AnyType
                 |  VoidType
                 |  NullType
                 |  BoolType
                 |  ByteType
                 |  CharType
                 |  IntType
                 |  RealType
```

4.4.1 Null

The null type is typically used to show the absence of something. It is distinct from void, since variables can hold the special `null` value (where as there is no special "`void`" value). The set of values defined by the type `null` is the singleton set containing exactly the `null` value. Variables of `null` type support only equality (§6.7) and inequality comparisons (§6.7). The `null` value is particularly useful for representing optional values and terminating recursive types.

Example. The following illustrates a simple example of the `null` type:

```
type Tree is null | { int data, Tree left, Tree right }

function height(Tree t) => int:
    if t is null:
        // height of empty tree is zero
        return 0
    else:
        // height is this node plus maximum height of subtrees
        return 1 + Math.max(height(t.left), height(t.right))
```

This defines `Tree` — a *recursive type* — which is either empty (i.e. `null`) or consists of a field `data` and two subtrees, `left` and `right`. The `height` function calculates the height of a `Tree` as the longest path from the root through the tree.

Notes. With all of the problems surrounding `null` and `NullPointerException`s in languages like Java and C, it may seem that this type should be avoided. However, it remains a very useful abstraction around (e.g. for terminating recursive types) and, in Whiley, is treated in a completely safe manner (unlike e.g. Java).

4.4.2 Booleans

The `bool` type represents the set of boolean values (i.e. `true` and `false`). Variables of `bool` type support equality (§6.7), inequality (§6.7), binary logical operators (§6.5) and logical not (§**??**).

Example. The following illustrates a simple example of the `bool` type:

```
// Determine whether item is contained in list or not
function contains([int] list, int item) => bool:
    // examine every element of list
    for l in list:
        if l == item:
            return true
    // done
    return false
```

This function determines whether or not a given integer value is contained within a list of integers. If so, it returns `true`, otherwise it returns `false`.

4.4.3 Bytes

The type **`byte`** represents the set of eight-bit sequences, whose values are expressed numerically using `0` and `1` followed by `b` (e.g. 00101b). The set of values defined by the **`byte`** type is the set of all 256 possible combinations of eight-bit sequences. Variables of **`byte`** type support equality (§6.7), inequality (§6.7), bitwise operators (§6.6), bitwise complement (§**??**) and shift operators (§6.11).

```
ByteType  ::=  byte
```

Example. The following illustrates a simple example of the **`byte`** type:

```
// convert a byte into a string
function toString(byte b) => string:
    string r = "b"
    for i in 0..8:
        if (b & 00000001b) == 00000001b:
            r = "1" ++ r
        else:
            r = "0" ++ r
        b = b >> 1
    return r
```

This illustrates the conversion from a **`byte`** into a **`string`**. The conversion is performed one digit at a time, starting from the rightmost bit.

Notes. Unlike for many languages, there is no representation associated with a byte. For example, to extract an integer value from a byte, it must be explicitly decoded according to some representation (e.g. two's compliment) using an auxillary function (e.g. `Byte.toInt()`).

4.4.4 Integers

The type **`int`** represents the set of arbitrary-sized integers, whose values are expressed as a sequence of one or more numerical or hexadecimal digits (e.g. `123456`, `0xffaf`, etc). Variables of **`int`** type support equality (§6.7), inequality (§6.7), comparators (§6.7), addition (§6.12), subtraction (§6.12), multiplication (§6.12), division (§6.12), remainder (§6.12) and negation (§**??**) operations.

```
IntType  ::=  int
```

Example. The following illustrates a simple example of the **`int`** type:

```
function fib(int x) => int:
    if x <= 1:
        return x
    else:
        return fib(x-1) + fib(x-2)
```

This illustrates the well-known recursive method for computing numbers in the *fibonacci* sequence.

Notes. Since integers in Whiley are of arbitrary size, *integer overflow* is not possible. This contrasts with other languages (e.g. Java) that used *fixed-width* number representations (e.g. 32bit two's complement). Furthermore, there is nothing equivalent to the constants found in such languages for representing the uppermost and least integers expressible (e.g. `Integer.MIN_VALUE` and `Integer.MAX_VALUE`, as found in Java).

4.4.5 Rationals

The type **`real`** represents the set of arbitrary-sized rationals, whose values are expressed as a sequence of one or more numerical digits separated by a period (e.g. `1.0`, `0.223`, `12.55`, etc). The set of values defined by the type **`real`** is the (infinite) set of all integer pairs, where the first element is designated the numerator, and the second designated the denominator. Variables of **`real`** type support equality (§6.7), inequality (§6.7), comparators (§6.7), addition (§6.12), subtraction (§6.12), multiplication (§6.12), division (§6.12), remainder (§6.12) and negation (§**??**) operations. Variables of type **`real`** also support the *rational destructuring assignment* to extract the numerator and denominator (illustrated below).

```
RealType  ::=  real
```

Example. The following illustrates a simple example of the **`real`** type:

```
function floor(real x) => int:
    int num / int den = x     // extract numerator and denominator
    int r = num / den         // integer division
    if x < 0.0 && den != 1:
        return r - 1
    else:
        return r
```

This illustrates the well-known function for computing the *floor* of a **`real`** variable `x` (i.e. the greatest integer not larger than `x`). The rational destructuring assignment is used to extract the numerator and denominator of the parameter `x`.

4.4.6 Characters

The type **`char`** represents the set of unicode characters, whose values are expressed as an arbitrary character between quotes (e.g. 'c', '0', '%', etc). Variables of **`char`** type support equality (§6.7), inequality (§6.7), comparators (§6.7), addition (§6.12), subtraction (§6.12), multiplication (§6.12), division (§6.12), remainder (§6.12) and negation (§**??**) operations.

```
CharType  ::=  char
```

Example. The following illustrates a simple example of the **`char`** type:

```
function isUpperCase(char c) => bool:
    return 'A' <= c && c <= 'Z'
```

This illustrates a very simple function for determining whether an ASCII character is uppercase or not.

4.4.7 Any

The type **any** represents the type whose variables may hold any possible value. Thus, **any** is the *top type* (i.e. $\top$) in the lattice of types and, hence, is the supertype of all other types. Variables of **any** type support only equality (§6.7), inequality comparisons (§6.7) and *runtime type tests*. Finally, unlike the majority of other types, there are no *values* of type **any**.

```
AnyType  ::=  any
```

Example. The following illustrates a simple example of the **any** type:

```
function toInt(any val) => int:
    if val is int:
        return val
    else if val is real:
        return Math.floor(val)
    else:
        return 0 // default value
```

Here, the function `toInt` accepts *any valid Whiley value*, which includes all values of type **int**, **real**, collections, records, etc. The function then inspects the value that it has been passed and, in the case of values of type **int** and **real**, returns an integer approximation; for all other values, it returns `0`.

Notes. The any type is roughly comparable to the `Object` type found in pure object-oriented languages. However, in impure object-oriented languages which support primitive types, such as Java, this comparison often falls short because `Object` is not a supertype of primitives such as **int** or `long`.

4.4.8 Void

The **void** type represents the type whose variables cannot exist (i.e. because they cannot hold any possible value). Thus, **void** is the *bottom type* (i.e. $\bot$) in the lattice of types and, hence, is the *subtype* of all other types. Void is used to represent the return type of a method which does not return anything. Furthermore, it is also used to represent the element type of an empty list of set. Finally, unlike the majority of other types, there are no *values* of type **void**.

```
VoidType  ::=  void
```

Example. The following example illustrates several uses of the **void** type:

```
// Attempt to update first element
method update1st(&[int] list, int value) => void:
    // First, check whether list is empty or not
    if *list != [void]:
```

```
        // Then, update 1st element
        (*list)[0] = x
    // done
```

Here, the method `update1st` is declared to return **`void`** — meaning it does not return a value. Instead, this method updates some existing state accessible through the reference `list`. Within the method body, the value accessible via this reference is compared against the `[void]` (i.e. the *empty list*).

4.5 Tuples

A tuple type describes a compound type made up of two or more elements in sequence, whose values are expressed as sequences of values separated by a comma (e.g. `1,2`, `2.0,3.32,3.45`, etc). Tuples are similar to records, except that fields are effectively anonymous. Variables of tuple type support equality (§6.7) and inequality (§6.7) operations, as well the *tuple destructuring assignment* to extract elements (illustrated below).

```
    TupleType  ::=  ( Type ( , Type )+ )

TuplePattern   ::=  ( TypePattern ( , TypePattern )+ ) [ Ident ]
```

Example. The following example illustrates several uses of tuples:

```
function swap(int x, int y) => (int,int):
    return y,x
```

This function accepts two integer parameters, and returns a tuple type containing two integers. The function simple reverses the order that the values occur in the tuple passed as a parameter.

4.6 Records

A record type describes a compound made of from one or more *fields*, each of which has a unique name and a corresponding type. Variables of record type support equality (§6.7), inequality (§6.7) and field access (§6.13) operations, as well as field assignment (§**??**).

```
RecordType  ::=  { MixedType ( , MixedType )* [ , ... ] }
```

Records use *mixed types* for defining fields (see §**??**), meaning that field names may be mixed within their type. This is primarily useful for fields of function or method type (see below). Records using the `...` notation are referred to as *open records* (e.g. `{int x, ...}`), otherwise they are referred to as *closed records* (e.g. `{int x, int y}`). Open records represent all records containing *at least* the given fields, whilst closed records represent those containing *exactly* the given fields.

Example. The following example illustrates an open record type:

```
type Writer is {
   method write([byte]) => int,
   ...
}
type PrintWriter is {
   method write([byte]) => int,
   method println(string) => void,
   ...
}
// Create print writer if not already one
function create(Writer writer) => PrintWriter:
    if writer is PrintWriter:
        return writer
    else:
        return new PrinterWriter(writer)
```

The above illustrates two open records `Writer` and `PrintWriter`. The former has one field (`write`), whilst the latter has two fields (`write` and `println`). The above also illustrates use of mixed types. For example, the field "`write`" is declared as "`method write([byte]) => int`" which mixes together the field name (i.e. "`write`") with its type (i.e. "`method([byte]) => int`").

4.7 References

Reference types in Whiley represent references to variables, such as those allocated in the heap. They are similar to references or pointers found in many imperative and object-oriented languages (e.g. C/C++, Java, C#, etc). A type `&T` represents a reference to a variable of type `T`. Variables of reference type support equality (§6.7), inequality (§6.7) and dereference (§**??**) operations, as well as dereference assignment (§**??**).

```
ReferenceType  ::=  &  Type
```

Example. The following example illustrates reference types:

```
// Swap contents of heap-allocated int variables
method swap(&int pX, &int pY):
    int tmp = *pX
    *pX = *pY
    *pY = tmp
```

The above illustrates a method which accepts two references to variables of type `int` that may refer to the same variable. The method simply swaps the contents of the variables to which they refer.

4.8 Nominals

Nominal types represent user-defined types declared within one or more Whiley source files. Nominal types provide a mechanism for enforcing *information hiding*, and also for constructing *recursive types* (§4.14). All nominal types have an underlying — or, *concrete* — type which may or may not be visible within a given Whiley source file. Nominal types with *visible* underlying types are indistinguishable from their underlying type, and support

all operations therein. Nominal types with *invisible* underlying types support only equality (§6.7) and inequality (§6.7) operations. Furthermore, to enforce information hiding, nominal types cannot be destructured using runtime type tests.

```
NominalType  ::=  Ident
```

Example. The following example illustrates nominal types:

```
// Using a nominal type to construct a recursive type
public type LinkedList is null | { int data, LinkedList next }

// Using a nominal type to enforce information hiding
protected type Hidden is { int x, int y }
```

Three different uses for nominal types are illustrated here. The type `LinkedList` is declared **`public`**, meaning that: firstly, it can be referred to by name from any other source file; secondly, its underlying type is visible to any other source file. Within the declaration of `LinkedList` a reference to itself is used to define a recursive type (§4.14). Finally, the type `hidden` has the concrete type "`{int x, int y}`" and is declared **`protected`**, meaning that: firstly, it can be referred to by name from any other source file; secondly, that its underlying type is invisible to all other source files.

4.9 Collections

Collection types in Whiley describe compound values constructed from arbitrarily many values.

```
CollectionType  ::=  SetType
                  |  MapType
                  |  ListType
```

4.9.1 Sets

A set type describes set values whose elements are subtypes of the element type. For example, `{1,2,3}` is an instance of set type `{int}`; however, `{1.345}` is not. Variables of set type support equality (§6.7), inequality (§6.7), union (§**??**), intersection (§**??**), difference (§**??**) and element-of (§**??**) operations.

```
SetType  ::=  {  Type  }
```

Example. The following example illustrates set types:

```
// Adjacency list representation
type Graph is ([{int}])

function depthFirstSearch(int v, Graph graph, {int} visited) => {int}:
    visited = visited + {v}
    // Traverse edges not yet visited
```

```
    for w in graph[v]:
        if !(w in visited):
            visited = depthFirstSearch(w,graph,visited)
    // Done
    return visited
```

The above illustrates a simple implementation of the well-known *depth-first search* algorithm. In the example, a `Graph` is a list of sets of edge targets, where any `w in g[v]` describes an edge from `v` to `w` in the graph. The `visited` set is used to maintain a list of previously seen vertices, in order to prevent the same vertex from being visited more than once.

4.9.2 Maps

A map represents a one-many mapping from variables of one type to variables of another type. For example, the map type `{int=>real}` represents a map from integers to real values. A valid instance of this type might be `{1=>1.2,2=>3.0}`. Variables of map type support equality (§6.7), inequality (§6.7), access (§6.13), union (§**??**), intersection (§**??**), difference (§**??**) and element-of (§**??**) operations.

```
MapType  ::=  { Type => Type }
```

Example. The following example illustrates map types:

```
type Expr is int | string // simple expression forms

function evaluate(Expr e, {string=>int} environment) => int:
    if e is int:
        // expression is constant, so return this directly
        return e
    else:
        // expression is variable, so look up its value in environment
        return environment[e]
```

The above illustrates a function for evaluating simple expressions which are either integer constants or variable names. To evaluate an expression which is an integer constant, we simply return that constant. To evaluate an expression which is a variable name, we look up the current value of that variable in an environment which maps variable names to integer constants.

4.9.3 Lists

A list type describes list values whose elements are subtypes of the element type. For example, `[1,2,3]` is an instance of list type `[int]`; however, `[1.345]` is not. Variables of list type support equality (§6.7), inequality (§6.7), append (§6.9), sublist (§**??**) and element-of (§**??**) operations.

```
ListType  ::=  [ Type ]
```

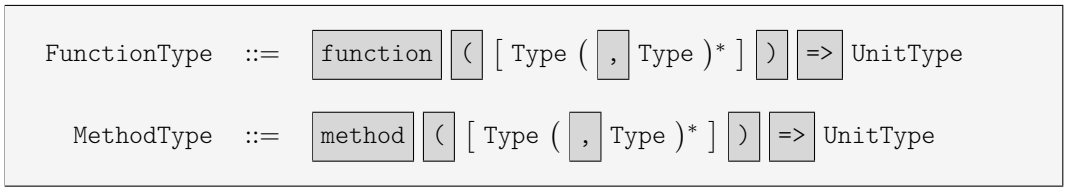
Example. The following example illustrates list types:

```
function add([int] v1, [int] v2) => ([int] v3):
    int i=0
    while i < |v1|:
        v1[i] = v1[i] + v2[i]
        i = i + 1
    return v1
```

The above illustrates a simple function which adds two integer lists together. The function's precondition requires that both input list have the same length, whilst its postconditions ensures that this matches the length of the output.

4.10 Functions and Methods

A function or method type describes the signature of a function or method. These types enable functions or methods to be passed around as values in Whiley and are often referred to as *functors*. This enables a degree of polymorphism in the language, where the exact function or method to be called is unknown. Variables of function or method type support equality (§6.7) and inequality (§6.7).



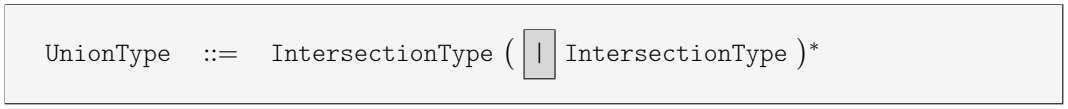
Example. The following example illustrates function types:

```
type Fun is function(int) => int

function map([int] items, Fun fn) => [int]:
    //
    for i,v in items:
        items[i] = fn(v)
    //
    return items
```

The above illustrates the well-known *map* function, which maps all elements of a list according to a given function.

4.11 Unions

A union type is constructed from two or more component types and contains any value held in any of its components. For example, the type `null|int` is a union which holds either an integer value or `null`. The set of values defined by a union type `T1|T2` is exactly the union of the sets defined by `T1` and `T2`. In general, variables of union type support only equality (§6.7), inequality comparisons (§6.7) and *runtime type tests* (see §4.15 for exceptions to this).



Example. The following example illustrates a union type:

```
// Return lowest index of matching item, or null if none
function indexOf([int] items, int value) => int|null:
    for i,v in items:
        if v == value:
            // match
            return i
    // item not found
    return null
```

Here, a union type is used to construct a more expressive return value. If no matching element is found, `null` is returned (rather than e.g. `-1`).

4.12 Intersections

An intersection type is constructed from two or more component types and contains any value held in all of its components. For example, the type `[int]&[bool]` is an intersection which hold any value which is both an instance of `[int]` and `[bool]` (in fact, only the empty list meets this criteria). Intersections are used to type variables on the true branch of a runtime type test. The set of values defined by an intersection type `T1&T2` is exactly the intersection of the sets defined by `T1` and `T2`. In general, variables of intersection type support only equality (§6.7), inequality comparisons (§6.7) and *runtime type tests* (see §4.15 for exceptions to this).

```
IntersectionType  ::=  TermType ( & TermType )*
```

Example. The following example illustrates an intersection type:

```
type Reader is {
   method read(int) => [byte],
   ...
}
type Writer is {
   method write([byte]) => int,
   ...
}
type ReaderWriter is Reader & Writer
```

Here, the type `Reader` is defined as any record containing a `read(int)` method, whilst the type `Writer` is defined as any record containing a `write([byte])` method. Then, the intersection type `ReaderWriter` is defined as any record containing *both* a `read(int)` and `write([byte])` method.

4.13 Negations

A negation type is constructed a component type and contains any value *not* held in its component. For example, the type `!int` is a negation which holds any non-integer value. Negations are used to type variables on the false branch of a runtime type test. The set of values defined by a negation type $!T_1$ is exactly the set of all values less those defined by T_1. In general, variables of negation type support only equality (§6.7), inequality comparisons (§6.7) and *runtime type tests* (see §4.15 for exceptions to this).

```
NegationType  ::=  ! TermType
```

Example. The following example illustrates a negation type:

```
function f(any item) => !null:
    if item is null:
        return 0
    else:
        return item
```

Here, the function `f()` accepts a parameter of any type, and returns a value which is permitted to be anything except `null`. The above also illustrates how the type test operator (§**??**) retypes variables on the false branch using negation types.

4.14 Recursive Types

Recursive types describe tree-like structures of arbitrary depth. For example, linked lists, binary trees, quad trees, etc can all be described using recursive types. Recursive types have no explicit syntax and, instead, are declared indirectly in terms of themselves using one or more nominal types (§4.8).

Example. The following example illustrates a simple recursive type:

```
type Node is { Tree left, Tree right, int data }
type Tree is null | Node

function sizeOf(Tree t) => int:
    if t == null:
        return 0
    else:
        return 1 + sizeOf(t.left) + sizeOf(t.right)
```

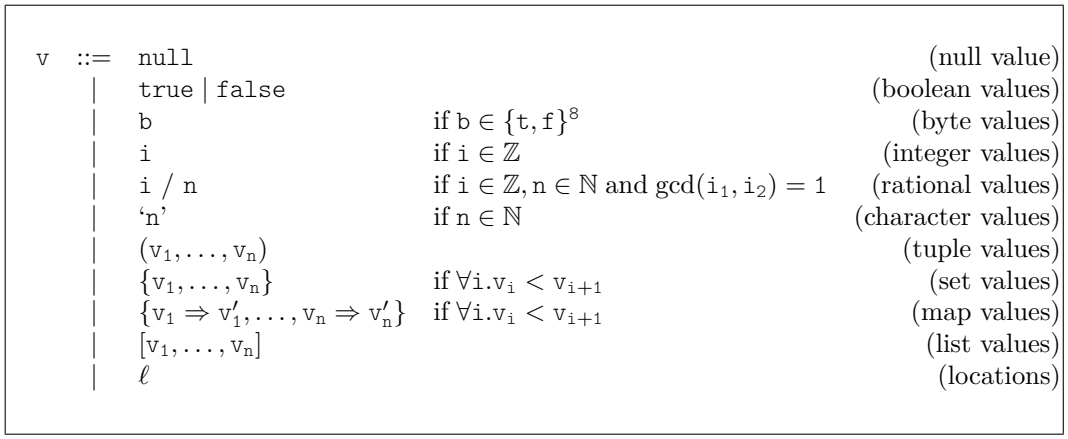
Here, the type `Tree` is recursive because it is defined in terms of itself. An instance of type `Tree` is a sequence of nested records which is arbitrarily deep, and whose branches are terminated by `null`. The function `sizeOf()` traverses an arbitrary instance of `Tree` and returns the number of `Node`s it contains.

4.15 Effective Types

An effective type is a union of types which all contain some property (e.g. a union of lists). This common property allows the effective type to support more operations than possible for an arbitrary union (§4.11).

4.15.1 Effective Tuples

An effective tuple is a union of tuple types. For example, `(int,int)|(real,real)` is an effective tuple. An effective tuple type supports all operations valid for a tuple type (§4.5).



v	::=	null	(null value)
	|	true | false	(boolean values)
	|	b if b ∈ {t, f}[8]	(byte values)
	|	i if i ∈ ℤ	(integer values)
	|	i / n if i ∈ ℤ, n ∈ ℕ and gcd(i1, i2) = 1	(rational values)
	|	'n' if n ∈ ℕ	(character values)
	|	(v1, ..., vn)	(tuple values)
	|	{v1, ..., vn} if ∀i.vi < vi+1	(set values)
	|	{v1 ⇒ v'1, ..., vn ⇒ v'n} if ∀i.vi < vi+1	(map values)
	|	[v1, ..., vn]	(list values)
	|	ℓ	(locations)

Figure 4.1: The language of abstract values used to formalise the meaning of types in Whiley, where $\mathbb{Z}$ is the (infinite) set of integers, $\mathbb{N}$ the (infinite) set of naturals and gcd() returns the Greatest Common Divisor of two values (e.g. using Euclid's well-known algorithm).

4.15.2 Effective Records

An effective record is a union of record types. For example, `{int f, int g}|{real f, int h}` is an effective record. An effective record provides access to fields common to all records in the union. For example, the type `{int f, int g}|{real f, int h}` can be viewed as having an effective type of `{int|real f, ...}` and, hence, read access to field `f` is given.

4.15.3 Effective Collections

An effective collection is a union of collection types. For example, `[int]|[real]` is an effective list. An effective collection supports all operations valid for a collection type (§4.9). For example, the type `[int]|[real]` can be viewed as having an effective type of `[int|real]` and, hence, read access to its length and elements is given.

4.16 Semantics

Although types are abstract entities we can (for the most part) imagine them as describing sets of *abstract values*. For example, `int|null` denotes the set of values containing exactly the (infinite) set of integers and `null` (i.e. $\mathbb{Z} \cup \{\texttt{null}\}$). This is often referred to as a set-theoretic interpretation of types[? ? ? ?]. Under this interpretation, for example, one type *subtypes* another if the set of values it denotes is a *subset* of the other (see § 4.16.2 for more).

We specify the meaning of types by formalising a set theoretic interpretation of them over the language of values given in Figure 4.1. To minimise confusion, care is taken in the figure to ensure that abstract values are represented canonically. For example, "`2 / 4`" is not a valid abstract value since "`1 / 2`" is its canonical representation. Likewise, "$\{2, 1\}$" is not a valid abstract value, with "$\{1, 2\}$" being its canonical representation. Figure 4.1 separates abstract values into distinct categories (e.g. integers, rationals, tuples, etc). These distinctions are significant. For example, "`0`" is distinct from "`0 / 0`". Similarly, byte values are not expressed using the digits `0` and `1` (as might be expected), but in terms of the characters `t` and `f`. This ensures binary values are distinct from integer values.

How zero represented?

An evaluation function $[\![\texttt{T}]\!]$ is defined which returns the set of values associated with a type `T`. For example, $[\![\texttt{bool}]\!] = \{\texttt{true}, \texttt{false}\}$, $[\![\texttt{int}]\!] = \mathbb{Z}$, etc. This function is defined as follows:

Definition 1 (Type Semantics) *Every type descriptor* $\mathtt{T}$ *is characterized by the set of values it accepts, given by* $[\![\mathtt{T}]\!]$ *and defined as follows:*

$$
\begin{array}{rcl}
[\![\mathtt{any}]\!] & = & \mathbb{D} \\
[\![\mathtt{void}]\!] & = & \emptyset \\
[\![\mathtt{null}]\!] & = & \{\mathtt{null}\} \\
[\![\mathtt{bool}]\!] & = & \{\mathtt{true}, \mathtt{false}\} \\
[\![\mathtt{byte}]\!] & = & \{\ \mathtt{b} \mid \mathtt{b} \in \{\mathtt{t}, \mathtt{f}\}^8\ \} \\
[\![\mathtt{int}]\!] & = & \mathbb{Z} \\
[\![\mathtt{real}]\!] & = & \{\mathtt{v_n}/\mathtt{v_d} \mid \mathtt{v_n} \in \mathbb{Z}, \mathtt{v_d} \in \mathbb{Z}\} \\
[\![\mathtt{char}]\!] & = & \mathbb{Z} \\
[\![(\mathtt{T_1}, \ldots, \mathtt{T_n})]\!] & = & \{\mathtt{v_1}, \ldots, \mathtt{v_n} \mid \mathtt{v_1} \in [\![\mathtt{T_1}]\!], \ldots, \mathtt{v_n} \in [\![\mathtt{T_n}]\!]\} \\
[\![\mathtt{T_1}|\ldots|\mathtt{T_n}]\!] & = & [\![\mathtt{T_1}]\!] \cup \ldots \cup [\![\mathtt{T_n}]\!] \\
[\![\mathtt{T_1}\&\ldots\&\mathtt{T_n}]\!] & = & [\![\mathtt{T_1}]\!] \cap \ldots \cap [\![\mathtt{T_n}]\!] \\
[\![!\mathtt{T_1}]\!] & = & \mathbb{D} - [\![\mathtt{T_1}]\!]
\end{array}
$$

Here, the *domain* of all possible values is given by $\mathbb{D}$, whilst the set of all integers is given by $\mathbb{Z}$. Furthermore, if $\mathtt{T}$ is a type descriptor, then $[\![\mathtt{T}]\!]$ is its underlying type.

4.16.1 Equivalences

Since types are defined in terms of the set of values they represent, it is perfectly possible for two distinct type descriptors to describe the same underlying type. For example, `int|null` is considered equivalent to `null|int`. Whilst this case is fairly easy to spot, there are some cases which are not so obvious. The definition of equivalence is given as follows:

Definition 2 (Type Equivalence) *Two type descriptors* $\mathtt{T_1}$ *and* $\mathtt{T_2}$ *are said to be* equivalent*, denoted by* $\mathtt{T_1} \equiv \mathtt{T_2}$*, iff* $[\![\mathtt{T_1}]\!] = [\![\mathtt{T_2}]\!]$.

Based on the above definition, we identify a number of such equivalences to illustrate:

- `!any` is equivalent to `void` and, conversely, `any` is equivalent to `!void`
- `int&!int` is equivalent to `void` and, conversely, `int|!int` is equivalent to `any`
- `{int | null f}` is equivalent to `{int f} | {null f}`
- `{int | null f}&{bool | null f}` is equivalent to `{null f}`

Under Definition 2, an infinite number of equivalences exist between the type descriptors of Whiley, and we cannot list them all here.

4.16.2 Subtyping

Types in Whiley support the notion of *subtyping* where one type may be a *subtype* for another. For example, the type `int` is a subtype of `any`. Likewise, `bool` is a subtype of `bool|null`. The *subtyping operator* is denoted by "$\leq$"; for example, $\mathtt{T_1} \leq \mathtt{T_2}$ indicates that type $\mathtt{T_1}$ is a subtype of $\mathtt{T_2}$. The subtyping operator is *reflexive*, *transitive* and *anti-symmetric* with respect to the underlying types involved.

The subtyping operator is regarded as an algorithm for determining whether the type described by one type descriptor is a subtype of another. The implementation of this algorithm is not straightforward and a full discussion of it is beyond the scope of this document. Indeed, there are many possible implementations of this operator. Nevertheless, there any valid implementation of this operator must exhibit two desirable properties:

Definition 3 (Subtype Soundness) *A subtype operator,* $\leq$*, is* sound *if, for any types* $\mathtt{T_1}$ *and* $\mathtt{T_2}$*, it holds that* $\mathtt{T_1} \leq \mathtt{T_2} \implies [\![\mathtt{T_1}]\!] \subseteq [\![\mathtt{T_2}]\!]$.

Definition 4 (Subtype Completeness) *A subtype operator, $\leq$, is* complete *if, for any types $\mathtt{T}_1$ and $\mathtt{T}_2$, it holds that $[\![\mathtt{T}_1]\!] \subseteq [\![\mathtt{T}_2]\!] \implies \mathtt{T}_1 \leq \mathtt{T}_2$.*

A subtype operator which exhibits both of these properties is said to be *sound* and *complete*.

Chapter 5

Statements

The execution of a Whiley program is controlled by *statement*s, which cause effects on the environment. However, statements in Whiley do not produce values. *Compound statement* statements may contain other statements.

5.1 Blocks

A statement block is a sequence of zero or more consecutive statements which have the same indentation. Statement blocks are used to group statements together when constructing compound statements. For example:

```
function sum([int] items) => int:
    // outer block begins
    int r = 0
    int i = 0
    while i < |items|:
        // inner block begins
        r = r + items[i]
        i = i + 1
        // inner block ends
    //
    return r
    // outer block ends
```

The above example contains two statement blocks, one nested inside the other. The outer block demarcates the body of the `sum()` function, whilst the inner block demarcates the body of the **`while`** statement.

5.2 Assert Statement

Represents an *assert statement* of the form "**`assert`** `e`", where `e` is a boolean expression. A *fault* will be raised at runtime if the asserted expression evaluates to `false`; otherwise, execution will proceed normally. At verification time, the verifier is forced to ensure that the asserted expression is true for all possible execution paths. This allows the programmer to specify and check something he/she believes to be true at a given point in the program.

```
AssertStmt  ::=  assert Expr
```

Example. The following illustrates an **assert** statement:

```
function abs(int x) => int:
    if x < 0:
        x = -x
    assert x >= 0
    return x
```

Here, an assertion is used to check that the value being returned by the `abs()` is non-negative. Since this is a true statement of the function, this statement will never raise a fault.

5.3 Assignment Statement

As *assignment statement* is of the form `leftHandSide = rightHandSide`. Here, the `rightHandSide` is any expression, whilst the `leftHandSide` must be an `LVal` — that is, an expression permitted on the left-hand side of an assignment. At runtime, the value generated by evaluating the right-hand side must be a subtype (§4.16.2) of the left-hand side.

```
AssignStmt  ::=  LVal [=] Expr
```

Example. The following illustrates different possible assignment statements:

```
x = y        // variable assignment
x.f = y      // field assignment
x[i] = y     // list assignment
x[i].f = y   // compound assignment
```

The last assignment here illustrates that the left-hand side of an assignment can be arbitrarily complex, involving nested assignments into lists and records.

5.4 Assume Statement

An *assume statement* is of the form "`assume e`", where `e` is a boolean expression. A fault will be raised at runtime if the assumed expression evaluates to `false`; otherwise, execution will proceed normally. At verification time, the verifier will automatically assume that the given expression holds. Thus, `assume` statements provide a way for the programmer to override the verifier. This is useful where the verifier is unable to establish something that the programmer knows to be true. Care must be taken to ensure that the assumed expression really does hold.

```
AssumeStmt  ::=  [assume] Expr
```

Example. The following illustrates an `assume` statement:

```
function abs(int x) => (int y) ensures y >= 0:
    //
    assume x >= 0
    return x
```

Here, the programmer has used an assumption to ensure this function passes verification. This would not appear to be safe in this case, and may lead to a fault at runtime.

5.5 Break Statement

A *break statement* transfers control out of the enclosing loop (i.e. **`do`**, **`for`**, **`while`**). It is a compile-time error if no such enclosing loop exists.

```
BreakStmt  ::=  break
```

Example. The following illustrates a **`break`** statement:

```
// Remove lowest element holding x from xs
function remove([int] xs, int x) => [int]:
    int i = 0
    while i < |xs|:
        if xs[i] == x:
           break
        else:
           i = i + 1
    return xs[0..i] ++ xs[i+1..]
```

Here, we see a **`break`** statement being used to exit a **`while`** loop when the first element matching parameter `x` is found.

Notes. Unlike many other programming languages (e.g. Java), **`break`** statements cannot be used to transfer control out of a **`switch`** statement (§5.14). This is because **`switch`** statements have *explicit*, rather than *implicit*, fall-through.

5.6 Continue Statement

A *continue statement* can be used either to transfer control to the next iteration of the enclosing loop (i.e. **`do`**, **`for`**, **`while`**), or to transfer control to the next case of the enclosing **`switch`** statement.

```
ContinueStmt  ::=  continue
```

Example. The following illustrates a **`continue`** statement:

```
function sumNonNegative([int] xs) => int:
    int i = 0
    int r = 0
    for i in 0 .. |xs|:
        if xs[i] < 0:
            continue
        r = r + xs[i]
    return r
```

Here, a **`continue`** statement is used to ensure the negative numbers are not included in the result of the function.

Notes. Unlike many other programming languages (e.g. Java), **`continue`** statements are used to transfer control to the next case of a **`switch`** statement (§5.14). This is because **`switch`** statements have *explicit*, rather than *implicit*, fall-through.

5.7 Debug Statement

A *debug statement* outputs the result of evaluating its expression to the *debug stream*. Debug statements are intended to be used purely for debugging, particularly from within (pure) functions. The debug stream is an imaginary output stream which does not exist in the true semantic of the language. Instead, from an operational semantics perspective, the debug statement is equivalent to the skip statement (§5.13).

```
DebugStmt ::= debug Expr
```

Example. The following illustrates a `debug` statement:

```
function f(int x) => int:
    debug "f(" ++ x ++ ")-called"
    if x == 1 || x == 0:
       return x
    else:
       return f(x-1) + f(x-2)
```

Here, we see a recursive implementation of the well-known *fibonacci* sequence. A `debug` statement is being used to investigate the parameter values passed to the function.

5.8 Do/While Statement

A do-while statement repeatedly executes a statement block until an expression (the condition) evaluates to `false`. Optional **`where`** clause(s) are permitted which, together, are commonly referred to as the loop invariant.

```
DoWhileStmt^ℓ ::= do : Block^γ while Expr ( where Expr )*
                                                (where ℓ < γ)
```

Example. The following illustrates an do-while statement:

```
function sum([int] xs) => int
// Input must not be empty list
requires |xs| > 0:
  //
  int r = 0
  int i = 0
  do:
    r = r + xs[i]
    i = i + 1
  while i < |xs| where i >= 0:
  //
  return r
```

Here, we see a simple do-while statement which sums the elements of variable `xs`, storing the result in variable `r`. A loop invariant is given which establishes that variable `i` is non-negative.

Notes. When multiple **`where`** clauses are given, these are combined using a conjunction to form the loop invariant. The combined invariant must hold on entry to the loop and after each iteration. Thus, when the condition evaluates to `false`, the loop invariant is guaranteed to hold. However, the loop invariant need not hold when the loop is exited using a **`break`** (§5.5) statement.

5.9 For Statement

A *for statement* iterates over all elements in a collection obtained from evaluating the *source expression*. Optional **`where`** clause(s) are permitted which, together, are commonly referred to as the loop invariant.

$$\texttt{ForStmt}^{\ell} \quad ::= \quad \boxed{\texttt{for}}\ \texttt{VarPattern}\ \boxed{\texttt{in}}\ \texttt{Expr}\ \big(\ \boxed{\texttt{where}}\ \texttt{Expr}\ \big)^{*}\ \boxed{\texttt{:}}\ \texttt{Block}^{\gamma} \qquad (\text{where } \ell < \gamma)$$

Example. The following illustrates a **`for`** statement:

```
function max([int] items) => int
// Input list cannot be empty
requires |items| > 0:
   //
   int r = items[0]

   for i,v in items:
        r = Math.max(r,v)

   return v
```

Here, we see a simple **`for`** loop which iterates over all elements of the list `items`. At each iteration, variable `i` holds the index whilst `v` contains the element at that index (i.e. `v == items[i]`).

Notes. When multiple **`where`** clauses are given, these are combined using a conjunction to form the loop invariant. The combined invariant must hold on entry to the loop and after each iteration. Thus, when the condition evaluates to `false`, the loop invariant is guaranteed to hold. However, the loop invariant need not hold when the loop is exited using a **`break`** (§5.5) statement.

5.10 If Statement

An **`if`** statement conditionally executes a statement block based on the outcome of one or more expressions. Chaining of **`if`** statements is permitted, and an optional **`else`** branch may be given. The expression(s) are referred to as *conditions* and must be boolean expressions. The first block is referred to as the *true branch*, whilst the optional **`else`** block is referred to as the *false branch*.

$$\texttt{IfStmt}^{\ell} \quad ::= \quad \boxed{\texttt{if}}\ \texttt{Expr}\ \boxed{\texttt{:}}\ \texttt{Block}^{\gamma}\ \big(\ \boxed{\texttt{else}}\ \boxed{\texttt{if}}\ \texttt{Expr}\ \boxed{\texttt{:}}\ \texttt{Block}^{\omega_i}\ \big)^{*}\ \big[\ \boxed{\texttt{else}}\ \boxed{\texttt{:}}\ \texttt{Block}^{\phi}\ \big]$$

(where $\ell < \gamma$ and $\forall i.\ell < \omega_i$ and $\ell < \phi$)

Example. The following illustrates an **if** statement:

```
function max(int x, int y) => int:
    if(x > y):
        return x
    else if(x == y):
        return 0
    else:
        return y
```

Here, we see an **if** statement with two conditional outcomes and one default outcome.

5.11 While Statement

A while statement repeatedly executes a statement block until an expression (the condition) evaluates to `false`. Optional **where** clause(s) are permitted which, together, are commonly referred to as the loop invariant.

$$\texttt{WhileStmt}^{\ell} \quad ::= \quad \boxed{\texttt{while}}\ \texttt{Expr}\ \big(\ \boxed{\texttt{where}}\ \texttt{Expr}\ \big)^{*}\ \boxed{\texttt{:}}\ \texttt{Block}^{\gamma}$$

(where $\ell < \gamma$)

Example. The following illustrates an **while** statement:

```
function sum([int] xs) => int:
  int r = 0
  int i = 0
  while i < |xs| where i >= 0:
    r = r + xs[i]
    i = i + 1
  return r
```

Here, we see a simple **while** statement which sums the elements of variable `xs`, storing the result in variable `r`. A loop invariant is given which establishes that variable `i` is non-negative.

Notes. When multiple **where** clauses are given, these are combined using a conjunction to form the loop invariant. The combined invariant must hold on entry to the loop and after each iteration. Thus, when the condition evaluates to `false`, the loop invariant is guaranteed to hold. However, the loop invariant need not hold when the loop is exited using a **break** (§5.5) statement.

5.12 Return Statement

A *return statement* has an optional expression referred to as the *return value*. At runtime, this statement returns control to the caller of the enclosing function or method. At verification time, the verifier will ensure the returned value meets the postcondition of the enclosing function or method.

```
ReturnStmt  ::=  return [ Expr ]
```

Example. The following illustrates a `return` statement:

```
function f(int x) => int:
    return x + 1
```

Here, we see a simple simple function which returns the increment of its parameter `x` using a `return` statement.

Notes. The returned expression (if there is one) must begin on the same line as the statement itself.

5.13 Skip Statement

A *skip statement* is a no-operation and has no effect on the environment. This statement can be useful for representing empty statement blocks (§5.1).

```
SkipStmt  ::=  skip
```

Example. The following illustrates a `skip` statement:

```
function abs(int x) => (int y)
// Return value cannot be negative
ensures y >= 0:
    //
    if x >= 0:
        skip
    else:
        x = -x
    //
    return x
```

Here, we see a `skip` statement being used to represent an empty statement block.

5.14 Switch Statement

A *switch statement* transfers control to one of several statement blocks, referred to as *switch cases*, depending on the value obtained from evaluating a given expresion. Each case is associated with one or more values which are used to match against. If no match is made, control either falls through to the next statement following the `switch` or is transferred to a `default` block if one is given.

```
SwitchStmt^ℓ  ::=  switch Expr : ( CaseBlock^γ | DefaultBlock^γ ) +
CaseBlock^ℓ   ::=  case ConstantExpr ( , ConstantExpr )* : Block^γ
DefaultBlock^ℓ ::=  default : Block^γ
                                                    (where ℓ < γ)
```

Example. The following illustrates a **switch** statement:

```
function toDescriptorString(JvmType.Primitive t) => string:
    switch t:
        case JvmType.Boolean:
            return "Z"
        case JvmType.Byte:
            return "B"
        case JvmType.Char:
            return "C"
        case JvmType.Short:
            return "S"
        case JvmType.Int:
            return "I"
        case JvmType.Long:
            return "J"
        case JvmType.Float:
            return "F"
        default:
            return "D"
```

Here, we see a simple **switch** statement which choose between a number of possible values of type `JvmType.Primitive`. A **default** case is given which catches the only remaining case (i.e. representing the value `JvmType.Double`).

5.15 Throw Statement

A *throw statement* causes an exception to thrown which, if not caught locally, causes an *abrupt termination* of the current function or method. Functions or methods which may terminate abruptly must declare appropriate an throws clause (§3.5.4,§3.5.5) which contains al potentially thrown exceptions.

```
ThrowStmt  ::=  throw Expr
```

Example. The following illustrates a `throw` statement:

```
function parseInt(int pos, string input) => (int,int)
// Throws a syntax error if the string is malformed
throws SyntaxError:
    //
    int start = pos
```

```
    // check for negative input
    if pos < |input| && input[pos] == '-':
        pos = pos + 1
    // match remainder
    while pos < |input| && Char.isDigit(input[pos]):
        pos = pos + 1
    // check for error
    if pos == start:
        throw SyntaxError("Missing number",start,pos)
    // done
    return Int.parse(input[start..pos]),pos
```

Here, we see a function which parses a string into an integer. The function declares that a `SyntaxError` may be thrown. This is required for two reasons: firstly, the input contains no digits then an `SyntaxError` is thrown by this function. Additionally, the function `Int.parse()` is declared to throw a `SyntaxError` and, since it is not caught, this declaration must be propagated.

5.16 Try Statement

A *try* statement demarcates a statement block to be executed in the context of one or more *exception handlers*. If an uncaught exception is raised within the block which matches one (or more) of the exception handlers, then control is transferred directly to that handler. Exception handlers are matched against raised exceptions in the order of declaration.

$$\texttt{TryStmt}^{\ell} \quad ::= \quad \boxed{\texttt{try}}\ \boxed{\texttt{:}}\ \texttt{Block}^{\gamma}\ \Big(\ \boxed{\texttt{catch}}\ \boxed{\texttt{(}}\ \texttt{Type Ident}\ \boxed{\texttt{)}}\ \boxed{\texttt{:}}\ \texttt{Block}^{\omega_i}\ \Big)^{+}$$

$$(\text{where } \ell < \gamma \text{ and } \forall i.\ell < \omega_i \text{ and } \ell < \phi)$$

Example. The following illustrates a `try` statement:

```
function parse(string input) => int|string|null
// Input cannot be empty
requires |input| > 0:
    //
    try:
        if Char.isDigit(input[0]):
            // must be integer
            return Int.parse(input)
        else:
            // must be string
            return input
    catch(SyntaxError ex):
        // We can get here from Int.parse()
        return null
```

Here, we see a function which parses a string as an integer (if it begins with a digit) or returns the input string (otherwise). The function `Int.parse(string)` throws a `SyntaxError` in the case that its parameter is not well-formed. A `try` statement is used to catch this and return `null` in such case.

Notes. Exceptions in Whiley differ from those found in other languages (e.g. Java) as they do not include runtime errors (e.g. divide-by-zero, out-of-bounds access, out-of-memory,

stack-overflow, etc), Instead, all exceptions are explicitly thrown using a `throw` statement. In contrast, runtime errors correspond to faults in Whiley, and are thrown implicitly when an unrecoverable error occurs.

5.17 Variable Declaration Statement

A *variable declaration* statement has an optional expression assignment referred to as a *variable initialiser*. If an initialiser is given, this will be evaluated and assigned to the declared variables when the declaration is executed.

```
VarDecl  ::=  TypePattern [ = Expr ]
```

Example. Some example variable declarations are:

```
int x
int y = 1
int z = x + y
int a, int b = x,y
```

Here we see four variable declarations. The first has no initialiser, whilst the remainder have initialisers. The final declaration illustrates a more complex use of type patterns where two variables of type `int` are initialised from a tuple expression

Chapter 6

Expressions

The majority of work performed by a Whiley program is through the execution of *expressions*. Every expression produces a *value* and may have additional side effects.

6.1 Abrupt Execution

6.2 Evaluation Order

6.3 Multi Expressions

A multi-expression is an expression composed of two or more unit expressions and which returns a tuple value (§4.5). The operands of a multi-expression are evaluated in a strict left-to-right order.

```
TupleExpr  ::=  UnitExpr ( , UnitExpr )+
```

Example. The following example illustrates the use of a multi-expression:

```
function scale(real x, real y, real p) => (real,real):
    return (x*p), (y*p)
```

This function accepts three `real` values and returns two. In the body, the `return` statement contains a multi-expression which produces a tuple composed of two `real` values.

6.4 Unit Expressions

A unit expression is an expression which returns exactly one value. There is a large range of possible unit expressions, including comparators, arithmetic operators, logical operators, etc.

```
UnitExpr  ::=  LogicalExpr
            |  BitwiseExpr
            |  ConditionExpr
            |  QuantifierExpr
            |  AppendExpr
            |  RangeExpr
            |  ShiftExpr
            |  AdditiveExpr
            |  MultiplicativeExpr
            |  AccessExpr
            |  TermExpr
```

6.5 Logical Expressions

A logical expression operates on values of **bool** type (§4.4.2) to produce another **bool** value. The *if-and-only-if (iff)* operator, `<==>`, returns `true` if either both operands are `true` or both are `false`. The *implication* operator, `==>`, returns `true` if either the left operand is `false`, or both operands are `true`. The *logical OR* operator returns `true` if either operand is `true`, whilst the *logical AND* operator returns `true` if both operands are `true`.

```
    LogicalExpr  ::=  LogicalOrExpr [ <==> LogicalExpr ]
                   |  LogicalOrExpr [ ==> LogicalExpr ]

  LogicalOrExpr  ::=  LogicalAndExpr
                   |  LogicalOrExpr || LogicalAndExpr

 LogicalAndExpr  ::=  BitwiseExpr
                   |  LogicalAndExpr && BitwiseExpr
```

Example. The following examples illustrate some of the logical operators:

```
function implies(bool x, bool y) => bool:
    return !x || y

function iff(bool x, bool y) => bool:
    return implies(x,y) && implies(y,x)
```

The function `implies()` implements the well-known equivalence between implication and logical OR. The function `iff()` implements the well-known equivalence between implication and iff.

6.6 Bitwise Expressions

A bitwise expression operates on values of **byte** type (§4.4.3). The *bitwise OR* operator, `|`, performs a logical OR between the respective bits of each operand, and produces a **byte**. The *bitwise AND* operator, `&`, performs a logical AND between the respective bits of each

operand, and produces a **byte**. The *bitwise exclusive-OR* operator, ^, performs a logical exclusive-OR between the respective bits of each operand, and produces a **byte**.

```
    BitwiseExpr  ::=  BitwiseOrExpr

  BitwiseOrExpr  ::=  BitwiseXorExpr
                  |   BitwiseOrExpr | BitwiseXorExpr

 BitwiseXorExpr  ::=  BitwiseAndExpr
                  |   BitwiseXorExpr ^ BitwiseAndExpr

 BitwiseAndExpr  ::=  ConditionExpr
                  |   BitwiseAndExpr && ConditionExpr
```

Example. The following example illustrates the bitwise OR operator:

```
function toUnsignedByte(u8 v) => byte:
    //
    byte mask = 00000001b
    byte r = 0b
    for i in 0..8:
        if (v % 2) == 1:
            r = r | mask
        v = v / 2
        mask = mask << 1
    return r
```

This function converts an unsigned integer in the range `0` ... `255` to a **byte**. The bitwise OR operator is used to construct the resulting **byte** by setting individual bits via the `mask`. This example also illustrates the left-shift operator (§6.11).

6.7 Condition Expressions

```
 ConditionExpr  ::=
```

Description.

Examples.

Notes.

6.8 Quantifier Expressions

```
QuantExpr  ::=  ( no | some | all ) {
                Ident in Expr ( , Ident in Expr )+ | LogicalExpr
                }
```

Description.

Examples.

Notes.

6.9 Append Expressions

```
AppendExpr  ::=  RangeExpr ( ++ RangeExpr )*
```

Description.

Examples.

Notes.

6.10 Range Expressions

```
RangeExpr  ::=  ShiftExpr [ .. ShiftExpr ]
```

Description.

Examples.

Notes.

6.11 Shift Expressions

```
ShiftExpr  ::=  AdditiveExpr [ ( « | » ) AdditiveExpr ]
```

Description.

Examples.

Notes.

6.12 Additive/Multiplicative Expressions

```
      AdditiveExpr  ::=
MultiplicativeExpr  ::=
```

Description.

Examples.

Notes.

6.13 Access Expressions

```
AccessExpr  ::=
```

Description.

Examples.

Notes.

6.14 Term Expressions

```
TermExpr  ::=
```

Description.

Examples.

Notes.

6.15 Dereference Expressions

The dereference operation "`e→f`" is a short-hand notation for "`(*e).f`" and can be used when `e` has effective record type (§4.15.2).

Chapter 7

Flow Typing

The Whiley programming language is *statically typed*, meaning that every expression has a type determined at compile time. Furthermore, evaluating an expression is guaranteed to yield a value of its type. Whiley's *type system* governs how the type of any variable or expression is determined. Whiley's type system is unusual in that it operates in a *flow-sensitive* manner allowing variables to have different types at different program points.

Chapter 8

Verification

The Whiley programming language supports *specifications* on functions, methods and data types which can be *statically verified* at compile time. Verification operates in an intra-procedural fashion based on a modified and extended version of Hoare logic [?]. To benefit from verification, programmers must provide specifications for their functions, methods and data types; additionally, they must provide loop invariants and other *assertion*s to guide the verifier.