The Whiley Language Specification

David J. Pearce
School of Engineering and Computer Science
Victoria University of Wellington, New Zealand
djp@ecs.vuw.ac.nz

January 16, 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 fewer errors that those developed by more convention means. 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 [1]; the 1988 worm which reeked havoc on the internet by exploiting a buffer overrun [2]; the 1991 Patriot missile failure where a rounding error resulted in the missile catastrophically hitting a barracks [3]; and, the Ariane 5 rocket which exploded shortly after launch because of an integer overflow, costing the ESA an estimated $500 million [4].

Around 2003, Hoare proposed the creation of a *verifying compiler* as a grand challenge for computer science [5]. 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 [6], Deutsch [7], the Gypsy Verification Environment [8] and the Stanford Pascal Verifier [9]. More recently, the Extended Static Checker for Modula-3 [10] which became the Extended Static Checker for Java (ESC/Java) — a widely acclaimed and influential work [11]. Building on this success was JML and its associated tooling which provided a standard notation for specifying functions in Java [12]. Finally, Microsoft developed the Spec# system which is built on top of C# [13].

1.2 Goals

The Whiley Programming Language has been designed from scratch in conjunction with a verifying compiler. The intention of this 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

Chapter 2

Lexical Structure

2.1 Indentation

2.2 Blocks

2.3 Whitespace

2.4 Identifiers

Chapter 3

Source Files

Whiley programs are split across one or more *source file*s 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

3.2 Packages & Imports

3.3 Declarations

Camel case

3.3.1 Access Control

3.3.2 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 filess 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:

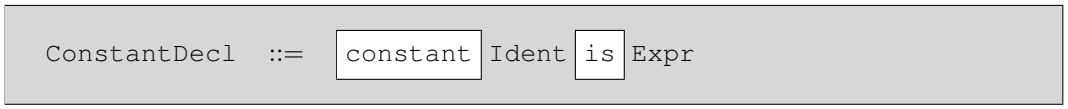
```
// Define a simple point type
type Point is { int x, int y }

// 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.

3.3.3 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 filess, although it may not refer to itself (either directly or indirectly).



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 example to illustrate constant declarations are:

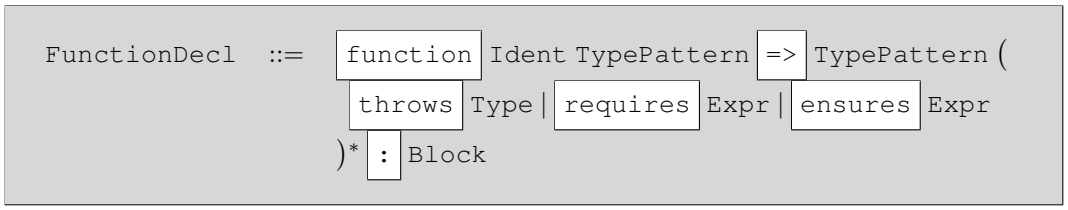
```
// Define the well-known mathematical constant to 10 decimal places.
constant PI is 3.141592654

// 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).

3.3.4 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 always 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. They also 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 the throws clause before the others.

- **Requires clause(s)**. These define constraints on the permissible values of the parameters on entry to the function or method, 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 ensure(s) clauses.
- **Ensures clause(s)**. These define constraints on the permissible values of the the function or method'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 the requires clause(s) after the others.

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.

3.3.5 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 functions above.

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):
    if *list is null:
```

```
        // reached the end of the list, so allocate new node
        *list = new { next: null, data: item }
    else:
        // continue traversing the list
        insertAfter(list→next, item)
```

Chapter 4

Types & Values

The Whiley programming language is *statically typed*, meaning that every expression has a type determined at compile time, such that any evaluation of that expression yields a value of that 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*, *intersection* and *negation types*, 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:

```
int|null id(null|int x):
    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.

4.1.1 Type 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 set-theoretic interpretation of types [14, 15, 16, 17]. Under this interpretation, for example, one type *subtypes* another if the set of values it denotes is a *subset* of the other.

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?

$$
\begin{array}{lcll}
\mathtt{v} & ::= & \mathtt{null} & & \text{(null value)} \\
& | & \mathtt{true} \mid \mathtt{false} & & \text{(boolean values)} \\
& | & \mathtt{b} & \text{if } \mathtt{b} \in \{\mathtt{t},\mathtt{f}\}^8 & \text{(byte values)} \\
& | & \mathtt{i} & \text{if } \mathtt{i} \in \mathbb{Z} & \text{(integer values)} \\
& | & \mathtt{i} \,/\, \mathtt{n} & \text{if } \mathtt{i} \in \mathbb{Z}, \mathtt{n} \in \mathbb{N} \text{ and } \gcd(\mathtt{i}_1, \mathtt{i}_2) = 1 & \text{(rational values)} \\
& | & \text{`}\mathtt{n}\text{'} & \text{if } \mathtt{n} \in \mathbb{N} & \text{(character values)} \\
& | & (\mathtt{v}_1, \ldots, \mathtt{v}_n) & & \text{(tuple values)} \\
& | & \{\mathtt{v}_1, \ldots, \mathtt{v}_n\} & \text{if } \forall \mathtt{i}.\mathtt{v}_i < \mathtt{v}_{i+1} & \text{(set values)} \\
& | & \{\mathtt{v}_1 \Rightarrow \mathtt{v}'_1, \ldots, \mathtt{v}_n \Rightarrow \mathtt{v}'_n\} & \text{if } \forall \mathtt{i}.\mathtt{v}_i < \mathtt{v}_{i+1} & \text{(map values)} \\
& | & [\mathtt{v}_1, \ldots, \mathtt{v}_n] & & \text{(list values)} \\
& | & \ell & & \text{(locations)}
\end{array}
$$

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).

Finally, an evaluation function $[\![\mathtt{T}]\!]$ is defined which returns the set of values associated with a type `T`. For example, $[\![\mathtt{bool}]\!] = \{\mathtt{true}, \mathtt{false}\}$, $[\![\mathtt{int}]\!] = \mathbb{Z}$, etc. In the remainder of this chapter, the body of this function will be give for each type as it is encountered.

4.1.2 Type Descriptors

As discussed above, 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  ::=  UnionType
             |  IntersectionType
             |  TermType

 TermType  ::=
             |  PrimitiveType
             |  TupleType
             |  RecordType
             |  ReferenceType
             |  NominalType
             |  CollectionType
             |  NegationType
             |  FunctionType
             |  MethodType
```

4.1.3 Type Patterns

In Whiley, *type pattern*s are used to associate variables with types and their subcomponents. For example, they 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
```

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.

4.1.4 Value Constructors

A *value* in Whiley may be stored in a variable of some kind (e.g. local variable, parameter, heap location, etc) and is the result of evaluating an expression. Every value is associated with a unique (i.e. most precise) *concrete type*. For example, the value `1` is associated with the type **`int`**, whilst the value $[1, 2.2]$ is associated with the type `[`**`int`**`|`**`real`**`]`. Is abstract?

Values are generated by evaluating *expressions* given in the program source. Values can be generated indirectly by performing operations on existing values, or directly by evaluating *constructor expressions*, whose top-level grammar is given below:

```
ConstructorExpr  ::=
                   |   NullExpr
                   |   BoolExpr
                   |   ByteExpr
                   |   IntExpr
                   |   RealExpr
                   |   CharExpr
                   |   TupleExpr
                   |   RecordExpr
                   |   SetExpr
                   |   MapExpr
                   |   ListExpr
                   |   NewExpr
```

4.2 Primitives

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

4.2.1 Null

The null type is a special type which should be 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). Variables of **`null`** type support only equality (`==`) and inequality comparisons (`!=`). The **`null`** value is particularly useful for representing optional values and terminating recursive types.

Examples. 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.

Semantics. The set of values defined by the type **`null`** is given as follows:

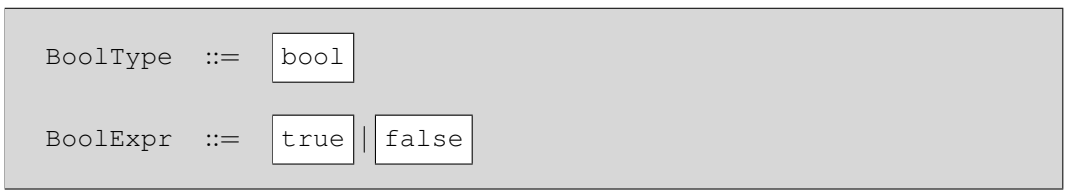
$$[\![\texttt{null}]\!] \;=\; \{\texttt{null}\}$$

In other words, the set of values defined by the **`null`** type is the singleton set containing exactly the **`null`** value.

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).

4.2.2 Booleans

The **`bool`** type represents the set of boolean values (i.e. `true` and `false`). Variables of **`bool`** type support equality (`==`), inequality (`!=`), logical and (`&&`), logical or (`||`), logical xor (`^`) and logical not (`!`).



Examples. 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`.

Semantics. The set of values defined by the type **`bool`** is given as follows:

$$[\![\texttt{bool}]\!] \quad = \quad \{\texttt{true}, \texttt{false}\}$$

In other words, the set of values defined by the **`bool`** type is the set containing exactly the values `true` and `false`.

4.2.3 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). Variables of **`byte`** type support equality (==), inequality (!=), bitwise and (&), bitwise or (|), bitwise xor (^), bitwise complement (~), left shift (<<) and right shift (>>).

```
ByteType  ::=  byte

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

Byte values 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 `00101b`).

Examples. 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.

Semantics. The set of values defined by the type **`byte`** is given as follows:

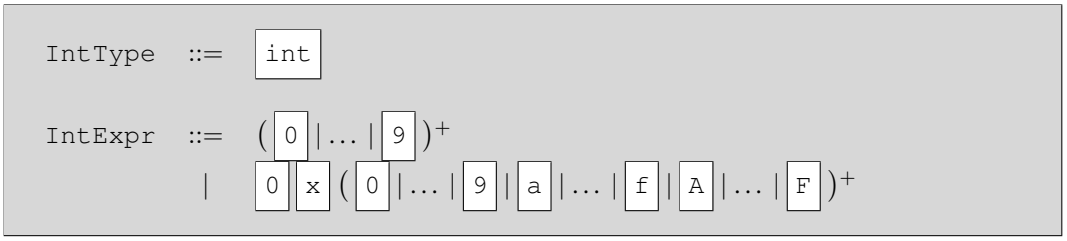
$$[\![\texttt{byte}]\!] \quad = \quad \{\ \texttt{b} \mid \texttt{b} \in \{\texttt{t}, \texttt{f}\}^8\ \}$$

In other words, the set of values defined by the **`byte`** type is the set of all 256 possible combinations of eight-bit sequences.

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()`).

4.2.4 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 (`==`), inequality (`!=`), comparators (`<`,`<=`,`>=`,`>`), addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), remainder (`%`) and negation (`-`) operations.



Examples. 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.

Semantics. The set of values defined by the type **`int`** is given as follows:

$$[\![\texttt{int}]\!] \;=\; \mathbb{Z}$$

In other words, the of values defined by the type **`int`** is exactly the (infinite) set of integers.

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).

4.2.5 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). Variables of **`real`** type support equality (`==`), inequality (`!=`), comparators (`<`,`<=`,`>=`,`>`), addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), remainder (`%`) and negation (`-`) operations. Variables of type **`real`** also support the *rational destructuring assignment* to extract the numerator and denominator (illustrated below).

```
RealType  ::=  real

RealExpr  ::=  ( 0 | ... | 9 )+ . ( 0 | ... | 9 )+
```

Examples. 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.

Semantics. The set of values defined by the type **real** is given as follows:

$$[\![\texttt{real}]\!] \;=\; \{\texttt{v}_\texttt{n}/\texttt{v}_\texttt{d} \mid \texttt{v}_\texttt{n} \in \mathbb{Z}, \texttt{v}_\texttt{d} \in \mathbb{Z}\}$$

In other words, the 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.

4.2.6 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 (==), inequality (!=), comparators (<,<=,>=,>), addition (+), subtraction (−), multiplication (*), division (/), remainder (%) and negation (−) operations.

```
CharType  ::=  char

CharExpr  ::=  ' . '
```

Examples. 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.

Semantics. The set of values defined by the type **int** is given as follows:

$$[\![\texttt{char}]\!] \;=\; \mathbb{Z}$$

In other words, the of values defined by the type **char** is exactly the (infinite) set of integers.

4.2.7 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 (==) and inequality comparisons (!=) as well as *runtime type tests*. Finally, unlike the majority of other types, there are no *values* of type **any**.

```
AnyType  ::=  any
```

Examples. 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`.

Semantics. The set of values defined by the type **any** is given as follows:

$$[\![\texttt{any}]\!] \;=\; \mathcal{D}$$

In other words, the set of values defined by the **any** type equals the *domain* (i.e. the set of all values).

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`.

4.2.8 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
```

Examples. 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*).

Semantics. The set of values defined by the type **`void`** is given as follows:

$$[\![\texttt{void}]\!] \quad = \quad \emptyset$$

In other words, the set of values defined by the **`void`** type equals the empty set.

4.3 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 (`==`) and inequality (`!=`) operations, as well the *tuple destructuring assignment* to extract elements (illustrated below).

```
    TupleType  ::=  ( Type ( , Type )+ )
TuplePattern   ::=  ( TypePattern ( , TypePattern )+ ) [ Ident ]
    TupleExpr  ::=  Expr ( , Expr )+
```

Examples. 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.

Semantics. The set of values defined by a tuple type is given as follows:

$$[\![(\texttt{T}_1,\ldots,\texttt{T}_n)]\!] \quad = \quad \{\texttt{v}_1,\ldots,\texttt{v}_n \mid \texttt{v}_1 \in [\![\texttt{T}_1]\!],\ldots,\texttt{v}_n \in [\![\texttt{T}_n]\!]\}$$

In other words, the set of values defined by the **`void`** type equals the empty set.

4.4 Records

A record is made up of a number of fields, each of which has a unique name. Each field has a corresponding type. One can think of a record as a special kind of ”fixed” map (i.e. where we know exactly which entries we have).

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

Examples.

Semantics.

Notes. Syntax for functions? Open versus closed records?

4.5 References

Represents a reference to an object in Whiley.

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

Examples.

Semantics.

Notes.

4.6 Nominals

The existential type represents the an unknown type, defined at a given position.

```
NominalType  ::=  Ident
```

Examples.

Semantics.

Notes.

4.7 Collections

4.7.1 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.

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

Examples.

Semantics.

Notes.

4.7.2 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}`.

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

Examples.

Semantics.

Notes.

4.7.3 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.

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

Examples.

Semantics.

Notes.

4.8 Functions

```
FunctionType  ::=  function ( [ Type ( , Type )* ] ) => Type
```

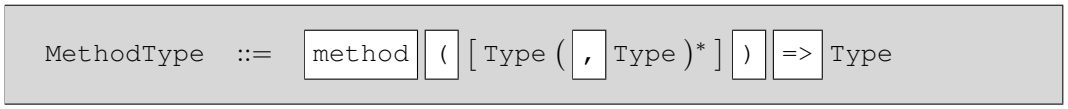
Description.

Examples.

Semantics.

Notes.

4.9 Methods



Description.

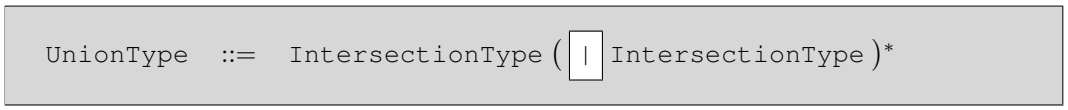
Examples.

Semantics.

Notes.

4.10 Unions

A union type represents a type whose variables may hold values from any of its ”bounds”. For example, the union type **`null`** | **`int`** indicates a variable can either hold an integer value, or **`null`**.

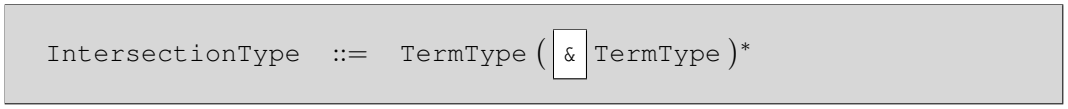


Examples.

Semantics.

Notes. There must be at least two bounds for a union type to make sense.

4.11 Intersections



Description.

Examples.

Semantics.

Notes.

4.12 Negations

Description. A negation type represents a type which accepts values *not* in a given type.

Examples.

Semantics.

Notes.

4.13 Abstract Types

4.13.1 Recursive Types

4.13.2 Effective Tuples

4.13.3 Effective Records

4.13.4 Effective Collections

4.14 Subtyping Algorithms

Discussion of soundness and completeness.

4.15 Equivalences

Discuss some obvious equivalences between types.

Chapter 5

Statements

5.1 Assert Statement

Represents an *assert statement* of the form "**assert** `e`", where `e` is a boolean expression.

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

Examples. The following illustrates:

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

Notes. Assertions are either *statically checked* by the verifier, or turned into *runtime checks*.

5.2 Assignment Statement

Represents an *assignment statement* of the form `lhs = rhs`. Here, the `rhs` is any expression, whilst the `lhs` must be an `LVal` — that is, an expression permitted on the left-side of an assignment.

```
AssignStmt  ::=  LVal = Expr
```

Examples. 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.

Semantics.

Notes.

5.3 Assume Statement

Represents an *assume statement* of the form "`assume e`", where `e` is a boolean expression.

```
AssumeStmt  ::=  assume Expr
```

Examples. The following illustrates a simple function which uses an `assume` statement to meet its postcondition:

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

Notes. Assumptions are *assumed* by the verifier and, since this may be unsound, are always turned into *runtime checks*.

5.4 Return Statement

Represents a *return statement* with an optional expression is referred to as the *return value*.

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

Examples. The following illustrates a simple function which returns the increment of its parameter `x`:

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

Here, we see a simple **`return`** statement which returns an **`int`** value.

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

5.5 Throw Statement

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

Description.

Examples.

Notes.

5.6 Variable Declarations

Represents a *variable declaration* which has an optional expression assignment referred to as an *variable initialiser*. If an initialiser is given, then this will be evaluated and assigned to the variable when the declaration is executed.

```
VarDecl ::= Type Ident [ = Expr ]
```

Examples. Some example variable declarations are:

```
int x
int y = 1
int z = x + y
```

Notes.

5.7 If Statement

Represents a classical **`if`** statement which supports chaining and an optional **`else`** branch. 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*.

```
IfStmt^ℓ ::= if Expr : Block^γ ( else if Expr : Block^ωi )*
             [ else : Block^ϕ ]

                                   (where ℓ < γ and ∀i.ℓ < ωi and ℓ < ϕ)
```

Examples. The following illustrates:

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

Notes.

5.8 While Statement

Represents a while statement with optional **`where`** clause(s) commonly referred to as loop invariants.

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

(where $\ell < \gamma$)

Examples. As an example:

```
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
```

Notes. When multiple **`where`** clauses are given, these are combined using a conjunction. The combined invariant defines a condition which must be true on every iteration of the loop.

5.9 Do/While Statement

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

(where $\ell < \gamma$)

Description.

Examples.

Notes.

5.10 For Statement

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

(where $\ell < \gamma$)

Description.

Examples.

Notes.

5.11 Switch Statement

```
SwitchStmt  ::=
```

Description.

Examples.

Notes.

5.12 Try/Catch Statement

```
TryCatchStmt  ::=
```

Description.

Examples.

Notes.

Chapter 6

Expressions

Expression blah blah.

6.1 Tuple Expressions

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

Description.

Examples.

Notes.

6.2 Unit Expressions

```
UnitExpr ::= LogicalExpr
```

Description.

Examples.

Notes.

6.3 Logical Expressions

```
    LogicalExpr  ::=  LogicalOrExpr

 LogicalOrExpr  ::=  LogicalAndExpr
                |    LogicalOrExpr || LogicalAndExpr

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

Description.

Examples.

Notes.

6.4 Bitwise Expressions

```
    BitwiseExpr  ::=  BitwiseOrExpr

 BitwiseOrExpr  ::=  BitwiseXorExpr
                |    BitwiseOrExpr | BitwiseXorExpr

BitwiseXorExpr  ::=  BitwiseAndExpr
                |    BitwiseXorExpr ^ BitwiseAndExpr

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

Description.

Examples.

Notes.

6.5 Condition Expressions

```
ConditionExpr  ::=
```

Description.

Examples.

Notes.

6.6 Quantifier Expressions

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

Description.

Examples.

Notes.

6.7 Append Expressions

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

Description.

Examples.

Notes.

6.8 Range Expressions

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

Description.

Examples.

Notes.

6.9 Shift Expressions

```
ShiftExpr ::= AdditiveExpr [ ( << | >> ) AdditiveExpr ]
```

Description.

Examples.

Notes.

6.10 Additive/Multiplicative Expressions

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

Description.

Examples.

Notes.

6.11 Access Expressions

```
AccessExpr  ::=
```

Description.

Examples.

Notes.

6.12 Term Expressions

```
TermExpr  ::=
```

Description.

Examples.

Notes.

Glossary

boolean expression An expression which evaluates to a value of type `bool`. 6, 23–25, 32

expression A combination of constants, variables and operators that, when evaluated, produce a single value. Expressions in certain circumstances may have side effects. 28, 32

loop invariant A boolean expression which must hold on every iteration of a loop. 7, 8, 26

package A unit of hierarchical organisation within the Whiley namespace.. 6

postcondition A logical condition over the parameters and returns of a function or method which must be true immediately after execution of that function or method.. 7, 8

precondition A logical condition over the parameters of a function or method which must be true immediately prior to execution of that function or method.. 7, 8

safety critical system A system which operates in a high-risk setting where failure can lead to loss of life, injury, significant damage or environmental harm. 3

source file A file in which source code is located. Source files for the Whiley programming language have the extension `.whiley`. In Whiley, source files must be compiled into a binary form before they can be executed.. 6–8, 32

type An abstract entity which represents the set of values a given variable may hold, or a given expression may evaluate to.. 10, 32

type descriptor A source-level description of an underlying type. Unlike many languages, type descriptors and types are quite distinct in Whiley as, for example, two distinct descriptors may describe the same underlying type. 10, 32

type pattern A source-level description of an underlying type (similar to a type descriptor) where one or more variables are associated with its subcomponent(s).. 11

variable declaration A statement which declares one or more variable(s) for use in a given scope. Each variable is given a type which limits the possible values it may hold, and may not already be declared in an enclosing scope. 25, 32

variable initialiser An optional expression used to initialise variable(s) declared as part of a variable declaration. 25

verifying compiler A compilers which employs automated mathematical and logical reasoning to check the correctness of the programs that it compiles. 3

WyIL file A compiled (i.e. binary) form of a Whiley source file. 6

Bibliography

[1] Nancy G. Leveson and Clark S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, 1993.

[2] Mark W. Eichin and Jon A. Rochlis. With microscope and tweezers: An analysis of the internet virus of November 1988. In *Proc. IEEE Symposium on Research in Security and Privacy*, pages 326–343, 1989.

[3] Software problem led to system failure at dhahran, saudi arabia, gao report #b-247094, 1992.

[4] Ariane 5: Flight 501 failure. report by the enquiry board. Technical report, European Space Agency, 1996.

[5] Tony Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, 2003.

[6] S. King. *A Program Verifier*. PhD thesis, Carnegie-Mellon University, 1969.

[7] L. Peter Deutsch. *An interactive program verifier*. Ph.d., 1973.

[8] D. I. Good. Mechanical proofs about computer programs. In *Mathematical logic and programming languages*, pages 55–75, 1985.

[9] D. C. Luckham, S. M. German, F. W. von Henke, R. A. Karp, P. W. Milne, D. C. Oppen, W. Polak, and W. L. Scherlis. Stanford pascal verifier user manual. Technical Report CS-TR-79-731, Stanford University, Department of Computer Science, 1979.

[10] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. SRC Research Report 159, Compaq Systems Research Center, 1998.

[11] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proc. PLDI*, pages 234–245, 2002.

[12] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, 55(1-3):185–208, March 2005.

[13] Mike Barnett, K. Rustan, M. Leino, and Wolfram Schulte. The spec# programming system: An overview. Technical report, Microsoft Research, 2004.

[14] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the ACM conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 31–41. ACM Press, 1993.

[15] Flemming M. Damm. Subtyping with union types, intersection types and recursive types. volume 789 of *LNCS*, pages 687–706. 1994.

[16] Castagna and Frisch. A gentle introduction to semantic subtyping. In *Proc. ICALP*, pages 198–199, 2005.

[17] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *JACM*, 55(4):19:1–19:64, 2008.